INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

BANGALORE

NATURAL LANGUAGE PROCESSING

AI 829

Story Summarization

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 the necessary preprocessing of the Xsum dataset in mandate 2. Along with extractive summarization using PageRank Algorithm.
Now in this Mandate the goal is to Do syntactical analysis of the dataset.

2. Syntactic Analysis:

Syntactic analysis, also known as parsing, is a crucial process in natural language processing (NLP) that involves analyzing the syntax of a sentence to understand its structure and the relationships between words. The main goal is to determine how words in a sentence group together to form meaningful units like phrases and clauses, and how these units interact.

2.1. POS tagging:

Before syntactic analysis, each word in a sentence is typically tagged with its part of speech (noun, verb, adjective, etc.). This step is foundational because it helps to determine the grammatical structure of sentences.

- Deep learning Approach (BERT):
    - There are various approaches for POS tagging like Rule-based POS tagging, Stochastic POS tagging, Lexical based POS tagging, Machine learning based POS tagging, Deep learning based POS tagging or even Hybrid POS tagging.
    - However I have used the Deep learning based POS tagging in my approach.

```python
from transformers import AutoModelForTokenClassification, AutoTokenizer, pipeline

# Load the tokenizer and model using the token for authentication
model_name = "QCRI/bert-base-multilingual-cased-pos-english"
tokenizer = AutoTokenizer.from_pretrained(model_name, use_auth_token=os.environ['HF_TOKEN'])
model = AutoModelForTokenClassification.from_pretrained(model_name, use_auth_token=os.environ['HF_TOKEN'])

# Setup the pipeline
pos_pipeline = pipeline("ner", model=model, tokenizer=tokenizer)

def transformer_pos_tag(example):
    # Initialize a structure to hold the tagged results
    tagged_sentences = []

    for sentence in example['document']:
        # Apply POS tagging
        results = pos_pipeline(sentence)
        # Store the results
        tagged_sentences.append({'sentence': sentence, 'pos_tags': results})

    # Return a dictionary with the results
    return {'document': tagged_sentences}
```

Figure 1: Code for POS tagging using BERT

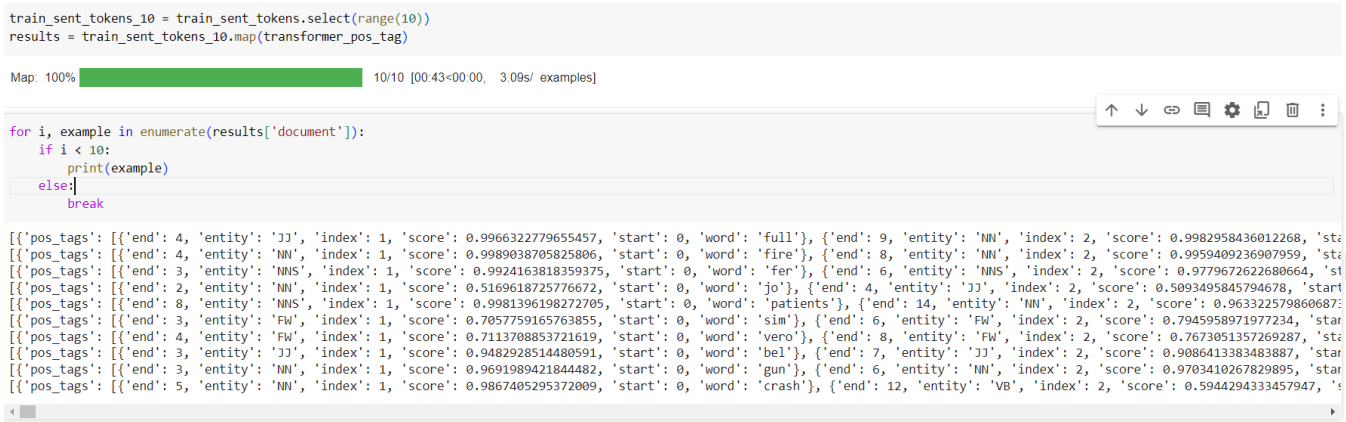

Figure 2: POS tags ('JJ', 'NN', 'NNS', etc) output by BERT

2.2. Parsing:

- **Using SpaCy:**
  - **Dependency Parsing**: SpaCy uses a statistical machine learning model to predict which words depend on which other words in a sentence. This involves identifying dependencies like subject, object, modifiers, and other grammatical relations. The output of a dependency parser is a tree structure (parse tree), where each node is a word, and edges define the dependencies.
  - **Pre-trained Models**: SpaCy offers several pre-trained models for different languages, which have been trained on annotated corpora like the Universal Dependencies dataset. These models are capable of performing various NLP tasks, including POS tagging and dependency parsing.
  - **Efficiency and Speed**: SpaCy's parsers are highly optimized for speed, making SpaCy one of the fastest libraries for NLP tasks, particularly suitable for large-scale applications and real-time processing.
  - **Integration with Other NLP Tasks**: Dependency parsing in SpaCy is well-integrated with other components of the NLP pipeline, such as tokenization and POS tagging. This integration ensures that the output of the parser is consistent with other linguistic annotations provided by SpaCy.
- **Integrating SpaCy with BERT tokenizer**:
  - For this I needed to create a custom tokenizer using BERT and then integrate it into SpaCy.
  - But when the custom tokenizer was used it even allowed token that SpaCy didn't know how to tag correctly.
  - For example the model was allowing 'ing' as a separate token and was tagging it as adjective ('JJ' tag) but SpaCy tags it as 'VERB' since it doesn't know of such tokens so it fails to POS tag it correctly.
  - This is clearly seen in the dependency parsing in the next section.

```python
import spacy
from transformers import pipeline

# Load SpaCy's English model for parsing
nlp = spacy.load("en_core_web_sm")

# Setup the POS tagging pipeline with Hugging Face
pos_pipeline = pipeline("ner", model="QCRI/bert-base-multilingual-cased-pos-english")

# Example text
text = "Apple is looking at buying U.K. startup for $1 billion."

# Apply POS tagging
pos_tags = pos_pipeline(text)

# Parse the text with SpaCy
doc = nlp(text)

for token, tag_info in zip(doc, pos_tags):
    token_tag = tag_info['entity'].split('-')[-1]  # Extract the POS tag
    print(f"{token.text} -> SpaCy POS: {token.pos_}, Hugging Face POS: {token_tag}, Dependency: {token.dep_}")
```

Figure 3: Code for comparing BERT POS tags with SpaCy inbuilt POS tagger tags using it's inbuilt tokenizer.

```
Apple -> SpaCy POS: PROPN, Hugging Face POS: NNP, Dependency: nsubj
is -> SpaCy POS: AUX, Hugging Face POS: VBZ, Dependency: aux
looking -> SpaCy POS: VERB, Hugging Face POS: VBG, Dependency: ROOT
at -> SpaCy POS: ADP, Hugging Face POS: IN, Dependency: prep
buying -> SpaCy POS: VERB, Hugging Face POS: VBG, Dependency: pcomp
U.K. -> SpaCy POS: PROPN, Hugging Face POS: VBG, Dependency: dobj
startup -> SpaCy POS: NOUN, Hugging Face POS: NNP, Dependency: dep
for -> SpaCy POS: ADP, Hugging Face POS: ., Dependency: prep
$ -> SpaCy POS: SYM, Hugging Face POS: NNP, Dependency: quantmod
1 -> SpaCy POS: NUM, Hugging Face POS: ., Dependency: compound
billion -> SpaCy POS: NUM, Hugging Face POS: NN, Dependency: pobj
. -> SpaCy POS: PUNCT, Hugging Face POS: NN, Dependency: punct
```

Figure 4 Comparing SpaCy and BERT POS tags for the same tokens.

```python
import spacy
from spacy.tokens import Doc
from transformers import AutoTokenizer

def create_custom_tokenizer(nlp, tokenizer):

    def custom_tokenizer(text):
        # Tokenize the text using the Hugging Face tokenizer
        encoded_input = tokenizer(text, return_tensors='pt', add_special_tokens=False)
        # Decode tokens into text tokens
        tokens = [tokenizer.decode(tok, skip_special_tokens=True).strip() for tok in encoded_input['input_ids'][0]]
        # Create a SpaCy Doc from the list of token texts
        return Doc(nlp.vocab, words=tokens)

    return custom_tokenizer

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

# Replace the tokenizer with custom tokenizer
nlp.tokenizer = create_custom_tokenizer(nlp,tokenizer)

# Test the tokenizer
doc = nlp("Apple is looking at buying U.K. startup for $1 billion.")
print([token.text for token in doc])
```

```
['Apple', 'is', 'looking', 'at', 'buy', '##ing', 'U', '.', 'K', '.', 'startu', '##p', 'for', '$', '1', 'billion', '.']
```

Figure 5: Code to setup SpaCy with custom tokenizer

2.2.1. Dependency Parsing:

This identifies the dependency relationships between words, such as which words are the subjects of verbs, which are objects, and so forth. A dependency parser will output a dependency tree that illustrates these relationships, showing how words in a sentence depend on one another.

- To remedy the incompatibility of our custom tokenizer with SpaCy's POS tagger, I decided to use the default tokenizer inbuilt in SpaCy.

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

# Test the tokenizer
doc = nlp("Apple is looking at buying U.K. startup for $1 billion.")
```

Figure 6: Using the default SpaCy tokenizer and POS tagger

```
# Process the text using the SpaCy pipeline
doc = nlp(text)

from spacy import displacy

# Visualize the dependency parse
displacy.render(doc, style='dep', jupyter=True, options={'distance': 100})
```

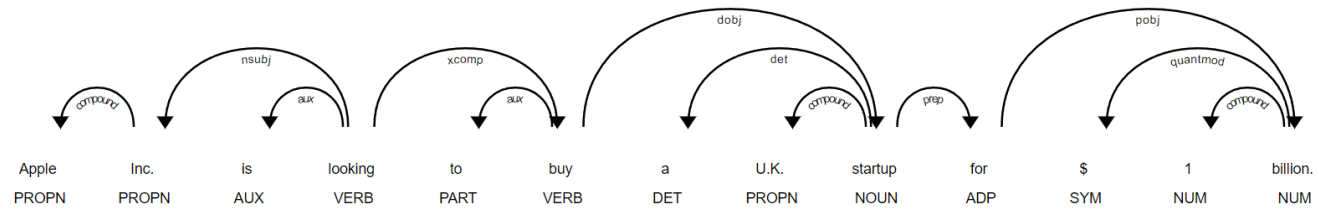

Figure 7 Dependency parse Visualization using Default SpaCy Configurations

```
# Process the text using the SpaCy pipeline
doc = nlp(text)

from spacy import displacy

# Visualize the dependency parse
displacy.render(doc, style='dep', jupyter=True, options={'distance': 100})
```

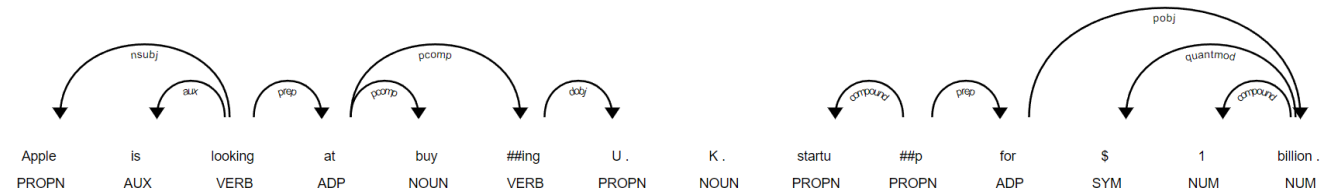

Figure 8: Output using BERT's tokenizer. Clearly showing SpaCy Tagged '##ing' as a VERB incorrectly.

2.2.2. Constituency Parsing:

Constituency parsing is a method in natural language processing that breaks down a sentence into its constituent parts, usually represented by a parse tree showing the syntactic structure according to a formal grammar. Unlike dependency parsing, which shows relationships between individual words, constituency parsing groups words into nested phrases.

```python
import stanza

# Download the English model
stanza.download('en')

# Set up the pipeline with the constituency parsing processor
nlp = stanza.Pipeline(lang='en', processors='tokenize,pos,constituency')

# Example sentence
text = "The quick brown fox jumps over the lazy dog."

# Process the text
doc = nlp(text)

# Access the constituency parse of the first sentence
constituency_parse = doc.sentences[0].constituency

# Print the parse tree
print(constituency_parse)
```

Figure 9: Code for Constituency Parsing using Stanza (Developed at Stanford)

```
(ROOT (S (NP (DT The) (JJ quick) (JJ brown) (NN fox)) (VP (VBZ jumps) (PP (IN over) (NP (DT the) (JJ lazy) (NN dog)))) (. .)))
```

Figure 10: Output of the Constituency Parse tree for the sentence 'The quick brown fox jumps over the lazy dog.'

3. Further Analysis

3.1. 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 11: Code for NER using SpaCy showing the Entities recognized in the output.

3.2. 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 12: Code for Coreference Resolution using Coreferee

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

Figure 13: 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'.

4. Challenges faced

Tried to use tokenizer used by “QCRI/bert-base-multilingual-cased-pos-english” Hugging face model as custom tokenizer in SpaCy. But since SpaCy uses its own parser it is not familiar with those tokens.. So ultimately I had to use the Tokenizer of SpaCy itself along with its POS tagger. However alternative could have been to train a custom parser. Which would have been computationally intensive.

5. Collab Notebook

Link

6. Mandate 4 Plan

Using LLM models and fine tuning them for creating abstractive summarization along with more Semantic analysis.