INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

BANGALORE

NATURAL LANGUAGE PROCESSING

AI 829

Story Summarization

(Final submission – Mandate 4)

Under the Guidance of:

Prof. Srinath Srinivasa

Submitted by:

Kartik Satish Gawande

MT2023045

kartik.gawande@iiitb.ac.in

Contents

1. Mandate 2 Recap:

I had done Syntactic analysis of the Xsum dataset. I did POS tagging using Deep learning approach (BERT). I found out dependency and constituency parsings.
Now in this Mandate the goal is to fine tune an Pretrained model to get abstractive summarization and compare results among various model performances. Achieving the final Objective of my Learning goal.

2. Semantic Analysis:

Semantic analysis in Natural Language Processing (NLP) is the process of understanding the meaning of text. This involves interpreting the meanings of words, phrases, and sentences within their specific contexts. Semantic analysis seeks to comprehend the intended message, relationships among different parts of the text, and the underlying concepts and entities. This is crucial for various NLP applications such as machine translation, summarization, question answering, and sentiment analysis, allowing computers to understand and respond to human language in a way that is meaningful.

2.1. NER

Named Entity Recognition (NER) is a key task in natural language processing (NLP) that involves identifying and classifying named entities in text into predefined categories. These entities can be proper nouns and other phrases that clearly identify items from the real world, such as people, organizations, locations, dates, quantities, monetary values, percentages, etc.

Key aspects of SpaCy's NER technique:

1. **Neural Network Model**:
   - SpaCy's NER system utilizes a type of neural network architecture called a **transition-based** model, similar in nature to what is used in dependency parsing. This model learns to make predictions based on the current state and a set of actions that can be applied to an input.
   - The model is a **feedforward deep neural network** with convolutional layers, specifically designed to work efficiently with the rest of the SpaCy pipeline.
2. **Embeddings**:
   - The neural network uses **word vectors** (also known as word embeddings) which are dense numerical representations of words based on their context. These embeddings capture semantic meanings and are crucial for effectively training the NER model.
   - Beyond word vectors, the model also utilizes **contextual features** from surrounding words to predict the entity type of each word in a sentence.
3. **Training**:
   - The NER model in SpaCy is trained on a labeled dataset where each entity in the text is annotated with its corresponding entity type (e.g., PERSON, ORGANIZATION).

- The training process involves learning to predict not just the labels but also where the boundaries of named entities start and end, which is crucial for accurately extracting entities from text.

```python
#entity recognition
import spacy

# Load SpaCy model
nlp = spacy.load('en_core_web_sm')

# Example text
text = "Apple Inc. is looking to buy a U.K. startup for $1 billion. Tim Cook announced this yesterday in San Francisco."

# Process the text
doc = nlp(text)

# Display the named entities
for ent in doc.ents:
    print(ent.text, ent.label_)
```

```
Apple Inc. ORG
U.K. GPE
$1 billion MONEY
Tim Cook PERSON
this yesterday DATE
San Francisco GPE
```

Figure 1: Code for NER using SpaCy showing the Entities recognized in the output.

2.2. Coreference Resolution

Coreference resolution is a critical task in natural language processing (NLP) that involves identifying which words in a text refer to the same entities. It focuses on resolving pronouns and other referring expressions (like nouns and proper names) to the entities they refer to, which could have been mentioned earlier in the text or are understood implicitly. This task helps in understanding the full context of discourse by determining the relationships among different parts of the text.

- **Coreferee:**

Coreferee is specifically designed to work with SpaCy 3.x and beyond.

1. **Integration with SpaCy**: Coreferee hooks directly into the SpaCy pipeline. This allows it to leverage SpaCy's tokenization, POS tagging, dependency parsing, and named entity recognition to inform its coreference resolution process.
2. **Model and Rule-Based System**: Coreferee uses a combination of model-based predictions and rule-based adjustments. It utilizes trained models where available but also includes rule-based strategies to handle coreferences, particularly in languages for which it does not have a trained model.

```
import spacy
import coreferee

nlp = spacy.load('en_core_web_md')
nlp.add_pipe('coreferee')
doc = nlp("Apple is looking at buying a startup. They are interested in its AI technology.")

print(doc._.coref_chains)
```

Figure 2: Code for Coreference Resolution using Coreferee

```
[0: [0], [8], 1: [6], [12]]
```

Figure 3: Output by Coreferee for the sentence "Apple is looking at buying a startup. They are interested in its AI technology." Showing that the zeroth word 'Apple' is being referred by the 8th word 'They'.

3. Fine tunning T5 pretrained Model

3.1. Why T5?

The T5 (Text-to-Text Transfer Transformer) model developed by Google is a versatile machine learning model designed primarily for natural language processing (NLP) tasks. The "small" variant of this model is a scaled-down version that offers a good balance between performance and computational efficiency. Due to it's small size, it's a little less compute intense to fine tune it.

The original model gave the following performance on Xsum dataset:

```
Original t5 Model ROUGE Scores:
rouge-1: {'r': 0.2134, 'p': 0.1207, 'f': 0.1513}
rouge-2: {'r': 0.0214, 'p': 0.01, 'f': 0.0133}
rouge-l: {'r': 0.1845, 'p': 0.1037, 'f': 0.1303}
```

```python
from transformers import pipeline
from rouge import Rouge
import torch

def evaluate_model(model, tokenizer, dataset, sample_size=50):

    # If there's a GPU available, move the model to the GPU

    if torch.cuda.is_available():
        print("GPU is available.")
        device = torch.device("cuda")
    else:
        print("GPU not available, using CPU.")
        device = torch.device("cpu")


    summarizer = pipeline("summarization", model=model, tokenizer=tokenizer, device=device)

    # Randomly select a sample of the dataset
    sampled_data = dataset["test"].shuffle(seed=42).select(range(sample_size))

    # Generate summaries
    generated_summaries = []
    for entry in sampled_data:
        summarized = summarizer(entry["document"], max_length=150, truncation=True, min_length=40)
        generated_summaries.append(summarized[0]['summary_text'])

    # Prepare references and candidates for ROUGE
    references = [entry["summary"] for entry in sampled_data]
    candidates = generated_summaries

    rouge = Rouge()
    scores = rouge.get_scores(candidates, references, avg=True)

    # Round each value in the scores to 4 decimal places
    rounded_scores = {key: {metric: round(value, 4) for metric, value in values.items()}
                      for key, values in scores.items()}

    return rounded_scores
```

Figure 4: Code used to evaluate model performance

3.2. Early Fine tunning attempts

At first I naively began fine tunning it with the default trainer by Hugging face. I arbitrarily chose reasonable Parameters for fine tunning. However the resultant model didn't perform well at all. In fact it performed worse than the original model.

```
def preprocess_function(examples):
    inputs = ["summarize: " + doc for doc in examples["document"]]
    model_inputs = tokenizer(inputs, max_length=512, truncation=True, padding="max_length")

    with tokenizer.as_target_tokenizer():
        labels = tokenizer(examples["summary"], max_length=128, truncation=True, padding="max_length")

    model_inputs["labels"] = labels["input_ids"]
    return model_inputs

tokenized_datasets = dataset.map(preprocess_function, batched=True)
```

Figure 5: function used to tokenize the dataset with max token size 512 in inputs

```
model_to_be_fine_tuned = T5ForConditionalGeneration.from_pretrained("t5-small")

from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01
)

trainer = Trainer(
    model=model_to_be_fine_tuned,
    args=training_args,
    train_dataset=tokenized_datasets["train"].select(range(1000)),
    eval_dataset=tokenized_datasets["validation"].select(range(1000)),
    tokenizer=tokenizer_before
)

trainer.train()
```

Figure 6: First fine tunning attempt code with 1000 training examples

[189/189 02:51, Epoch 3/3]

Epoch	Training Loss	Validation Loss
1	No log	3.555805
2	No log	0.859342
3	No log	0.827318

```
TrainOutput(global_step=189, training_loss=4.331529405381945, metrics={'train_runtime': 171.8493, 'train_samples_per_second': 17.457, 'train_steps_per_second': 1.1, 'total_flos':
4060254044160000.0, 'train_loss': 4.331529405381945, 'epoch': 3.0})
```

Figure 7: Output while first fine tunning attempt. Loss was decreasing with each epoch.

```
Fine-tuned Model ROUGE Scores: {'rouge-1': {'r': 0.22063405265145067, 'p': 0.11705052019213884, 'f': 0.15085126782756195},
                                'rouge-2': {'r': 0.021389659535311707, 'p': 0.009645475063951777, 'f': 0.012978985934649568},
                                'rouge-l': {'r': 0.18195206454936566, 'p': 0.09598073282756586, 'f': 0.12376672617136635}}
```

Figure 8: First attempt fine tuned T5 model ROUGE Performance

As is clearly seen above the ROUGE performance metric was even worse than the original model performance.
The reason for this poor performance was the less number of training examples I used. This gave me an idea of the magnitude of the problem. This is clearly gonna require a lot more compute than I had anticipated.

In my second attempt I increased the number of training examples and re-run the code.

```python
model_to_be_fine_tuned = T5ForConditionalGeneration.from_pretrained("t5-small")

from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
)

trainer = Trainer(
    model=model_to_be_fine_tuned,
    args=training_args,
    train_dataset=tokenized_datasets["train"].select(range(10000)),
    eval_dataset=tokenized_datasets["validation"].select(range(10000)),
    tokenizer=tokenizer_before
)

trainer.train()
```

Figure 9: Code of second fine tunning attempt with 10,000 training examples

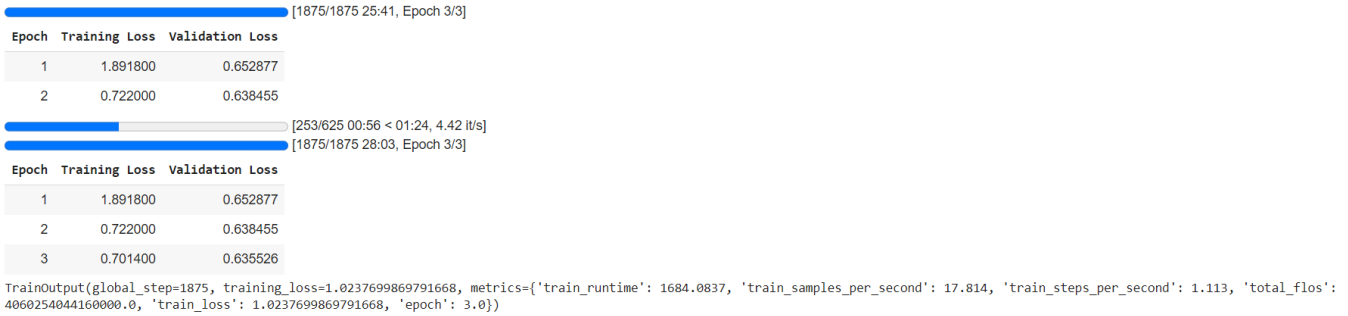

Figure 10: Output of the second fine tunning attempt code. Took more time than the first attempt.

```
Fine-tuned Model ROUGE Scores: {'rouge-1': {'r': 0.22063405265145067, 'p': 0.11705052019213884, 'f': 0.15085126782756195},
                                'rouge-2': {'r': 0.021389659535311707, 'p': 0.009645475063951777, 'f': 0.012978985934649568},
                                'rouge-l': {'r': 0.18195206454936566, 'p': 0.09598073282756586, 'f': 0.12376672617136635}}
```

Figure 11: ROUGE performance of the second fine tunning attempt model. Again came out to be worse than the original.

The ROUGE performance still came out to be poor even for the second attempt.
The reason for this bad performance even after having increased the number of training examples was that I failed to understand that T5 is a seq2seq Model and it expects the input to be the same size at all times. So I had to first do padding and truncation as necessary.
This understanding came from the work of a hugging face user named 'Daviadi Auzan Fadhlillah'[2].

3.3. About best attempt

So next I used DataCollator by Hugging face for seq2seq models. This helped in padding and truncating the input and labels as was required by T5.
Also I Increased training even more to 40k. Significantly more than all previous attempts.

```python
from transformers import DataCollatorForSeq2Seq

data_collator = DataCollatorForSeq2Seq(tokenizer=tokenizer, model="t5-small")
```

Figure 12: Code for DataCollator that changes the input and labels in the lenght expected by T5

```python
prefix = "summarize: "

def preprocess_function(examples):
    inputs = [prefix + doc for doc in examples["document"]]
    model_inputs = tokenizer(inputs, max_length=1024, truncation=True)
    labels = tokenizer(text_target=examples["summary"], max_length=128, truncation=True)
    model_inputs["labels"] = labels["input_ids"]
    return model_inputs
```

Figure 13: function used to tokenize dataset with increased token size to 1028 for inputs

```python
from datasets import load_dataset
train_data = load_dataset("xsum", split="train[:20%]")
test_data = load_dataset("xsum", split="test[:20%]")
```

Figure 14: Used 20% of the Xsum dataset. Significantly increaasing the training examples for fine tunning (almost 40k)

Next improvement over the last attempt was to use a custom compute metric for each epoch. This also removes the padding characters put in by the data_collator for an accurate computation of ROUGE score.

```python
import numpy as np


def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    decoded_preds = tokenizer.batch_decode(predictions, skip_special_tokens=True)
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)

    result = rouge.compute(predictions=decoded_preds, references=decoded_labels, use_stemmer=True)

    prediction_lens = [np.count_nonzero(pred != tokenizer.pad_token_id) for pred in predictions]
    result["gen_len"] = np.mean(prediction_lens)

    return {k: round(v, 4) for k, v in result.items()}
```

Figure 15: Custom compute_metric code for fine tunning process epochs.

To use DataCollatorForSeq2Seq hugging face recommends Seq2SeqTrainer and Seq2SeqTrainingArguments methods to fine tune Seq2Seq models like T5:

```
training_args = Seq2SeqTrainingArguments(
    output_dir="/content/drive/MyDrive/MyModel",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    weight_decay=0.01,
    save_total_limit=3,
    num_train_epochs=4,
    predict_with_generate=True,
    fp16=True,
    push_to_hub=False,
)

trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_train,
    eval_dataset=tokenized_test,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)

trainer.train()
```

Figure 16: Improved Fine tunning code using data_collator and custom compute_metrics this also conviniently is set to store the model on my Google collab Mounted google drive as mentioned in "output_dir" parameter

```
#mount google drive to save the fine tuned model to
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Figure 17: Code to mount Google drive to Google collab to store resultant Fine tunned model

```
from transformers import T5Tokenizer, T5ForConditionalGeneration

path_to_model = '/content/drive/MyDrive/MyModel'

tokenizer = T5Tokenizer.from_pretrained(path_to_model)
model_fine_tuned = T5ForConditionalGeneration.from_pretrained(path_to_model)
```

Figure 18: Code to load the saved Fine tuned model on my Google Drive mounted to Google Collab

All the above stated changes in my fine tunning approach finally resulted in a better fine tunned model than the original. The rouge score was tested on the data that the model never saw in the training process.
The model took 6 hours to get fine tunned over almost 40k training examples from Xsum dataset.

[24560/24560 5:58:17, Epoch 4/4]

Epoch	Training Loss	Validation Loss	Rouge1	Rouge2	Rougel	Rougelsum	Gen Len
1	2.178900	2.262195	0.250100	0.077700	0.196000	0.196000	18.862400
2	2.149500	2.243245	0.254600	0.079300	0.199300	0.199300	18.868800
3	2.128700	2.236404	0.255500	0.079500	0.199400	0.199400	18.874700
4	2.148500	2.232910	0.256600	0.080400	0.200600	0.200700	18.890300

Figure 19: Output during fine tunning. Fine tunning took 6 hours (5hrs 58minutes) in total.

```
Fine-tuned Model ROUGE Scores:
rouge-1: {'r': 0.3132, 'p': 0.2035, 'f': 0.2408}
rouge-2: {'r': 0.0708, 'p': 0.0398, 'f': 0.0497}
rouge-l: {'r': 0.2577, 'p': 0.166, 'f': 0.1971}
```

Figure 20: ROUGE performance by the fine tuned model finally came out to be better than the Original T5 model

4. Using different pretrained Models

4.1. BART

```
from transformers import BartTokenizer, BartForConditionalGeneration
```

```
tokenizer_bart = BartTokenizer.from_pretrained('facebook/bart-large-cnn')
model_bart = BartForConditionalGeneration.from_pretrained('facebook/bart-large-cnn')
```

```
# Evaluate the BART model
model_bart.eval()
tokenizer_bart.model_max_length = 1024
bart_scores = evaluate_model(model_bart, tokenizer_bart, dataset)
```

Figure 21: Tested BART for ROUGE score on Xsum dataset

```
print("BART Model ROUGE Scores:")
for metric, score in bart_scores.items():
    print(f"{metric}: {score}")
```

```
BART Model ROUGE Scores:
rouge-1: {'r': 0.2496, 'p': 0.1398, 'f': 0.1766}
rouge-2: {'r': 0.0353, 'p': 0.0168, 'f': 0.0223}
rouge-l: {'r': 0.2234, 'p': 0.1245, 'f': 0.1575}
```

Figure 22: ROUGE performance by BART on Xsum. Found to be better than Original T5 model and poor than our fine tuned T5 model

4.2. BlenderBot by Facebook (Meta)

```
from transformers import BlenderbotTokenizer, BlenderbotForConditionalGeneration

model_name = 'facebook/blenderbot-400M-distill'
tokenizer_blenderbot = BlenderbotTokenizer.from_pretrained(model_name)
model_blenderbot = BlenderbotForConditionalGeneration.from_pretrained(model_name)

# Evaluate the BART model
model_blenderbot.eval()
blenderbot_scores = evaluate_model(model_blenderbot, tokenizer_blenderbot, dataset)
```

Figure 23:Tested BlenderBot for ROUGE score on Xsum dataset

```
print("BlenderBot Model ROUGE Scores:")
for metric, score in blenderbot_scores.items():
    print(f"{metric}: {score}")
```

```
BlenderBot Model ROUGE Scores:
rouge-1: {'r': 0.1155, 'p': 0.0929, 'f': 0.1016}
rouge-2: {'r': 0.0042, 'p': 0.0034, 'f': 0.0037}
rouge-l: {'r': 0.1021, 'p': 0.0819, 'f': 0.0896}
```

Figure 24: ROUGE performance by BlenderBot on Xsum. Found to be even worse than original T5. Worst of all models.

.

5. Comparing Performances

So in total I tested four models namely:

- T5 by Google
- T5 Model that I fine tunned
- BART
- BlenderBot by Facebook (Meta)

The Performances of these models are as follows:

```
Original t5 Model ROUGE Scores:
rouge-1: {'r': 0.2134, 'p': 0.1207, 'f': 0.1513}
rouge-2: {'r': 0.0214, 'p': 0.01, 'f': 0.0133}
rouge-l: {'r': 0.1845, 'p': 0.1037, 'f': 0.1303}
Fine-tuned Model ROUGE Scores:
rouge-1: {'r': 0.3132, 'p': 0.2035, 'f': 0.2408}
rouge-2: {'r': 0.0708, 'p': 0.0398, 'f': 0.0497}
rouge-l: {'r': 0.2577, 'p': 0.166, 'f': 0.1971}
BART Model ROUGE Scores:
rouge-1: {'r': 0.2496, 'p': 0.1398, 'f': 0.1766}
rouge-2: {'r': 0.0353, 'p': 0.0168, 'f': 0.0223}
rouge-l: {'r': 0.2234, 'p': 0.1245, 'f': 0.1575}
BlenderBot Model ROUGE Scores:
rouge-1: {'r': 0.1155, 'p': 0.0929, 'f': 0.1016}
rouge-2: {'r': 0.0042, 'p': 0.0034, 'f': 0.0037}
rouge-l: {'r': 0.1021, 'p': 0.0819, 'f': 0.0896}
```

Figure 25: Performances of all models.

My fine tunned model performed the best then best performance was by BART followed by original T5 model then BlenderBot did the worst.

6. Challenges faced

During fine tunning, I couldn't improve the performance of my fine tunned models. That was due to not taking into account the nature of Seq2Seq. Such models expects input and label of same lengths. This issue was solved by the use of data_collator.

Then at first I was very reluctant to increase number of training examples in the fine tunning process to save on training time. Finally for better results I had to use significantly more examples.

During the evaluation of BART, I couldn't get it to give outputs. This issue was solved by referring to Abhishek Gupta (MT2023046) and Kuldip Bhatale (MT2023087)'s NLP mandate Group[1]. I then came to understand that I had to increase the model_max_length parameter to 1028 for it to properly give output.

7. Collab Notebooks

Mandate 2
Mandate 3
Mandate 4 (This submission)

8. Future Scope

This entire project can be given a front end to make it easier for any layman person to observe responses by different Models.

Even better computational resources can be used to make the fine tunned model even better. I only used the free resources given by Google Collab.

9. References

[1] Abhishek Gupta (MT2023046) and Kuldip Bhatale (MT2023087)'s Group's mandate-3 Submission
[2] Daviadi Auzan Fadhlillah's Work on hugging face