Individual Group Report

1. **Introduction**
2. **Description of individual work**
3. **Detailed Description of your work**
4. **Results**
5. **Summary and conclusions**
6. **Percentage of code copied from the internet.**
7. **References**

Introduction

Legal documents are very tough to interpret as they are very long and have a lot of important information. Humans find it quite challenging to quickly go through legal documents and understand key aspects without missing vital information. Most legal documents have a lot of information that is more relevant such as dates, deadlines, names of people etc. Attorneys, judges, lawyers, and others in the justice system are constantly surrounded by large amounts of the legal text, which can be difficult to manage across many cases. They face the problem of organizing briefs, judgements, and acts.

Due to the huge amount of legal information available on the internet, as well as other sources, the research community needs to do more extensive research on the area of legal text processing, which can help us make sense of the vast amount of available data. This information growth has compelled the requirement to develop systems that can help legal professionals, as well as ordinary citizens, get relevant legal information with very little effort [1].

Description of individual work

Text Summarization: Broadly speaking there are two types of summarizations: -

Extractive summarization: - Extractive summarization involves identifying important sections from text and generating them verbatim which produces a subset of sentences from the original text [2]. Extractive summarization techniques select and combine existing sentences from a text to create a summary.

Abstractive summarization: - Abstractive techniques generate new sentences while keeping the essence of the original text intact. Essentially, in the abstractive summarization the machine writes its own sentences [3]. Abstractive summarization uses natural language techniques to interpret and understand the important aspects of a text and generate a more "human" friendly summary.

Abstractive summarization is better generally as it leverages contextual learning to generate powerful summaries. These summaries are more human-readable, making them easier for agents to consume.

It may be tempting to use summarizations for all texts to get useful information from them and spend less time reading. However, for now, NLP summarization has been a successful use case in only a few areas. Text summarization works great if a text has a lot of raw facts and can be used to filter important information from them. The NLP models can summarize long documents and represent them in small simpler sentences. News, factsheets, and mailers fall under these categories [4].

However, for texts where each sentence builds up upon the previous, text summarization does not work that well. Research journals, and medical text are good examples of texts where summarization might not be very successful. Finally, if we take the case of summarizing fiction, summarization methods can work fine. However, it might miss the style and the tone of the text that the author tried to express. Hence, Text summarization is helpful only in a handful of use cases. The model used for

Detailed Description of your work in the project

Code

#Code/Summarization/train.py

```
from transformers import PegasusForConditionalGeneration, PegasusTokenizer,
Trainer, TrainingArguments

import torch

from datasets import load_dataset

from rouge import Rouge



class PegasusDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels['input_ids'][idx])  #
torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels['input_ids'])  # len(self.labels)

def show_samples(dataset, num_samples=3, seed=42):
    """
```

```
    Show num_samples random examples
    """
    sample = dataset['train'].shuffle(seed=seed).select(range(num_samples))
    for example in sample:
        print(f"\n'>> Article: {example['Text']}'")
        print(f"'>> Summary: {example['Summary']}'")


def prepare_data(model_name,
                 train_texts, train_labels,
                 val_texts, val_labels,
                 test_texts, test_labels):
    """
    Prepare input data for model fine-tuning
    """
    tokenizer = PegasusTokenizer.from_pretrained(model_name)

    prepare_val = False if val_texts is None or val_labels is None else True
    prepare_test = False if test_texts is None or test_labels is None else True

    def tokenize_data(texts, labels):
        encodings = tokenizer(texts, truncation=True, padding=True)
        decodings = tokenizer(labels, truncation=True, padding=True)
        dataset_tokenized = PegasusDataset(encodings, decodings)
        return dataset_tokenized

    train_dataset = tokenize_data(train_texts, train_labels)
```

```
    val_dataset = tokenize_data(val_texts, val_labels) if prepare_val else None
    test_dataset = tokenize_data(test_texts, test_labels) if prepare_test else
None

    return train_dataset, val_dataset, test_dataset, tokenizer


def prepare_fine_tuning(model_name, tokenizer, train_dataset, val_dataset,
freeze_encoder=True,
                        output_dir='./pegasus_indian_legal'):
    """
    Prepare configurations and base model for fine-tuning
    """
    torch_device = 'cuda' if torch.cuda.is_available() else 'cpu'
    model =
PegasusForConditionalGeneration.from_pretrained(model_name).to(torch_dev
ice)

    if freeze_encoder:
        for param in model.model.encoder.parameters():
            param.requires_grad = False

    if val_dataset is not None:
        training_args = TrainingArguments(
            output_dir=output_dir,  # output directory
            num_train_epochs=5,  # total number of training epochs
            per_device_train_batch_size=1,  # batch size per device during training,
can increase if memory allows
```

```
        per_device_eval_batch_size=1,  # batch size for evaluation, can increase
if memory allows
        save_steps=5,  # number of updates steps before checkpoint saves
        save_total_limit=5,  # limit the total amount of checkpoints and deletes
the older checkpoints
        evaluation_strategy='steps',  # evaluation strategy to adopt during
training
        eval_steps=5,  # number of update steps before evaluation
        warmup_steps=5,  # number of warmup steps for learning rate
scheduler
        weight_decay=0.01,  # strength of weight decay
        logging_dir='./logs',  # directory for storing logs
        logging_steps=5
    )

    trainer = Trainer(
        model=model,  # the instantiated 🤗 Transformers model to be trained
        args=training_args,  # training arguments, defined above
        train_dataset=train_dataset,  # training dataset
        eval_dataset=val_dataset,  # evaluation dataset
        tokenizer=tokenizer
    )

  else:
    training_args = TrainingArguments(
        output_dir=output_dir,  # output directory
        num_train_epochs=5,  # total number of training epochs
```

```
        per_device_train_batch_size=1,  # batch size per device during training, can increase if memory allows
        save_steps=5,  # number of updates steps before checkpoint saves
        save_total_limit=5,  # limit the total amount of checkpoints and deletes the older checkpoints
        warmup_steps=5,  # number of warmup steps for learning rate scheduler
        weight_decay=0.01,  # strength of weight decay
        logging_dir='./logs',  # directory for storing logs
        logging_steps=5,
    )

    trainer = Trainer(
        model=model,  # the instantiated 🤗 Transformers model to be trained
        args=training_args,  # training arguments, defined above
        train_dataset=train_dataset,  # training dataset
        tokenizer=tokenizer
    )

  return trainer

def calculate_rouge(hypothesis, reference):
  """
  Calculate ROUGE scores
  """
  rouge = Rouge()
  scores = rouge.get_scores(hypothesis, reference, avg=True)
```

```
    return scores


def evaluate_model(trainer, test_dataset, tokenizer):
    """
    Evaluate the fine-tuned model on the test dataset and print ROUGE scores
    """
    model = trainer.model
    test_dataloader = trainer.get_test_dataloader(test_dataset)
    rouge_scores = {'rouge-1': {'r': 0.0, 'p': 0.0, 'f': 0.0}, 'rouge-2': {'r': 0.0, 'p': 0.0,
'f': 0.0},
                    'rouge-l': {'r': 0.0, 'p': 0.0, 'f': 0.0}} #r-recall, p-precision, f-f1 score

    for batch in test_dataloader:
        inputs = tokenizer.batch_decode(batch['input_ids'],
skip_special_tokens=True)
        targets = tokenizer.batch_decode(batch['labels'],
skip_special_tokens=True)
        predictions = model.generate(batch['input_ids'])

        for pred, target in zip(predictions, targets):
            pred_text = tokenizer.decode(pred, skip_special_tokens=True)
            rouge_batch_scores = calculate_rouge(pred_text, target)

            # Accumulate ROUGE scores
            for rouge_key, rouge_score in rouge_batch_scores.items():
                rouge_scores[rouge_key]['r'] += rouge_score['r'] #Recall
                rouge_scores[rouge_key]['p'] += rouge_score['p'] #Precision
```

```
            rouge_scores[rouge_key]['f'] += rouge_score['f'] #F1-Score


    # Normalize ROUGE scores
    num_samples = len(test_dataset)
    for rouge_key in rouge_scores.keys():
        rouge_scores[rouge_key]['r'] /= num_samples
        rouge_scores[rouge_key]['p'] /= num_samples
        rouge_scores[rouge_key]['f'] /= num_samples


    # Print ROUGE scores
    print("ROUGE Scores:")
    print("ROUGE-1 (Recall):", rouge_scores['rouge-1']['r'])
    print("ROUGE-2 (Recall):", rouge_scores['rouge-2']['r'])
    print("ROUGE-L (Recall):", rouge_scores['rouge-l']['r'])
    print("ROUGE-1 (Precision):", rouge_scores['rouge-1']['p'])
    print("ROUGE-2 (Precision):", rouge_scores['rouge-2']['p'])
    print("ROUGE-L (Precision):", rouge_scores['rouge-l']['p'])
    print("ROUGE-1 (F1-Score):", rouge_scores['rouge-1']['f'])
    print("ROUGE-2 (F1-Score):", rouge_scores['rouge-2']['f'])
    print("ROUGE-L (F1-Score):", rouge_scores['rouge-l']['f'])


if __name__ == '__main__':
    # Use first 1000 docs as training data
```

```
dataset = load_dataset("ninadn/indian-legal")
show_samples(dataset)
dataset = dataset.filter(lambda x: x["Summary"] is not None) #Remove rows which have no summary
train_texts, train_labels = dataset['train']['Text'][:1000], dataset['train']['Summary'][:1000]
val_texts, val_labels = dataset['train']['Text'][1000:1250], dataset['train']['Summary'][1000:1250]
test_texts, test_labels = dataset['train']['Text'][1250:1500], dataset['train']['Summary'][1250:1500]

# use Pegasus model as base for fine-tuning
model_name = 'nsi319/legal-pegasus'
train_dataset, val_dataset, test_dataset, tokenizer = prepare_data(model_name, train_texts, train_labels, val_texts, val_labels, test_texts, test_labels)
trainer = prepare_fine_tuning(model_name, tokenizer, train_dataset, val_dataset)
trainer.train()

#Push model to hugging face hub
trainer.push_to_hub()

# Save model locally
trainer.save_model('pegasus_indian_legal')

# Evaluate the model on the test dataset
```

```
    evaluate_model(trainer, test_dataset, tokenizer)
#Code/streamlit.py
def extract_text_from_document(file):
    if file is not None:
        # Read the content of the file as bytes
        content_bytes = file.read()

        if content_bytes:
            # Decode the bytes into a string
            content = content_bytes.decode('utf-8')
            return content
        else:
            st.error("File is empty. Please choose a file with content.")
            return None
    else:
        return None
def generate_response_with_selected_model(model, tokenizer,
input_tokenized):
    summary_ids = model.generate(input_tokenized,
                                 num_beams=9,
                                 no_repeat_ngram_size=3,
                                 length_penalty=2.0,
                                 min_length=150,
                                 max_length=250,
                                 early_stopping=True)
    summary = [tokenizer.decode(g, skip_special_tokens=True,
clean_up_tokenization_spaces=False) for g in summary_ids][0]
```

```
    return summary
if model_choice == "Pegasus Legal":
        tokenizer = AutoTokenizer.from_pretrained("nsi319/legal-pegasus")
        model = AutoModelForSeq2SeqLM.from_pretrained("nsi319/legal-pegasus")
        input_tokenized = tokenizer.encode(document_text, return_tensors='pt', max_length=1024, truncation=True)
        summary = generate_response_with_selected_model(model, tokenizer, input_tokenized)
    elif model_choice == "Pegasus Indian Legal":
        tokenizer = AutoTokenizer.from_pretrained("akhilm97/pegasus_indian_legal")
        model = AutoModelForSeq2SeqLM.from_pretrained("akhilm97/pegasus_indian_legal")
        input_tokenized = tokenizer.encode(document_text, return_tensors='pt', max_length=1024,
                                           truncation=True)
        summary = generate_response_with_selected_model(model, tokenizer, input_tokenized)
```

Results

Pegasus

Rouge 1 Recall	Rouge 2 Recall	Rouge L Recall	Rouge1 Precision	Rouge2 Precision	RougeL Precision	Rouge 1F1-score	Rouge 2 F1-score	Rouge L F1-score
16.57	6.49	14.94	52.99	26.46	48.18	24.48	10.01	22.16

Table 1 Metrics for the final fine-tuned model

Model	Dataset	Rouge-1 (Precision)	Rouge-2 (Precision)	Rouge-L (Precision)
Pegasus (google/pegasus-large · Hugging Face)	CNN Daily Mail (cnn_dailymail · Datasets at Hugging Face)	45.68	14.56	20.07
Legal Pegasus (nsi319/legal-pegasus · Hugging Face)	US-Litigation releases (Litigation Releases | U.S. Securities and Exchange Commission)	62.97	28.42	33.22
Pegasus Indian Legal (Fine-tuned) (akhilm97/pegasus_indian_legal · Hugging Face)	Indian-Legal documents (ninadn/indian-legal · Hugging Face)	52.99	26.4	48.1

Table 2 ROUGE (Precision) scores of all the Summarization models

Based on the memory constraints and the size of the dataset used for text summarization of the base model, the base model Pegasus could not be directly used for fine-tuning as it was giving a CUDA out-of-memory error. Each checkpoint file in the Pegasus model was around 2.2GB and it has 568M parameters [6]. Therefore, I used the fine-tuned version of Pegasus (legal-pegasus) which was trained on the US litigation releases website as the base model for the checkpoint.

Summary and Conclusions

The fine-tuned PEGASUS also yielded fair summarization with a ROUGE-1 (Precision) score of 52.99% on Indian legal documents. The only limitation is that while inferencing this model on the test set the model was generating summaries where the last sentence was incomplete.

Percentage of code copied from the internet.

Some part of the code was copied from [5]. Around 60.4% of the code was used in train.py.

References

[1] AliguliyevRamiz M, FangChangjian, GalganiFilippo, RashediEsmat, TurtleHoward, AustinJohn Langshaw, BhattacharyaPaheli, Wikipedia, Press, ManiInderjeet, FarzindarAtefeh, KanapalaAmbedkar, AllahyariMehdi, GambhirMahak, NenkovaAni, LuhnHans Peter, EdmundsonH.P., … RushAlexander M. (2021, March 9). Summarization of legal documents:

Where are we now and the way forward. Computer Science Review. https://www.sciencedirect.com/science/article/abs/pii/S1574013721000289

[2] https://www.prodigaltech.com/blog/extractive-vs-abstractive-summarization-how-does-it-work
[3] https://iris.ai/technology/tech-deep-dive-abstractive-summarization/
[4] https://turbolab.in/types-of-text-summarization-extractive-and-abstractive-summarization-basics/
[5] https://gist.github.com/jiahao87/50cec29725824da7ff6dd9314b53c4b3