Analysis of Pretrained Language Representation Models for Sentence-level Commonsense Validation

Jamie Grech

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Science in Artificial Intelligence
of the
University of Aberdeen

Department of Computing Science

2021

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed: Jamie Grech

Date: 15th September 2021

Abstract

The task of commonsense validation requires a system to distinguish between natural language statements which make sense to humans and statements that do not based on common sense. This task was originally proposed as part of the International Workshop on Semantic Evaluation in 2020. The original task was presented in the form of pairs of sentences where one is nonsensical and one is sensical and the system to be developed is required to identify which of the sentences is the nonsensical one. This thesis expands the scope of this task by separating these pairs of sentences in order to investigate whether a system can effectively classify individual sentences as either sensical or nonsensical. Transformer-based pretrained language models BERT and ALBERT were used as the base of our classification system and fine-tuned on the dataset provided for the aforementioned challenge. Two different types of models were trained, one for the original sentence pair classification task and one for the individual sentence classification. The best performing models for both tasks were based on the ALBERT language model, with the sentence pair model achieving a test accuracy of 95.6%. This accuracy places the model amongst the top performing models for this task, without using additional data other than the provided dataset, whilst most of the better performing models were supplemented with additional data. The individual sentence model achieved a test accuracy of 89.3%, indicating that the individual sentence nonsense classification task is indeed possible albeit more challenging than the sentence pair task. Moreover, a qualitative evaluation of the individual sentence model was also conducted and a number of suggestions for potential future expansion of this research were made based on observations from the quantitative and qualitative analysis conducted.

Acknowledgements

I would like to thank my academic supervisor Dr. John Paul Vargheese for his guidance throughout the course of this project. I would also like to thank my family, close friends and last but certainly not least my girlfriend for their endless support and encouragement.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Common sense can be defined in many ways, though it is most commonly associated with sound and reasonable judgement in practical matters. However, one key point about common sense is that it is usually developed through practical experience rather than formal training or by following a set of rules. For instance, although tomatoes are scientifically classified as fruits, common sense dictates that it would be rather unusual to put tomatoes in your fruit salad.

Similarly, whilst the sentence "The man poured orange juice on his cereal" is not grammatically or syntactically (Hoque, 2015) incorrect by any means, most humans would agree that it is unusual based on intuition or common sense. This phenomenon inspired Wang et al. (2020a) to present SemEval-2020 Task 4: Commonsense Validation and Explanation (**ComVE**). This task challenged participants to deliver AI-driven systems which can “distinguish a natural language statement that *makes sense* to humans from one that does not, and provide the reasons” (Wang et al., 2020a). This problem is non-trivial due to the aforementioned point that common sense reasoning is typically based on intuition or experience rather than following a formula or a specific set of rules. The challenge was presented in the form of three subtasks described as follows:

Subtask A - Validation: Given two similar statements, identify which of the two statements does not make sense.

Subtask B - Explanation (Multi-Choice): Given a nonsensical statement and three other statements discussing it, identify which of the three statements give the best explanation of why the statement does not make sense.

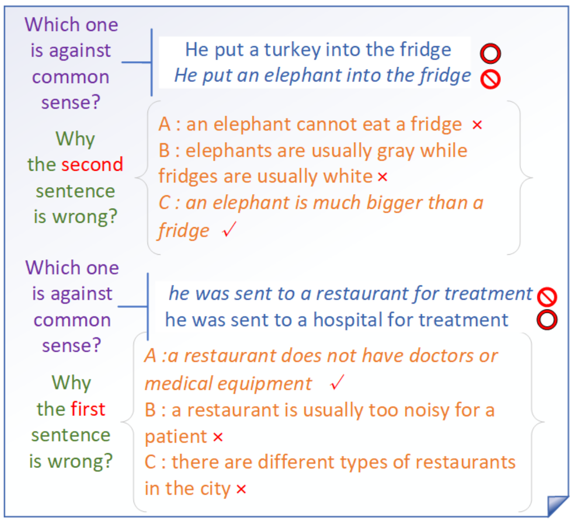
Subtask C - Explanation (Generation): Given a nonsensical statement, generate a statement which gives a reason for why it does not make sense.

Although the scope of this thesis will be limited to the first component of this challenge (Commonsense Validation), it aims to delve deeper into this task by exploring the challenge of identifying nonsensical sentences on an individual basis as opposed to selecting one sentence out of a pair as nonsensical.

1.1 Motivation

The popularity of writing assistant software such as Grammarly is definitely not to be disregarded, with over 30 million people[1] using Grammarly on a daily basis. These tools often review your

[1]https://www.grammarly.com/about



Which one is against common sense?

He put a turkey into the fridge ○

He put an elephant into the fridge ⊘

Why the second sentence is wrong?

A : an elephant cannot eat a fridge ×

B : elephants are usually gray while fridges are usually white ×

C : an elephant is much bigger than a fridge ✓

Which one is against common sense?

he was sent to a restaurant for treatment ⊘

he was sent to a hospital for treatment ○

Why the first sentence is wrong?

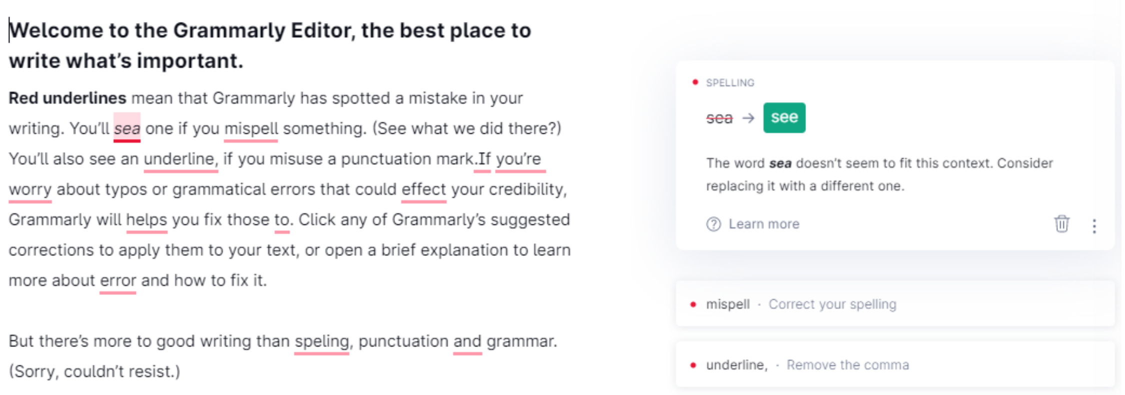
A :a restaurant does not have doctors or medical equipment ✓

B : a restaurant is usually too noisy for a patient ×

C : there are different types of restaurants in the city ×

Figure 1.1: Example of SemEval-2020 Task 4 data (Wang et al., 2020b)

writing for spelling, grammar and punctuation errors and sometimes even go so far as to review the style and tone of your writing. However, determining whether your text includes nonsensical statements such as the one shown in the previous section often escapes the scope of these writing assistants. Although one could argue that such mistakes are not as objectively classifiable as factors like grammatical correctness and punctuation errors, having the option to highlight such statements as potential errors could contribute to further improving the functionality and perhaps popularity of such tools.



Welcome to the Grammarly Editor, the best place to write what's important.

Red underlines mean that Grammarly has spotted a mistake in your writing. You'll sea one if you mispell something. (See what we did there?) You'll also see an underline, if you misuse a punctuation mark.If you're worry about typos or grammatical errors that could effect your credibility, Grammarly will helps you fix those to. Click any of Grammarly's suggested corrections to apply them to your text, or open a brief explanation to learn more about error and how to fix it.

But there's more to good writing than speling, punctuation and grammar. (Sorry, couldn't resist.)

SPELLING

~~sea~~ → see

The word ***sea*** doesn't seem to fit this context. Consider replacing it with a different one.

mispell · Correct your spelling

underline, · Remove the comma

Figure 1.2: Grammarly writing assistant interface

Other than the aforementioned direct application of this research, consisting of implementing such functionality into writing assistant software, the ability to imbue Natural Language Processing (NLP) systems with commonsense knowledge could also result in improved performance on other NLP tasks (Sap et al., 2020), serving as further motivation for research in this area.

However, both of the aforementioned would likely require a system that is capable of identifying individual nonsensical statements to be significantly effective, which is why this thesis aims to delve further into this problem and not confine itself to the original scope of the commonsense validation subtask.

1.2 Aims and Objectives

Building off the points made in the previous two sections, the scope of this thesis can be divided into two primary objectives as follows:

1. Explore and attempt the original SemEval-2020 Task 4 Subtask A, aiming to achieve results comparable or superior to those of the highest ranked competitors.

2. Implement and investigate the effectiveness of a system for identifying individual nonsensical sentences instead of selecting one nonsensical sentence out of a sentence pair.

Tackling these objectives and evaluating their results should allow us to answer the research questions “Is it possible to classify individual sentences as sensical or nonsensical using pretrained language models? If so, to what degree of effectiveness can this be done?”.

1.3 Outline

This thesis is structured into the following six chapters:

1. Introduction: An overview of the objectives of the project being tackled and the motivation behind it.

2. Background Research: An explanation of the fundamental concepts relating to the project and the solutions to be developed.

3. Literature Review: An overview of the state-of-the-art technologies relevant to the scope of the project as well as the methods adopted and results achieved by the participants of the task being tackled in the project.

4. Methodology: A description of the decisions taken in the design of the solution ranging from data preparation to model implementation, training and evaluation.

5. Evaluation: A critical analysis of the results achieved when training and testing the developed models.

6. Conclusion: A concise summary of the work done and results obtained, as well as the limitations encountered and suggestions for related further research.

Chapter 2

Background Research

The material covered in this chapter should help give an understanding of the concepts behind the key technologies and methods to be used in this project.

2.1 Neural Networks

2.1.1 Fundamental Concepts

The concept of neural networks originally stems from a paper written by two scientists, a neurophysiologist and a mathematician, inspired by the human nervous system (McCulloch and Pitts, 1943). In human biology, neurons, also known as nerve cells, serve as "the fundamental units of the brain and nervous system" (Woodruff, 2019).

Figure 2.1 shows a visualisation of the first computational neuron model proposed by McCulloch and Pitts (1943), consisting of a neuron which takes an arbitrary number n of boolean inputs $x_n \in \{0,1\}$. The first part of the neuron then aggregates the inputs, after which the second part of the neuron outputs a decision in the form of a boolean value y based on the aggregated value.

This computational model inspired by the biological neuron served as the inspiration for Artificial Neural Networks (ANNs), more commonly known as simply Neural Networks (NNs). Neural networks typically consist of at least two interconnected layers each consisting of a number of neurons, with the first layer being the input layer and the last layer being the output layer. Multilayer Perceptrons (MLPs), which are considered to be the most commonly used class of neural networks, consist of at least one hidden layer between the input layer and output layer (Fyfe, 2005; Zou et al., 2008). Moreover, when a neural network contains more than one hidden layer between the input and output layers, it is considered a Deep Neural Network (DNN) (Bengio,

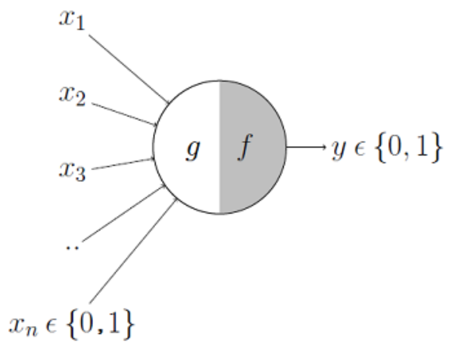


Figure 2.1: The McCulloch-Pitts Neuron model (Chandra, 2018)

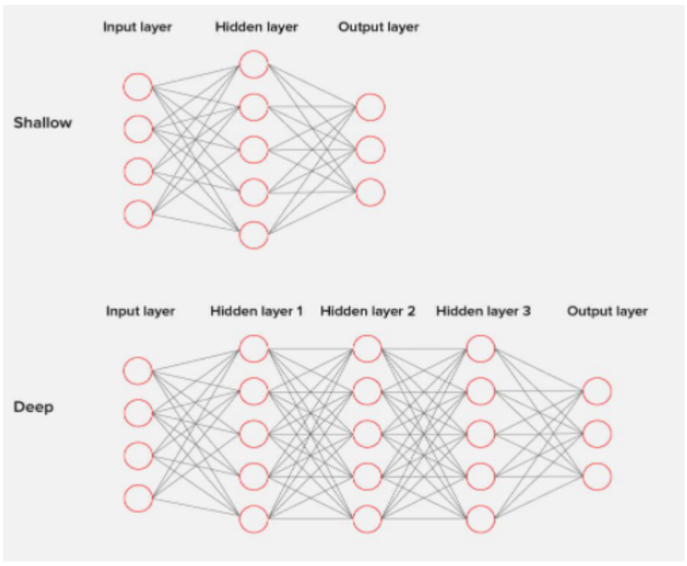


Figure 2.2: The difference between shallow and deep neural networks (Gavrilova, 2020)

2009). Neural networks are often employed for a wide range of tasks including but not limited to classification, optimisation, data compression and approximation (Fyfe, 2005).

The other key components which are found in all neural networks besides layers and neurons are edges, weights and activation functions. Although not being a necessary component of a neural network, bias neurons can also be found in the vast majority of neural networks (Gavrilova, 2020). These aforementioned components can be described as follows:

Edge These serve to connect the neurons in the network by feeding the output from a layer's neurons as input to the next layer's neurons.

Weight Each edge has a weight attached to it which determines how influential its input is in the receiving neuron.

Activation Function The activation function is the function which maps the aggregated result of a neuron's inputs to the neuron's output. A number of different activation functions can be used and the choice of activation functions can vary based on a number of factors, such as where in the network the neuron is and what the purpose of the particular network is.

Bias The bias is a constant which is added to the inputs of a neuron before feeding them into the activation function. It is hence used to shift the activation function in the same way the constant or y-intercept would shift a linear function.

2.1.2 Training

The training of most neural networks is done using an algorithm known as backpropagation (Fyfe, 2005). This process typically starts by initialising all weights in the network to small random numbers, after which a set of input data used to train the network (referred to as the *training set*) is applied to the input layer and propagated forward through the network. Once the activation from this input data reaches the output neurons, the output from the output layer is compared to the desired output, and the error or *loss* is calculated based on some loss function, with a higher loss indicating a larger difference between the achieved output and the desired output. This loss is then propagated back through the network towards the input layer, adjusting the weights on the network accordingly. One full pass of the training data through the backpropagation process is typically referred to as a training *epoch*. This process can be carried out in one of three ways, batch, online or minibatch mode, with the minibatch approach being the most commonly favoured approach for most modern applications (Fyfe, 2005; Li et al., 2014).

Batch The entire training set is presented to the network, the total loss is calculated and the weights are only updated once at the end of every training epoch.

Minibatch The training set is passed through the network in smaller batches known as minibatches, the size of which determined by a manually set parameter (*hyperparameter*) referred to as the *batch size*. The weights of the model are updated in a similar manner to the batch mode however they are updated after every minibatch as opposed to at the end of every full training epoch.

Online The training set is passed through the network one example at a time and the weights of the model are updated after every training example.

Other than the training set, two other sets of data referred to as the validation set and test set are often used in the process of training a neural network model. The validation set is a set of data which is not used to update the weights of the model, but typically passed through the model after every training epoch to gauge the effectiveness of the model on unseen data and thus give an indication of what kind of changes need to be made to the model or training process, if any. Similarly, the test set is also a set of data unseen by the model during training, but it is held out of the training process in its entirety and only used on the final model so as to allow for a truly unbiased estimation of the model's capability to generalise on unseen data (James et al., 2013).

2.1.3 Regularisation

Possibly the most commonly encountered issue in machine learning and especially deep learning is the issue of overfitting, which in its simplest terms can be defined as the development of a model which is either more complex or flexible than it needs to be (Hawkins, 2004). What is specifically meant by the issue of a model being too flexible is when a model adapts too well to the training data, to the point where its ability to generalise is hindered, causing a decrease in performance when used on previously unseen data (Ramsundar and Zadeh, 2018). Regularisation can hence be described as a strategy employed to reduce or prevent overfitting in a model and thus improve the model's ability to generalise.

Among the most prevalent methods of regularisation in neural network models since its introduction by Hinton et al. (2012) is *dropout*. This method consists of randomly deactivating a percentage (known as the *dropout rate*) of the neurons in a layer, with Hinton et al. opting for a dropout rate of 0.5 or 50% on the hidden layers of the networks they evaluated, whilst also reporting promising results with a lower dropout rate of 0.2 or 20% applied to the input layer. Employing dropout in a neural network model serves as a highly effective way of reducing or eliminating co-adapatation, which can be described as a phenomenon whereby the model trained is heavily reliant on some particular units rather than the structure of the network as a whole.

Perhaps the simplest method of regularisation and certainly another amongst the most commonly used is the *early stopping* method. The premise of this method is simply to monitor the validation loss after every training epoch and cease training in the case of a consistent decline in validation performance. In the case of an early termination of the training process due to early stopping, the model can either be taken as is or reverted to the last state before the downturn in validation performance (Ramsundar and Zadeh, 2018).

Weight regularisation is another technique often employed for regularisation of neural network models which involves the modification of the loss function by adding a weight penalty. There are many different types of weight regularisation strategies available, with L1 regularisation and L2 regularisation being the most common. *Weight decay* is another type of weight regularisation strategy which is commonly confused with or thought to be equivalent to L2 regularisation, however, this is not the case. Other types of weight regularisation exist and the penalty applied to the loss function varies between them, however, the core idea behind weight regularisation is that it serves to control the weight coefficients in the network so as to prevent them from taking extreme values (Ramsundar and Zadeh, 2018; Loshchilov and Hutter, 2017, 2018).

2.1.4 Transformer models

The work done by Vaswani et al. (2017) revolutionised the field of machine learning based NLP by introducing a new neural network architecture known as the transformer architecture. This new model architecture has since inspired a radical shift in state-of-the-art approaches for text related machine learning tasks from the previously favoured Recurrent Neural Network (RNN) models such as Long Short-Term Memory (LSTM) (Qiu et al., 2020).

The transformer architecture can be split into two main segments, the encoder and decoder stack. In the original model proposed by Vaswani et al., the encoder and decoder stacks each consisted of 6 identical layers, with the purpose of the encoder being to encode the input sequence (which is first transformed into a learned embedding) into a high-dimensional intermediate representation, specifically a 512-dimensional vector. The decoder then takes this final encoded state as well as the target sequence, effectively learning a mapping from the high-dimensional intermediate representation to the desired output. This kind of model can also be referred to as a sequence-to-sequence (Seq2Seq) model and is often used for machine translation tasks as can be seen in Figure 2.3 (Versloot, 2021).

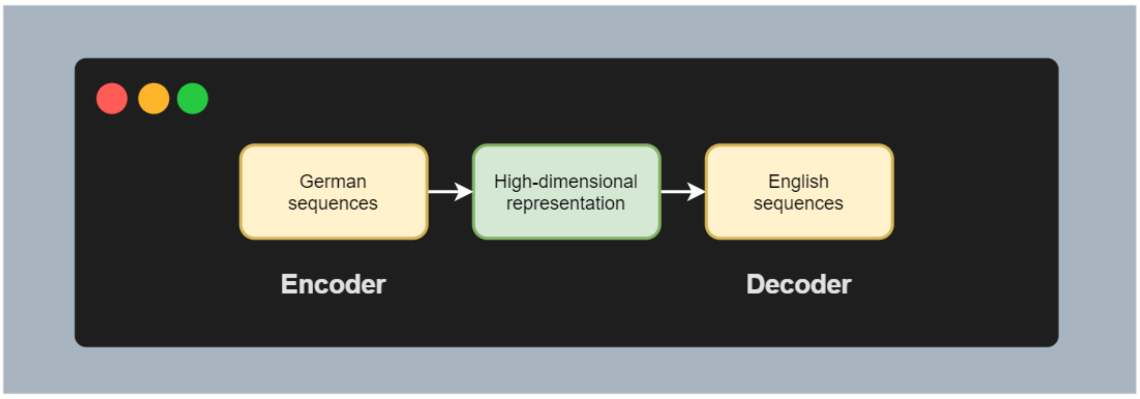


Figure 2.3: Example of an encoder-decoder architecture (Versloot, 2021)

Whilst the introduction of the transformer architecture has driven great advances in Seq2Seq modelling, it also inspired researchers to devise new classes of transformers better suited for a range of other NLP tasks. More specifically, these can be divided into autoregressive transformers and autoencoding transformers. The key differences between these classes of transformer models are that autoregressive models such as GPT (Radford et al., 2018) rely mainly on the decoder portion of the original transformer model and cater towards tasks involving repeated generation such as question answering tasks in Natural Language Generation (NLG). On the other hand, autoencoding models such as BERT (Devlin et al., 2018) are more reliant on the encoder portion of the transformer model and are more focused on tasks dependent on Natural Language Understanding (NLU) such as sentence classification and multiple choice classification[1] (Versloot, 2021).

One final important distinction which must be made when discussing the different types of transformers is that the difference between autoregressive, autoencoding and Seq2Seq transformers is the task which the models are pretrained on rather than the architectures of the models themselves. This means that whilst it might not always be optimal, the same architecture used for an autoencoding model can be used for an autoregressive model and vice versa. Furthermore, a model trained for autoencoding can also be fine-tuned on a task typically suited more for an autoregressive model and vice versa (Versloot, 2021).

2.2 Transfer Learning

2.2.1 Introduction and Definitions

Inductive transfer or *transfer learning* in the field of machine learning is the concept focused on retaining knowledge gained from solving a particular problem and reapplying it in order to help solve a different but related problem (West et al., 2007). Due to the popularity and effectiveness of large scale deep neural networks in the fields of computer vision (Redmon et al., 2016; He et al., 2016; Chollet, 2017) and NLP as well as the extensive quantity of data often required to train such models well, the use of transfer learning has gathered a particular interest within these fields (Weiss et al., 2016; Tan et al., 2018; Alyafeai et al., 2020).

In practice, transfer learning for a classification task can be summed up in a few short steps (Jimoh, 2020):

1. Remove the output layer of the pretrained network you are using.
2. Add a new fully-connected/dense layer to the end of the network with an output dimension corresponding to the number of classes in your new task.
3. Randomize the weights of the new output layer.
4. Freeze the weights of the pretrained layers.
5. Train the resultant network on the data for the new task to update the weights of the new output layer.

The aforementioned steps should serve as a rough outline to how the process of transfer learning should generally be approached. However, certain steps may differ according to the

[1]https://huggingface.co/transformers/model_summary.html

strategy being employed or the pretrained model being used. For instance, in certain cases one might want to remove a few more layers than just the last layer (Jimoh, 2020). Another such example is that in some cases, one might want to keep the weights of some or all of the pretrained layers unfrozen (Devlin et al., 2018; Radford et al., 2018).

2.2.2 Transfer Learning with Pretrained Language Representations

Devlin et al. (2018) state that there are two main existing strategies in the context of transfer learning to apply pretrained language models to down-stream tasks, the *feature-based* approach and the *fine-tuning* approach. Whilst the feature-based approach, such as the one adopted by ELMo (Peters et al., 2018), makes use of "task-specific architectures that include the pre-trained representations as additional features", the fine-tuning approach aims to minimize task-specific parameters by "simply fine-tuning *all* pre-trained parameters" (Devlin et al., 2018).

Since the proposal of the BERT language representation model in 2018 (Devlin et al., 2018), a number of pretrained language models have emerged which have achieved state-of-the-art performance on some of the most popular benchmark datasets for NLP tasks such as the GLUE benchmark tasks (Wang et al., 2018), the SQuAD v1.1 task and its extension in the SQuAD v2.0 task (Rajpurkar et al., 2016), and the SWAG dataset (Zellers et al., 2018). Moreover, using a fine-tuning approach with BERT-based models in combination with supplementary domain-specific pretraining corpora has also proven to allow for state-of-the-art performance in certain domain specific tasks, such as the Legal-BERT (Zheng et al., 2021) model which provides state-of-the-art performance on the Terms of Service legal database (Lippi et al., 2019).

Chapter 3

Literature Review

This chapter will cover the relevant literature to the challenge at hand and the technologies and methods which have been used to solve it, as well as the models which will be used to tackle the objectives of this thesis.

3.1 SemEval-2020 Task 4

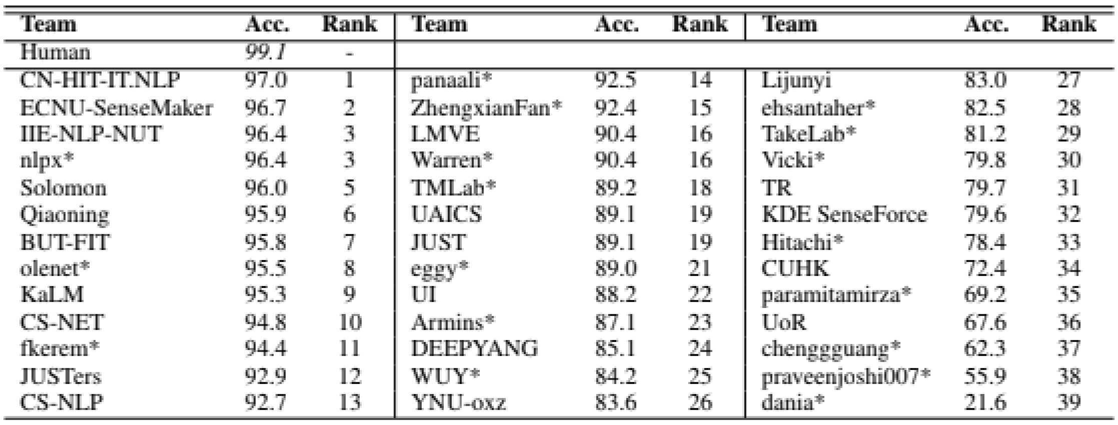
3.1.1 Task Description

The challenge brought forward by Wang et al. (2020a) known as SemEval-2020 Task 4: Commonsense Validation and Explanation consisted of three subtasks. As described earlier in the Introduction, the first subtask, which this thesis will be focusing on, deals with the problem of sentence pair classification. More specifically, identifying the nonsensical sentence out of a pair of similar sentences. Meanwhile, Subtask B and Subtask C deal with multiple choice classification and sentence generation respectively, with the former requiring the selection of 1 out of 3 sentences which correctly describes why a source sentence is nonsensical, while the latter requires the system in question to generate such a reason itself.

The dataset[1] used in the challenge was constructed using the crowdsourcing platform Amazon Mechanical Turk. The first step involved asking workers to write a sensical and nonsensical statement as is required for Subtask A, as well as a further three sentences, one of which is meant to explain why the nonsensical statement does not make sense while the other two are meant to be the incorrect/confusing options for Subtask B. The next step involved asking two separate workers to write a sentence each explaining why one of the nonsensical statements does not make sense. These two reasons would then be used alongside the correct statement from Subtask B to evaluate the performance of the systems on Subtask C. The authors stated that the reason separate workers were asked to write the reference reasons for Subtask C as opposed to having the same worker from the first step write all three reasons was to encourage diversity in the reference answers (Wang et al., 2020a).

The conclusion of the Mechanical Turk experiment resulted in a dataset which was preemptively split into training, testing and validation/development sets by the researchers proposing the challenge. The training set consisted of 10,000 rows of data each having seven sentences, with the first two being the sensical and nonsensical statements, the next three being the multiple choice options for Subtask B out of which one is the correct reason, and the final two being the additional

[1]https://github.com/wangcunxiang/SemEval2020-Task4-Commonsense-Validation-and-Explanation/tree/master/ALL%20data



Team	Acc.	Rank	Team	Acc.	Rank	Team	Acc.	Rank
Human	*99.1*	-						
CN-HIT-IT.NLP	97.0	1	panaali*	92.5	14	Lijunyi	83.0	27
ECNU-SenseMaker	96.7	2	ZhengxianFan*	92.4	15	ehsantaher*	82.5	28
IIE-NLP-NUT	96.4	3	LMVE	90.4	16	TakeLab*	81.2	29
nlpx*	96.4	3	Warren*	90.4	16	Vicki*	79.8	30
Solomon	96.0	5	TMLab*	89.2	18	TR	79.7	31
Qiaoning	95.9	6	UAICS	89.1	19	KDE SenseForce	79.6	32
BUT-FIT	95.8	7	JUST	89.1	19	Hitachi*	78.4	33
olenet*	95.5	8	eggy*	89.0	21	CUHK	72.4	34
KaLM	95.3	9	UI	88.2	22	paramitamirza*	69.2	35
CS-NET	94.8	10	Armins*	87.1	23	UoR	67.6	36
fkerem*	94.4	11	DEEPYANG	85.1	24	chenggguang*	62.3	37
JUSTers	92.9	12	WUY*	84.2	25	praveenjoshi007*	55.9	38
CS-NLP	92.7	13	YNU-oxz	83.6	26	dania*	21.6	39

Figure 3.1: SemEval-2020 Task 4 Subtask A participant results (Wang et al., 2020a)

reference reasons explaining why the nonsensical statement does not make sense. Similarly, the test set and validation set contained 1,000 and 997 rows of data respectively in the same format. As pointed out by the team which placed first in Subtask A (Zhang et al., 2020), it is worth noting that out of the 10,000 rows of data in the training set, five contain identical sentences for the sensical and nonsensical statement.

Another point of note is that Wang et al. (2020a) also preemptively formatted the data in three separate ways to facilitate the completion of each individual subtask. Most notably for the purpose of this thesis, the training, testing and validation sets for subtask A each consist of two separate files. The first file in each of these sets contains only the sensical and nonsensical statements from the original dataset in a random order, whilst the second file contains the respective binary label, with a 0 indicating that the first of the two sentences is the nonsensical one and a 1 indicating the opposite. If we ignore the aforementioned five repeated sentence pairs, this leaves us with a set of 9,995 pairs of sentences and their respective labels in the training set, or 19,990 individual sentences, with a perfect one to one split of sensical and nonsensical statements.

3.1.2 Challenge Results

The table displayed in Figure 3.1 shows the results obtained on Subtask A of SemEval-2020 Task 4 by the 39 teams which opted to participate, where the teams marked with an asterisk (*) are those which did not submit a system description paper to accompany their submitted systems. It is also worth noting that the human benchmark which the systems are being compared against was obtained from a trial dataset separate from the test dataset which the systems are evaluated on.

The two highest ranked teams, CN-HIT-IT.NLP (Zhang et al., 2020) and ECNU-SenseMaker (Zhao et al., 2020) both used structured data from an external knowledge base known as ConceptNet (Speer et al., 2017) to enhance their models, a strategy inspired by K-BERT (Liu et al., 2020b). This external data in the form of knowledge graph triples was used in addition to transformer models to help inject them with commonsense understanding, with ALBERT (Lan et al., 2019) and RoBERTa (Liu et al., 2019) being used as the encoders by CN-HIT-IT.NLP and ECNU-SenseMaker respectively.

Despite the highest performing teams using additional structured data to achieve optimal performance, it can be seen from their results that fine tuning the pretrained language models on the dataset without using additional resources can produce results very closely comparable to the best performing systems. For instance, CN-HIT-IT.NLP (Zhang et al., 2020) also produced an ALBERT model without knowledge graph triples which achieved an accuracy of 96%, just 1% less than the performance of their K-BERT inspired model. Similarly, Solomon (Srivastava et al., 2020) achieved the same accuracy of 96% using RoBERTa as an encoder without the use of additional resources.

As can be expected, autoencoding transformer based systems were the most popular approach for this subtask by an overwhelming margin. Out of the 21 teams which submitted system description papers including a system for Subtask A:

- Five teams (Cusmuliuc et al., 2020; Bai and Zhou, 2020; Ou and Li, 2020; Mendbayar and Aono, 2020; Wang et al., 2020c) presented BERT-based (Devlin et al., 2018) systems.
- Four teams (Zhang et al., 2020; Jon et al., 2020; Liu et al., 2020a; Li et al., 2020) presented ALBERT-based (Lan et al., 2019) systems.
- Ten teams (Zhao et al., 2020; Xing et al., 2020; Srivastava et al., 2020; Wan and Huang, 2020; Dash et al., 2020; Fadel et al., 2020; Saeedi et al., 2020; Doxolodeo and Mahendra, 2020; Teo, 2020; Markchom et al., 2020) presented RoBERTa-based (Liu et al., 2019) systems.

The remaining two teams (Liu, 2020; Mohammed and Abdullah, 2020) presented systems composed of ensembles of the aforementioned three transformer models alongside the autoregressive transformer XLNet (Yang et al., 2019).

Although it is an autoregressive pretrained model, XLNet has produced state-of-the-art results on a number of text classification tasks, as discussed further in Section 3.2.3. Notwithstanding this, it was relatively unrepresented in this challenge, being used in only two systems, both of which utilised it alongside its autoencoding counterparts in BERT, ALBERT and RoBERTa as part of an ensemble system.

3.2 Pre-trained Language Models

In this section the literature pertaining to the models which will form the core of the methodology of this thesis (BERT and ALBERT) will be explored on a technical level. It will also give a less detailed overview of some similar models which could prove useful for tasks such as the one being tackled in this thesis if further work were to be done on it.

3.2.1 BERT

In 2018, a group of researchers from Google AI Language introduced a new language representation model called BERT, short for Bidirectional Encoder Representations from Transformers. This model was not only the "first fine-tuning based representation model that achieves state of the art performance on a large suite of sentence-level *and* token-level tasks" (Devlin et al., 2018), but also the inspiration for numerous models which would go on to push this state-of-the-art even further (Lan et al., 2019; Liu et al., 2019; Sanh et al., 2019; Conneau et al., 2019; Le et al., 2019; Lewis et al., 2019; Liu et al., 2020c).

Devlin et al. (2018) argued that previously proposed models such as ELMo (Peters et al., 2018) and GPT (Radford et al., 2018) restricted the power of pre-trained language representations

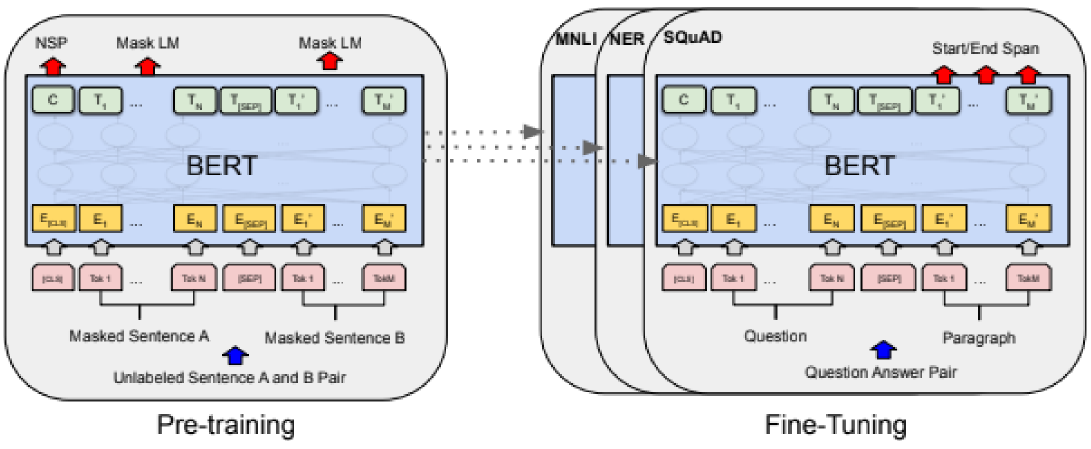


Figure 3.2: Pretraining and fine-tuning process of the original BERT model (Devlin et al., 2018)

for down-stream tasks, largely due to their unidirectional nature. Unidirectional language models are typically trained to predict a word given the words before it in the sentence, also known as the *left context* (Wu, 2018).

The researchers behind BERT aimed to alleviate the aforementioned constraint imposed by these unidirectional models by instead training their model using a *masked language model* (MLM) objective. The first step of the training process for this objective involves corrupting a specified percentage of tokens in the input sentences (15% in the case of BERT) by masking/replacing them with either a special mask token (80% likelihood), a random token different from the one masked (10% likelihood) or the same token (10% likelihood). After this step is complete, the sentence with the masked tokens is then given to the model, from which the model must predict the original sentence. Due to the fact that the model has access to the full context (left and right) for every token, this training objective allows for the training of a deep *bidirectional* transformer, as opposed to typical left-to-right language model pretraining (Devlin et al., 2018).

Other than the MLM task, the BERT pretraining procedure also includes a second training task: Next Sentence Prediction (NSP) (Devlin et al., 2018). During pretraining, the model is given two sentences at a time, as can be seen in Figure 3.2. For each pair of sentences, there is an equal chance that sentence B is either the next sentence in the corpus after sentence A or a random sentence from the same corpus. Hence, for each training example, the model is tasked with predicting the original sentence corresponding to two masked sentences as well as whether the second sentence follows the first one. A special [CLS] token is added to the beginning of each sentence pair sequence which the model is intended to learn to map to the output of this task during the training process. Moreover, a [SEP] token is added at the end of each sentence to separate the first sentence from the second as well as mark the end of the second sentence. This additional training task allows for increased performance on a variety of downstream tasks which are reliant on the relationship between two sentences like question answering or sentence pair classification (such as SemEval-2020 Task 4 Subtask A) tasks.

At the time of BERT's introduction, fine-tuned BERT-based models advanced the state of the art by a significant margin on 11 significant NLP benchmark tasks. Specifically, eight of these tasks were from the GLUE benchmark (Wang et al., 2018), whilst the other three tasks were

SQuAD v1.1, SQuAD v2.0 (Rajpurkar et al., 2016) and SWAG (Zellers et al., 2018).

3.2.2 ALBERT

As one might presume from the name, the A Lite BERT (ALBERT) architecture (Lan et al., 2019) is a BERT-inspired architecture designed with significantly less parameters to address the worrying issue of increasing memory requirements for training of new NLP models. Moreover, it also addresses the issues of BERT's NSP pretraining task highlighted in other prominent research (Yang et al., 2019; Liu et al., 2019) pertaining to language modelling improvements.

In order to tackle the issue of model size, Lan et al. (2019) make two key design choices in the model architecture of ALBERT. The first of these is the decomposition of the vocabulary embedding matrix into two smaller matrices by implementing an embedding layer prior to the hidden layers with a smaller size E than the hidden layer size H of the model. Given a vocabulary size V, this technique reduces the embedding parameters from $(V \times H)$ to $(V \times E + E \times H)$. This decrease in parameters is increasingly significant the larger the hidden size is than the embedding size, especially considering the relatively much larger vocabulary size which is typically deemed as a necessity for such language models. For instance, in the case of the ALBERT-xxlarge model which had *V = 30000*, *H=4096* and *E=128*, the model would have $30000 \times 4096 = 122,880,000$ embedding parameters without this optimization whilst with the embedding layer this is brought down to $30000 \times 128 + 128 \times 4096 = 4,364,288$ embedding parameters. When comparing to BERT with all other conditions equal, this optimization allowed ALBERT to achieve an 80% reduction in embedding parameters with only a miniscule drop in performance (Lan and Soricut, 2019) which is far outweighed by the much more significant potential of an increased hidden layer size.

The second approach taken by Lan et al. (2019) to reduce model size is the implementation of parameter sharing between layers. This decision was taken based on the observation of redundancy in Transformer-based models like BERT (Devlin et al., 2018), RoBERTa (Liu et al., 2019) and XLNet (Yang et al., 2019) whose layers often appeared to perform similar operations to each other whilst using different parameters. This is eliminated by ALBERT's approach of sharing parameters across the layers by stacking the same layer on top of each other. Similarly to the previous optimization, parameter sharing between layers allowed ALBERT to achieve a 70% reduction in overall parameters whilst only slightly diminishing the accuracy (Lan and Soricut, 2019).

Subsequent studies to BERT (Yang et al., 2019; Liu et al., 2019) found that the impact of NSP as an additional training task alongside the MLM task was unreliable and opted to eliminate it entirely, which resulted in an improvement in performance across many downstream tasks. The researchers behind ALBERT make the following speculation about NSP's ineffectiveness:

> We conjecture that the main reason behind NSP's ineffectiveness is its lack of difficulty as a task, as compared to MLM. As formulated, NSP conflates *topic prediction* and *coherence prediction* in a single task. However, topic prediction is easier to learn compared to coherence prediction, and also overlaps more with what is learned using the MLM loss (Lan et al., 2019).

.

Model	# Layers	# Parameters	Hidden Size	Encoding Size
bert-large-uncased (Devlin et al., 2018)	24	336M	1024	1024
distilbert-base-uncased (Sanh et al., 2019)	6	66M	768	768
roberta-large (Liu et al., 2019)	24	355M	1024	1024
xlnet-large-cased (Yang et al., 2019)	24	340M	1024	1024
albert-xxlarge (Lan et al., 2019)	12	223M	4096	128

Table 3.1: Summary of most popular pretrained language models from recent literature

Notwithstanding this speculation, as opposed to the approaches taken by XLNet (Yang et al., 2019) and RoBERTa (Liu et al., 2019), Lan et al. (2019) maintain their belief in the importance of inter-sentence modelling for the purpose of language understanding and propose a replacement for the NSP loss instead of eliminating it entirely. They propose the Sentence-Order Prediction (SOP) loss as a replacement, a task which is relatively similar to NSP but neglects the *topic prediction* task in favour of focusing on the *coherence prediction* aspect. Similarly to NSP, this task uses two consecutive sentences from the same document as the positive cases, but uses the same two sentences with the order swapped as the negative case instead of two unrelated sentences. In the evaluation carried out by Lan et al. (2019), it was found that the model trained on the SOP task can solve the NSP task "to a reasonable degree", whilst the same cannot be said for the NSP model on the SOP task. This resulted in ALBERT models experiencing an improvement in downstream task performance for multi-sentence encoding tasks (Lan et al., 2019).

The most successful ALBERT model, ALBERT-xxlarge, advanced the state of the art on the majority of GLUE benchmark (Wang et al., 2018) tasks, as well as SQuAD v1.1 and v2.0 (Rajpurkar et al., 2016) tasks (Lan et al., 2019). Notably, this model also made a very significant advancement in the state-of-the-art for the RACE dataset (Lai et al., 2017), which is a dataset intended to evaluate the NLU capability of a model by administering a test similar to a reading comprehension test. The performance of ALBERT-xxlarge on this dataset yielded a performance slightly worse than that of the previous state-of-the-art model in RoBERTa when trained on the same dataset as BERT, however this performance increased significantly when ALBERT-xxlarge was pretrained on the same larger dataset as RoBERTa, surpassing it's performance of 83.2% by an absolute 6.2% and setting a new state-of-the-art score at 89.4% (Lan and Soricut, 2019).

3.2.3 Related models

Whilst the previously described BERT and ALBERT models will be the ones implemented in this project, this section will give an overview of some of the other popular pretrained language models from recent literature as can be seen in Table 3.1.

The first model of note which has also been referred to in Section 3.1.2 is RoBERTa (Liu et al., 2019). This model was the most represented model in SemEval-2020 Task 4 (Wang et al., 2020a). RoBERTa is a BERT-inspired model similar to ALBERT, however unlike ALBERT the differences between RoBERTa and BERT all lie within the pretraining process rather than within the model architecture itself. The first of these differences is the elimination of the NSP loss as previously mentioned in Section 3.2.2. RoBERTA also uses a larger batch size, as well as masking tokens differently at each training epoch whereas BERT masks them once and for all. The final distinction between BERT and RoBERTa is that RoBERTa utilises Byte-Pair Encoding (BPE)

(Sennrich et al., 2015) using an implementation proposed by Radford et al. (2019) for the vocabulary embedding process. This technique allows RoBERTa to learn a subword vocubulary which can “encode any input text without introducing any ‘unknown’ tokens” (Liu et al., 2019) with a byte-level vocabulary containing 50,000 units as opposed to the character-level 30,000 unit vocabulary used by BERT and ALBERT (Devlin et al., 2018; Lan et al., 2019). These optimizations allowed RoBERTa to achieve state-of-the-art in all the GLUE (Wang et al., 2018) benchmark tasks as well as the RACE (Lai et al., 2017) dataset upon its introduction, although it was overtaken in most of these tasks by ALBERT (Lan et al., 2019) later on.

The last language model which was significantly represented in SemEval-2020 Task 4 and not yet described is XLNet (Yang et al., 2019). This is a non-traditional autoregressive model which takes inspiration from BERT and employs strategies which allow it to learn bidirectional contexts and utilize its autoregressive nature to overcome certain limitationss of BERT observed by Yang et al. (2019). Whilst its performance on previously mentioned benchmark tasks such as GLUE, SQuAD, SWAG and RACE (Wang et al., 2018; Rajpurkar et al., 2016; Zellers et al., 2018; Lai et al., 2017) surpasses that of BERT, it falls short on these tasks when compared to ALBERT, as was also the case with RoBERTa. It also appears to fall short of ALBERT and RoBERTa in the SemEval-2020 Task 4 (Wang et al., 2020a) being tackled in this project. However, XLNet still provides state-of-the-art performance on numerous other text classification datasets, particularly those in the field of sentiment analysis such as AG News, DBPedia, Yelp Reviews, Amazon Reviews (Maas et al., 2011), Stanford Sentiment Treebank (SST) (Socher et al., 2013) and the IMDB dataset (Zhang et al., 2015), despite being an autoregressive transformer.

Chapter 4

Methodology

This chapter will cover the decisions made when designing and implementing the solution to tackle the objectives and research questions mentioned in Section 1.2.

4.1 Experimental Setup

4.1.1 Hardware and Software

The training experiments conducted for the purpose of this thesis were run using a mix of Google Colaboratory[1] (Colab) with GPU support and a personal computer with the following components:

GPU Nvidia GeForce RTX 2070 8GB GDDR6

CPU Intel Core i7-8750H Hexa-Core Processor

RAM 16GB DDR4 2666MHz

In terms of software, both the Google Colab environment and the local environment on the PC setup were running python 3.7 and making use of the following python libraries:

PyTorch (Paszke et al., 2019) One of the most popular open source machine learning and deep learning libraries for python which also offers CUDA GPU support.

Huggingface Transformers [2] An NLP library which offers access to the state-of-the-art pre-trained language models previously discussed in the Literature Review with inbuilt support for PyTorch.

Pandas (McKinney et al., 2010) A widely used python library for data analysis and manipulation, used to load and preprocess our data.

Scikit-learn (Pedregosa et al., 2011) A machine learning and data analysis oriented python library, used to extract metrics such as accuracy, precision, recall and more for our evaluation.

Tensorboard [3] An experiment tracking toolkit which offers tools for tracking and visualizing hyperparameters and metrics throughout the training process.

---

[1]https://colab.research.google.com

[2]https://huggingface.co/transformers

[3]https://www.tensorflow.org/tensorboard

On a final note, some of the code from the python notebooks provided in the Huggingface Transformers documentation[4] were used in the development of the implementations detailed in this chapter.

4.1.2 Reproducibility

In order to make all experiments conducted in this experiment reproducible, a function was defined to assign all randomization seeds affecting the experiment to a given value as shown in Listing 4.1 below.

```python
def set_seed(seed):
  """ Set all seeds to make results reproducible """
  torch.manual_seed(seed)
  torch.cuda.manual_seed_all(seed)
  torch.backends.cudnn.deterministic = True
  torch.backends.cudnn.benchmark = False
  np.random.seed(seed)
  random.seed(seed)
  os.environ['PYTHONHASHSEED'] = str(seed)
```

Listing 4.1: Function to set all seeds affecting random operations in our project environment

Since the way randomization occurs in the environment can significantly affect the results obtained, such a function allows for reproducible results and hence helps uphold scientific integrity of the claims made in this report. The seed used throughout all experiments in this project is ‘8’, which is a seed which was randomly selected before experiments began and not changed at any point throughout the process unless otherwise specified.

4.2 Model Development

The following subsections will describe the process behind the implementation of our solution, from the loading and processing of the input data to the training, tuning and evaluation of the classification models.

4.2.1 Data Preparation

```python
def load_sentence_pairs(X_path, y_path):
  X = pd.read_csv(X_path).drop(columns=["id"])
  y = pd.read_csv(y_path, header=None).drop(columns=[0])
  X = X.rename(columns={"sent0": "sentence1", "sent1": "sentence2"})
  y = y.rename(columns={1: "label"})
  df = pd.concat([X, y], axis=1)

  return df
```

Listing 4.2: Function for loading sentence pairs from dataset .csv file

The first step to preparing the data for use in training and evaluating our models is loading the data from the comma separated values (CSV) files provided by the organizers of the SemEval task. This was done using the read_csv function from the pandas library to load the data into a pandas dataframe. To train models for classifying sentence pairs as required for SemEval-2020 Task 4 Subtask A, this simply involved loading the two separate files containing the sentences and

[4] https://huggingface.co/transformers/v4.1.1/notebooks.html

the labels, removing the unnecessary ID columns, renaming the columns (‘sentence1’, ‘sentence2’ and ‘label’), and merging them into a single dataframe. This function can be seen in Listing 4.2.

```
for index, row in y.iterrows():
  # Ignore rows where both sentences are the same
  if X["sent0"][index].lower() != X["sent1"][index].lower():
    X_new.append(X["sent0"][index])
    X_new.append(X["sent1"][index])
    if y[1][index] == 0:
      y_new.append(1)
      y_new.append(0)
    else:
      y_new.append(0)
      y_new.append(1)
  else:
    print(index)

df = pd.DataFrame({"sentence1": X_new, "label": y_new})
```

Listing 4.3: Loading data as individual sentences with their respective labels

On the other hand, in order to train models for the classification of individual sentences the data must be loaded in an appropriate format. This was done by iterating through the original data and adding each sentence from the sentence pairs into a list, as well as the appropriate label in another list, then merging them into a single two-column dataframe. Specifically, we opt to represent nonsensical statements with the positive label (1) and sensical statements with the negative label (0). This maintains the convention set by the originally provided data, whereby the 0 label indicates that the first sentence (sent0) is the nonsensical one whilst the 1 label indicates that the second sentence (sent1) is the nonsensical one. Hence, when creating the dataframe for the individual labels, the sentences were always added in the same order as they appeared in the original sentence pairs, whilst the order the labels are added in is dependent on the label of the original pair.

When adding the sentences and labels from the original task data to our dataframe we also run a case insensitive comparison of the sentence pairs so as to identify and eliminate the duplicate sentence pairs mentioned by Zhang et al. (2020). Through this measure we identify the sentence pairs with indexes 4075, 4121, 4313, 6398 and 7700 from the SemEval-2020 Task 4 Subtask A (Wang et al., 2020a) training dataset as the only duplicate sentence pairs and manually remove them from our version of the dataset used for all of the experiments detailed throughout this chapter.

```
def MaxWords(df):
  lensentences = []
  lenpairs = []
  for index, row in df.iterrows():
    sen1len = len(df["sentence1"][index].split())
    sen2len = len(df["sentence2"][index].split())
    totallen = sen1len + sen2len
    lensentences.append(sen1len)
    lensentences.append(sen2len)
    lenpairs.append(totallen)
```

```
print("Max length of sentence pairs: ", max(lenpairs))
print("Max length of individual sentence: ", max(lensentences))
```

Listing 4.4: Identifying the maximum length of sentence pairs and individual sentences

After loading the sentence pairs from the dataset, the function shown in Listing 4.4 was used to identify the maximum length in words of sentence pairs as well as individual sentences in the data. Python's split function was used to separate the strings by spaces and hence convert them into lists of individual words, the length of which was then used to determine the number of words in each sentence. For every pair of sentences, the length of the individual sentences was stored as well as the combined length of the two sentences. The longest sentence pair as well as the longest individual sentence between the training, validation and test sets were both found to be in the training set, with the longest sentence pair having 45 words between both of the sentences and the longest individual sentence having 23 words. This information is pivotal in deciding our maximum sequence length for the tokenization process.

```
class CustomDataset(Dataset):

  def __init__(self, data, maxlen, with_labels=True, bert_model='albert-
    xxlarge-v2'):

    self.data = data  # pandas dataframe
    # Initialize the tokenizer
    self.tokenizer = AutoTokenizer.from_pretrained(bert_model)

    self.maxlen = maxlen
    self.with_labels = with_labels

  def __len__(self):
    return len(self.data)
```

Listing 4.5: Creating our custom subclass of the PyTorch dataset class

Since we are working with the PyTorch ML framework, we will be creating our own custom subclass of the PyTorch Dataset class which we will use to create our DataLoaders to load data in a format which can be used to train and evaluate our models[5]. We first override the initialisation function __init__ which allows us to customise properties of the custom dataset object upon initialisation, such as the maximum length the input sequences should be padded or truncated to in the tokenization step, as well as the BERT or ALBERT model the dataset is going to be used for so as to use the appropriate pretrained tokenizer. Moreover, we also override then __len__ function to return the size of our inputted dataset as this function is expected to behave as such by the default PyTorch DataLoader implementation.

```
  def __getitem__(self, index):

    # Selecting sentence1 and sentence2 at the specified index in the data
    frame
    sent1 = str(self.data.loc[index, 'sentence1'])
    # Account for single sentence or sentence pair problems
```

[5]https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset

```
    if 'sentence2' in self.data.columns:
        sent2 = str(self.data.loc[index, 'sentence2'])

        # Tokenize the pair of sentences to get token ids, attention masks and token type ids
        encoded = self.tokenizer(sent1, sent2,
                                 padding='max_length',  # Pad to max_length
                                 truncation=True,  # Truncate to max_length
                                 max_length=self.maxlen,
                                 return_tensors='pt')  # Return torch.Tensor objects
    else:
        # Tokenize the sentence to get token ids, attention masks and token type ids
        encoded = self.tokenizer(sent1,
                                 padding='max_length',  # Pad to max_length
                                 truncation=True,  # Truncate to max_length
                                 max_length=self.maxlen,
                                 return_tensors='pt')  # Return torch.Tensor objects

    token_ids = encoded['input_ids'].squeeze(0)  # tensor of token ids
    attn_masks = encoded['attention_mask'].squeeze(0)
    # binary tensor with "0" for padded values and "1" for the other values
    token_type_ids = encoded['token_type_ids'].squeeze(0)
    # binary tensor with "0" for the 1st sentence tokens & "1" for the 2nd sentence tokens

    if self.with_labels:  # True if the dataset has labels
        label = self.data.loc[index, 'label']
        return token_ids, attn_masks, token_type_ids, label
    else:
        return token_ids, attn_masks, token_type_ids
```

Listing 4.6: The __getitem__ function of our custom Dataset subclass

The final function overriden in our custom subclass is the __getitem__ function, which is the function used by map-style datasets such as the one we are implementing to access and fetch data entries. This function which can be seen in Listing 4.6 first fetches the sentence at the given index from the 'sentence1' column, checks if a 'sentence2' column exists in the dataframe and fetches the sentence in the column if it is found. The sentence or pair of sentences is then tokenized with the pretrained tokenizer from the aforementioned initialisation function. In the case of the BERT and ALBERT models we are using, the tokenizer first sets the sentence to lowercase since the models being used are trained on uncased text. Moreover, a [CLS] token is added to the beginning of each tokenized sequence as well as a [SEP] token after every sentence, as is done in the pretraining process of BERT and ALBERT and described in Section 3.2.1. Finally, the input sentences are padded to the maximum sequence length specified when creating the dataset object using special <pad> tokens. Based on the maximum sequence lengths discussed earlier, the maximum sequence length for the tokenizers in the experiments conducted was set to 32 for individual sentences and 64 for sentence pairs. These values were selected as they allow all sentences and sentence pairs from our dataset to be tokenized without being truncated, whilst still allowing for reasonable time and memory requirements when training our models, as opposed to the maximum sequence length of 512 often used in the pretraining process for transformer-based language models (Devlin et al., 2018; Lan et al., 2019; Liu et al., 2019; Yang et al., 2019).

As shown in the code listing, the BERT and ALBERT tokenizers return three separate tensors for each inputted sequence: the input IDs, the attention masks and the token type IDs. The first tensor, the input IDs, consists of the unique identifier for each word of the sequence within the pre-trained model's vocabulary. If a word is not in the model's vocabulary, it is instead replaced with a special <unk> (unknown) token. Meanwhile, the attention masks and token type IDs tensors, with the former containing 0s for all padded tokens and 1s for all other tokens and the latter containing 0s for tokens belonging to the first sentence and 1s for tokens belonging to the second sentence in the case of sentence pair sequences. These three tensors together are the expected input of the BERT and ALBERT-based models which we shall be using from the Huggingface library.

4.2.2 Model Definition

```python
class BertClassifier(nn.Module):

    def __init__(self, bert_model="albert-xxlarge-v2", freeze_bert=False, dropout_rate=0.2):
        super(BertClassifier, self).__init__()
        #  Instantiating BERT-based model object
        self.bert_layer = AutoModel.from_pretrained(bert_model)

        #  Fix the hidden-state size of the encoder outputs (If you want to add other pre-trained models here, search for the encoder output size)
        if bert_model == "albert-base-v2":  # 12M parameters
            hidden_size = 768
        elif bert_model == "albert-large-v2":  # 18M parameters
            hidden_size = 1024
        elif bert_model == "albert-xlarge-v2":  # 60M parameters
            hidden_size = 2048
        elif bert_model == "albert-xxlarge-v2":  # 235M parameters
            hidden_size = 4096
        elif bert_model == "bert-base-uncased": # 110M parameters
            hidden_size = 768
        elif bert_model == "bert-large-uncased": # 336M parameters
            hidden_size = 1024

        # Freeze bert layers and only train the classification layer weights
        if freeze_bert:
            for p in self.bert_layer.parameters():
                p.requires_grad = False

        self.dropout = nn.Dropout(p=dropout_rate)

        # Classification layer
        self.cls_layer = nn.Linear(hidden_size, 1)
```

Listing 4.7: Subclassing the PyTorch Module class to create a custom BertClassifier class

After overriding the PyTorch Dataset class to create a custom dataset subclass suited to the needs of our project, we similarly subclass the PyTorch module[6] class to create a custom class for BERT and ALBERT-based classification models, as can be seen in Listing 4.7. Similar to what was done with the tokenizer when making our custom dataset class, we use the AutoModel.from_pretrained function from the Huggingface library to load the appropriate pretrained encoder to be used as the base of our classification model according to the model name passed through the bert_model parameter as a string. This string also determines the input size of the linear classification layer which will be added after the BERT or ALBERT encoder to learn our nonsense classification task, as the input size of this layer should match the hidden layer size of the encoder being used. The other two parameters for the initialisation of a BertClassifier object are the freeze_bert parameter, which freezes the weights of all the layers in the pretrained encoder so as to only update the weights of the linear classification layer during training, and the dropout_rate, which determines the dropout rate between the pooled [CLS] token representation from the encoder and the aforementioned linear classification layer.

```
    @autocast()  # run in mixed precision
    def forward(self, input_ids, attn_masks, token_type_ids):
        '''
        Inputs:
            -input_ids : Tensor  containing token ids
            -attn_masks : Tensor containing attention masks to be used to focus on non-padded values
            -token_type_ids : Tensor containing token type ids to be used to identify sentence1 and sentence2
        '''

        # Feeding the inputs to the BERT-based model to obtain contextualized representations
        cont_reps, pooler_output = self.bert_layer(input_ids, attn_masks, token_type_ids)

        # Feeding to the classifier layer the last layer hidden-state of the [CLS] token further processed by a
        # Linear Layer and a Tanh activation. The Linear layer weights were trained from the sentence order prediction (ALBERT)
        # or next sentence prediction (BERT) objective during pre-training.
        logits = self.cls_layer(self.dropout(pooler_output))

        return logits
```

Listing 4.8: Defining the forward pass function of our BertClassifier class

After defining the initialization of the BertClassifier class we then define the forward pass function of our classifier as shown in Listing 4.8. The first step of the forward pass consists of passing the three tensors obtained from tokenising the input sequence, as described in the previous section, to the pretrained BERT or ALBERT encoder. This results in two separate outputs, the contextualized representations and the pooler output. Whilst the former of these contains the last

[6]https://pytorch.org/docs/stable/generated/torch.nn.Module.html

hidden layer representation of each token in the original sequence, the latter contains the last hidden layer representation of the [CLS] token only, however this representation is further processed through an additional pooling layer. This pooling layer consists of an additional linear layer with a Tanh activation function, the weights of which are trained from the NSP task in the case of a BERT model or the SOP task in the case of an ALBERT model as described in Section 3.2.1 and Section 3.2.2 respectively. Given the classification nature of our downstream task, we simply pass the pooler layer output to the linear classification layer, after applying dropout according to the rate specified when initializing the BertClassifier object. The forward pass function then returns the raw prediction outputted by the linear classification layer, to which a sigmoid function can be applied in order to obtain a probability ranging between 0 and 1, corresponding to the model's prediction for which of the respective labels the sequence belongs to. This means that the closer the probability is to 1, the higher the likelihood that the sentence belongs to the nonsensical class, and conversely, a probability closer to 0 indicates a higher likelihood that the sentence belongs to the sensical class.

It is worth noting the @autocast() python decorator applied to the forward pass function. This is part of the PyTorch CUDA Automatic Mixed precision (AMP) API, and allows the operations within the forward pass of the model to run in mixed precision (Micikevicius et al., 2017). This is an amalgamation of training techniques, which involves running certain float point calculations in half precision, using fp16 for the float datatype, as opposed to the single precision or fp32 which is used by PyTorch by default. Using this amalgamation of techniques results in significantly lower training times when training using a GPU with tensor cores, as well as a reduction in GPU memory requirements, without compromising the accuracy of the model.

4.2.3 Training Function

Listing 4.9 below shows the training loop of our model training function, i.e. the loop which controls the iteration over the training set and updating of the model weights at every training epoch. As discussed in the previous section, autocast is invoked for each forward pass of the model, including the computation of the training loss. However, the backward pass is not wrapped in the autocast wrapper as recommended in the PyTorch AMP documentation[7]. Since all models we will be training are binary classification models and the loss is being calculated directly on the raw predictions, i.e. the logits of the model, we use Binary Cross Entropy (BCE) with logits as our loss criterion.

```
 for it, (seq, attn_masks, token_type_ids, labels) in enumerate(tqdm(
train_loader)):

   # Converting to cuda tensors
   seq, attn_masks, token_type_ids, labels = \
     seq.to(device), attn_masks.to(device), token_type_ids.to(device),
labels.to(device)

   # Enables autocasting for the forward pass (model + loss)
   with autocast():
     # Obtaining the logits from the model
     logits = net(seq, attn_masks, token_type_ids)
```

[7]https://pytorch.org/docs/stable/amp.html

```

      # Computing loss
      loss = criterion(logits.squeeze(-1), labels.float())
      # Log training loss after every batch
      writer.add_scalar("Train/loss", loss.item(), (it + 1) + ((ep) * nb_iterations))
      loss = loss / iters_to_accumulate  # Normalize the loss because it is averaged

    # Backpropagating the gradients
    # Scales loss.  Calls backward() on scaled loss to create scaled gradients.
    scaler.scale(loss).backward()

    if (it + 1) % iters_to_accumulate == 0:
      # Optimization step
      # scaler.step() first unscales the gradients of the optimizer's assigned params.
      # If these gradients do not contain infs or NaNs, opti.step() is then called,
      # otherwise, opti.step() is skipped.
      scaler.step(opti)
      # Updates the scale for next iteration.
      scaler.update()
      if lr_scheduler is not None:
        # Adjust the learning rate based on the number of iterations.
        lr_scheduler.step()
      # Clear gradients
      opti.zero_grad()
```

Listing 4.9: The training loop component of the model training function

The iters_to_accumulate variable is a parameter of our training function which allows for the use of gradient accumulation, i.e. accumulating the loss over multiple training batches and only updating the weights of the network every certain number of batches. This effectively allows for training with higher batch sizes without the higher memory requirements, however if gradient accumulation is not desired this parameter can simply be set to 1 which results in the typical model training process.

The scaler object shown in the listing is an instance of the GradScaler class, which is one of the techniques involved in the mixed precision training process. This scales the loss by multiplying it by a scalar number to increase its magnitude and prevent underflowing to 0. This is done before performing the backpropagation or optimization step and the scaling is then reverted before applying the weight update. Since the loss of the network is usually much higher early on in the process, the GradScaler doubles the scalar of multiplication every so often. If no infinite gradients or NaN gradients are found after the unscaling step, the optimizer's step function is invoked, updating the weights of the network. Otherwise, this step is skipped, and in the case that infinite gradients were found, the scalar which the gradients are multiplied by is half. The purpose of this process is to counteract the higher rounding error incurred by half-precision floating point values.

After each training epoch, the model is evaluated on the validation set to obtain the validation loss and accuracy as shown in Listing 4.10. The get_probs_from_logits function shown in this listing simply applies the sigmoid function to the logits from the network and returns the resultant probabilities. These validation set metrics are then used to determine whether early stopping should be invoked to stop training before the specified maximum number of epochs in order to prevent overfitting the model.

```python
with torch.no_grad():
  for it, (seq, attn_masks, token_type_ids, labels) in enumerate(tqdm(dataloader)):
    # Converting data to cuda tensors
    seq, attn_masks, token_type_ids, labels = \
      seq.to(device), attn_masks.to(device), token_type_ids.to(device), labels.to(device)

    # Passing input data through network and calculating loss from output + labels
    logits = net(seq, attn_masks, token_type_ids)
    mean_loss += criterion(logits.squeeze(-1), labels.float()).item()

    # Getting predictions from outputted probabilities
    probs = get_probs_from_logits(logits.squeeze(-1)).squeeze(-1).tolist()
    preds = (pd.Series(probs) >= 0.5).astype('uint8')
    all_preds = all_preds.append(preds, ignore_index=True)
    # Converting labels to CPU tensor so that it can be converted to Series
    all_labels = all_labels.append(pd.Series(labels.cpu()).astype('uint8'), ignore_index=True)

    count += 1

return mean_loss / count, accuracy_score(all_labels, all_preds)
```

Listing 4.10: Evaluating the model on the validation set

After the model is trained for the specified number of epochs or training is stopped due to early stopping, the state dictionary of the model's best state based on validation set metrics is saved to a .pt file using PyTorch's save and state_dict functions. Similarly, these models can then be loaded using the PyTorch load and load_state_dict functions to use the trained models, either to evaluate them and make predictions or to train further.

Model encoder	Learning Rate	Weight Decay	Epochs	Dropout Rate	Batch Size
bert-large-uncased	5e-5	1e-2	5	0.1	16
albert-xxlarge-v2	1e-5	1e-2	4	0.1	16

Table 4.1: Configurations of initial model hyperparameters

4.2.4 Initial Experimentation

After preparing all the functions necessary for model training, the last step necessary before starting to train models was to decide the initial set of hyperparameters from which to begin our experimentation. The selection of the initial hyperparameters was mainly inspired by the hyperparameters used by the best performing ALBERT (Zhang et al., 2020) and BERT (Cusmuliuc et al., 2020) models in SemEval-2020 Task 4 Subtask A. All of our models were trained using the Adam optimizer (Kingma and Ba, 2014), specifically the AdamW implementation by Loshchilov and Hutter (2018) which includes a correct implementation of the weight decay regularisation as opposed to most other implementations of Adam according to its authors. We use a batch size of 16 for all of our models, even though a higher batch size is sometimes used in the aforementioned literature such as Cusmuliuc et al. (2020), as higher batch sizes often resulted in memory issues due to hardware limitations.

The aforementioned literature used as reference for hyperparameter selection did not specify whether dropout or weight decay was used, therefore we used the Huggingface ALBERT for sentence pair classification fine-tuning notebook as mentioned in Section 4.1.1 as a reference for these hyperparameters.

The hyperparameter configurations shown were initially used to train two separate models with each configuration, one for sentence pair classification and one for individual sentence classification. The first models were trained with the encoder layers frozen, though this conceivably resulted in mediocre results for all models, likely due to the difference between the tasks the encoders were trained on and our downstream task. After training again with the encoder layers unfrozen, the ALBERT models showed a massive improvement in performance, achieving a validation accuracy above 90% for the sentence pair model and above 80% for the individual sentence model. However, the BERT models failed to converge and maintained an accuracy consistently around 50%.

During this initial training experiment, it was noted that the ALBERT models' training loss was consistently decreasing throughout training even after the validation loss reached its apparent minimum and began increasing again, indicating a likely situation of overfitting. To counteract this, the models were trained again using an increased dropout rate of 0.2, which resulted in improved validation set performance for both the individual and sentence pair models. This experiment was then repeated with the dropout rate being increased to 0.3, however this showed negative results in the validation set performance of both models. Based on these experiments, the dropout rate was changed to 0.2 for the rest of the models trained.

After making this change to the dropout rate, the ALBERT models were trained again, this time with the addition of a learning scheduler (cosine with hard restarts and warmup) as used by Zhang et al. (2020) in their baseline ALBERT model for the sentence pair nonsense classification task which achieved a test accuracy of 96%. However, in our experiments, using this learning rate scheduler resulted in a significant decrease in performance for both individual and sentence pair models.

Finally, the ALBERT models were trained again with a higher weight decay of 1e-1, in order to examine whether this would help with the potential overfitting observed earlier, similar to the effect of increasing the dropout rate to 0.2. This resulted in a lower validation loss for the individual sentence model, albeit paired with a lower validation accuracy, and a decrease in validation performance across the board for the sentence pair model.

It should be noted that whilst early stopping was not being used at this point and the models were being trained for the full number of epochs shown in Table 4.1, a copy of the state dictionary of the model was saved and overwritten after every epoch, which resulted in an improvement in validation loss. When saving these models, the state of the model after the epoch with the lowest validation loss was saved as opposed to the state of the model after the final epoch.

Several attempts were made to achieve promising results using a BERT-based model, including experiments such as eliminating weight decay and dropout entirely and using gradient accumulation to mimic the batch size of 32 used by Cusmuliuc et al. (2020), however these models consistently failed to diverge. Whilst their model was the best performing BERT-based model in SemEval-2020 Task 4 Subtask A, their system description paper did not go into great detail about the model configuration other than the learning rate, number of epochs, batch size and optimizer used.

4.2.5 Further Hyperparameter Tuning

In a final attempt to improve the performance of our ALBERT-based models as well as achieve meaningful results from the BERT-based models, a hyperparameter search was conducted. This hyperparameter search involved combinations of the learning rate values [5e-5, 4e-5, 3e-5, 2e-5, 1e-5] and the weight decay values [1e-2, 1e-1]. The range of learning rate values was selected based on the values used by the best performing systems from the competition mentioned in the previous section as well as values used by Lan et al. (2019) and Devlin et al. (2018) in the fine-tuning of ALBERT and BERT on benchmark datasets. Meanwhile, the two weight decay values were chosen based on the weight decay experimentation described in the previous section, where a weight decay of 1e-2 achieved overall better results, however a weight decay of 1e-1 showed an improvement in validation loss in the ALBERT-based individual sentence model. Other than these ranges of values for the learning rate and weight decay, the batch size, maximum epochs and dropout rate were set at 16, 5 and 0.2 respectively for all of the models trained in this experiment.

Whilst this hyperparameter search space is far from an exhaustive search of all possibly optimal hyperparameter configurations, it was kept limited due to the limited timeframe and hardware availability for this project. Depending on the GPU allocated to the particular runtime, training our ALBERT-based models on Colab can take anywhere between 14-25 minutes for a single training epoch, whilst the BERT-based models tend to take between 3-6 minutes per epoch. Moreover, the GPU of the personal computer mentioned in Section 4.1.1 lacks the memory required to train the models effectively. Therefore, even simply training the 40 different models (2 tasks $\times$ 2 encoders $\times$ 2 weight decays $\times$ 5 learning rates) involved in this limited hyperparameter search required several days worth of training since Colab requires interactive use in order to avoid a runtime disconnection and thus cannot be used for overnight training.

In order to help alleviate the training time requirement without major sacrifices in model performance, a custom form of early stopping was implemented based on observations made in the initial training experiments. First, training would cease for a particular model after only 1 epoch without improvements in validation performance from the previous epoch. This was done as it was consistently noted in the initial model training that once a model's validation performance experiences its first decrease between epochs, it ceases to make any further improvements and instead continues to overfit to the training set. Moreover, the models are only to stop training if no improvement is noted in either validation loss or accuracy, as opposed to solely validation loss. This is due to the fact that in the initial experimentation, models were sometimes noted to undergo a sizeable increase in validation accuracy despite a slight increase in validation loss. The results of this final experiment and the performance of the models produced will be discussed in detail in the next chapter.

Chapter 5

Evaluation

In this chapter, the results from the training of the models described in the previous chapter will be reported and discussed in both a quantitative and qualitative manner in order to address the research questions put forward in Section 1.2.

5.1 Training Results

The parallel coordinates plot shown in Figure 5.1 shows the results of the hyperparameter tuning experiment described in Section 4.2.5, with the lines coloured on a gradient determined by the validation accuracy of the model, going from red for the models with a high accuracy to blue for the models with a low accuracy. The results of the hyperparameter search as shown seem to indicate that the fine-tuning process for this downstream task is significantly sensitive to the hyperparameter configuration, with a significant portion of the models failing to converge entirely and falling within a validation accuracy range of 45-55%, similar to what was observed in the initial experimentation with BERT-based models as described in Section 4.2.4.

It was noted that some of the models trained for the individual sentence classification task which failed to converge were repeatedly predicting the same label, thus being stuck at a validation accuracy of exactly 50%. This phenomenon is often caused by a class imbalance in the training data or an overly complex model architecture, however due to the fact that our training data for the individual sentence nonsense classification task has a perfectly split class balance as described in Section 3.1.1, this factor is highly unlikely to be the cause of these results. Moreover,

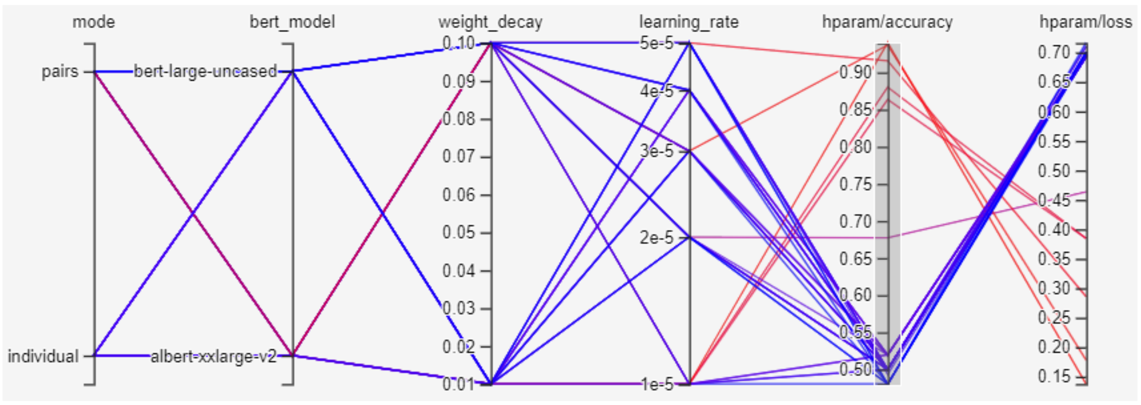


Figure 5.1: Parallel coordinates plot showing validation metrics of models trained during hyperparameter tuning

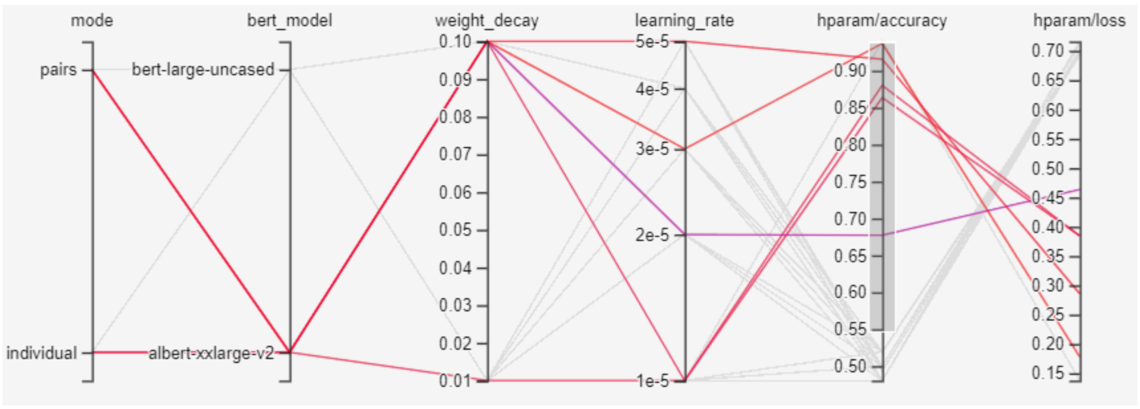


Figure 5.2: Parallel coordinates plot from hyperparameter tuning with models under 55% validation accuracy greyed out

the same model architecture also obtains favourable results with certain hyperparameter configurations. Based on these observations, the cause of this model divergence is speculated to be related to the models' sensitivity to hyperparameters.

As shown in Figure 5.2, when filtering out models with a validation accuracy lower than 55%, all BERT-based models from the hyperparameter tuning are greyed out. This indicates that the hyperparameter configurations explored were not sufficient to alleviate the divergence issues encountered in our BERT-based model training as described in Section 4.2.4. When referring to the work done by Bai and Zhou (2020), which describes their approach in creating the second best BERT-based model in SemEval-2020 Task 4 Subtask A, it was noted that they use cross-validation with a significant number of different random seeds and an otherwise consistent hyperparameter configuration to obtain their final model. On the other hand, all trained model results reported in this project make use of the same random seed as described in Section 4.1.2.

Based on the aforementioned observations, it is highly likely that a BERT-based model could be successfully trained to convergence for our two classification tasks since this was done by teams in the original SemEval-2020 Task 4 Subtask A without using additional external resources such as knowledge graph triples or additional pretraining data (Cusmuliuc et al., 2020; Bai and Zhou, 2020). However, it was noted that the validation performance of our ALBERT-based models on the sentence pair classification task surpasses that of these BERT-based models, whilst also achieving significantly satisfactory validation results on the individual sentence classification task. Therefore, when conducting a quantitative analysis of the trained models on the test set as well as a qualitative analysis of the outputs on individual sentences, we will opt to focus on the ALBERT-based models which achieved a validation accuracy of at least 55% so as to filter out the models which failed to converge entirely.

5.2 Test Set Results

In order to conduct a thorough quantitative analysis of our systems with a validation accuracy above 55%, the following metrics will be used:

Accuracy The proportion of correctly predicted samples in the test set ($accuracy = \frac{tp+tn}{tp+tn+fp+fn}$).

Precision The proportion of all samples in the test set predicted as positive by the model that are actually positive ($precision = \frac{tp}{tp+fp}$).

Recall The proportion of all positive samples in the test set correctly identified as positive by the model ($recall = \frac{tp}{tp+fn}$).

F1-Score The harmonic mean of the precision and recall scores ($F_1 = 2 \times \frac{precision \times recall}{precision + recall}$).

Area Under Receiver Operating Characteristic (ROC) Curve The ROC curve is a probability curve which maps the Recall (also known as True Positive Rate or TPR) against the False Positive Rate (FPR), which is the proportion of all negative samples in the test rate which were incorrectly identified, at different prediction thresholds. The prediction threshold is the minimum prediction probability which must be given by the model for a sample to be classified as positive. The area under this curve, often referred to as the AUC or AUC-ROC, is hence commonly used as a measure of a classification model's capability to confidently differentiate between classes (BhandariI, 2020).

It should be noted that in the metric descriptions shown above, 'tp' and 'fp' represent true positives and false positives, meaning the sum of samples correctly and incorrectly predicted with the positive label (1) respectively. Similarly, 'tn' and 'fn' represent true negatives and false negatives, which are the sum of samples correctly and incorrectly predicted with the negative label (0).

Whilst all these metrics will be used to evaluate the individual sentence classification models, only the accuracy will be used when evaluating the sentence pair classification models and comparing these to the models from the SemEval-2020 Task 4 Subtask A participants. This is due to the fact that this model is always predicting which of two sentences is the nonsensical one rather than predicting whether a particular sentence belongs to the sensical or nonsensical class, which means that there is no significant real-world distinction between the positive and negative labels in the case of the sentence pair classification task. As a matter of fact, the models presented by the participating teams in the original challenge were evaluated solely on accuracy (Wang et al., 2020a).

5.2.1 Sentence Pair Classification

As can be seen from Table 5.1, the best performing model on the sentence pair classification task was the model with the final hyperparameter configuration from the original experiments done before the hyperparameter tuning as described in Section 4.2.4. However, it was noted that any increase in the learning rate of this configuration, whilst keeping the weight decay constant, results

Learning Rate	Weight Decay	Accuracy (%)
1e-5	1e-1	66.8
4e-5	1e-1	81.6
5e-5	1e-1	93.1
3e-5	1e-1	93.4
2e-5	1e-1	95.0
1e-5	**1e-2**	**95.6**

Table 5.1: Accuracy of ALBERT-based models on the sentence pair classification test set

in the model failing to converge. On the other hand, all five learning rate values explored resulted in a model with a validation accuracy above 55% when paired with an increased weight decay.

The test accuracy of 95.6% achieved by our best performing model on this task is only an absolute difference of 0.4% away from the 96% achieved by the best performing ALBERT-based model (Zhang et al., 2020) on the original subtask which did not use additional data apart from the datasets provided.

Our model's accuracy also trails just behind the second best participating ALBERT-based model (Jon et al., 2020) which achieved an accuracy of 95.8%, however with a test accuracy of 95.6% our model would still place comfortably at 8th place out of 40 in the original challenge leaderboards. Moreover, out of the seven teams with a better test accuracy, only three teams achieve an accuracy above 95.6% without using additional sources of data like knowledge graph triples or extra pretraining corpora.

Whilst the performance of this model could likely be improved further even by simply experimenting with different random seeds when training, this was not deemed necessary within the scope of this project. This is because the core focus of this research is to investigate the potency of such models in identifying individual sentences as nonsensical or sensical, as will be described in the following section.

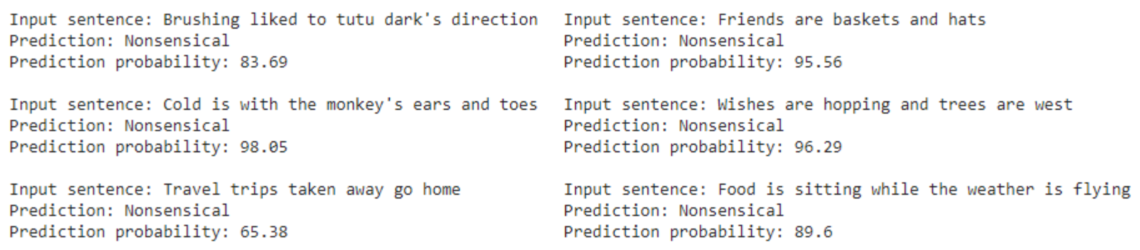
5.2.2 Individual Sentence Classification

As can be seen in Table 5.2, the best accuracy achieved by our trained models on the individual sentence nonsense classification task was 89.3%. It is also worth noting that the second best model, with an accuracy of 88.7%, makes use of the same hyperparameter configuration. The difference between these models is that the former, marked with an asterisk (*) in the table, was trained before the hyperparameter tuning stage and hence without the addition of validation accuracy improvement as a consideration for which version of the model to save. Whilst this model configuration resulted in a slight increase in validation loss between the second and third training epochs, this increase in loss also came with an increase in validation accuracy, which resulted in the latter model. However, this increase in validation accuracy was not reciprocated in the test set performance of the two models, as can be seen from the aforementioned table.

Both of these described models achieved relatively promising results, with the former of the two reporting a higher precision but lower recall. This indicates that the model trained for one epoch longer correctly classified a larger proportion of the nonsensical sentences, however at the expense of a larger proportion of false positives. Despite this trade-off in precision and recall between the two models, we consider the model marked with an asterisk, i.e. the model trained for one less epoch, to be the overall best performing model on this task due to the higher accuracy, F1-score and AUC-ROC, which serve as a more reliable indication of overall model performance.

Learning Rate	Weight Decay	Accuracy (%)	Precision (%)	Recall (%)	F1-Score	Area Under ROC Curve
1e-5	1e-1	85.3	89.4	80.0	0.844	0.930
1e-5	1e-2	88.7	89.5	**87.7**	0.886	0.951
1e-5*	**1e-2**	**89.3**	**92.3**	85.6	**0.888**	**0.957**

Table 5.2: Evaluation metrics of ALBERT-based models on the individual sentence classification test set



```
Input sentence: Brushing liked to tutu dark's direction
Prediction: Nonsensical
Prediction probability: 83.69

Input sentence: Cold is with the monkey's ears and toes
Prediction: Nonsensical
Prediction probability: 98.05

Input sentence: Travel trips taken away go home
Prediction: Nonsensical
Prediction probability: 65.38

Input sentence: Friends are baskets and hats
Prediction: Nonsensical
Prediction probability: 95.56

Input sentence: Wishes are hopping and trees are west
Prediction: Nonsensical
Prediction probability: 96.29

Input sentence: Food is sitting while the weather is flying
Prediction: Nonsensical
Prediction probability: 89.6
```

Figure 5.3: Predictions of the final model on gibberish sentences

This is based on the intuition that a false positive in our case is equally as costly as a false negative, whereas in a context where a false negative was significantly more costly than a false positive, such as in a medical diagnosis tool, the higher recall model would likely be preferable.

On a final note, it appears that the models for the individual sentence classification task were more sensitive to changes in learning rate, as all learning rates higher than 1e-5 resulted in these models failing to converge, even with an increased weight decay rate. Furthermore the ideal hyperparameter configuration from those explored was found to be the same as the ideal hyperparameter configuration for the sentence pair task. Despite the added difficulty of this task due to the model being given an individual sentence which could be either sensical or nonsensical, as opposed to a pair of sentences which always has one sensical and one nonsensical statement, our best model still achieved a test accuracy of 89.3% on this task, which is only an absolute difference of 6.3% from the sentence pair task. In the next section we will analyse this particular model in more detail from a qualitative perspective.

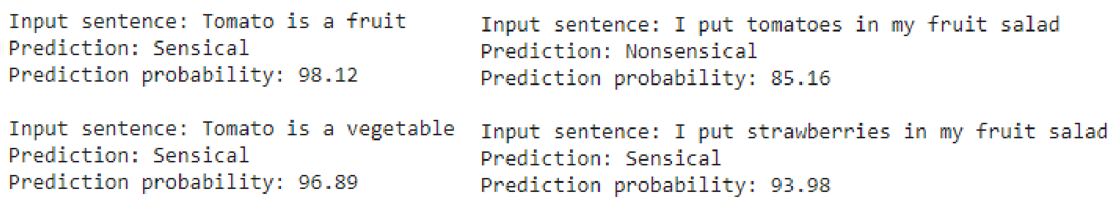
5.3 Qualitative Analysis

In order to achieve a better understanding of the trained model's performance, we manually observe the output of the model for a number of given sentences. Initially, we give the model a number of gibberish[1] sentences, which are made up of real words but have no real or significant meaning behind them. As can be seen in Figure 5.3, the model successfully identifies all six of such sentences shown to it as nonsensical.

Following this initial test, the model was given a less trivial scenario to understand its capability for commonsense reasoning. It was first given the statements "Tomato is a fruit" and "Tomato is a vegetable", which were both classified as sensical. This output is satisfactory as tomatoes are classified as fruits botanically however they are deemed to be vegetables in a culinary context. Next, the model is given the sentences "I put tomatoes in my fruit salad" and "I put strawberries in my fruit salad". Whilst tomatoes are scientifically classified as fruits which the model was shown to understand, most would agree that common sense dictates they are not typically included in a fruit salad. As shown in Figure 5.4, the model has shown to be capable of making this distinction, as it successfully classified the statement about tomatoes in fruit salad as nonsensical whilst classifying the sentence about strawberries in fruit salad as sensical.

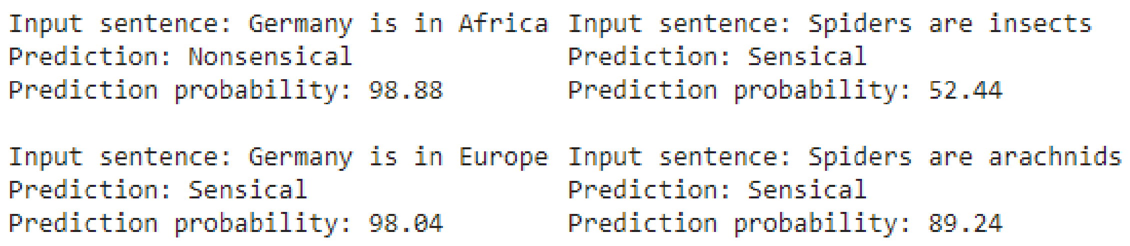
Next, we tested the model with some correct and incorrect facts to gauge the model's reaction to objectively correct and incorrect statements. As can be seen in Figure 5.5, the model successfully identified the statements "Germany is in Africa" and "Germany is in Europe" as nonsensical

[1]https://examples.yourdictionary.com/examples-of-gibberish.html



```
Input sentence: Tomato is a fruit
Prediction: Sensical
Prediction probability: 98.12

Input sentence: Tomato is a vegetable
Prediction: Sensical
Prediction probability: 96.89

Input sentence: I put tomatoes in my fruit salad
Prediction: Nonsensical
Prediction probability: 85.16

Input sentence: I put strawberries in my fruit salad
Prediction: Sensical
Prediction probability: 93.98
```

Figure 5.4: Predictions of the final model in a nontrivial scenario



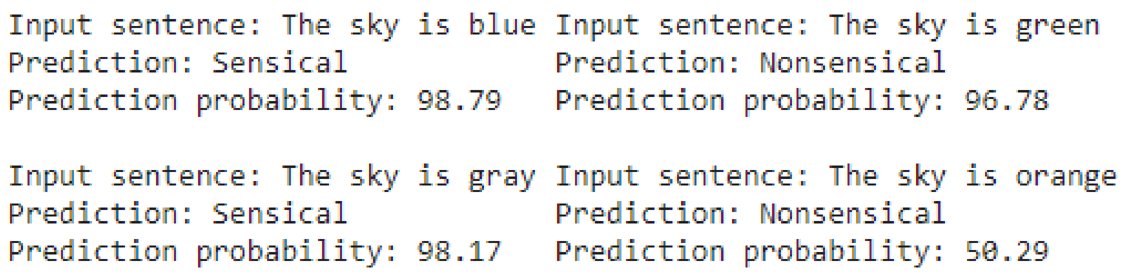
```
Input sentence: Germany is in Africa
Prediction: Nonsensical
Prediction probability: 98.88

Input sentence: Germany is in Europe
Prediction: Sensical
Prediction probability: 98.04

Input sentence: Spiders are insects
Prediction: Sensical
Prediction probability: 52.44

Input sentence: Spiders are arachnids
Prediction: Sensical
Prediction probability: 89.24
```

Figure 5.5: Predictions of the final model on correct and incorrect objective statements

and sensical respectively with a high degree of confidence. On the other hand, when prompted with the sentences “Spiders are insects” and “Spiders are arachnids”, both these sentences were classified as sensical, when spiders are distinctly classified as arachnids and not insects. However, the prediction probability for the statement about spiders being insects is very close to 50%, indicating that the model was very uncertain about this prediction. Moreover, spiders are commonly misclassified as insects even by humans, to which regard an argument can be made about whether this statement can truly be classified as common sense. The model’s prediction of this statement can highly probably be attributed to this common human misconception, since it is likely reflected in the corpora which the ALBERT model was pretrained on.

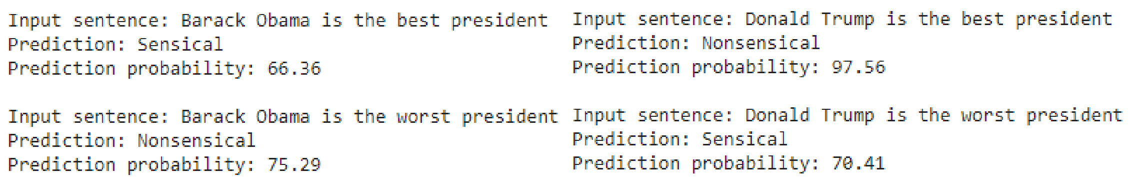
In Figure 5.6, we can see the performance of the model on four statements each beginning with “The sky is” and followed by a colour. The statements describing the sky as blue and gray are predicted as sensical with a very high probability, which should be expected as these are the most commonly used colours to describe the sky. Conversely, the statement describing the sky as green is predicted as nonsensical with a similarly high probability. The final sentence, describing the sky as orange, is also predicted as nonsensical, however with a probability of 50.29%. Whilst the sky cannot be described as orange most of the time, it can often appear and be described as orange during a sunset, therefore this sentence should not be deemed as nonsensical based on this intuition. However, as mentioned previously the probability for this prediction is immensely close to the 50% threshold, indicating that this prediction has a significant likelihood of changing with even minor alterations to the model.

Finally, the model was evaluated on a set of subjective statements based on opinions with varying popularity. Specifically, the model was given statements describing two recent U.S. presidents, in the format “Barack Obama/Donald Trump is the best/worst president”. Whilst the statements describing Barack Obama as the best and worst president were predicted as sensical and nonsensical respectively, the opposite outcome was given for the sentences describing Donald Trump. Most of these predictions were given with a probability of around 70%, with the exception of the statement describing Donald Trump as the best president, which was deemed nonsensical



```
Input sentence: The sky is blue Input sentence: The sky is green
Prediction: Sensical            Prediction: Nonsensical
Prediction probability: 98.79   Prediction probability: 96.78

Input sentence: The sky is gray Input sentence: The sky is orange
Prediction: Sensical            Prediction: Nonsensical
Prediction probability: 98.17   Prediction probability: 50.29
```

Figure 5.6: Predictions of the final model on different sentences describing the colour of the sky



```
Input sentence: Barack Obama is the best president  Input sentence: Donald Trump is the best president
Prediction: Sensical                                Prediction: Nonsensical
Prediction probability: 66.36                       Prediction probability: 97.56

Input sentence: Barack Obama is the worst president Input sentence: Donald Trump is the worst president
Prediction: Nonsensical                             Prediction: Sensical
Prediction probability: 75.29                       Prediction probability: 70.41
```

Figure 5.7: Predictions of the model on controversial opinion-based statements

with a very high probability of 97.56%. Whilst it is fair to say that this sentiment is not a very common one, such statements are still subjective in nature despite the varying degrees of popularity of different opinions. Therefore, such a system should not flag such opinionated statements as sensical or nonsensical based on the observed popularity of the opinions behind them from the training corpora. This indicates that such a model could potentially benefit from the use of a subjectivity analysis component, trained on something like the SUBJ (Pang and Lee, 2004) dataset, to identify statements which are highly likely to be subjective and take this into consideration before predicting them as nonsensical.

Overall, despite the model's prediction on subjective statements appearing to be significantly impacted by the popularity of the opinions behind the statements, it appeared to perform very well on all other tests conducted. Whilst there were some other predictions made by the model which were arguable, the prediction probability was very close to 50% in all of these situations. This indicates that whilst the overall performance is good, there is a significant potential for improvement in such a model as changes like using additional pretraining data for the encoder could be enough to alter these uncertain predictions.

Chapter 6

Conclusion

This thesis has described the research and implementation efforts behind a project which involved taking a relatively recent research challenge and expanding its scope further. Specifically, the task involved imbuing a machine learning model with enough NLU capability to identify sentences which defy common sense.

A combination of deep learning and transfer learning was used to tackle this challenge, with large-scale pretrained language models used as a base for classification models to identify one sentence from a pair as nonsensical as well as to identify whether an individual sentence is nonsensical. Whilst the first of these tasks had previously been attempted by numerous teams of researchers, to the best of our knowledge there does not seem to be any significant quantity of research to date tackling the latter task. Therefore, the data used for the original task (Wang et al., 2020a) was repurposed to allow for the training and evaluation of a model for the latter task.

The main research objective of this thesis was to determine whether it is possible to create a system which can identify individual sentences as sensical or nonsensical, and if so, to what degree of effectiveness. Our best performing model for this task, fine-tuned on the individual sentences from the original sentence pair dataset, achieved an accuracy of 89.3% when evaluated on the sentences from the test set of the aforementioned dataset, as well as favourable results in a number of other metrics including precision, recall, F1-score and AUC-ROC.

Moreover, a qualitative analysis was carried out by investigating the model's predictions on a number of different types of statements such as gibberish sentences, incorrect facts and subjective, opinion-based statements. The model classified most of these sentences as expected, with most of the incorrect decisions having a probability very close to 50% indicating the model was not confident in the predictions which were deemed to be incorrect. The exception to this was with the opinion-based statements, where the results appear to indicate that the model was prone to predicting unpopular but valid opinions as nonsensical. In this regard, it was proposed that potential future systems to tackle similar tasks could benefit from the inclusion of a subjectivity analysis component.

6.1 Limitations

The main limitations encountered throughout this project were hardware and time. Although the models developed were based on pretrained models, the fine-tuning process for a single model could still take up to almost three hours depending on the model configuration. Moreover, due to hardware limitations these models were trained on Google Colab, which requires interactive use, hence making it impossible to train models overnight. These factors combined with the time required for other tasks such as research, evaluation and writing, and the limited time frame allocated for the project meant that the amount of model optimization which could be done was limited.

6.2 Future Work

A number of different paths can be taken to build on the work carried out in this project. Perhaps the most obvious option would be to further investigate whether the metrics and issues shown in the quantitative and qualitative evaluation of the model could be improved and fixed respectively. This could possibly be done in a number of different ways, such as a more exhaustive hyperparameter search or using additional pretraining data for the model encoder. Other encoders such as RoBERTa and XLNet could also be explored, as well as a wide variety of other techniques as shown by the original participants of SemEval-2020 Task 4 (Wang et al., 2020a).

Another potential approach to building on this research could be to investigate whether subjectivity analysis (Pang and Lee, 2004) can be used to help the model differentiate between nonsensical statements and a valid opinion which is simply unpopular. This proposal is mainly based on the observations made in the qualitative analysis of the final model when it was tasked with giving a prediction for such statements.

Expanding the current system by creating a separate NLG model to generate text explaining why a given sentence is nonsensical could also prove to be an interesting path for furthering this research. This is the same challenge which was proposed Subtask C of SemEval-2020 Task 4, however, based on the results from the submissions for this subtask it was reported that there is still “a long way to go to generate human acceptable reasons” (Wang et al., 2020a).

Finally, if a model such as the one developed were refined further, it could potentially be used to enhance the functionality of writing assistant software such as Grammarly or Microsoft’s spell check to identify potential mistakes related to the logic and meaning behind text as opposed to solely grammatical or spelling errors.

Bibliography

Alyafeai, Z., AlShaibani, M. S., and Ahmad, I. (2020). A survey on transfer learning in natural language processing. *arXiv preprint arXiv:2007.04239*.

Bai, Y. and Zhou, X. (2020). Deepyang at semeval-2020 task 4: Using the hidden layer state of bert model for differentiating common sense. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 516–520.

Bengio, Y. (2009). Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127.

BhandariI, A. (2020). Auc-roc curve in machine learning clearly explained.

Chandra, A. L. (2018). Mcculloch-pitts neuron - mankind's first mathematical model of a biological neuron. *Medium*.

Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258.

Conneau, A., Khandelwal, K., Goyal, N., Chaudhary, V., Wenzek, G., Guzmán, F., Grave, E., Ott, M., Zettlemoyer, L., and Stoyanov, V. (2019). Unsupervised cross-lingual representation learning at scale. *arXiv preprint arXiv:1911.02116*.

Cusmuliuc, C.-G., Coca, L.-G., and Iftene, A. (2020). Uaics at semeval-2020 task 4: Using a bidirectional transformer for task a. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 609–613.

Dash, S. R., Routray, S., Varshney, P., and Modi, A. (2020). Cs-net at semeval-2020 task 4: Siamese bert for comve. *arXiv preprint arXiv:2007.10830*.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Doxolodeo, K. and Mahendra, R. (2020). Ui at semeval-2020 task 4: Commonsense validation and explanation by exploiting contradiction. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 614–619.

Fadel, A., Al-Ayyoub, M., and Cambria, E. (2020). Justers at semeval-2020 task 4: Evaluating transformer models against commonsense validation and explanation. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 535–542.

Fyfe, C. (2005). Artificial neural networks. In *Do smart adaptive systems exist?*, pages 57–79. Springer.

Gavrilova, Y. (2020). A guide to deep learning and neural networks. *Serokell Software Development Company*.

Hawkins, D. M. (2004). The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In

Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Hoque, M. (2015). Components of language.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An introduction to statistical learning*, volume 112. Springer.

Jimoh, H. (2020). How transfer learning works. *Medium*.

Jon, J., Fajčík, M., Dočekal, M., and Smrž, P. (2020). But-fit at semeval-2020 task 4: Multilingual commonsense. *arXiv preprint arXiv:2008.07259*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lai, G., Xie, Q., Liu, H., Yang, Y., and Hovy, E. (2017). Race: Large-scale reading comprehension dataset from examinations. *arXiv preprint arXiv:1704.04683*.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. (2019). Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*.

Lan, Z. and Soricut, R. (2019). Albert: A lite bert for self-supervised learning of language representations. *Google AI Blog*.

Le, H., Vial, L., Frej, J., Segonne, V., Coavoux, M., Lecouteux, B., Allauzen, A., Crabbé, B., Besacier, L., and Schwab, D. (2019). Flaubert: Unsupervised language model pre-training for french. *arXiv preprint arXiv:1912.05372*.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. (2019). Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*.

Li, J., Wang, B., and Ding, H. (2020). Lijunyi at semeval-2020 task 4: An albert model based maximum ensemble with different training sizes and depths for commonsense validation and explanation. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 556–561.

Li, M., Zhang, T., Chen, Y., and Smola, A. J. (2014). Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670.

Lippi, M., Pałka, P., Contissa, G., Lagioia, F., Micklitz, H.-W., Sartor, G., and Torroni, P. (2019). Claudette: an automated detector of potentially unfair clauses in online terms of service. *Artificial Intelligence and Law*, 27(2):117–139.

Liu, P. (2020). Qiaoning at semeval-2020 task 4: Commonsense validation and explanation system based on ensemble of language model. *arXiv preprint arXiv:2009.02645*.

Liu, S., Guo, Y., Li, B., and Ren, F. (2020a). Lmve at semeval-2020 task 4: Commonsense validation and explanation using pretraining language model. *arXiv preprint arXiv:2007.02540*.

Liu, W., Zhou, P., Zhao, Z., Wang, Z., Ju, Q., Deng, H., and Wang, P. (2020b). K-bert: Enabling language representation with knowledge graph. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 2901–2908.

Liu, Y., Gu, J., Goyal, N., Li, X., Edunov, S., Ghazvininejad, M., Lewis, M., and Zettlemoyer, L. (2020c). Multilingual denoising pre-training for neural machine translation. *Transactions of the Association for Computational Linguistics*, 8:726–742.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Loshchilov, I. and Hutter, F. (2017). Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Loshchilov, I. and Hutter, F. (2018). Fixing weight decay regularization in adam.

Maas, A., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pages 142–150.

Markchom, T., Dhruva, B., Pravin, C., and Liang, H. (2020). Uor at semeval-2020 task 4: pre-trained sentence transformer models for commonsense validation and explanation.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

McKinney, W. et al. (2010). Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX.

Mendbayar, K. and Aono, M. (2020). Kde senseforce at semeval-2020 task 4: Exploiting bert for commonsense validation and explanation. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 551–555.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al. (2017). Mixed precision training. *arXiv preprint arXiv:1710.03740*.

Mohammed, R. and Abdullah, M. (2020). Teamjust at semeval-2020 task 4: Commonsense validation and explanation using ensembling techniques. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 594–600.

Ou, X. and Li, H. (2020). Ynu-oxz at semeval-2020 task 4: Commonsense validation using bert with bidirectional gru. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 626–632.

Pang, B. and Lee, L. (2004). A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. *arXiv preprint cs/0409058*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*.

Qiu, X., Sun, T., Xu, Y., Shao, Y., Dai, N., and Huang, X. (2020). Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, pages 1–26.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. (2016). Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.

Ramsundar, B. and Zadeh, R. B. (2018). *TensorFlow for deep learning: from linear regression to reinforcement learning*. " O'Reilly Media, Inc.".

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788.

Saeedi, S., Panahi, A., Saeedi, S., and Fong, A. C. (2020). Cs-nlp team at semeval-2020 task 4: Evaluation of state-of-the-art nlp deep learning architectures on commonsense reasoning task. *arXiv preprint arXiv:2006.01205*.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2019). Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.

Sap, M., Shwartz, V., Bosselut, A., Choi, Y., and Roth, D. (2020). Commonsense reasoning for natural language processing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*, pages 27–33, Online. Association for Computational Linguistics.

Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642.

Speer, R., Chin, J., and Havasi, C. (2017). Conceptnet 5.5: An open multilingual graph of general knowledge. In *Thirty-first AAAI conference on artificial intelligence*.

Srivastava, V., Sahoo, S. K., Kim, Y. H., Rr, R., Raj, M., and Jaiswal, A. (2020). Team solomon at semeval-2020 task 4: Be reasonable: Exploiting large-scale language models for commonsense reasoning. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 585–593.

Tan, C., Sun, F., Kong, T., Zhang, W., Yang, C., and Liu, C. (2018). A survey on deep transfer learning. In *International conference on artificial neural networks*, pages 270–279. Springer.

Teo, D. (2020). Tr at semeval-2020 task 4: Exploring the limits of language-model-based common sense validation. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 601–608.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Versloot, C. (2021). Differences between autoregressive, autoencoding and sequence-to-sequence models in machine learning. *MachineCurve*.

Wan, J. and Huang, X. (2020). Kalm at semeval-2020 task 4: Knowledge-aware language models for comprehension and generation. *arXiv preprint arXiv:2005.11768*.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. (2018). Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Wang, C., Liang, S., Jin, Y., Wang, Y., Zhu, X., and Zhang, Y. (2020a). Semeval-2020 task 4: Commonsense validation and explanation.

Wang, C., Liang, S., Zhang, Y., Li, X., and Gao, T. (2020b). Does it make sense? and why? a pilot study for sense making and explanation.

Wang, H., Tang, X., Lai, S., Leung, K. S., Zhu, J., Fung, G. P. C., and Wong, K.-F. (2020c). Cuhk at semeval-2020 task 4: Commonsense explanation, reasoning and prediction with multi-task learning. *arXiv preprint arXiv:2006.09161*.

Weiss, K., Khoshgoftaar, T. M., and Wang, D. (2016). A survey of transfer learning. *Journal of Big data*, 3(1):1–40.

West, J., Ventura, D., and Warnick, S. (2007). Spring research presentation: A theoretical foundation for inductive transfer. *Brigham Young University, College of Physical and Mathematical Sciences*, 1(08).

Woodruff, A. (2019). What is a neuron? *Queensland Brain Institute*.

Wu, M. (2018). The bidirectional language model. *Medium*.

Xing, L., Xie, Y., Hu, Y., and Peng, W. (2020). Iie-nlp-nut at semeval-2020 task 4: Guiding plm with prompt template reconstruction strategy for comve. *arXiv preprint arXiv:2007.00924*.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., and Le, Q. V. (2019). Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32.

Zellers, R., Bisk, Y., Schwartz, R., and Choi, Y. (2018). Swag: A large-scale adversarial dataset for grounded commonsense inference. *arXiv preprint arXiv:1808.05326*.

Zhang, X., Zhao, J., and LeCun, Y. (2015). Character-level convolutional networks for text classification. *Advances in neural information processing systems*, 28:649–657.

Zhang, Y., Lin, J., Fan, Y., Jin, P., Liu, Y., and Liu, B. (2020). Cn-hit-it. nlp at semeval-2020 task 4: Enhanced language representation with multiple knowledge triples. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 494–500.

Zhao, Q., Tao, S., Zhou, J., Wang, L., Lin, X., and He, L. (2020). Ecnu-sensemaker at semeval-2020 task 4: Leveraging heterogeneous knowledge resources for commonsense validation and explanation. *arXiv preprint arXiv:2007.14200*.

Zheng, L., Guha, N., Anderson, B. R., Henderson, P., and Ho, D. E. (2021). When does pretraining help? assessing self-supervised learning for law and the casehold dataset of 53,000+ legal holdings. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Law*, pages 159–168.

Zou, J., Han, Y., and So, S.-S. (2008). Overview of artificial neural networks. *Artificial Neural Networks*, pages 14–22.

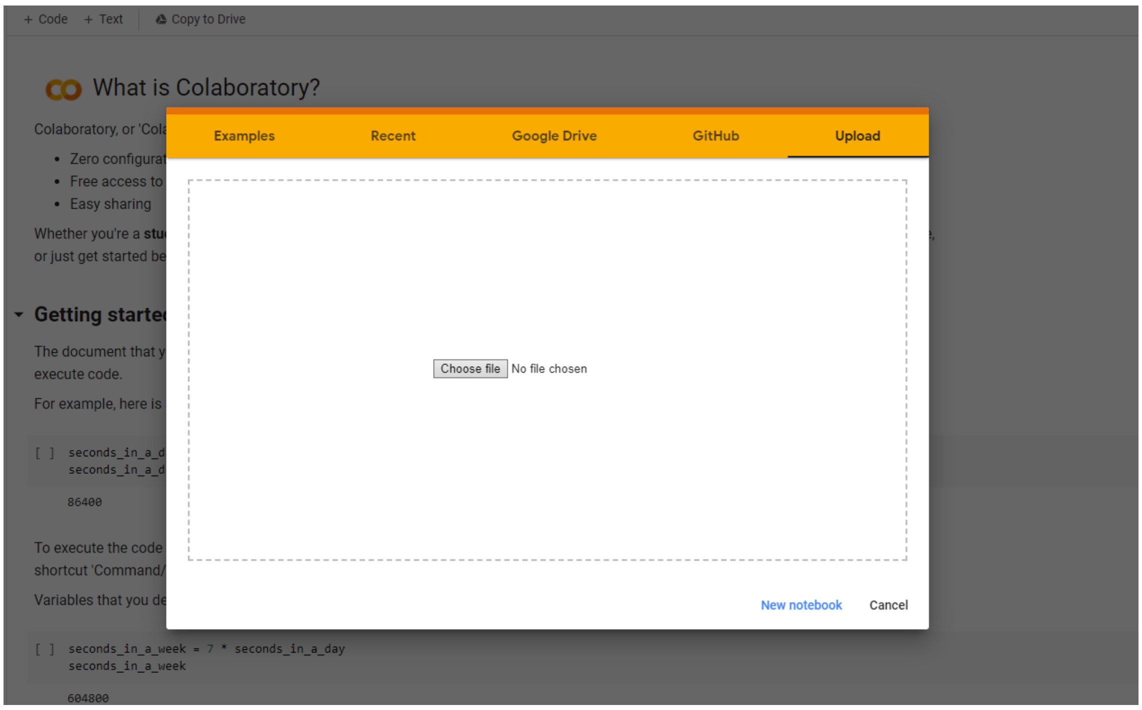
Appendix A

Code User Manual

This appendix will guide you through the process of using the provided notebooks and datasets to train and evaluate your own models.

1. Upload the given 'Data' folder to your Google Drive.

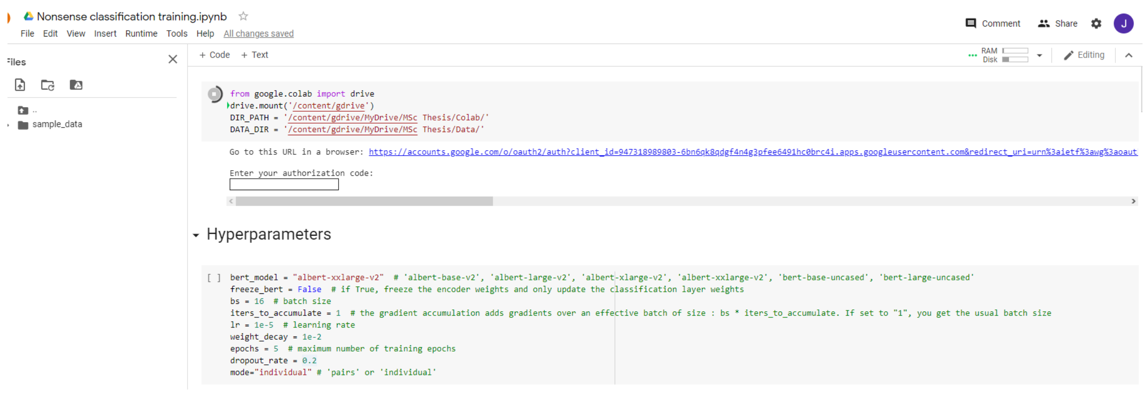
2. Open Google Colaboratory and upload the 'Nonsense classification training.ipynb' notebook.



```
[ ]  seconds_in_a_d
     seconds_in_a_d

     86400

[ ]  seconds_in_a_week = 7 * seconds_in_a_day
     seconds_in_a_week

     604800
```

3. **Recommended:** In the menu bar at the top of the page, click **Runtime**, followed by **Change runtime type** and set the hardware accelerator to GPU.

4. Run the first code block and follow the instructions given to authorise Colab to save and load files to and from your Google Drive.

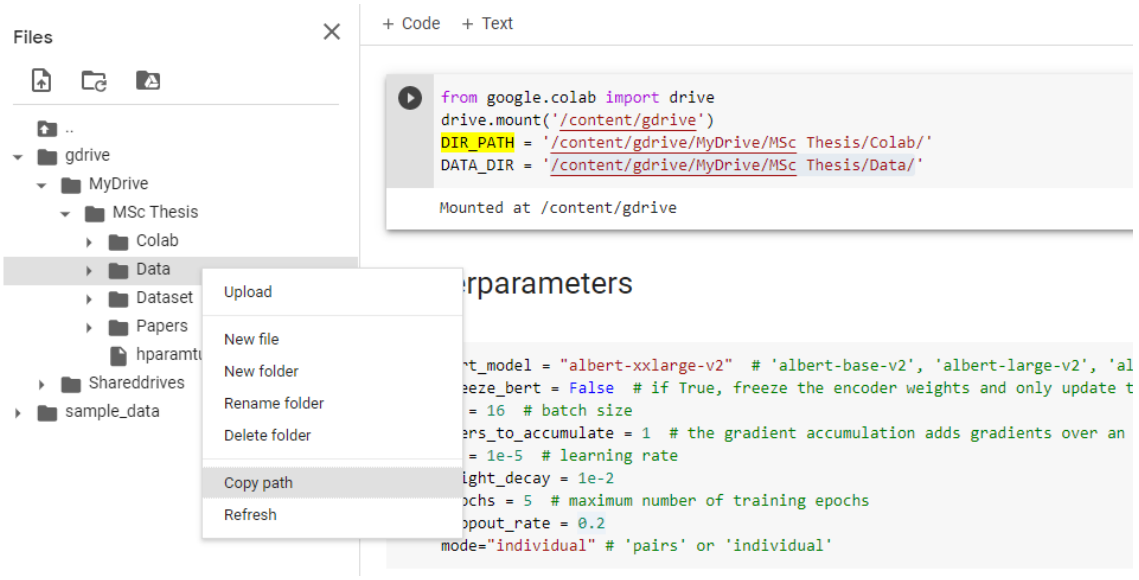

Nonsense classification training.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
Files
sample_data
+ Code + Text
RAM Disk
Editing
Hyperparameters

```
from google.colab import drive
drive.mount('/content/gdrive')
DIR_PATH = '/content/gdrive/MyDrive/MSc Thesis/Colab/'
DATA_DIR = '/content/gdrive/MyDrive/MSc Thesis/Data/'
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoaut

Enter your authorization code:
```

```
[ ] bert_model = "albert-xxlarge-v2"  # 'albert-base-v2', 'albert-large-v2', 'albert-xlarge-v2', 'albert-xxlarge-v2', 'bert-base-uncased', 'bert-large-uncased'
    freeze_bert = False  # if True, freeze the encoder weights and only update the classification layer weights
    bs = 16  # batch size
    iters_to_accumulate = 1  # the gradient accumulation adds gradients over an effective batch of size : bs * iters_to_accumulate. If set to "1", you get the usual batch size
    lr = 1e-5  # learning rate
    weight_decay = 1e-2
    epochs = 5  # maximum number of training epochs
    dropout_rate = 0.2
    mode="individual" # 'pairs' or 'individual'
```

5. Set the DIR_PATH variable to the directory in your google drive where you would like the trained model to be saved, and the DATA_DIR variable to the path to the Data folder in your Google Drive. This can be done by copying the path from the file directory on the left of the page. **Please note that both of these variables should end with a slash symbol (/).**

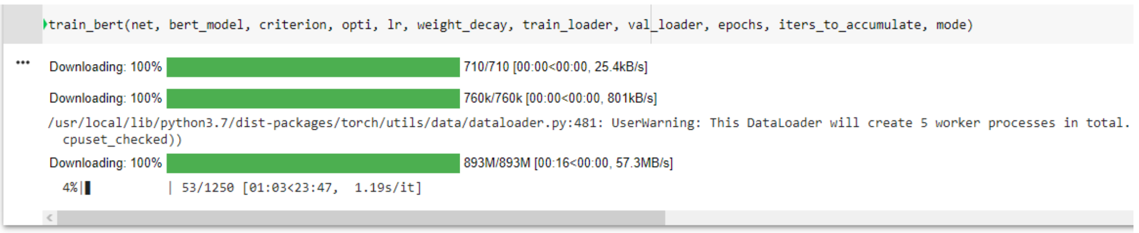


```
from google.colab import drive
drive.mount('/content/gdrive')
DIR_PATH = '/content/gdrive/MyDrive/MSc Thesis/Colab/'
DATA_DIR = '/content/gdrive/MyDrive/MSc Thesis/Data/'

Mounted at /content/gdrive
```

```
t_model = "albert-xxlarge-v2"  # 'albert-base-v2', 'albert-large-v2', 'al
eeze_bert = False  # if True, freeze the encoder weights and only update t
 = 16  # batch size
ers_to_accumulate = 1  # the gradient accumulation adds gradients over an
 = 1e-5  # learning rate
ight_decay = 1e-2
ochs = 5  # maximum number of training epochs
opout_rate = 0.2
mode="individual" # 'pairs' or 'individual'
```

6. In the following code block, set the hyperparameters to the desired values. After this is done, click the **Runtime** button in the menu bar at the top of the page and then click **Run all**.

7. Scroll down to the bottom of the notebook. The training progress will be displayed and the path to the saved model will be displayed when it is finished training.



```
train_bert(net, bert_model, criterion, opti, lr, weight_decay, train_loader, val_loader, epochs, iters_to_accumulate, mode)

Downloading: 100% 710/710 [00:00<00:00, 25.4kB/s]
Downloading: 100% 760k/760k [00:00<00:00, 801kB/s]
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarning: This DataLoader will create 5 worker processes in total.
  cpuset_checked))
Downloading: 100% 893M/893M [00:16<00:00, 57.3MB/s]
  4%|▌          | 53/1250 [01:03<23:47,  1.19s/it]
```

Following the steps shown above will provide a trained model for either the sentence pair classification task or the individual sentence classification task, depending on whether the 'mode' hyperparameter was set to ‘pairs’ or ‘individual’ respectively. The next steps can be followed to evaluate the trained model on the test set as well as on any individual sentences of your choice.

1. Open the ’Nonsense classification evaluation.ipynb’ notebook in Google Colab as done with the training notebook.

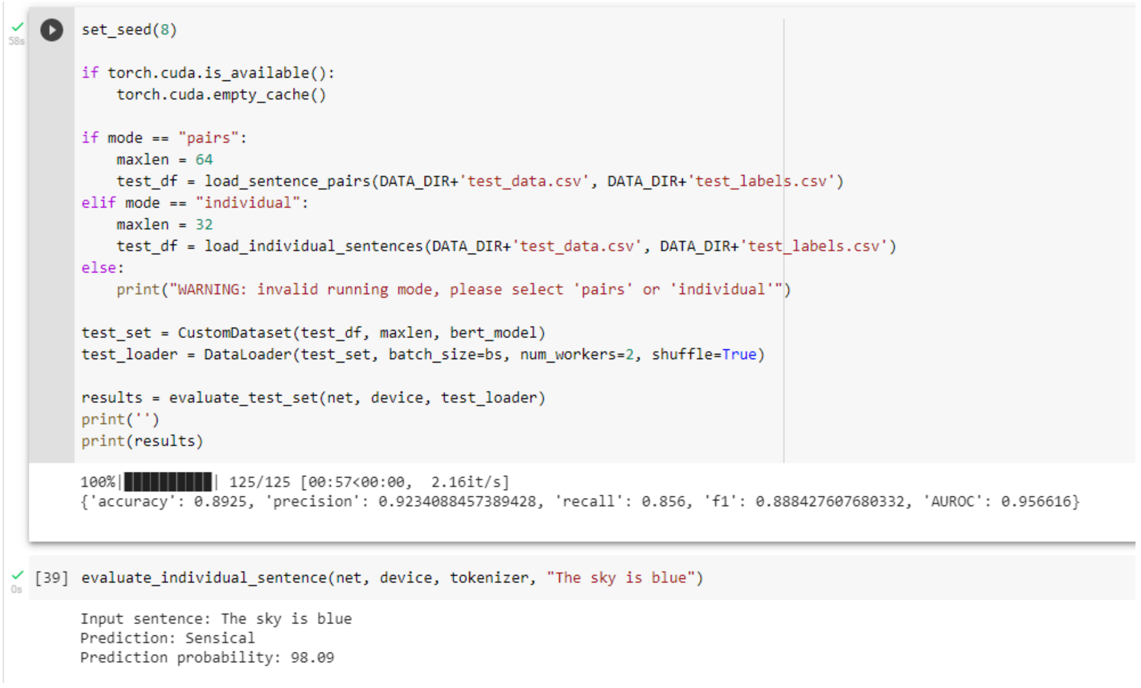
2. Run the first code block to connect to Google Drive as done in step 4 of the training process.

3. Set the ‘DATA_DIR’ variable to the same value you set it to in the training notebook. Set the ‘path_to_model’ variable to the path to the trained model, with the filename and file extension included. Finally, set the ‘bert_model’ and ‘mode’ parameters to their respective values for the model you are evaluating. For example, if they were set to ‘albert-xxlarge-v2’ and ‘individual’ when training your model, they should be set to the same values when running the evaluation. The batch size can be changed from the original value however setting it too high might result in the code failing to execute due to memory limitations.

4. Click **Runtime** in the menu bar at the top of the page followed by **Run all**.

5. Scroll to the bottom of the page. The model will be evaluated on the test set and the accuracy, precision, recall, F1-score and AUC-ROC/AUROC will be displayed.

6. The last code block in the notebook can be used to get a prediction from the model on any given input sentence by changing the string at the end of the function call (Set to ‘The sky is blue’ in the screenshot below’). However, it should be noted that this function is only intended to be used with models which were trained on the ‘individual’ mode.



```
set_seed(8)

if torch.cuda.is_available():
    torch.cuda.empty_cache()

if mode == "pairs":
    maxlen = 64
    test_df = load_sentence_pairs(DATA_DIR+'test_data.csv', DATA_DIR+'test_labels.csv')
elif mode == "individual":
    maxlen = 32
    test_df = load_individual_sentences(DATA_DIR+'test_data.csv', DATA_DIR+'test_labels.csv')
else:
    print("WARNING: invalid running mode, please select 'pairs' or 'individual'")

test_set = CustomDataset(test_df, maxlen, bert_model)
test_loader = DataLoader(test_set, batch_size=bs, num_workers=2, shuffle=True)

results = evaluate_test_set(net, device, test_loader)
print('')
print(results)
```

```
100%|██████████| 125/125 [00:57<00:00,  2.16it/s]
{'accuracy': 0.8925, 'precision': 0.9234088457389428, 'recall': 0.856, 'f1': 0.888427607680332, 'AUROC': 0.956616}
```

```
[39] evaluate_individual_sentence(net, device, tokenizer, "The sky is blue")
```

```
Input sentence: The sky is blue
Prediction: Sensical
Prediction probability: 98.09
```