1. Cover Page

Natural Language Processing, Natural Language Processing (NLP) with Python: Implementation and Comparison of Modern Techniques, Askerbay Arman, 18.02.2025

2. Introduction

Natural Language Processing (NLP) is a critical subfield of artificial intelligence (AI) that focuses on the interaction between computers and human languages. It involves teaching machines to understand, interpret, and generate human language in a way that is both meaningful and useful. NLP is integral to various applications across industries, including machine translation, chatbots, sentiment analysis, speech recognition, and information retrieval, among others. As the amount of textual data continues to grow, effective NLP models are becoming increasingly important for businesses, researchers, and governments to extract valuable insights and make informed decisions.

In recent years, Deep Learning has revolutionized the field of NLP, enabling the development of more sophisticated and accurate models. Traditional NLP approaches often relied on rule-based systems and shallow machine learning techniques, which struggled with the complexity of natural language. However, the advent of deep learning models, particularly transformers, has significantly advanced the ability of machines to understand language context and semantics. Models such as BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pretrained Transformer) have set new benchmarks in a variety of NLP tasks, achieving state-of-the-art performance in areas like named entity recognition, sentiment analysis, and text generation.

Python has emerged as one of the most popular programming languages for NLP due to its extensive ecosystem of libraries that simplify the process of text processing and model development. Key Python libraries used for NLP include NLTK (Natural Language Toolkit), spaCy, and transformers. These libraries provide powerful tools for text preprocessing, tokenization, part-of-speech tagging, named entity recognition (NER), and sentiment analysis. NLTK is particularly useful for educational purposes and research, offering a wide range of algorithms for text processing and analysis. On the other hand, spaCy is a modern, fast, and efficient library that excels at industrial-level NLP tasks like named entity recognition, dependency parsing, and text categorization. Transformers, a library by Hugging Face, offers access to pre-trained transformer models such as BERT, GPT, and T5, making it easier to apply state-of-the-art deep learning techniques to NLP problems without the need for extensive computational resources.

In this report, we explore the capabilities of these Python libraries for performing key NLP tasks such as tokenization, named entity recognition, and sentiment analysis, with a focus on how traditional NLP methods compare to the advanced transformer models.

3. Implementation and Code Snippets

1) Text Preprocessing with NLTK and spaCy.

This task involves preprocessing text using two popular Python libraries: NLTK and spaCy. Preprocessing includes tokenization, lemmatization, and stopword removal.

```
[23]:
import nltk
import spacy
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

nlp = spacy.load("en_core_web_sm")
```

+ Code   + Markdown

```
[26]:
text = """Text preprocessing is an essential step in Natural Language Processing.
It involves tasks like tokenization, lemmatization, and stopword removal.
These steps help in improving the quality of textual data."""
```

```
[29]:
nltk_tokens = word_tokenize(text)
print("NLTK Tokenized Text:", nltk_tokens)
```

```
NLTK Tokenized Text: ['Text', 'preprocessing', 'is', 'an', 'essential', 'step', 'in', 'Natural', 'Language', 'Processing', '.', 'It', 'involves', 'tasks', 'like', 'tokenization', ',', 'lemmatizatio
n', ',', 'and', 'stopword', 'removal', '.', 'These', 'steps', 'help', 'in', 'improving', 'the', 'quality', 'of', 'textual', 'data', '.']
```

```
[13]:
spacy_doc = nlp(text)
spacy_tokens = [token.text for token in spacy_doc]
print("spaCy Tokenized Text:", spacy_tokens)
```

```
spaCy Tokenized Text: ['Text', 'preprocessing', 'is', 'an', 'essential', 'step', 'in', 'Natural', 'Language', 'Processing', '.', '\n', 'It', 'involves', 'tasks', 'like', 'tokenization', ',', 'lemma
tization', ',', 'and', 'stopword', 'removal', '.', '\n', 'These', 'steps', 'help', 'in', 'improving', 'the', 'quality', 'of', 'textual', 'data', '.']
```

```
[14]:
nltk.data.path.append('/usr/local/nltk_data')
```

```
[15]:
from nltk.corpus import wordnet
print(wordnet.synsets("word"))
```

Figure 1: Tokenization using NLTK and spaCy

This image shows the output of tokenization with both NLTK and spaCy, displaying how each library splits the text into individual tokens.

```
[Synset('word.n.01'), Synset('word.n.02'), Synset('news.n.01'), Synset('word.n.04'), Synset('discussion.n.02'), Synset('parole.n.01'), Synset('word.n.07'), Synset('son.n.02'), Synset('password.n.0
1'), Synset('bible.n.01'), Synset('give_voice.v.01')]
```

```
[16]:
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
nltk_lemmatized = [lemmatizer.lemmatize(word) for word in nltk_tokens]
print("NLTK Lemmatized Text:", nltk_lemmatized)
```

```
NLTK Lemmatized Text: ['Text', 'preprocessing', 'is', 'an', 'essential', 'step', 'in', 'Natural', 'Language', 'Processing', '.', 'It', 'involves', 'task', 'like', 'tokenization', ',', 'lemmatizatio
n', ',', 'and', 'stopword', 'removal', '.', 'These', 'step', 'help', 'in', 'improving', 'the', 'quality', 'of', 'textual', 'data', '.']
```

```
[19]:
spacy_lemmatized = [token.lemma_ for token in spacy_doc]
print("spaCy Lemmatized Text:", spacy_lemmatized)
```

```
spaCy Lemmatized Text: ['text', 'preprocessing', 'be', 'an', 'essential', 'step', 'in', 'Natural', 'Language', 'Processing', '.', '\n', 'it', 'involve', 'task', 'like', 'tokenization', ',', 'lemmat
ization', ',', 'and', 'stopword', 'removal', '.', '\n', 'these', 'step', 'help', 'in', 'improve', 'the', 'quality', 'of', 'textual', 'datum', '.']
```

```
[20]:
stop_words = set(stopwords.words('english'))
nltk_filtered = [word for word in nltk_lemmatized if word.lower() not in stop_words]
print("NLTK Processed Text (No Stopwords):", nltk_filtered)
```

```
NLTK Processed Text (No Stopwords): ['Text', 'preprocessing', 'essential', 'step', 'Natural', 'Language', 'Processing', '.', 'involves', 'task', 'like', 'tokenization', ',', 'lemmatization', ',', '
stopword', 'removal', '.', 'step', 'help', 'improving', 'quality', 'textual', 'data', '.']
```

```
[21]:
spacy_filtered = [token.lemma_ for token in spacy_doc if not token.is_stop]
print("spaCy Processed Text (No Stopwords):", spacy_filtered)
```

```
spaCy Processed Text (No Stopwords): ['text', 'preprocessing', 'essential', 'step', 'Natural', 'Language', 'Processing', '.', '\n', 'involve', 'task', 'like', 'tokenization', ',', 'lemmatization',
',', 'stopword', 'removal', '.', '\n', 'step', 'help', 'improve', 'quality', 'textual', 'datum', '.']
```

Figure 2: Lemmatization and Stopword Removal using NLTK and spaCy

This image shows the output after lemmatization and stopword removal with both NLTK and spaCy, highlighting the differences in the preprocessing results.

Tokenization: This process splits the text into individual words or tokens.

NLTK Tokenization: ‘word_tokenize()’ from NLTK is used to split the text.

spaCy Tokenization: The nlp object processes the text and the tokens are extracted using token.text.

Lemmatization: This step converts words into their base form (e.g., "running" becomes "run").

‘NLTK’ Lemmatization: The ‘WordNetLemmatizer’ is used to lemmatize words in the text.

‘spaCy’ Lemmatization: ‘spaCy’ automatically lemmatizes the tokens when processed.

Stopword Removal: Stopwords are commonly used words that are usually removed because they do not add meaningful value in text analysis.

‘NLTK’ Stopword Removal: ‘stopwords.words('english')’ provides a list of stopwords, and the text is filtered to remove them.

‘spaCy’ Stopword Removal: ‘spaCy’ has a built-in ‘.is_stop’ attribute for each token to check if it's a stopword.

2) Named Entity Recognition (NER) with spaCy

Named Entity Recognition (NER) is a crucial task in Natural Language Processing (NLP) that involves identifying and classifying entities (such as names of people, organizations, locations, dates, etc.) in a text. In this task:

- We use spaCy’s pre-trained NER model to extract named entities from a given text.
- The named entities are then visualized using the displacy module, which provides a neat graphical representation of the identified entities.

Figure 3. Named Entity Recognition output using ‘spaCy’ and Visualization of Named Entities using displacy

This image shows the named entities extracted from the sample text, such as "Apple Inc." (ORG), "Austin" (GPE), and "January 15, 2024" (DATE) and shows the graphical representation of the named entities in the text, with entities highlighted in different colors.

Loading spaCy's Pre-trained Model: The spacy.load("en_core_web_sm") loads a small English model which includes pre-trained components for tokenization, tagging, parsing, and named entity recognition (NER).

Text Processing: The nlp(text) function processes the given input text, analyzing it and identifying various linguistic features, including named entities.

Entity Extraction: We loop through the doc.ents object, which contains the named entities identified by spaCy. Each entity is printed along with its corresponding label (e.g., PERSON, ORG, GPE, DATE).

Visualization: The 'displacy.render(doc, style="ent", jupyter=True)' method visualizes the named entities in a graphical format within a Jupyter notebook. Each entity is displayed with its label and can be visually differentiated using different colors.

3) Text Vectorization using Transformers

In this task, we use a pre-trained transformer model (BERT) from Hugging Face's transformers library for text vectorization. The goal is to:

- Load a pre-trained BERT model and tokenizer.
- Tokenize and encode a sample sentence using the tokenizer.
- Extract word embeddings from the model's hidden states, which represent the semantic information of the input sentence.

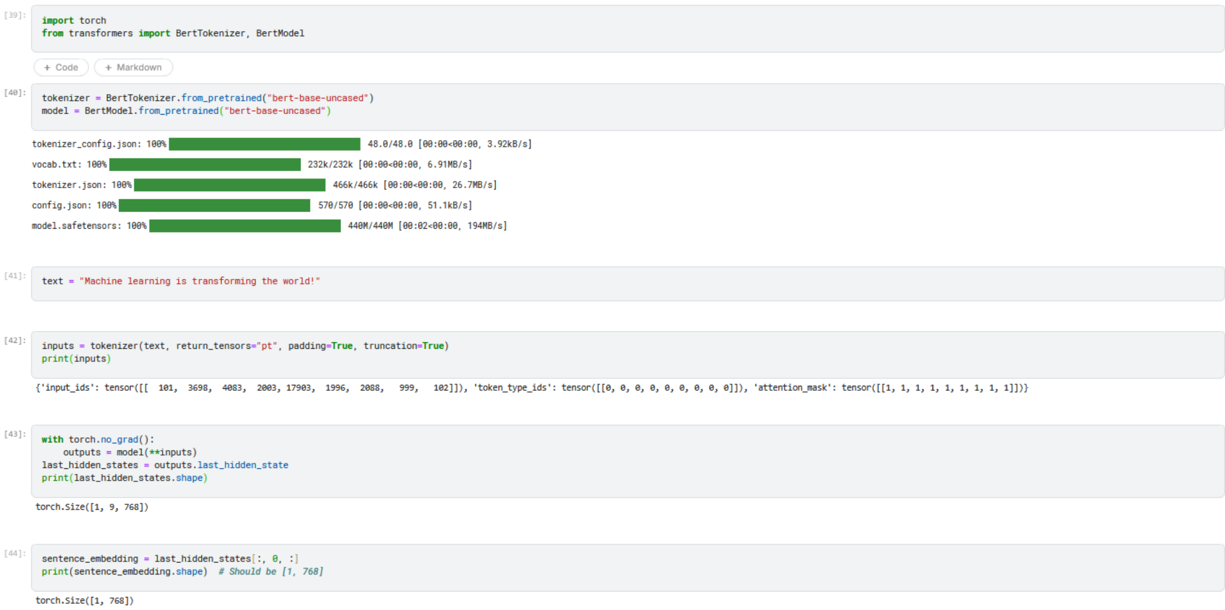

Figure 4. Tokenization and Input Encoding using BERT, Extracted Word Embeddings, Sentence Embedding (CLS Token)

This image shows the tokenized input, where the sentence is split into tokens and mapped to their corresponding IDs, ready for model processing. Displays the shape of the embeddings produced by the BERT model. The output tensor has a shape of (1, N, 768), where N is the number of tokens, and 768 represents the dimensionality of each token's embedding and shows the extracted sentence embedding, typically represented by the first token’s embedding (CLS), which is a 768-dimensional vector.

Loading the Pre-trained BERT Model and Tokenizer:

‘BertTokenizer.from_pretrained("bert-base-uncased")’ loads the pre-trained BERT tokenizer, which is responsible for converting text into tokens that the model can understand.

‘BertModel.from_pretrained("bert-base-uncased")’ loads the pre-trained BERT model that has been fine-tuned on a large corpus of text.

Tokenization:

The ‘tokenizer(text, return_tensors="pt", padding=True, truncation=True)’ function tokenizes the input text and returns it as a PyTorch tensor. The ‘padding=True’ ensures that the tokens are padded to the correct length, and truncation=True ensures the input is truncated if it exceeds the model's maximum length.

Forward Pass:

The forward pass through the model ‘(model(**inputs))’ generates the embeddings. with ‘torch.no_grad()’ ensures that the model does not calculate gradients, which saves memory and computation during inference.

Extracting Hidden States:

The ‘outputs.last_hidden_state’ gives the hidden states (embeddings) of all tokens in the sentence. These embeddings capture the contextual meaning of each token.

Sentence Embedding:

Typically, the first token ([CLS]) in the sequence is used as a sentence embedding, which is extracted by ‘last_hidden_states[:, 0, :]’. This embedding represents the entire sentence's semantic meaning in a vector of size 768.

4) Sentiment Analysis with Transformers

Task Explanation:

In this task, we perform sentiment analysis using two different approaches:

Transformers-based approach: Using Hugging Face's pre-trained sentiment analysis pipeline to analyze the sentiment of sample sentences.

Traditional approach: Using NLTK's VADER (Valence Aware Dictionary and sEntiment Reasoner) sentiment analyzer to classify the sentiment of the same sentences.

The goal is to compare the results from both methods and discuss the differences.

[46]:

```python
from transformers import pipeline
sentiment_pipeline = pipeline("sentiment-analysis")
sentences = [
    "I love this product! It's amazing.",
    "This is the worst experience I've ever had.",
    "The movie was okay, but not great."
]
results = sentiment_pipeline(sentences)
for text, result in zip(sentences, results):
    print(f"Text: {text}\nSentiment: {result['label']} (Confidence: {result['score']:.4f})\n")
```

```
No model was supplied, defaulted to distilbert/distilbert-base-uncased-finetuned-sst-2-english and revision 714eb0f (https://huggingface.co/distilbert/distilbert-base-uncased-finetuned-sst-2-english).
Using a pipeline without specifying a model name and revision in production is not recommended.
config.json: 100% 629/629 [00:00<00:00, 43.8kB/s]
model.safetensors: 100% 268M/268M [00:01<00:00, 219MB/s]
tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 3.79kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 3.82MB/s]
Device set to use cpu
Text: I love this product! It's amazing.
Sentiment: POSITIVE (Confidence: 0.9999)

Text: This is the worst experience I've ever had.
Sentiment: NEGATIVE (Confidence: 0.9998)

Text: The movie was okay, but not great.
Sentiment: NEGATIVE (Confidence: 0.9984)
```

+ Code + Markdown

[47]:

```python
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer
nltk.download("vader_lexicon")
sia = SentimentIntensityAnalyzer()
for text in sentences:
    score = sia.polarity_scores(text)
    sentiment = "POSITIVE" if score["compound"] >= 0.05 else "NEGATIVE" if score["compound"] <= -0.05 else "NEUTRAL"
    print(f"Text: {text}\nSentiment: {sentiment} (Score: {score['compound']:.4f})\n")
```

```
[nltk_data] Downloading package vader_lexicon to /root/nltk_data...
Text: I love this product! It's amazing.
Sentiment: POSITIVE (Score: 0.8516)

Text: This is the worst experience I've ever had.
Sentiment: NEGATIVE (Score: -0.6249)

Text: The movie was okay, but not great.
Sentiment: NEGATIVE (Score: -0.6112)

/usr/local/lib/python3.10/dist-packages/nltk/twitter/__init__.py:20: UserWarning: The twython library has not been installed. Some functionality from the twitter package will not be available.
  warnings.warn("The twython library has not been installed. "
```

Figure 5. Sentiment Analysis using Hugging Face Transformer and Sentiment Analysis using NLTK VADER

This image displays the sentiment label and confidence score (positive or negative sentiment) for each sentence as predicted by the Hugging Face transformer model and shows the sentiment classification (positive, neutral, or negative) based on VADER's compound score for each sentence.

Transformers-based Sentiment Analysis:

We use Hugging Face's pipeline function to load a pre-trained sentiment analysis model. The pipeline automatically tokenizes the text and performs sentiment classification.

The 'pipeline("sentiment-analysis")' method returns a list of dictionaries containing the sentiment label ('POSITIVE', 'NEGATIVE') and a confidence score for each sentence.

Traditional Sentiment Analysis using VADER:

The VADER tool in NLTK is a lexicon and rule-based sentiment analysis model. The SentimentIntensityAnalyzer is used to compute the sentiment scores for each sentence.

The polarity_scores method returns a dictionary with scores for negative, neutral, positive, and a compound score, which is used to classify the overall sentiment as positive, neutral, or negative.