University of Kent

Fine-tuning pre-trained Large Language Models for customer support dialog summarization

Marouane Merno

Supervised by Pr. Frank Z. Wang

MSc Computer Science

2023/2024

ABSTRACT

Large Language Models have revolutionised the field of natural language processing. These models, based on the recently introduced transformers architecture, are trained on massive amounts of text data and can perform well on various tasks including.

Due to the large amount of data used to train these models, they often lack in knowledge depth when confronted about specific tasks or domains. This led to a shift in the research community to leverage more effectively the capabilities of these models and allow their usability on specific tasks.

Fine-tuning has become an increasingly important method to adapt pre-trained language models, by adjusting their parameters to a specific task or domain. It allows the model to acquire a deeper understanding of the context, terminology and general information about this task, and makes it able to generate text accordingly.

The goal of this project is to fine-tune a pre-trained large language model, and make it able to generate quality abstractive summaries, using a formatted dialog summarization instruction dataset to train the model, and recognised evaluation benchmarks to assess the quality of its outputs.

We show in this project that fine-tuning an open-source model on the specialised instruction dataset has a visible effect on the quality of the output text and increases its performance in summarising dialogs within the context of customer service.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to thank my supervisor Pr. Frank Zhigang Wang for his guidance and continuous support throughout the development of my project.

I would also like to thank the School of Computing's staff that were helpful in providing me with the necessary resources to conduct this project.

Finally, I would like to thank my family for their continuous support throughout my studies and for the completion of my project.

1. INTRODUCTION

Large Language Models, based on transformers architecture, have revolutionised the field of Natural Language Processing. These models demonstrated remarkable ability in learning complex language patterns and generating human-like text across a wide range of subjects. This makes them a first choice in building various natural language processing applications.

Despite their general efficiency, when applied to domain-specific tasks with a particular text format and terminology, these models become limited and may not produce satisfying results that can be used for a specific purpose. Fine-tuning refers to the process of adapting a pre-trained to a specific task by training it on a related dataset. This model starts to be able to learn in-depth to generate text in the terminology

The development of artificial intelligence solutions is transforming the field of customer service (Aishwarya et Al, 2019). The ability to efficiently analyse and manage many customer interactions has become a critical task that companies would gain in improving (Douglas,2023). The most effective way to analyse these interactions is to gather the main information from the conversation through summarisation.

Text summarisation in the context of Large Language Model has been subject to several past research, and dialogs summarisation can also be considered similar. Effective summarisation of dialogs can improve the information retrieval process, which can be used to gather insights about elements such as the quality of goods and services, and the quality of customer support within various business fields. In the field Natural Language Processing, previous research in dialog summarisation pointed out the importance in adapting every model to a specific task (Ankan, 2019).

The motivation for this project is to develop a practical solution that can be adopted by organisations to improve their internal workflow through customer support insights. It explores the fine-tuning process of a commercial Large Language Model, to enhance its performance on dialog summarisation in a context of customer service interactions. The primary aim of the project is to apply fine-tuning methodologies to adapt the model in generating text in the required format, taking as a reference abstractive summary from a domain-specific dataset, and backing our claims with recognised evaluation metrics in the context of natural language processing.

This project has the potential to contribute to the field of Large Language Models applied to enterprise customer service, by developing a quality analysis tool that could provide insight about conversations between AI or human agents and customers.

The content of this dissertation is organized as follows:

- Background Research: A comprehensive review of relevant literature, that includes foundational concepts in Natural Language Processing, Transformers architecture and training of Large Language Models.
- Design: An introduction to our development projects, including an exhaustive list of the tools and resources that will be used throughout the development project.
- Implementation Strategy: A detailed description of all the steps gone through to develop or software, including the dataset preparation and the fine-tuning process, and the results.
- Evaluation of the results: An analysis of the results of the projects, highlighting the impact of our fine-tuning on the performance of the model in dialog summarization.
- Discussion, Conclusion and Future Work: An interpretation of the results, a discussion of the limitations of our development project, and a proposition of future possible work on the same topic.

2. BACKGROUND RESEARCH

In this section, we will provide the state-of-the-art theoretical background about Large Language Models, their development and customisation to specific tasks. We will briefly cover the main components of the transformers architecture which represents the key foundation of Large Language Models, and we will discuss related research papers to provide a relevant context for our development project.

2.1. Tokenization

Large Language Models main feature is their ability to perform natural language processing tasks. When trying to understand the meaning of text information, the models need first to convert the text data into numerical values to make them understandable by machines, this process is called tokenization.

2.1.1.Tokenizer

In natural language processing, the tokenizer prepares the input text to be processed by the model by undergoing a tokenization, which divides it into smaller tokens, usually words but sometimes sub-words or letters, and assigns a specific index number to each unique token.

To demonstrate how the process works, we can consider the example sentence: "Deep Learning is a vast field". The most prevalent approach and the one used for transformer models is to tokenize each word of the sentence, thus the input sentence will be split into six tokens stored in the following array: ["Deep","learning","is","a","vast","field"]. This array is called the vocabulary and contains all different words of the input sequence.

Next, we can assign integer values to each token in the vocabulary based on the order of their occurrence. In this case the token "Deep" would be assigned the value 1 and the token "field" the value 6. The result of the mapping would be: {"Deep": 1, "learning": 2, "is": 3, "a": 4, "vast": 5, "field": 6}

The resulting mapping from the last tokenization can be used to encode other input's text data into numerical tokens. As an example, the tokenization of the input sequence "Deep learning is vast" would return the mapping [1, 2, 3, 5].

There are several types of tokenization algorithms that can be used, among them Byte-pair encoding, used in the tokenizer of the Llama 2 model.

2.1.2.Byte-Pair Encoding

Byte-Pair Encoding (Rico et al.,2016)) is originally a data compression algorithm and is currently used in natural language processing to handle the tokenization of text. Its main function is to merge the most frequent pair of consecutive bytes or character in the input text until a defined vocabulary size is reached. The resulting tokens can then be used to represent the original text more effectively. As an example of tokenization using byte-pair encoding, considering the input sequence "High Higher Highest":

The first step is to split all characters in the sequence, including spaces, which gives us the vocabulary array: ['h','i','g','h',' ','h','i','g','h','e','r',' ','h','i','g','h','e','s','t']. In this step, the initial vocabulary consists of all the bytes or characters in the text corpus: Vocabulary = {'h','i','g','e','r','s','t'}

In the second step, we calculate the frequency of each character. This gives us the following data structure: {h: 6, i:3, g:3, e:2, r:2, s:1, t:1}.

The third step is to find the most frequent pair of two characters, merge the pair and update the frequency counts: {hi:3, ig:3, gh:3, he:2, er:1, es:1, st:1}. There are three most frequent pairs of characters: hi, ig and gh. We use lexicographical order to select one pair, so "gh" is selected and added to the vocabulary: Vocabulary = {"h", "i", "g", "e", "r", "s", "t", "gh"}. This third step is repeated until the pre-defined vocabulary size is reached. At the fourth iteration, the state of the vocabulary is the following one: vocabulary = {"h", "i", "g", "e", "r", "s", "t", "gh", "hi", "ig", "he"}.

In the end, the original text corpus can be represented using these sub-words:
"hi" -> "hi"
"ig" -> "ig"
"gh" -> "gh"
"he" -> "he"
"er" -> "er"
"es" -> "e" + "s"
"st" -> "s" + "t"

The main benefit of byte pair encoding is that it can be adapted to any corpus of text or languages as long as it is encoded in bytes.

Once the input text corpus is entirely tokenized, these tokens are passed through the model which typically includes an embedding layer and transformer blocks. In the next section, we will see how these token values are given a meaning when passed through the model.

2.2. Transformers Architecture

2.2.1. Introduction

In this subsection we will present the fundamentals of transformer architecture introduced in the paper "Attention is all you need" (Vashwani et al., 2017), which is the foundation of Large Language Models. It replaced the recurrent neural networks (Elman, 1990) as the go-to architecture for natural language processing, which processes the data sequentially one step at a time, and maintains a hidden state that gets updated when a new input comes in.

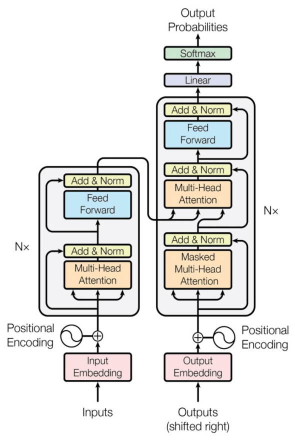

Figure 1: The Transformers Architecture

The transformer architecture relies entirely on self-attention to draw global dependencies between an input and an output. It relies entirely on this mechanism to understand the relationship among words within the same sentence and determine their meaning. This process necessitates the neural network to be able to gather information about each word in a sentence, this can be done using the tokenizer previously introduced that turns the input sequence.

2.2.2. Embedding Layers

We previously discussed how tokenizers turn the input sequence into tokens to make it possible to perform natural language processing tasks. This process happens prior to the transformer pipeline and ensure that all tokens in the input sequence are assigned an integer value. These tokens are then passed as an input to the embedding layer.

The goal of the embedding layer is to turn the input tokens into information usable by machines. It converts the tokens into vectors embeddings, these vectors are used by the model to capture the nuances, connections, and semantic relationships between words in the input sequence to be able to determine each word's meaning depending on its position.

The embedding layer takes as parameters the variables *sequence_length* representing the number of unique tokens in the dictionary, and *vector_dim* referring to the dimensionality of embedding vectors in the embedding layer, which is a parameter that can be adjusted. Each input token is mapped into a dense vector, and together these vectors result in an embedding matrix *sequence_length * vector_dim* representing the output of the layer.

2.2.3. Positional Encoding

We previously covered how transformers use embedding vectors to represent each token, and so each word in the input. But when reading a sequence of words, each word is dependent on the words around it, and some words have different meanings in different contexts. Unlike Recurrent Neural Networks, transformers do not capture the distance between tokens in the input sequence. Positional encoding makes it possible for transformers to get information about the position of each word within the text sequence.

After each token in the dictionary is turned into a *vector_dim* dimensions word embedding in the embedding layer, positional encodings are added to each vector embedding to enable the model to understand the order of the words in the whole input sequence. They are created for each vector using sine and cosine functions, with the aim to generate values within the range of [-1,1], making it easier for the model to learn.

These positional encodings provide each token with a unique positional representation and are added to the vector embeddings. The output of the positional encoding layer is a set of encoding vectors containing both semantic and positional information about the tokens.

2.2.4. Multi-head Attention Layer

We previously covered how the embedding layer turn the input tokens into a matrix containing vectors, and how the positional encoder turns these vectors into positional vectors allowing the model to get the order of each token within the input sequence. These two steps enable the multi-head attention layer to understand how tokens relate to each other in a sequence.

Multi-head attention refers to the capacity of the attention layer in transformer architecture to focus simultaneously on many parts of the input text sequence.

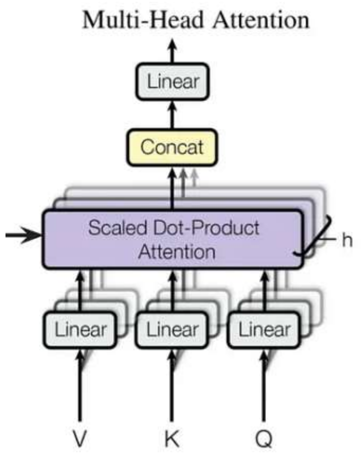

Figure 2: Multi-Head Attention Layer

In this layer, the attention procedure is repeated multiple times in parallel across multiple “heads”, each attention is called an Attention Head.

The input of the multi-head attention layer is the embedding matrix resulting from the embedding and positional encoding layers.

The number of attention heads in the layer *num_heads* is an adjustable parameter; each attention head processed the input matrix and outputs a matrix of dimensions *input_length * attention_head_length* where *attention_head_length = vector_dim / num_heads.*

Then, after all the attention heads have processed the input, their outputs are concatenated to produce the output of the Multi-Head Attention layer, which is a matrix with the same dimensions as the input matrix.

$$\begin{aligned}\mathrm{MultiHead}(Q,K,V) &= \mathrm{Concat}(\mathrm{head_1},\ldots,\mathrm{head_h})W^O \\ \text{where}\ \mathrm{head_i} &= \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)\end{aligned}$$

Since many different types of attentions depending on the model’s architecture, the attention mechanism will be covered more accurately in the encoder-decoder pipelines section.

2.2.5.Linear Classifier

The linear classifier is the final layer of the transformer architecture, it performs a classification of the output from the multi-head self-attention layer.

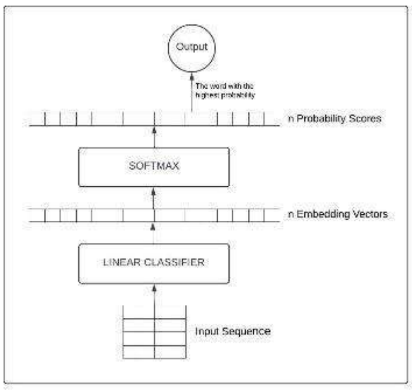

Figure 3 : Linear Classifier

In the linear classifier, this output in the form of a sequence of vectors is introduced to a Softmax function, which turns the vector's values into a range of probability scores between 0 and 1 representing the probability for every possible token. The highest value among these probability scores is then identified, and its index points to the word from the vocabulary that the model predicts as most likely being the next one in the sentence. This represents the final output of the linear classifier.

2.3. Encoder, Decoder, and Encoder-Decoder pipeline

2.3.1. Encoder Self-Attention : Scaled-Dot Product Attention

Scaled dot product attention determines which words suit our sequence given the context, that is to decide which words of a sentence the transformer should focus on.

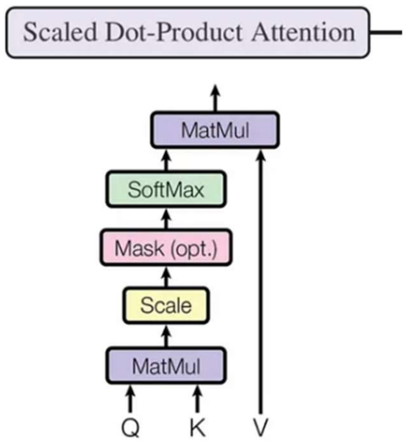

Figure 4: Scaled Dot-Product Attention Layer

The Figure 4 shows the structure of a scaled dot product attention operation, where n is the number of tokens in the input sequence and d is the dimensionality of those tokens.

Multiple sets of Query, Key, Value vectors are generated and processed independently:

- The query vector (Q) : Represents the position in the sequence where the attention is focused.
- The key vector (K) : Identifies all the elements in the input sequence.

- The value vector (V) : Contains information associated with each position in the sequence, to be averaged based on attention

The Q, K and V vectors are then taken as an input to each multi-head attention block. Each head calculates attention scores and processes the input sequence independently. The results from all the heads are then concatenated and linearly transformed to produce the final output. The scaled-dot product attention is calculated using the following formula:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

2.3.2. Decoder Attention : Masked multi-head self-attention

A normal encoder-decoder transformer uses two different parts: one part to encode the input, called the encoder, and the other part to generate the output, the decoder. It applies many types of attention mechanisms: The self-attention and encoder-decoder attention.

Decoder-only transformers have a single part that is used both to encode the input and to generate the output. They use Masked self-attention, a variant of self-attention and apply it to both the input and the output. Compared to normal Self-attention which allows all tokens within the sequence to be considered when computing attention scores, masked self-attention modifies this pattern by preventing the current tokens to access the tokens located next in the sequence.

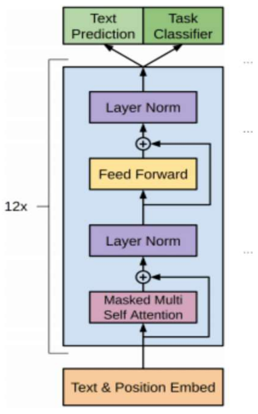

Figure 5: Masked Multi-Head Attention Layer

As an example, considering our input sequence: “deep learning is a growing field”.

Assuming we are trying to compute attention scores for the token “learning”. Usual self-attention will compute an attention score between the current token “learning” and every other token of the input sequence. However, masked self-attention only compute attention scores for the tokens “deep” and “learning”, so it keeps the attention layer from looking forward in the sequence during self-attention. The masked multi-head attention is calculated with the same formula as the MultiHead attention.

2.4. Large Language Models: Training and Adaptation

The transformer pipeline allows Large Language Models to learn an extremely vast corpus of information during the training process. Although these models show exceptional abilities on natural language processing tasks, they are only able to perform well on generalized information. For this reason, other models based on these pre-trained models are being developed. This process necessitates way less resources and time than full-scale training and can give satisfying results.

2.4.1. Supervised finetuning

Supervised Finetuning is the process of updating pre-trained models for specific tasks by training them on a smaller dataset. It is often used for-in domain information, or to make the model generate an output in a specific format. It differs from unsupervised learning as the data used to train the model has been checked, in contrary to the initial training of large language models.

During the finetuning phase, the model is exposed to a new labelled dataset training it for the target tasks, and it calculates the error or difference between its output prediction and the actual output given in the dataset. This error margin is then used to adjust the model's weights via an optimization algorithm like "Adam" (Diederik et Al., 2015).

The model performs multiple iterations, called "epochs" on the dataset where it continues to adjust its weights in the purpose of tuning to a new weight configuration that minimizes the error value in comparison to the training dataset outputs.

Supervised finetuning makes it possible to leverage the previously learned knowledge and adapt it to the nuances and specific patterns present in the new dataset, therefore making the model more specialized and effective for the target task.

During this fine-tuning, the Large Language Model is updated with the labelled data. The weights of the model change based on the different between its guesses and the actual answers in the training dataset. This helps the model learn details found in the labelled data to improve at the specific task.

2.4.2. Instruction Fine-Tuning

The supervised fine-tuning of large language models can be done using several methods, among them instruction fine-tuning. Instruction finetuning allows to train a model by providing instructions/outputs examples showing it how it should respond to a specific query. In this case, the dataset contains 3 types of information: The instruction, containing both the task that we want the model to perform e.g. translating or summarizing a text and the input text that we provide the model with, followed by an output which is an example of how we want the model to respond to the instruction input query. Training the model with these instruction/output pairs allows it to generate text in an updated way, to match our purpose.

2.4.3. LoRA : Low-rank Adaptation

Training large language model with parameter efficient finetuning is computationally challenging and involves adjusting millions of parameters, this approach requires a lot of computational resources and time. Low Rank Adaptation (Edwards et al., 2021) was recently introduced as an effective solution to this problem.

LoRA is a training technique that reduces significantly the number of trainable parameters in a transformer model by inserting a small number of new weights into the model and training them.

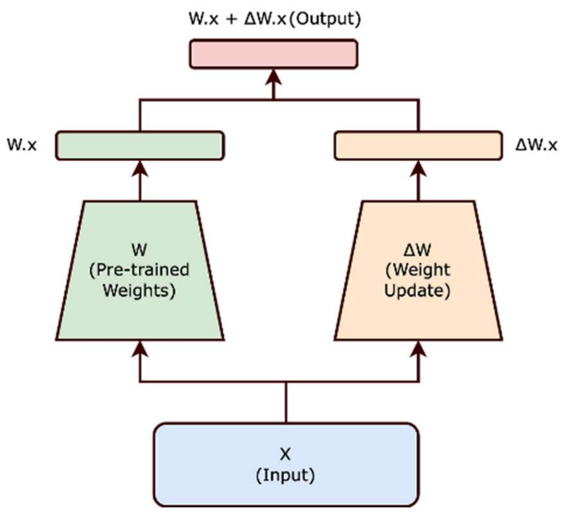

Figure 6: Weight Update in LoRA

In traditional parameter-efficient fine-tuning, a pre-trained model weights are modified to adapt it to a new task. This modification is done by changing the original weight matrix W of the model, by introducing another matrix ΔW, the resulting updated weight matrix W' is thus expressed as (W + Δw)

In LoRA fine-tuning, the training matrix ΔW is decomposed in a product of two smaller matrices (A) and (B), resulting in the following expression of the updated matrix (W' = W + AB). The original weight matrix W is kept frozen during the training process, so the product of the lower-dimension matrices A and B represents a low-rank approximation of ΔW.

Training a model using lower rank matrices considerably reduces the number of trainable parameters. Considering that W is a two-dimensional matrix n*n, updating it would involve n^2 parameters. But if we use a lower rank matrix of size r, the number of parameters would be equal to the sum of the parameters in matrices A (r*n) and B (n*r), which is significantly less considering that the value of r is much smaller than the value of n.

2.4.4. QLoRA: Quantized Low-rank Adaptation

Quantized Low-Rank Adaptation or QLoRA (Tim et al., 2023) takes Low-Rank adaptation a step further by introducing to it 3 new concepts, aiming to reduce memory and at the same time keeping the same quality performance.

The first concept is the introduction of the 4-bit NormalFloat (NF4) data type used to store the parameters of trained models; these parameters were usually stored in a 32-bit format, but QLoRA compresses them to a 4-bit format. The NF4 format uses 4 bits for the exponent and 4 bits for the mantissa, and performs bitwise operations for arithmetic computations during the training. This enables to make the training process faster and to significantly reduce the memory footprint of the large language models.

The second breakthrough of QLoRA is Double quantization, it involves using 4-bit quantization two times during the training process: one time on the base model to reduce memory usage, and another time to the low-rank adapter layers. This approach allows compressing large language models with minimal incidence on their performance, and makes it possible to run them on devices with limited memory. The last concept is a memory management technique called Paged Optimizers, and used during the training process. When training LLMs with a lot of parameters it's a common problem to run out of GPU memory, in these cases, paged optimizers allow to manage these memory spikes by swapping data in and out of memory, only keeping the most relevant data every time. Hence, paged optimizers work like regular CPU paging meaning that they become active only if we run out of GPU memory.

3. DESIGN

3.1. Interpretation of requirements

Based on the description of the software, the requirements could be described as follows :

- The software must be able to generate relevant abstractive summaries of a dialog
- The software must show clear improvement of its performance in dialog summarisation after being trained on a relevant corpus dataset
- The software must be adaptable to different organisations in the context of customer support

3.2. Pre-trained Base Model: Llama 2 7b chat

The first step in fine-tuning an LLM is to choose a base model that we are going to train. There are numerous available open-source Large Language Models, differing by their architecture and the number of parameters they have been trained on.

The Llama 2 7b model was chosen for this project, Llama2 is a family of pre-trained and fine-tuned large language models (LLMs) released by Meta AI in 2023. It was released free of charge for research and commercial use and is able to perform various natural language processing tasks, ranging from text generation to sentiment analysis or code programs generation. This model was chosen first because it requires significantly less memory and computing resources to be trained than other models, it has enhanced performance capabilities for the smaller model with 7b parameters, and then because of the large amount of documentation available online that would help us in conducting our project. We will use the chat version of the Llama 7B model to be able to use it in the form of a conversational agent.

3.3. TWEETSum Dataset

To train the model for dialog summarization, we decided to use the TWEETSUM dataset. TWEETSUM is an open-source dataset released under the CDLA licence, its aim is to summarise text transcripts of dialogs between customers and human support agents. The dataset contains 1100 dialogs, and around 6500 summaries generated by human annotators. The dialogs are reconstructed from conversations that appear in the “Kaggle Customer Support on Twitter” dataset, they cover a wide range of products or services, and various companies from different sectors.

The summaries included contain both extractive summaries, where fragments from the original text are used to form a summary, or abstractive summaries in which new expressions are generated to summarize the text. For the training purpose, only the abstractive summaries were used and the first abstractive summary for each dialog was selected as a representative sample to train the model.

The dataset was released with train, validation and testing sets. The training set is made of 879 dialog/summary rows and was used to train the model with instruction fine-tuning, the validation set was used to calculate the validation loss during the training, and the testing set was used to evaluate the quality of outputs of the trained model.

3.4. Instruction Format

To be able to train the Llama2 7b using an instruction dataset, we need to use a specific training prompt containing an instruction text, and an output text.

```
<s>
    [INST]
            <<SYS>> {{ System prompt }}<</SYS>>
     {{ user prompt }}
    [/INST]
</s>
```

The <s> </s> tag represents the beginning and the end of the input sequence. It allows the model to understand where the input starts and ends.

The [INST] [/INST] tag is the marker for instructions, also called prompts. They indicate that the text enclosed within them contains instructions that the model needs to follow.

The <<SYS>> <</SYS>> tag is optional and indicates that the text enclosed within them contains the system prompt.

The {{ system prompt }} is also optional and represents the system prompt variable, and it can be used by the user to give an overall context to model response.

The {{ user prompt }} placeholder acts as a variable that gets replaced with the actual user input when construction the input sequence for the model.

4. IMPLEMENTATION

Training Large Language Models necessitates advanced hardware resources, so an access to the University of Kent's Hydra cluster was granted, the cluster uses Slurm as a workload manager and contains several GPU options.

The finetuning code was written in a Jupyter Notebook running on the Hydra cluster. The training was conducted using a single GPU of type Ampere (A100), 24GB of memory were allocated from the cluster to fit the memory requirements of the experiment.

4.1. Data Processing

4.1.1. Data Cleaning

Cleaning the dataset mainly involved modifying the conversation data between the user and agent. Although the content of the dialogs can be considered clean as not irrelevant or bad conversations were included, there were still some modifications to be done. The textual conversations happen between humans on Twitter, so the dataset may contain elements that may create noise in the dataset as they are not relevant for our purpose of generating text content. Some of these elements were identified as URL's, usernames preceded by a @ sign, names or initials of the user or agent preceded by an “^”, were removed from the dataset.

4.1.2. Data Reduction

The original TweetSum dataset contains a lot of information, 5 rows in total and much more sub-rows. Some of the information present in the dataset is redundant, and some of it doesn’t fit our purpose. Hence, the only used subsets of the dataset were the “abstractive summaries”, the “user utterance” and the “system response”. Hence, to reduce the volume and complexity of the data, we made sure to remove all other unnecessary data in our data pre-processing code and keep only the elements necessary for the training process.

4.1.3. Data formatting

The original format of the TweetSum dataset didn’t match the dataset format requirements for our training task. The finetuning of Large Language Models requires the use of an instruction dataset in the format “Instruction/Output”. Thus, we had to reformat the dataset to match the instruction format, this was done using python code to and Pandas library for visualisation purpose. The instruction part of the dataset contains the full dialog between the user and the agent, the dialog was created by gathering all text present in “user utterance” and “system response”, and formatting them in the form of a conversation between two parties. The output part of the dataset contains the respective summary for each full dialog, which is the first element in “abstractive summaries” column.

4.2. Training the model

4.2.1.Training Environment

Training Large Language Models requires advanced hardware resources, an access to the University of Kent’s Hydra cluster was granted, the cluster uses Slurm as a workload manager and contains several GPU options.

The finetuning code was written in a Jupyter Notebook running on the Hydra cluster. The training was conducted using a single GPU of type Ampere (A100), 24GB of memory were allocated from the cluster to fit the memory requirements of the experiment.

4.2.2. Dataset Split

The finetuning was done using the training split from the previously formatted instruction dataset. The validation split was also used to measure the performance of the model on unseen data.

4.2.3. Loading the quantized model

Prior to training the Llama2 base model it needs to be loaded locally. However, even for the relatively small model we’re currently using, the Llama2 7 billion parameters, running it requires approximately 30GB of GPU memory resources, higher than our available 24GB of RAM.

The solution is to quantize the model like previously stated in the background research, quantization allows use to represent the model's weight with a lower precision data type, here NormalFloat4 equivalent to 4-bit, instead of the original 32-bit weights.

Double quantization makes it possible for us to quantize gradients in addition to parameters, this ensures that we maximize the compression of the model while minimizing its loss of performance.

Hence, we create a configuration using the "BitsAndBytes" library where we set the respective parameters for both NF4 and double quantization. This configuration is then used to load the base model Llama 2 7b chat from the Hugging Face Hub official repository.

4.2.4. Creating a LoRA Configuration

As previously stated in the background research, the LoRA method for finetuning Large Language Models involves frozing the original weight matrix W and decomposing the training matrix W' into two smaller low-rank matrices A and B in the attention layer, to reduce the number of parameters that need to be finetuned. In the configuration used, we used the superclass PeftConfig to set the values of the required parameters, the parameter r represents the dimensionality of each low-rank matrices, and the task_type variable indicates the nature of the task the model is fine-tuned for, which here is "causal language modelling".

4.2.5. Declaring the training Arguments

The "*TrainingArguments"* class is created and contains all the hyperparameters and training arguments to set for fine-tuning the model, some of these are built-in and have default values that can still be adjusted. The default optimal training arguments values are available in the trainer class documentation from the Transformers library in Hugging Face Hub, these values were slightly adjusted during the training, this table contains the values used for some of them:

Parameter	**Value**
Epochs	3
Training and Evaluation Batch sizes	8
Gradient Accumulation	1
Optimizer	paged_adamw_32bit
Learning Rate	2e-4
Evaluation Strategy	steps
Evaluation Steps	0.1
Logging Steps	20
Max Gradient	1.0

Table 1 : Fine-Tuning Arguments

The "epoch" is used to describe one complete pass through the entire training dataset. Our dataset has 879 instruction/output pairs, one epoch means that the model has had the opportunity to learn from each pair once. An epoch too low can lead to the model not learning enough from the training that, and an epoch too high can cause the model to perform poorly on unseen data. The recommended epoch value is 3, we choose to set the parameter to this value as it was minimising the training loss.

The ”batch_size” represents the number of training examples used in one step during the training. Training a model with all data at once can be too much to handle for a computer, so we divide the dataset into batches. Our dataset contains 879 pairs of instruction/output and we choose a batch size of 8, this means that there will be 879/8 = 109 + 7/8 so a total of 110 batches during the training.

4.2.6. Training Metrics

The fine-tuning of the model generates training metrics computed over the course of the training, monitoring these metrics can provide us with insights about the new model’s effectiveness and generalization capabilities. By assessing these metrics, we can measure how well the fine-tuned model performs on task-specific data and identify potential areas for improvement.

The training loss measures how good our model is performing on the training data, it is computed during the finetuning process, we can use this metric to improve the training process by updating the training arguments like the number of epochs and/or the batch size. The training loss is updated every step during the training, its value should be decreasing over time to show that the model is learning from the data.

The training loss is computed as the average loss across all training data points, using the cross-entropy function:

$$H(p, q) = -\sum_{x \epsilon classes} \overset{\text{True probability distribution (one-shot)}}{p(x)} \log \underset{\text{Your model's predicted probability distribution}}{q(x)}$$

However, this metric is not sufficient to demonstrate the quality of the model. Although it can show that a model learned from the data, this model may be overfitting and may perform poorly on new unseen data.

For this reason, the validation loss was also included in the training metrics. The validation loss is a measure of how well our model is performing on the validation dataset, containing 110 instruction/output pairs. It allows to assess how well the model performs on new data that it not included in the training set. If the validation loss starts increasing while the training loss is decreasing, it’s a sign that the model is overfitting on the training data.

The validation loss is computed similarly to the training loss, using the cross-entropy loss function.

4.3. Results

4.3.1. Training Loss

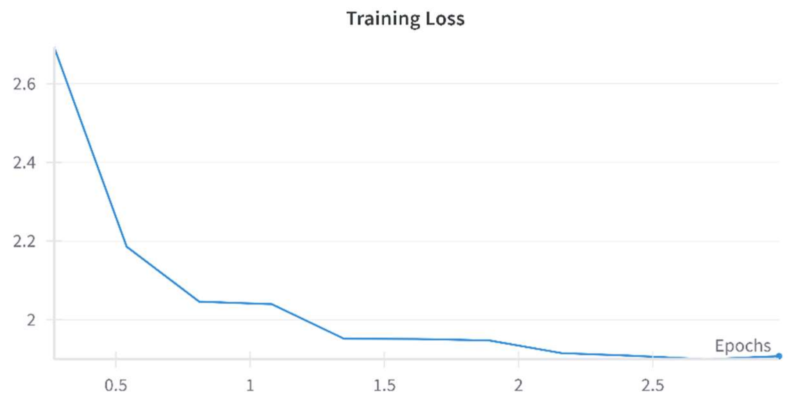

Figure 7: Training Loss

The value of the training loss keeps decreasing overall keeps decreasing, it stagnates at around 0.8 Epochs but never starts increasing. We can say that the model is effectively learning from the fine-tuning. The final value of the training loss is 1.95 which can be considered satisfying in the context of fine-tuning.

4.3.2. Validation Loss

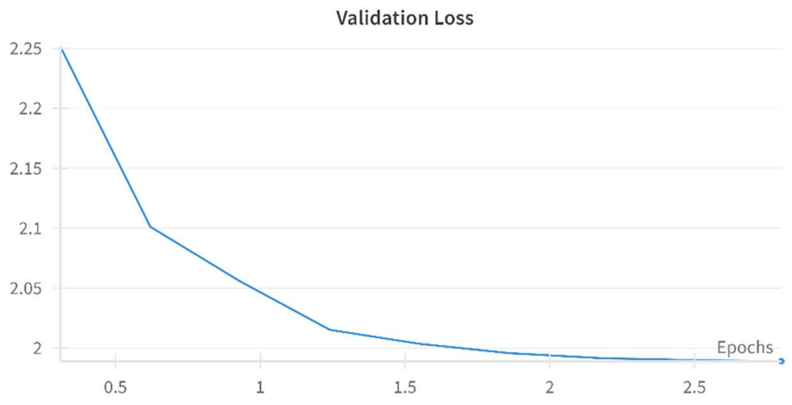

Figure 8: Validation Loss

The validation loss is constantly decreasing throughout the Epochs, it shows us that there is no overfitting that could lead the model to perform poorly on unseen data. The final validation loss value is 2, which is a also satisfying result in the context of fine-tuning.

4.3.3. GPU Memory Allocated:

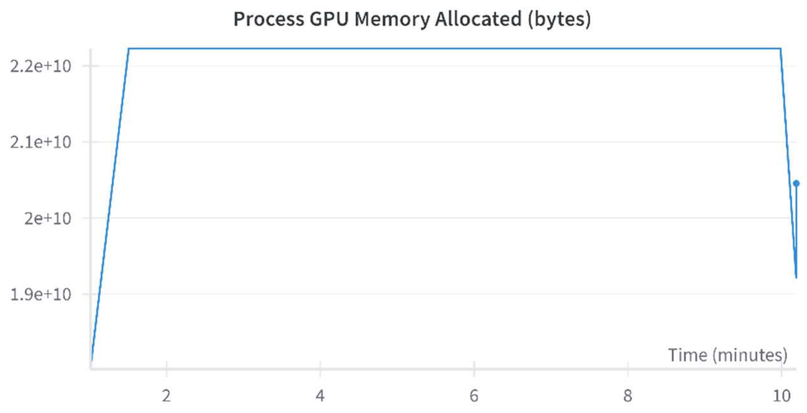

Figure 9: GPU Memory Allocated

This measure was done to ensure the maximum available GPU memory of 24 GB is enough to fine-tune the model. The main elements that have an impact on the GPU memory consumption are the training hyperparameters (number of epochs, batch size…), and the size of the training dataset. The maximal GPU memory allocated measured was of 2.2e+10 = 21 GB. We can thus say that the GPU memory resources are sufficient for our training.

4.3.4. Fine-tuned model inference

After training the model on our dialog summarization dataset, we want to start using it for this specific purpose.

Pipeline is a class from the transformers library that allows the use of models for inference, it can run the model for several tasks.

To use the pipeline, we specify the model that we want to use along with its tokenizer, we specify the type of task to perform, in our case *"text-generation"* we are looking to perform, and we can also a maximum length for the output text to avoid overflow in case of errors during the training.

We also proceeded to create an example input consisting of the usual Llama2 prompt format tags, the system prompt previously included in the training dataset, and a textual dialog, preferably coming from the testing set to make sure that the model was not exposed to it.

5. EVALUATION

After being able to generate outputs using the fine-tuned model, our objective is to evaluate the performance of this model, compared to the base model that didn't go through weights updating. The purpose of this evaluation is to ensure that the model gained new knowledge following the training, this can be done at first simply by comparing the outputs for dialog summarization tasks for the two models, and then by using recognised benchmarks to assess their performance.

5.1. Output Comparison

5.1.1.Design

In our approach of comparing the summaries generated by the two distinct models and their respective tokenizers in two different notebooks. The base model was imported from the original repository on Hugging Face and was loaded into a Jupyter Notebook. The trained model was directly used in the Jupyter notebook file containing the training code to ensure the authenticity of its answers.

To ensure that both models generate text in the same context, and to avoid any irrelevant information, we will used the Llama2 7b chat prompt format, including the same system prompt used in the formatted dataset, hence, the prompt format of our inputs should be similar to the following one: [INST] <<SYSS>> {{ system_prompt }} <</SYSS>> {{ input }} [/INST].

5.1.2.Results

The following tables shows the outputs of Llama2 model, compared to the output of our base model, and a random reference output taken from the testing set of our dataset.

	Input	Output
Llama2	user: I'm locked out of my email and cannot seem to get the forms right.. Help agent: Hello, JaeJae! It's our pleasure to help. Just to clarify, which account are you pertaining to? Hello, JaeJae! Are you still experiencing the same issue? We are always here to help. user: Yes, I am.. All I want is this email so I can access accounts online agent: Have you tried to recover your account using this link: Hello, JaeJae! Were you able to access your account after filling out the form? user: After 3 attempts no. agent: 1/2 Oh! That's unusual. When was the last time you were able to access your account? Are you traveling or using a new device 2/2 when you tried to access your account? Please reply to us via direct message. Thank you. Hi, Jae! How are you? Were you able to access your Microsoft account? Please send us a direct message if you need assistance.	“The human, JaeJae, is locked out of their email account and is having trouble filling out the forms to recover it. The agent is there to help and asks JaeJae to provide more information about the issue, including the last time they were able to access their account and any changes they have made to their device or travel. The agent offers to help JaeJae recover their account and asks them to send a direct message for further assistance.”
Fine-tuned Model		“Customer is complaining that he is locked out of his email and cannot seem to get the forms right. Agent updated that they are always here to help and asked to send a direct message if they need assistance.”
Reference Summary		“Customer is complaining about the email which has been locked and is unable to get the forms. Agent ask several questions and updates to reply them via DM for further assistance.”

Table 2: Output Results

We can clearly see that the two summaries are different, and that the summary generated by the fine-tuned model shows higher similarity to the reference summary. This first evaluation can only highlight the difference in summaries generated after training a model, but it’s difficult to use it to determine which one of them is better. In the evaluation methods, we will use a set of benchmarks to assess each model’s performance summarising customer support dialogs effectively, using the summaries in the testing set as a reference.

5.2. ROUGE Evaluation

After noticing the difference in outputs from the base model and the trained model for dialog summarization tasks, each dialog summary will need to be evaluated to determine the most accurate one.

5.2.1. ROUGE Metrics

The ROUGE Benchmark, standing for Recall-Oriented Understudy for Gisting Evaluation (Lin C, 2004) is a set of metrics used to assess the quality of summarisation tasks. It compares summaries generated by machine learning models, against reference summaries typically created by humans to count the number of overlaps, word sequences and word pairs to measure the precision, recall, and F1 score for each summary. As an example, we can use the following reference and machine learning model summaries:

The weather is rainy today.

The weather is rainy.

The Rouge value is 4 since there are 4 overlapping words between the two sentences, but this value can't give us any insight about the similarity between the two sentences.

ROUGE-1 refers to the overlaps in unigrams (single words) between the system and reference summaries. The ROUGE-1 contains Recall, Precision and F1 Score metrics. The value included in our calculations corresponds to the F1 Score that takes into consideration both the recall and precision.

Recall = (number of overlapping words / number of words in reference summary) = 4/5 = 0.8
Precision = (number of overlapping words / number of words in model summary) = 4/4 = 1
F1 score = 2 * [(recall * precision) / (recall + precision)] = 2 * (0.8 / 1.8) = 1.6 / 1.8 = 0.8

In this example, the ROUGE-1 score will be equal to its F1 score, hence 0.8. We then proceed to make similar calculations for the other ROUGE metrics depending on their specifications.

ROUGE-2 refers to the overlaps in bigrams (pair of words) between the system and reference summaries.

Rouge-L measures the longest matching sequence of words using the longest possible sequence that can be found in the two summaries (LCS). This metric is used to determine how much of the content generated by the trained model is reflective of the content considered important in the reference summary.

Rouge-Lsum is a variant of the ROUGE-L specifically adapted for summarisation task, it splits the text into sentences based on newlines and computes the LCS for each sentence, then it takes the average score of all the sentences.

5.2.2. Design

The performance of our model in dialog summarization tasks was qualitatively assessed through this series of metrics, providing a direct comparison of its effectiveness of our fine-tuned model against the Llama 2 base model.

The evaluation was conducted on the Evaluation code file. Two external libraries were used as dependencies: the *"evaluate"* library which simplifies the process of comparing models when evaluating their respective performances, and the *"rouge_score"* package, designed to replicate ROUGE scoring implementations.

In the purpose of generating the summaries to evaluate, two model inferences were run in the Evaluation Notebook: The Llama2 7b chat base model and the fine-tuned model. The testing set from our original dataset, containing 110 pairs of dialogs and abstractive summaries was also loaded to use it for the testing step.

Two python function were created to run a text generating pipeline with each model's inference and taking as an input the dialogs from the testing set. These functions were returning a list of 110 summaries, each one of them corresponding to the model's output for the respective dialog from the testing set.

The "rouge-score" library was used along with a function to compute the average ROUGE scores for 110 summaries generated. First, we computed the ROUGE scores of the base Llama 2 model with the reference summaries from the testing set, and then the scores of our fine-tuned model with this same testing set. This resulted in two distinct sets of ROUGE scores that were assessed to determine which of the two models is more performant for summarization.

It should be noted that because we were not getting consistent outputs from model inference in my first trials, some of these important lines of code do not appear on the Evaluation code file. It should also be considered that outputs from the base model were often too long and unpredictable. This was partially fixed by specifying a maximum length in the system prompt, and by ensuring that the list of outputs contains relevant information. Considering that the evaluation was done using 110 summaries, any other exceptional case would not have a significant impact on the results.

5.2.3. Results

This section illustrates the results in dialog summarization performance quantified with ROUGE metrics.

The rouge-score library was used to generate ROUGE metrics using outputs from two different model inferences, our fine-tuned model and the Llama2 7b base model.

Metric	Fine-tuned Model	Lama2 7b chat Model
ROUGE-1	0.4035	0.2338
ROUGE-2	0.1585	0.0513
ROUGE-L	0.3539	0.1761
ROUGE-Lsum	0.3885	0.2005

Table 3: ROUGE Score results

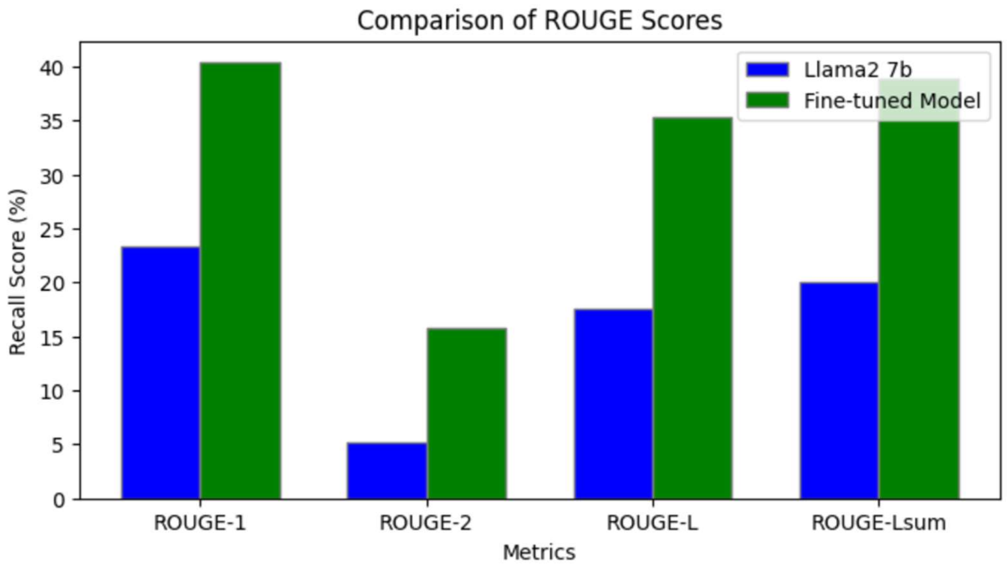

Figure 10: ROUGE Score results comparison

Higher ROUGE scores for a model show a better alignment of the generated summaries of this model with the human-made reference summaries. The comparison displayed in the Figure clearly shows significant difference in all ROUGE metrics, pointing out an improved performance of the fine-tuned model in capturing details more effectively for summarising dialogs in a customer support context.

5.3. BERTScore Evaluation

5.3.1. BERTScore

BERTScore (Tiyani et Al., 2020), represents a collection of evaluation metrics used in Natural Language Processing to measure the similarity between generated and reference text content. Unlike ROUGE used previously, which is an n-gram metric, it uses the vector embedding of the pre-trained Large Language Model BERT to capture the text meaning in both the candidate and reference texts, and matches words in candidates and reference sentences by cosine similarity. It has been shown to correlate with human judgement on sentence-level and system-level evaluation. Moreover, BERTScore computes precision, recall and F1 measure, which can be useful for evaluating different language generation tasks.

Unlike the ROUGE evaluation methods which is based upon the measure of n-gram similarity, it leverages the vector embeddings of the pre-trained Large Langue Model BERT (Bidirectional Encoder Representations from Transformers) to capture the semantic and contextual information of words and sentences in generated and reference text data. The approach makes it more effective in measuring the performance of generated text due to its ability to assess the overall meaning, fluency, and order of the output, while ROUGE metrics only assess the word match.

BERTScore includes three different metrics: BERT-Precision P_{BERT} measures how well the model avoids introducing irrelevant information in the generated text, BERT-Recall R_{BERT} measures how well the model avoids omitting relevant information in the generation text, and BERT-F1 F_{BERT} combines both.

$$R_{\text{BERT}} = \frac{1}{|x|} \sum_{x_i \in x} \max_{\hat{x}_j \in \hat{x}} \mathbf{x}_i^\top \hat{\mathbf{x}}_j \ , \quad P_{\text{BERT}} = \frac{1}{|\hat{x}|} \sum_{\hat{x}_j \in \hat{x}} \max_{x_i \in x} \mathbf{x}_i^\top \hat{\mathbf{x}}_j \ , \quad F_{\text{BERT}} = 2 \frac{P_{\text{BERT}} \cdot R_{\text{BERT}}}{P_{\text{BERT}} + R_{\text{BERT}}}$$

5.3.2.Results

It should be pointed out to the reader that the BERTScore was not included in the Corpus Submission files, nevertheless the evaluation steps are To calculate the BERT, running each model's and collecting outputs for the inputs in the dataset. The bert-score library was used to evaluate the model, then we created a function that takes as parameters the evaluated summaries and the reference summaries, and returns the BERT-Precision, BERT-Recall and BERT-F1 for each output of the

The two following tables contain the average BERTScores calculated for 109 outputs from each model:

Precision	Recall	F1 Score
0.7368	0.7489	0.7403

Table 4: Average BERTScore Llama 2 -7b

Precision	Recall	F1 Score
0.8352	0.8323	0.8334

Table 5: Average BERTScore fine-tuned model

The BERTScore values show an improvement of the dialog summarization task in our fine-tuned model for the Precision, Recall and F1 Score. Although this improvement is smaller than the one displayed in the ROUGE metrics, it is still enough consequent to be considered.

6. DISCUSSION

6.1. Dataset quality

The fine-tuned model was trained on an instruction dataset consisting a full dialog as an instruction, and a summary as an output. The dialogs in the dataset were already curated, and we cleaned the dataset by removing all the unnecessary elements. However, since the dialogs come from real-world online discussion between customers and agents, there were still some elements in the dataset that may cause noise in the data and make it harder for the model to understand the instruction, these could be grammatical errors, emojis, or names. These models can have a deleterious effect on the performance of the trained model.

Furthermore, the abstractive summaries used to train the model to generate outputs were made by humans, and although their format is usually consistent, it is possible that theses summaries contain human mistakes or biases that could hinder the model in generating correct and contextual information.

Finally, the training dataset contains 879 instruction/output pairs, it can be considered a limited amount in the context of fine-tuning Large Language Models but can be explained by the large length of the instruction text containing a full conversation between a customer and an agent.

6.2. Training Hyperparameters

The trained model used in the evaluation was fine-tuned on a specific set of hyperparameters. These training hyperparameters are set before the training begins and have a significant impact on the performance of the model. There are at least a dozen arguments, and each one of them could have numerous values, so trying different combinations of arguments to find the optimal value is difficult and costly. The thing we can do instead is to understand the role of each argument to guess which values would give us the best results then looking at the training metrics (training loss, validation loss, learning rate, …) to confirm our assumptions. This was done to adjust most of the original arguments during fine-tuning.

6.3. ROUGE Limitations

In the paper "The Limits of Automatic Summarisation According to ROUGE", Natalie S. (2017), the author explored the effectiveness of summarisation systems evaluated by the ROUGE metric. Several key issues about the ROUGE metric are highlighted in the paper.

The first issue is that ROUGE primarily focuses on word match between the generated and reference summaries such as recall and overlap in n-grams. This approach often overlooks the meaning of each word, and the coherence of the summaries.

Secondly, while ROUGE measures the extent to which generated summaries have the same content as reference summaries, it does not necessarily evaluate if the information is presented in a useful or informative manner, and so this can lead to high scores for summaries that are irrelevant.

6.4. BERTScore Limitations

BERTScore is calculated using the embedding vectors of the pre-trained model BERT. The score can't be explained because the embedding space of BERT can't be known. Even though the metric provides a numerical score, it doesn't explain how or why the particular score was assigned. In contrast, the ROUGE evaluation using the n-gram based metrics can easily be calculated to understand its value.

7. CONCLUSION

This project demonstrates the efficiency of refining pre-trained available Large Language models to match specific needs. Through performance evaluation with ROUGE and BERTScore metrics, we showed that our training was effective and that our model exceeded the performance of the pre-trained base model in the same task.

By using the dialog summarisation dataset that we cleaned, reduced and formatted to tailor it for our needs, we were able to train the model and adapt it not only to summarize text, but to understand and generate words in the context of customer service.

The findings validate the fine-tuning strategies used in our project and that quality of the dataset used. It opens the way for many other potential use cases of generative AI tools in the context of customer support tasks.

To conclude, this project could answer to the current need in companies to automate the quality analysis of their customer support services done by AI or human agents by gathering the main insights of each conversation. This can also be completed by other functionalities such as the ability to evaluate whether the support agent follows the guidelines of the company.

8. FUTURE WORK

Based on the knowledge and insights gained from the development of this project, there are still other research that can be conducted, and other functionalities that can be implemented.

Our model was trained using a specific set of training arguments values which were taken from the recommended set of values in finetuning. However, the optimal values for each training argument depends on many factors such as the size of the dataset, the tasks, the model architecture,… In the purpose of maximising the efficiency of the fine-tuning we can try to optimize the values of each argument, in a way to improve some training metrics (training and validation loss, learning rate,…), and evaluate the performance with the ROUGE and BERTScore metrics for each combination of This process would require a lot of time and computational resources, but would be beneficial in improving the quality of the trained model.

Finally, in our purpose to develop a tool that would be used in customer support quality analysis, it would be interesting to add other features to the model that we trained. Among the features that would benefit the current project is to give the model access to a knowledge base, which can be a file or database containing all the necessary customer service directives of the company. This kind of functionality requires the implementation of retrieval-augmented generation (RAG) system (Partick, et al. 2020), to give the model access to a knowledge base giving him access to elements outside of its training data to generate a response. By adding a default prompt to the RAG pipeline that instructs the model to determine if the content of the summary generated corresponds to the directives of the company available in the knowledge base, it would allow it to generate content that describes more accurately the interaction and provide the reader with more insights about the quality of the customer service workflow.

9. REFERENCES

Ankan, M., Ayan, KB., Raghav, R., Ravi,K., Prasenjit, D., Pawan, G., Niloy, G. (2024). *Long Dialog Summarization: An Analysis.* https://doi.org/10.48550/arXiv.2402.16986

Douglas, C. (2023). *Analysis of the effectiveness od e-customer service platforms on customer satisfaction at ABSA, Bostwana.*

Aishwarya, M., Jutendra, M, Naveem, L., Sagarika, M. (2023). *Artificial Intelligence Transforming Customer Service Management: Embracing the Future*. The Oriental Studies

Rico, S., Barry, H., and Alexandra, B. (2016). *Neural Machine Translation of rare words with subwords units*. https://doi.org/10.48550/arXiv.1508.07909

Ashish, V., Noam, S., Niki, P. Jakob, U., Llion, J., Aidan, G., Lukasz, K., Illia, P., (2017). *Attention Is All You Need*. https://doi.org/10.48550/arXiv.1706.03762

Elman, J.L. (1990). *Finding Structure in Time*. Coginitive Science 14, 179-211.

Diederik, P.K., Jimmy, B. (2014). *Adam: A Method for Stochastic Optimization.* https://doi.org/10.48550/arXiv.1412.6980

Edwards, J.H., Yelong, S., Philip, W., Zeyuan, A., Yuanzhi, L., Shean, W., Lu, W., Weizhu, C. (2021). *LoRA: Low-Rank Adaptation of Large Language Models. Microsoft Corporation.* https://doi.org/10.48550/arXiv.2106.09685

Tim, D., Artidoro, P., Ari, H., Luke, Z. (2023). QLoRA: Efficient Finetuning of Quantized LLMs. University of Washington. https://doi.org/10.48550/arXiv.2305.14314

Jianguo, Z., Kun, Q., Zhiwei, L., Shelby, H., Rui, M. Ye, L., Zhou, Y., Huan, W., Silvio, S., Caiming, X. (2023). *DialogStudio: Towards Richest and Most Diverse Unified Dataset Collection for Conversational AI.*

Gui, F., Chulaka, G., Benjamin, S., Sachindra, J., David, K., Rainit, A. (2021). TWEETSUMM - A *Dalog Summarization Dataset for Customer Service.* Association for Computational Linguistics 245--260, https://aclanthology.org/2021.findings-emnlp.24

Hugo, T., Louis, M., Kevin, S., Peter, A., Amjad, A., Yasmine, B., Nikolay, B., Soumya, B., Prajjwal, B., Shruti, B., Dan, B., Lukas, B., Cristian, C.F., Moya, C., Guillem, C., David, E., Jude, F., Jeremy, F., Wenyin, F., Brian, F. (2023). *Llama 2: Open Foundations and Fine-Tuned Chat Models*. https://huggingface.co/meta-llama/Llama-2-7b-chat-hf/blob/main/README.md

Tiyani, Z., Varsha, K., Felix, W., Kilian, Q.W., Yoav, A. (2020). *BERTScore: Evaluating Text Generation.* https://doi.org/10.48550/arXiv.1904.09675

TiyaniPatrick, L., Ethan. P, Alekssandra, P., Fabio, P., Vladimir, K., Naman, G., Heinrich, K., Mike, L., Wen-tau, Y., Tim, R., Sebastian, R., Douwe, K. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP tasks. https://doi.org/10.48550/arXiv.2005.11401

Maxime, L., Younes, B. (2023). Fine-tune Llama2 in Google Colab. https://github.com/mlabonne/llm-course/blob/main/Fine_tune_Llama_2_in_Google_Colab.ipynb