Term Project for CS535 - Big Data
Automated Text Summarization

Tanjim Bin Faruk, Zarin Tasnim Promi & Tony Kappen

April 30, 2023

1 Introduction

1.1 Problem Formulation

The problem that this project solved is that of extracting summaries from large blocks of text. The process of extracting meaningful information from a newspaper or journal article is currently a time-consuming and manual task. Our project generates a simple summary of a given body of text while preserving the original context which can then be easily understood by human consumers.

1.2 Background

We live in a world where we are constantly overloaded with vast amounts of information. Navigating this information maze and extracting the necessary parts to digest is a challenging task. An Automated Text Summarizer (ATS) can help in this regard by quickly and efficiently extracting the most important bits of information. This will reduce the need for manual effort and will save time. Another point where ATS can play a crucial role is in helping people with disabilities by providing accessible content. It can also prove to be useful for overcoming language barriers. Furthermore, even though our world is becoming more globally connected, there is a growing need to cater to local demands by providing region-specific content. ATS can fill this gap with the help of bilingual or cross-lingual summarization.

1.3 Why is it an interesting problem from a Big Data perspective?

This is an interesting problem to solve from a Big Data standpoint as high-performing models need to be trained on large datasets, which are time-consuming operations. A distributed training architecture facilitates the effective distribution of training workloads, allowing for a faster-trained model.

1.4 Potential Users

Potential users of an ATS include, but are not limited to:

- **Journalists** can create concise summaries to quickly update their readers in real-time.
- **Researchers** have to read lots of papers on a regular basis. They can decide whether to dive deep into a paper by going through the summary of the paper generated by an ATS.
- ATS can help **Students** to help prepare compact notes that they can refer to before the exams.
- **Lawyers** may benefit from ATS by obtaining summaries of legal contracts or documents.
- In the era of short-form content, **Content Creators** can utilize an ATS to create content for meaningful engagement.

2 Methodology

2.1 Dataset

The dataset that we used to train the ATS model is the CNN DailyMail dataset [6].

2.1.1 Format

This dataset consists of three columns:

1. An `article` column containing the entire article.
2. A `highlights` column, containing the human-generated summary of the article.
3. A `id` column generated by hashing the article itself to uniquely identify each row.

This dataset is suitable for training a text summarizer model because it provides a diverse variety of article topics and a diverse array of human-generated summaries. The summaries frequently vary in length and detail, which forces the model to learn various summarization techniques and reduces the possibility of overfitting.

2.1.2 Size

This dataset consists of roughly $300,000$ rows. The train set is $287,113$ rows, the validation set is $13,368$ and finally, the test set is $11,490$ rows.

2.1.3 Source

This dataset was originally compiled as part of this paper [2] by researchers in collaboration between Stanford University and Google Brain.

Although the authors of this dataset have made it publicly on GitHub [7], we opted to utilize the preloaded dataset object provided by *Hugging Face* [8] through its python *datasets* library. Using the preloaded dataset from Hugging Face offered several advantages:

- Hugging Face provides PyTorch binders that enable us to easily load the dataset into Pytorch. This allowed us to avoid writing a custom data loader.
- Dataset library automatically caches the processed dataset on disk, enabling us to reuse it across multiple experiments.
- Dataset works seamlessly with `transformers` library, making it easier to load, preprocess and fine-tune pre-trained models.

2.2 Data Preprocessing

In the CNN Dailymail dataset, the input articles are of various character and word lengths and contain a diverse array of special characters, punctuation, and contractions. Therefore, a series of preprocessing steps were taken to normalize the dataset over a common character space. These steps were applied uniformly to the input `article` as well as its corresponding `highlights`. We also utilized the popular `NLTK` library for this step. The preprocessing steps are described below:

1. The first preprocessing step that was taken was to replace all contractions with their full form representation. For example, the phrase *I'll* was replaced with *I will*.
2. Secondly, characters such as parentheses, double quotes, as well as any other punctuation marks were removed. Any words that represented a URL or were HTML tags were also removed.
3. As a final step, very short words (defined as less than 3 characters in length) as well as stopwords (from the `NLTK` library) were removed. Figure 1 shows the preprocessing steps that were applied to the dataset.

2.3 Tokenizer

In natural language processing, tokenization is a step that needs to be taken to convert the input sequences of text into smaller chunks. These individual chunks represent the resolution that the model will "see" and so the scale at which this decision is made has a huge impact on the performance of the model.

```python
def clean_text(text, should_remove_stopwords=True, should_remove_very_short_words=True):
    lower_text = lambda text: text.lower()
    split_text = lambda text: text.split()
    expand_words = lambda words: [contractions[word] if word in contractions else word for word in words]
    join_words = lambda words: ' '.join(words)
    remove_parentheses = lambda text: re.sub(r'\([^)]*\)', '', text)
    remove_double_quotes = lambda text: re.sub('"', '', text)
    remove_URL = lambda text: re.sub(r'https?:\/\/.*[\r\n]*', '', text, flags=re.MULTILINE)
    remove_HTML = lambda text: re.sub(r'<.*?>', ' ', text)
    remove_special_characters = lambda text:  re.sub(r'[_"\-;%()|+&=*%.,!?:#$@\[\]/]', ' ', text)
    remove_apostrophes = lambda text: re.sub(r'\'', ' ', text)
    remove_non_alphabetical_characters = lambda text: re.sub(r'[^a-zA-Z]', ' ', text)

    compose = lambda *F: reduce(lambda f, g: lambda x: g(f(x)), F)

    composed_function = compose(
        lower_text,
        remove_parentheses,
        remove_double_quotes,
        remove_URL,
        remove_HTML,
        remove_special_characters,
        remove_apostrophes,
        remove_non_alphabetical_characters,
        split_text,
        expand_words,
    )

    words = composed_function(text)

    if should_remove_stopwords:
        stops = set(stopwords.words('english'))
        words = [word for word in words if word not in stops]

    if should_remove_very_short_words:
        words = [word for word in words if len(word) >= 3]

    return join_words(words)
```

Figure 1: Data Preprocessing

There are various schemes that can be followed when deciding how to split up a block of input text. The possible tokenization schemes include splitting the text into characters, words and finally sub-words.

For this project, we opted to utilize a pre-trained tokenizer. A pre-trained tokenizer is one that is trained on a particular dataset and builds its vocabulary or corpus, from that dataset. By using this understanding of the language corpus, the tokenizer learns what sequence of characters to consider as one token of meaning.

As part of this project, our team experimented with several of the tokenizers currently used in models available through the Hugging Face library. We experimented with 3 pre-trained tokenizers, each accompanied by its pre-trained model counterpart:

1. **sshleifer/distilbart-cnn-6-6**
2. **t5-small**
3. **bart-base-cnn**

Figure 2, shows an example sample output of one of the tokenizers that was utilized. The special character 'Ġ' is used to indicate the end of the preceding token. It can be seen that some words are split into sub-tokens, e.g. the word *tokenizer*. Based on its understanding of the vocabulary, the tokenizer is able to intelligently split words into their smallest unit of meaning. It can also be observed that the tokenizer split a word that was outside the vocabulary, '14eD', into sub-tokens until each separate token was found in the vocabulary.

```
from transformers import BartTokenizer

tokenizer = BartTokenizer.from_pretrained("sshleifer/distilbart-cnn-6-6")

tokenizer.tokenize("This is a sample text to test the tokenizer. 14eD")
```

```
['This',
 'Ġis',
 'Ġa',
 'Ġsample',
 'Ġtext',
 'Ġto',
 'Ġtest',
 'Ġthe',
 'Ġtoken',
 'izer',
 '.',
 'Ġ14',
 'e',
 'D']
```

Figure 2: Tokenizer Example Output

2.4 Encoder/Decoder

The job of the encoder layer in a natural language processing problem is to encode each token into a corresponding numerical value which is in turn passed to the model. The encoder layer is correspondingly matched with a decoding layer to convert the numerical output of the model into text.

There are several common encoding schemes that are available to use, the simplest among these is to simply assign each token a number based on its index in processing. If a token has been seen previously it is assigned the number that was assigned to it previously. If a brand new token is seen, then the index counter is incremented and the token is assigned that number. The problem with such encoding schemes is that they fail to capture the similarity of a word in relation to similar words.

For example, a word like *car* is contextually closer to another word like *road* than a word like *seagull*. With the index-based encoding schemes, there is no guarantee that contextually similar words will be numerically close.

```python
def process_data_to_model_inputs(batch):
    encoder_max_length = 512
    decoder_max_length = 128

    # tokenize the inputs and labels
    inputs = tokenizer(batch["article"], padding="max_length", truncation=True, max_length=encoder_max_length)
    outputs = tokenizer(batch["highlights"], padding="max_length", truncation=True, max_length=decoder_max_length)

    batch["input_ids"] = inputs.input_ids
    batch["attention_mask"] = inputs.attention_mask
    batch["decoder_input_ids"] = outputs.input_ids
    batch["decoder_attention_mask"] = outputs.attention_mask
    batch["labels"] = outputs.input_ids.copy()

    batch["labels"] = [[-100 if token == tokenizer.pad_token_id else token for token in labels] for labels in batch
    ["labels"]]

    return batch
```

Figure 3: Encoding and Decoding

Each of the different models that we experimented with, `t5-small`, `bart-base-cnn`, and `sshleifer/distilbart-cnn-6-6` used a similar technique to embed its tokenized input sequence. Each token is mapped to a high-dimensional vector using an embedding matrix, which is learned during training. BART varies slightly from T5 in that it also utilizes a fixed set of positional encodings that are added to the embeddings to indicate the position of each token in the input sequence. These positional encodings are based on a fixed set of functions, such as sine and cosine functions, which allows the model to distinguish between tokens based on their position in the sequence.

The decoding strategy is essentially the inverse of the embedding operation. The generated token is mapped to its numeric counterpart from the trained embedding matrix.

2.5 Deep Learning Models

2.5.1 Model Selection

Our team considered several model architectures. The model architectures that were considered are listed below, along with our rationale for selecting or excluding them:

- **Generative Adversarial Network**: This methodology, outlined in [5] seeks to train two models, a *generator* model, and a *discriminator* model. The generator network takes a full article as input and generates a summary as output. The discriminator model compares the summary output of the generator to the ground truth summary and attempts to distinguish them. The two models compete with each other until the generator is able to produce samples that successfully fool the discriminator.
- **BART**: The *Bidirectional and Auto-Regressive Transformer* model, first outlined in [3] uses a BERT-like, bidirectional encoding layer, and a GPT-like, left-to-right decoding layer. In essence, BART works by corrupting its input document and then training to map it back to its original document.

- **PEGASUS**: The *Pre-training with Extracted Gap-sentences for the Abstractive Summarization* model is an abstractive summarization model that uses a hybrid approach of extractive and abstractive techniques and a novel gap-sentence training structure to generate summaries.

- **T5**: *The Text-To-Text Transfer Transformer* model is a type of transformer model architecture that is similar to BART. The T5 model is unique in that it formulates all Natural Language Processing tasks in a text-to-text format, where both the input and output are sequences of text.

Ultimately, the team decided to utilize the two variants of the BART model: DistilBART and BART and the T5 Model. This decision was made due to their lighter size and simplicity of training as compared to a GAN. Although our initial choice included GAN, further exploration revealed that it is not a suitable choice for text summarization purposes. BART was preferred over Pegasus due to the slightly better performance it displayed when training with the smaller model sizes [10].

2.5.1.1 Model Architectures

Both BART and T5 use a transformer architecture consisting of a stack of encoders and decoders, but there are key differences in their design related to their size and their input processing.

The T5 model is a text-to-text transformer, which means it is trained to solve a variety of natural language tasks in a unified framework. It has a single encoder-decoder architecture, where the input is first processed by the encoder and then decoded by the decoder. The encoder and decoder consist of a stack of transformer layers. To support this unified format, T5 includes an additional pre-processing step where the input text is converted into a task-specific format, such as a question-answering format or a summarization format. The task is specified by the presence of a prefix on the input text.

On the other hand, BART is a combination of the transformer encoder and the transformer decoder. It is designed to solve natural language processing tasks that require a combination of both autoregressive and bidirectional processing. The BART architecture consists of a stack of transformer encoder layers that encode the input, followed by a stack of transformer decoder layers that decode the encoded input. Figure 4 outlines the BART architecture.

BART is trained by first corrupting a document and then optimizing the reconstruction loss - the cross entropy between the decoder's output and the original document. BART is designed to allow for any type of data corruption. At the point of maximum data corruption (all data is removed), the model becomes a simple generative language model.

As part of the training phase, several methods of data corruption are applied to the input document. These include token masking, token deletion, text infilling, document rotation, and sentence permutation. Token masking

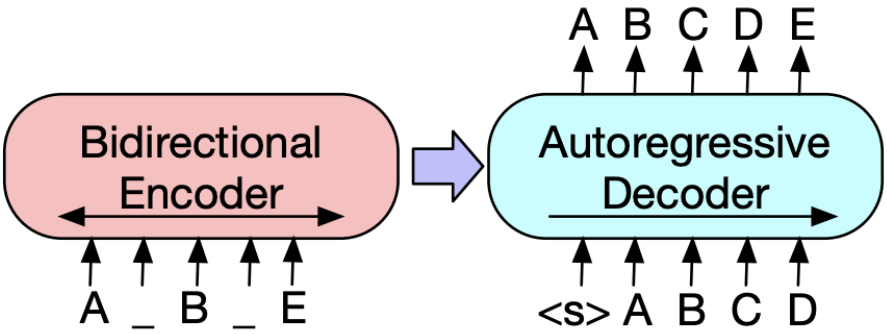

Figure 4: BART Architecture [3]

involves replacing random tokens with the `[MASK]` token. Token deletion randomly deletes tokens. The text infilling operation replaces a certain span of characters with the `[MASK]` token. The size of the span is chosen by a Poisson distribution. Document rotation involves choosing a random point in the text and then rotating the document such that this point becomes the starting point. Finally, sentence permutation involves dividing the document into sentences based on full stops and then randomly shuffling the order of these documents.

2.5.1.1.1 Cost Function

The choice of cost function has a huge impact on model performance. It is a numerical representation of how far the model's current output is from its desired output. By attempting to minimize this function, the model is trained to behave in the desired way.

In the context of the text summarization problem, the variants of the BART model used in this project utilized the masked language model loss function [1]. In this function, a sample of the tokens is replaced with the `[MASK]` token. Then the discrepancy between the model's prediction of what the `[MASK]` token should be and what it actually is, constitutes the loss. More precisely, the loss function is represented as:

$$\text{Loss} = -\log(P(x \mid input))$$

Where $P(x \mid input)$ is the probability of predicting the correct masked token given the input tokens.

The T5 model, like the BART model, utilizes a variant of cross-entropy loss as its loss function.

2.6 Framework

We chose PyTorch as our deep learning framework because of its simplicity and it has a shallow learning curve compared to TensorFlow. PyTorch has a similar syntax as NumPy and it also integrates nicely in our CS cluster. The team members had prior experience with PyTorch and it helped us to quickly get up to speed.

```
from transformers import T5Tokenizer, T5ForConditionalGeneration
tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = T5ForConditionalGeneration.from_pretrained("t5-small")
```

Figure 5: Pre-trained Model and Tokenizer from transformers library

Other than PyTorch, we relied heavily on Hugging Face's `transformers` library for loading the pre-trained model and tokenizer and training. Figure 5 shows an example of loading the T5 model and tokenizer using the `transformers` library.

```
training_args = TrainingArguments(
    output_dir="./updated_squad_fine_tuned_model",
    evaluation_strategy="epoch",
    learning_rate=5.6e-05,
    lr_scheduler_type="linear",
    warmup_ratio=0.1,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=2,
    weight_decay=0.01,
    local_rank=local_rank,
    fp16=True,
    remove_unused_columns=False,
    save_total_limit=3,
    save_strategy="epoch"
)

data_collator = DefaultDataCollator()

train_dataset = tokenized_cnndm["train"].select(range(15000))
train_sampler = DistributedSampler(train_dataset, num_replicas=int(os.environ['WORLD_SIZE']), rank=global_rank)
train_dataloader = DataLoader(train_dataset, batch_size=8, sampler=train_sampler)

eval_dataset = tokenized_cnndm["validation"].select(range(5000))
eval_sampler = DistributedSampler(eval_dataset, num_replicas=int(os.environ['WORLD_SIZE']), rank=global_rank)
eval_dataloader = DataLoader(eval_dataset, batch_size=8, sampler=eval_sampler)

trainer = CustomTrainer(
    model=model,
    args=training_args,
    train_dataloader=train_dataloader,
    eval_dataloader=eval_dataloader,
    train_dataset=tokenized_cnndm["train"].select(range(15000)),
    eval_dataset=tokenized_cnndm["validation"].select(range(5000)),
    tokenizer=tokenizer,
    data_collator=data_collator
)

trainer.train()
```

Figure 6: Training using transformers library

Figure 6 shows the code snippet for distributed training. Rather than writing a custom loop for the training process, the `TrainingArguments` and `Trainer` provides an easy-to-use interface for specifying the training parameters and they abstract away all the training-related logic. Some of the major parameters provided by the `TrainingArguments` class are:

- Learning Rate
- Batch Size
- Epochs
- Evaluation Strategy
- Model Saving Strategy
- Weight Decay

In addition to training arguments, the `Trainer` class takes in the training dataset, evaluation dataset, train data loader, evaluation data loader, tokenizer, and data collator (for padding and truncation). Finally, the training process initiates by calling the `trainer.train()` function.

2.7 Distributed Training

The various components concepts for our distributed training are described below:

```
import torch.distributed as dist

def setup_distributed_environment():
    dist.init_process_group(backend='nccl')

    torch.manual_seed(42)
```

Figure 7: Distributed Environment Setup

- `torch.distributed` is a distributed training package available directly in PyTorch. It provides various communication primitives and synchronization functions. `torch.distributed` allows for distributing the training process across multiple GPUs or multiple nodes. It supports several backend communication protocols, e.g. - `NCCL`, `Gloo`, and `MPI`, which are used to communicate between processes from multiple GPUs or multiple nodes.

`torch.distributed` not only provides various collective operations (e.g. - `allreduce`), but it also supports peer-to-peer communication, thus enabling parallelization of computation and communication. It acts as a building block on top of which other distributed training techniques can build, e.g. - `Distributed Data-Parallel (DDP)`.

- `torchrun` is a command line tool designed to simplify the initialization process for distributed training jobs. It can handle both multiple GPU and multiple node training. It provides the option for specifying the number of processes per GPU, number of nodes, node rank, rendezvous id, backend, and endpoint. It automatically creates the process (obviating the need for spawning the process in the code using `torch.multiprocessing`), sets up the required environment variables for distributed communication (doing away with specifying `MASTER ADDR` and `MASTER PORT` in the code) and manages the entire training lifecycle.

Figure 8: Torchrun command for distributed training

- `Distributed Data-Parallel (DDP)` replicates the deep learning model across multiple GPUs or multiple nodes and each device processes a unique subset of data. `DDP` essentially provides parallel computation by diving the data into smaller chunks and aggregating the gradients through a synchronization process to update the model parameters. `DDP` provides a high-level wrapper on top of the base model for distributed training. It is possible that the dataset can be of such a size that it is not possible to load it into a single machine. `DDP` can help in this situation by breaking the dataset down into smaller chunks that can be loaded into multiple machines.

- `Distributed Sampler` splits the data from the dataset across multiple processes or devices. It ensures that data does not get duplicated across multiple processes or nodes, improving the training process by preventing redundant computation.

- In the context of distributed training, `World Size` refers to the total number of different participating nodes.

- `Local rank` refers to the rank of a process within a single machine.

- `Global rank` uniquely identifies a process when all processes from all of the participating machines are taken into consideration.

- To ensure that each of the workers has the same initial condition and performs the same operations on its subset of data, we need to set the `Manual Seed`. Setting manual seed ensures consistency across workers and

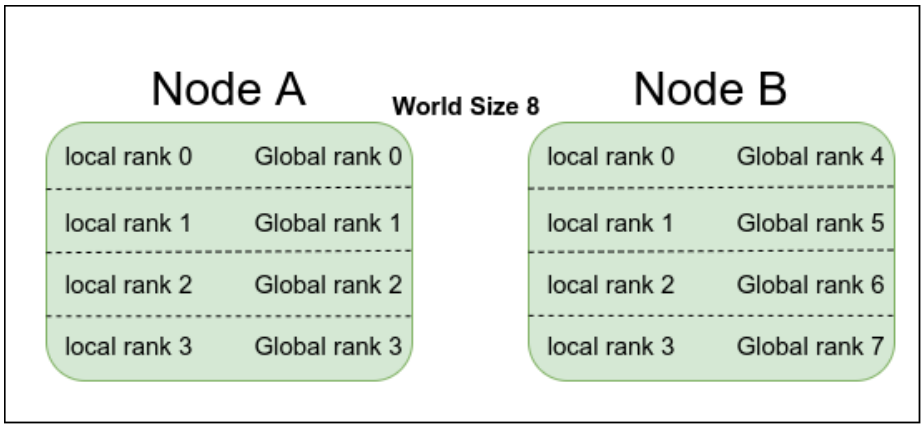

Figure 9: Local vs Global Rank [9]

provides deterministic behavior when the synchronization process takes place.

3 Results and Evaluation

3.1 Evaluation Criterion

The most commonly used set of metrics used to evaluate the performance of text summarization models are the ROUGE (Recall Oriented Understudy for Gisting Evaluation) metrics [4]. ROUGE scores are based on the overlap between the generated summary and the human-generated summary. ROUGE-1 measures the overlap of words, ROUGE-2 measures the overlap of pairs of consecutive words and ROUGE-L measures the longest common subsequence between generated and reference summaries. Finally ROUGE-LSum, like ROUGE-L, measures the longest common sequence but is tailored to more effectively account for similarities across multi-line blocks of text.

```
def generate_summary(batch, model):
  inputs = tokenizer(batch["article"], padding="max_length", truncation=True, max_length=512, return_tensors="pt")
  inputs = inputs.to(model.device) # Ensure that tensors are on the same device as the model
  summary_ids = model.generate(inputs.input_ids, num_beams=4, max_length=128, early_stopping=True)
  batch["predicted_highlights"] = tokenizer.batch_decode(summary_ids, skip_special_tokens=True)
  return batch
```

Figure 10: Summary Generation

We felt ROGUE was a suitable choice of metric for text summarization purposes for the following reasons:

- ROGUE makes it easier to compare different models as it provides a numeric score.

- ROGUE has multiple variants, which can be utilized to capture different quality aspects of the produced summary.
- ROGUE is recall oriented, which means it evaluates the produced summary on the basis of what important words has it covered from the original text.
- It is widely adopted in the research community.

3.2 Experiment Results

Model	ROUGE-1	ROUGE-2	ROUGE-L	ROUGE-LSum
sshleifer/distilbart-cnn-6-6	0.03955	0.00039	**0.03487**	0.03808
t5-small	0.01254	**0.00077**	0.012057	0.01243
bart-base-cnn	**0.04336**	0.00044	0.034717	**0.03974**

Table 1: Various ROUGE scores between tested models

The common experiment setup for each of the models is listed below:

- evaluation strategy: epoch
- learning rate: 5.6×10^{-5}
- lr scheduler type: linear
- warmup ratio: 0.1
- per device train batch size: 8
- per device eval batch size: 8
- train epochs: 2
- weight decay: 0.01
- fp16: True
- save strategy: epoch

Only the t5-small model was trained with a batch size of 16 while other parameters for t5 remained the same. t5 allowed for larger batch size and less training time compared to DistilBART and BART.

3.3 Result Analysis

As a higher ROGUE score indicates better model performance, our results were not expected. We believe there are three major reasons behind the overall poor performance of all three models:

- Due to huge training time, we limited the training data to 15000 examples which are only 5% of the total training data. Insufficient training data could have led to poor performance.
- Due to memory constraints, we opted for lighter models (`bert-base`: $110M$ parameters, `t5-small`: $60M$ parameters, and `distilbert-cnn-6-6`: $66M$ parameters) which could have contributed to subpar performance. Using a larger model could have significantly improved results.

4 Challenges Faced

1. **Setting up the Distributed Environment**: We faced major challenges in setting up the distributed environment and running the experiments in a distributed fashion. Since we previously had no experience with distributed training, we had a hard time configuring the setup. We initially decided to use Apache Spark. But this proved quite difficult as it required integration between PyTorch and Spark. So we switched to using `torch.distributed` package. But even then we struggled to run the experiment as we mistakenly ended up implementing code for multiple CPUs in a single machine, instead of multiple GPUs with multiple nodes. It took some time to figure out different components such as Distributed Data-Parallel (DDP), Data Sampler, Backend communication, etc. After some trial and error, we were finally able to run the experiment in a distributed manner in multiple nodes.
2. **Cuda Memory Issues**: We used the CS Lab 120 machines for running our experiments, as did most of our classmates. As a result, we often found the lab machines running out of GPU memory and hence we could not use them to run our training experiments. Moreover, the models we were initially planning to use such as `t5-base` or `t5-large` take up a large amount of disk space. For example, `t5-large` takes roughly 2.5 GB of disk space. As most of the lab machines had either a 10 GB or 12 GB GPU - loading both the model and the dataset exhausted the GPU memory. Furthermore, we found the required training time to be almost 6/7 hours. If someone else was using the machine during this time, the `NCCL` API would throw errors. That's why we opted to run our experiments late at night when lab machines were less used.
3. **Huge Training Time**: The size of our training dataset is almost 1.2 GB [6]. As a result, it was taking a significant amount of time to tokenize and train the model. If anything went wrong in the middle of the training

process, we would have to start the whole process from the start. This provided quite a challenge as training was taking almost 7 hours to complete and we did not have enough time to spare. So, we decided to run our experiments on a small subset of the data.

5 Future Work

- **Experimentation using other Models:** We were not able to run the experiment using larger and more complex models such `pegasus-large`, `t5-large`, due to CUDA memory issues and huge training time. In the future, if we can manage enough memory and a number of machines to run experimentation, we will extend our experiments to use these complex models and compare results.

6 Contribution

A detailed list of all the tasks completed by the team members is described below. The entire project span across roughly four weeks. We divided the project task milestones as below:

- Week 1 (03/25 - 03/31):
  - **Tanjim**: Understood the concepts required for distributed environment setup.
  - **Zarin**: Explored the CNN DailyMail dataset.
  - **Tony**: Looked into data preprocessing
- Week 2 (04/01 - 04/07):
  - **Tanjim**: Wrote some example code to understand the distributed environment setup using `torch.distributed` for multiple GPUs.
  - **Zarin**: Worked on a prototype using the CNN DailyMail dataset. Looked into model selection.
  - **Tony**: Completed data preprocessing.
- Week 3 (04/08 - 04/14):
  - **Tanjim**: Completed the distributed environment setup. Switch to using `torchrun` for multiple nodes.
  - **Zarin**: Finalized model selection. Started looking into computing metrics.
  - **Tony**: Understood the concepts required for encoder and decoder input and output

- Week 4 (04/15 - 04/21):
  - **Tanjim**: Debugged various errors related to distributed environment setup
  - **Zarin**: Fine-tuned the model hyperparameters for training. Computed ROGUE metric by comparing generated summaries with original ones.
  - **Tony**: Debugged various errors related to distributed environment setup
- Week 5 (04/22 - 04/29):
  - **Tanjim**: Ran experiments and evaluated model performance. Finalized report writing.
  - **Zarin**: Ran experiments and evaluated model performance. Finalized report writing.
  - **Tony**: Ran experiments and evaluated model performance. Finalized report writing.

References

[1] Hugging Face. *Hugging Face Transformers: BART model implementation*. `https://github.com/huggingface/transformers/blob/main/src/transformers/models/bart/modeling_bart.py`. Accessed: April 29, 2023. 2021.

[2] Karl Moritz Hermann et al. "Teaching Machines to Read and Comprehend". In: *NIPS*. 2015, pp. 1693–1701. URL: `http://papers.nips.cc/paper/5945-teaching-machines-to-read-and-comprehend`.

[3] Mike Lewis et al. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension". In: *CoRR* abs/1910.13461 (2019). arXiv: `1910.13461`. URL: `http://arxiv.org/abs/1910.13461`.

[4] Chin-Yew Lin. "ROUGE: A package for automatic evaluation of summaries". In: *Text Summarization Branches Out*. Association for Computational Linguistics. 2004, pp. 74–81.

[5] Linqing Liu et al. "Generative Adversarial Network for Abstractive Text Summarization". In: *CoRR* abs/1711.09357 (2017). arXiv: `1711.09357`. URL: `http://arxiv.org/abs/1711.09357`.

[6] Ramesh Nallapati, Bing Xiang, and Bowen Zhou. "Sequence-to-Sequence RNNs for Text Summarization". In: *CoRR* abs/1602.06023 (2016). arXiv: `1602.06023`. URL: `http://arxiv.org/abs/1602.06023`.

[7] Abi See. *cnn-dailymail*. `https://github.com/abisee/cnn-dailymail`. 2017.

[8] Abigail See. *cnn-dailymail*. https://huggingface.co/datasets/cnn_dailymail. 2022.

[9] Yunchao Yang. *Understanding Attention Mechanism in Neural Networks*. https://github.com/YunchaoYang/Blogs/issues/3. 2019.

[10] Jingqing Zhang et al. "PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization". In: *CoRR* abs/1912.08777 (2019). arXiv: 1912.08777. URL: http://arxiv.org/abs/1912.08777.