Natural Language Processing

AI 829

REPORT

Keyphrase Extraction from Computer Science Research Papers Via Extractive Summarization

By

Simrath Kaur

(MT2023066)

(simrath.kaur@iiitb.ac.in)

Sejal Khandelwal

(MT2023069)

(sejal.khandelwal@iiitb.ac.in)

Abstract

This project aims to improve how we find and understand information in computer science. We have used extractive summarization techniques for keyphrase extraction from technical documents and research papers. The aim was to create a strong system that can automatically pick out important keyphrases, making it easier to access and understand computer science research topics and only study relevant ones.

Automatically extracting keyphrases from scholarly documents offers a succinct representation that both humans and machines can utilize for tasks like information retrieval, article clustering, and classification. This project focuses on determining the optimal input for keyphrase extraction methods within scientific articles. While recent deep learning techniques favor titles and abstracts due to computational complexities, traditional methods can handle full-texts. Titles and abstracts are dense with keyphrases but may overlook crucial details, whereas full-texts contain more keyphrases but are often noisy. To strike a balance, we have employed extractive summarization models on full-texts.

Overview

In this report/mandate-4, we provide an outline of the entire project, covering all the work from start to finish. This involved summarizing the dataset and conducting lexical, syntactic, and semantic analyses. We utilized different NLP methods and attempted to integrate them into our model. Initially, our focus was mainly on the supervised Bi-LSTM model. However, we encountered difficulties incorporating NLP techniques into it. Then we also tried tuning BERT model, but faced dimension compatibility issues. As a result, we ultimately turned to unsupervised algorithms like MultipartiteRank and TextRank.

For our extractive summarization task, we implemented a transformer-based model known as DistilRoBERTa. This model, based on the robust RoBERTa architecture, is distilled for efficiency while retaining strong performance in understanding contextual language. By leveraging DistilRoBERTa, we aimed to generate concise summaries by identifying and extracting key information from the text, thereby facilitating efficient comprehension of the document's main points.

After implementing the DistilRoBERTa model for extractive summarization, we proceeded with lexical analysis. This involved several preprocessing steps to clean the text and enhance its readability. We replaced contractions with their full forms, removed punctuation marks, brackets, and their contents, eliminated newline characters and tabs, and removed any references present in the text. These preprocessing steps aimed to streamline the text and ensure that the subsequent analysis and summarization process focused solely on the essential content of the document.

We employed a range of NLP techniques to enhance our project's capabilities. These included incorporating GloVe embeddings, which capture semantic relationships between words, and BERT embeddings, renowned for their contextual understanding of language. Additionally, we utilized phrase identification models such as n-grams to identify meaningful word sequences, while TF-IDF vectorization provided a numerical representation of the importance of words in documents. To gain insights into the grammatical structure of sentences, we applied part-of-speech tagging.

Named entity recognition was employed to identify and classify entities such as people, organizations, and locations within the text. Furthermore, topic modeling techniques were employed to uncover latent topics present within the text data, facilitating a deeper understanding of its underlying themes and concepts.

Lastly, we constructed semantic similarity matrices, which later served as the foundation for the TextRank method, aiding in extracting key phrases.

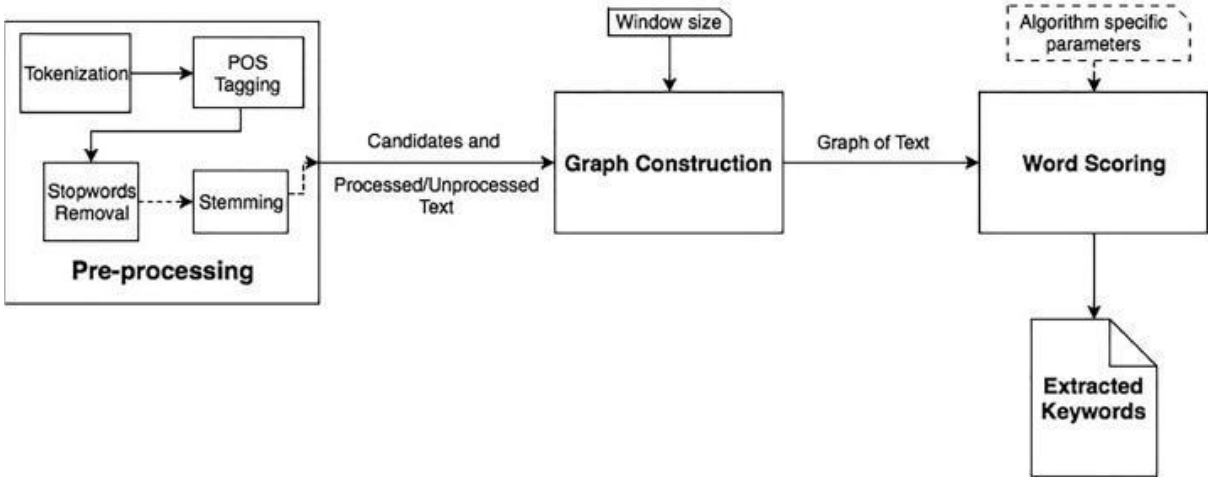


Figure 1. Basic Architecture of the project

Extractive summarization

Extractive summarization is a text summarization technique that involves selecting and extracting important sentences or passages from a document to create a concise summary. Instead of generating new sentences like in abstractive summarization, extractive summarization relies on identifying and retaining the most relevant information from the original text.

DistilRoBERTa is a variant of the RoBERTa (Robustly optimized BERT approach) model, which itself is based on the Transformer architecture. Developed by Facebook AI, DistilRoBERTa is designed to be a smaller and more efficient version of RoBERTa while maintaining competitive performance in various natural language processing (NLP) tasks.

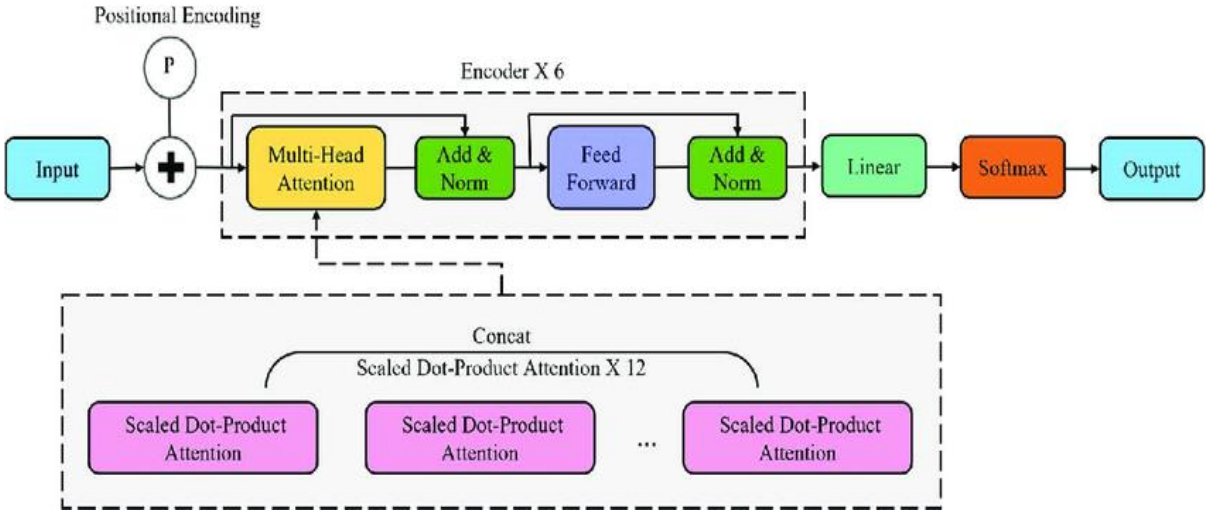


Figure 2. Architecture of DistilRoBERTa

We began by loading our extractive summarization model and applied it to the NUS dataset, which we accessed via the Hugging Face platform:

```
# ============================================================================================================================
# Define summarizer (transformersum)
# ============================================================================================================================

# using extractive model "distilroberta-base-ext-sum"
import torch

# Load the checkpoint file with map_location=torch.device('cpu')
checkpoint = torch.load("models\\epoch=3.ckpt", map_location=torch.device('cpu'))

# Rename the key from 'pytorch-ligthning_version' to 'pytorch-lightning_version'
checkpoint['pytorch-lightning_version'] = checkpoint.pop('pytorch-ligthning_version')

# Save the modified checkpoint
torch.save(checkpoint, "models\\epoch=3_modified.ckpt")

# Now load the model using ExtractiveSummarizer.load_from_checkpoint
from extractive import ExtractiveSummarizer

# Load the model checkpoint
checkpoint_path = "models\\epoch=3_modified.ckpt"
model = ExtractiveSummarizer.load_from_checkpoint(checkpoint_path)
```

```
file = 'datasets\\NUS.json'  # TEST data to evaluate the final model


# ============================================================================================
# Read data
# ============================================================================================

json_data = []
for line in open(file, 'r', encoding="utf8"):
    json_data.append(json.loads(line))

# convert json to dataframe
data = json_normalize(json_data)

print(data)



# count the running time of the program
start_time = time.time()
```

```
for index, abstract in enumerate(tqdm(data['abstract'])):
    # combine abstract + main body
    abstract_mainBody = abstract + ' ' + data['fulltext'][index]

    # remove '\n'
    abstract_mainBody = abstract_mainBody.replace('\n', ' ')

    # summarize abstract and full-text
    summarize_fulltext = model.predict(abstract_mainBody, num_summary_sentences=14)

    data['abstract'].iat[index] = summarize_fulltext
```

```
  0%|                                                                                                | 0/211 [00:00<?, ?it/s]Token indices sequence length is
longer than the specified maximum sequence length for this model (13802 > 512). Running this sequence through the model will result in indexing errors
100%|████████████████████████████████████████████████████████████████████████████████████████████| 211/211 [03:30<00:00,  1.00it/s]
```

Lexical Processing

We then proceeded to do some basic lexical processing and stemming, and used phrase identification model n-grams.

- Replace contractions:

```
def get_contractions():
    contraction_dict = {"ain't": "is not", "aren't": "are not", "can't": "cannot", "'cause": "because",
                        "could've": "could have", "couldn't": "could not", "didn't": "did not",
                        "doesn't": "does not", "don't": "do not", "hadn't": "had not", "hasn't": "has not",
                        "haven't": "have not", "he'd": "he would", "he'll": "he will", "he's": "he is",
                        "how'd": "how did", "how'd'y": "how do you", "how'll": "how will", "how's": "how is",
                        "I'd": "I would", "I'd've": "I would have", "I'll": "I will", "I'll've": "I will have",
                        "I'm": "I am", "I've": "I have", "i'd": "i would", "i'd've": "i would have",
                        "i'll": "i will", "i'll've": "i will have", "i'm": "i am", "i've": "i have",
                        "isn't": "is not", "it'd": "it would", "it'd've": "it would have", "it'll": "it will",
                        "it'll've": "it will have", "it's": "it is", "let's": "let us", "ma'am": "madam",
                        "mayn't": "may not", "might've": "might have", "mightn't": "might not",
                        "mightn't've": "might not have", "must've": "must have", "mustn't": "must not",
                        "mustn't've": "must not have", "needn't": "need not", "needn't've": "need not have",
                        "o'clock": "of the clock", "oughtn't": "ought not", "oughtn't've": "ought not have",
                        "shan't": "shall not", "sha'n't": "shall not", "shan't've": "shall not have",
                        "she'd": "she would", "she'd've": "she would have", "she'll": "she will",
                        "she'll've": "she will have", "she's": "she is", "should've": "should have",
                        "shouldn't": "should not", "shouldn't've": "should not have", "so've": "so have",
                        "so's": "so as", "this's": "this is", "that'd": "that would",
                        "that'd've": "that would have", "that's": "that is", "there'd": "there would",
                        "there'd've": "there would have", "there's": "there is", "here's": "here is",
                        "they'd": "they would", "they'd've": "they would have", "they'll": "they will",
                        "they'll've": "they will have", "they're": "they are", "they've": "they have",
                        "to've": "to have", "wasn't": "was not", "we'd": "we would", "we'd've": "we would have",
                        "we'll": "we will", "we'll've": "we will have", "we're": "we are", "we've": "we have",
                        "weren't": "were not", "what'll": "what will", "what'll've": "what will have",
                        "what're": "what are", "what's": "what is", "what've": "what have", "when's": "when is",
                        "when've": "when have", "where'd": "where did", "where's": "where is",
                        "where've": "where have", "who'll": "who will", "who'll've": "who will have",
                        "who's": "who is", "who've": "who have", "why's": "why is", "why've": "why have",
                        "will've": "will have", "won't": "will not", "won't've": "will not have",
                        "would've": "would have", "wouldn't": "would not", "wouldn't've": "would not have",
                        "y'all": "you all", "y'all'd": "you all would", "y'all'd've": "you all would have",
                        "y'all're": "you all are", "y'all've": "you all have", "you'd": "you would",
                        "you'd've": "you would have", "you'll": "you will", "you'll've": "you will have",
                        "you're": "you are", "you've": "you have", "nor": "not", "'s": "s", "s'": "s"}

    contractions_re = re.compile('(%s)' % '|'.join(contraction_dict.keys()))
    return contraction_dict, contractions_re


def replace_contractions(text):
    contractions, contractions_re = get_contractions()

    def replace(match):
        return contractions[match.group(0)]

    return contractions_re.sub(replace, text)
```

- Remove newline, tabs, punctuations:

```
newLine_tabs = '\t' + '\n'
newLine_tabs_table = str.maketrans(newLine_tabs, ' ' * len(newLine_tabs))
punctuation = string.punctuation  # + '\t' + '\n'
#punctuation = punctuation.replace("'", '')  # do not delete '
table = str.maketrans(punctuation, ' '*len(punctuation))

def remove_punct_and_non_ascii(text):
    clean_text = text.translate(table)
    clean_text = clean_text.encode("ascii", "ignore").decode()  # remove non-ascii characters
    # remove all single letter except from 'a' and 'A'
    clean_text = re.sub(r"\b[b-zB-Z]\b", "", clean_text)
    return clean_text
def remove_brackets_and_contents(doc):
    """
    remove parenthesis, brackets and their contents
    :param doc: initial text document
    :return: text document without parenthesis, brackets and their contents
    """
    ret = ''
    skip1c = 0
    # skip2c = 0
    for i in doc:
        if i == '[':
            skip1c += 1
        # elif i == '(':
        # skip2c += 1
        elif i == ']' and skip1c > 0:
            skip1c -= 1
        # elif i == ')'and skip2c > 0:
        # skip2c -= 1
        elif skip1c == 0:  # and skip2c == 0:
            ret += i
    return ret

def remove_newline_tabs(text):
    return text.replace('\n', ' ').replace('\t', ' ')
```

- Remove references: Since we are dealing with research papers, which tend to have many references and that might not be contextually relevant, it is important to remove them.

```
def remove_references(doc):
    """
    remove references of publications (in document text)
    :param doc: initial text document
    :return: text document without references
    """
    # delete newline and tab characters
    clear_doc = doc.translate(newLine_tabs_table)

    # remove all references of type "Author, J. et al., 2014"
    clear_doc = re.sub(r'[A-Z][a-z]+,\s[A-Z][a-z]*\. et al.,\s\d{4}', "REFPUBL", clear_doc)

    # remove all references of type "Author et al. 1990"
    clear_doc = re.sub("[A-Z][a-z]+ et al. [0-9]{4}", "REFPUBL", clear_doc)

    # remove all references of type "Author et al."
    clear_doc = re.sub("[A-Z][a-z]+ et al.", "REFPUBL", clear_doc)

    return clear_doc
```

- Stopword Removal:
  Stopword removal specifically deals with filtering out common and less informative words, known as stopwords, from the text. Stopwords are words that occur frequently in a language but often carry little semantic meaning or contribution to the understanding of the text. Examples of stopwords include articles (e.g., "the," "a"), prepositions (e.g., "in," "on"), conjunctions (e.g., "and," "but"), and some common verbs (e.g., "is," "have").

```
stoplist = list(string.punctuation)
stoplist += ['-lrb-', '-rrb-', '-lcb-', '-rcb-', '-lsb-', '-rsb-']
stoplist += stopwords.words('english')
```

- Phrase identification Model: n-gram and stemming:
  An n-gram phrase identification model is a technique used in natural language processing (NLP) to identify meaningful phrases or combinations of words of length n within a text corpus.

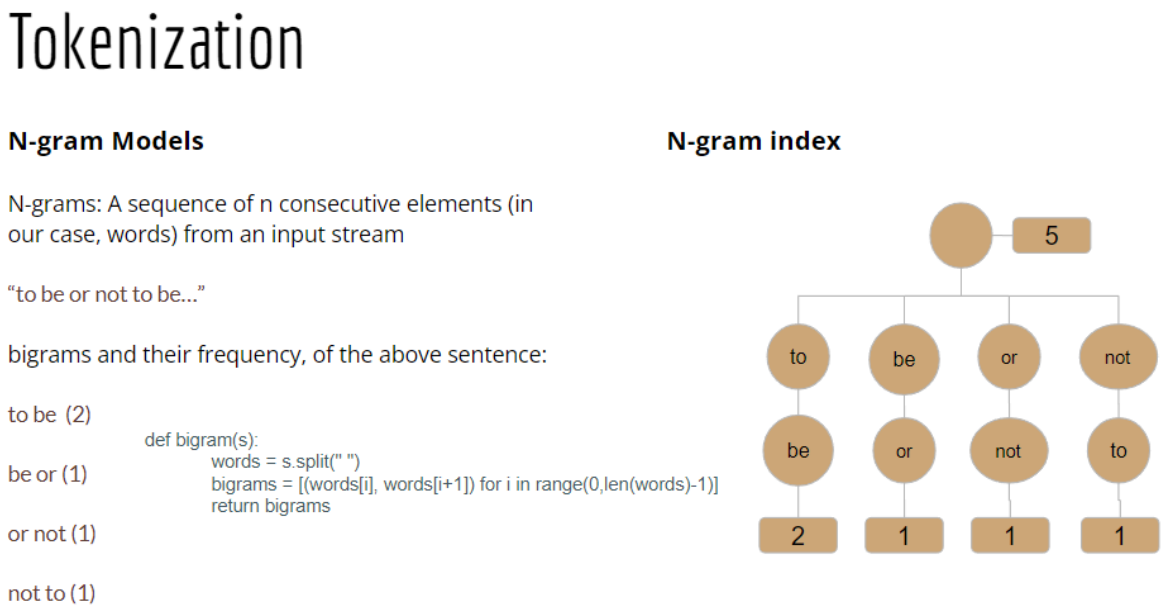


Figure 3. N-gram

We're using `CountVectorizer` from scikit-learn with `ngram_range=(1, 2)`, which means we are considering both unigrams (single words) and bigrams (sequences of two words) in our analysis. This allows us to capture not only individual words but also pairs of words that commonly appear together, potentially identifying meaningful phrases in our text data.

The `tokenize_and_stem` function tokenizes each document into words and then applies stemming to reduce words to their root forms.

The `CountVectorizer` then converts these tokenized and stemmed documents into a matrix where each row represents a document, and each column represents a unique word (or n-gram, in this case, with n ranging from 1 to 2), with each cell containing the count of the corresponding word in the document.

```
stemmer = PorterStemmer()

def tokenize_and_stem(text):
    tokens = word_tokenize(text)
    stems = [stemmer.stem(t) for t in tokens]
    return stems

# Tokenize and create n-grams
cv = CountVectorizer(tokenizer=tokenize_and_stem, ngram_range=(1, 2))
cv.fit(data_abstract['abstract'])

X = cv.transform(data_abstract['abstract'])
```

Syntactic Processing

Vectorization:

Vectorization in the context of natural language processing (NLP) refers to the process of converting text data into numerical vectors, enabling algorithms to process and analyze textual information. It transforms raw text into a format that machine learning models can understand and operate on efficiently

The purposes of following vectorization techniques i.e. TF-IDF and bag-of-words (CountVectorizer) are to represent text data in a numerical format suitable for machine learning tasks such as document classification, clustering, and information retrieval.

- **Tf-idf Vectorization:**

TF-IDF is a common technique used in Natural Language Processing (NLP) for converting text documents into numerical vectors, which can then be used for various NLP tasks such as document classification, clustering, and information retrieval.

tf-idf

Commonly used scoring model for text indexing and search

$$tfidf_d(t) = tf_d(t) \times idf(t)$$

In the following code, the `TfidfVectorizer` is a class provided by the scikit-learn library. This class is used to convert a collection of raw documents (in this case, abstracts) into a matrix of TF-IDF features

```
def phrase_identification_tfidf(data):
    vectorizer = TfidfVectorizer(stop_words='english')
    tfidf_matrix = vectorizer.fit_transform(data['abstract'])
    return tfidf_matrix
```

- **Bag of Words(BoW):**

  A bag of words is one of the popular word embedding techniques of text where each value in the vector would represent the count of words in a document/sentence. In other words, it extracts features from the text. We also refer to it as vectorization.

  We're using Python's `CountVectorizer` from scikit-learn to create a bag-of-words representation of text data.

- **Padding:**
  Padding is a technique commonly used in natural language processing, especially in tasks involving deep learning models like recurrent neural networks (RNNs) and transformers.
  Padding plays a crucial role in syntactic processing, especially in the context of deep learning models like BERT, which require fixed-length input sequences.
  By padding the sequences, the code ensures that the syntactic structure of the input text is preserved, and no information is lost during the preprocessing stage. This allows the BERT model to effectively capture both syntactic and semantic information from the input text.

```
# Pad sequences to a fixed length
max_length = max(len(tokens) for tokens in tokenized_texts)
padded_tokenized_texts = [tokens + [0] * (max_length - len(tokens)) for tokens in tokenized_texts]

# Convert token IDs to tensors
input_ids = torch.tensor(padded_tokenized_texts)
```

- **Sentence Segmentation:**
  Sentence segmentation, also known as sentence boundary detection or sentence tokenization, is the process of dividing a text or a document into its constituent sentences.
  Many NLP tasks, such as parsing, part-of-speech tagging, and named entity recognition, operate at the sentence level. Accurate sentence segmentation facilitates syntactic analysis by providing well-defined boundaries for sentence-based processing.
  We are using spaCy's NLP pipeline to do this.

- **POS tagging:**
  Part-of-speech (POS) tagging is a fundamental technique in natural language processing (NLP) that assigns grammatical labels to words in a text. It's basically like automatically diagramming sentences, but for computers. The purpose of POS tagging is to analyse and understand the syntactic structure of a sentence, which helps in various downstream NLP tasks.

```
pos = {'NOUN', 'PROPN', 'ADJ'}
stoplist = list(string.punctuation)
stoplist += ['-lrb-', '-rrb-', '-lcb-', '-rcb-', '-lsb-', '-rsb-']
stoplist += stopwords.words('english')
extractor.candidate_selection(pos=pos)
```

- **Dependency Parsing:**
  Dependency parsing is a natural language processing (NLP) technique that aims to analyze the grammatical structure of a sentence by determining the relationships between words. In particular, it identifies the syntactic dependencies between words, revealing how each word in a sentence relates to other words and what role it plays in the overall sentence structure.
  In dependency parsing, each word in a sentence is treated as a node in a graph, and the relationships between words are represented as directed edges between these nodes. These directed edges typically indicate the grammatical relationships such as subject, object, modifier, etc.

  In following code, dependency parsing is implicitly happening within the spaCy's NLP pipeline when we call `nlp(text)` on the input text. Dependency parsing is one of the tasks performed by spaCy's pipeline.
  spaCy processes the text through its pipeline, which includes all the linguistic analysis steps. This includes breaking the text into tokens (tokenization), assigning part-of-speech tags to each token, identifying named entities, and parsing the syntactic structure of the sentence to determine the relationships between words (dependency parsing).

```
# Load the SpaCy model
nlp = spacy.load("en_core_web_sm")
def extract_named_entities(text, filter_entities=True):
    doc = nlp(text)
```

- **Candidate Weighting:**

  The candidate weighting process assigns weights to candidate keyphrases based on various factors, such as their co-occurrence frequency and context. This process involves analyzing the syntactic and semantic relationships between words in the text to determine the importance of each keyphrase candidate.

```
keyphrases = []
for abstract in data['abstract']:
    extractor.load_document(input=abstract, normalization="stemming")
    extractor.candidate_selection(pos=pos)
    extractor.candidate_weighting(alpha=1.1, threshold=0.74, method='average')
    pred_kps = extractor.get_n_best(n=10)
    keyphrases.append([kp[0].split() for kp in pred_kps])
```

Word Embeddings:

Word embeddings are dense, low-dimensional vector representations of words that capture semantic and syntactic relationships between words in a corpus of text.
Word embeddings encode both semantic and syntactic information about words. Semantic information captures the meaning of words, while syntactic information captures their structural relationships within sentences.
We have used Glove and BERT Embeddings. GloVe embeddings are used directly in the candidate weighting step of the keyphrase extraction algorithm, while BERT embeddings are used to extract embeddings for each document.

- **Glove embeddings:**
  GloVe embeddings capture the semantic meanings of words in a continuous vector space. By representing words in this way, GloVe embeddings can capture the semantic similarity between words.

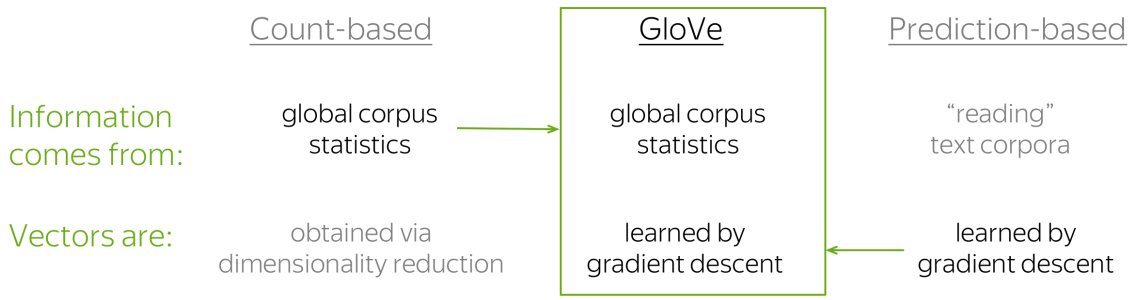


Figure 4. Glove Embedding

The following function computes the similarity between candidate keyphrases and the entire document using GloVe embeddings. This similarity score reflects how closely related a candidate keyphrase is to the content of the document. Keyphrases with higher similarity scores are more likely to be relevant to the document's content.

```
def load_glove_model(glove_file):
    word_embeddings = {}
    with open(glove_file, encoding="utf8") as f:
        for line in f:
            values = line.split()
            word = values[0]
            coefs = np.asarray(values[1:], dtype='float32')
            word_embeddings[word] = coefs
    return word_embeddings
```

- **BERT embeddings:**
  We alternatively also tried BERT embedding. BERT captures the contextual meaning of words by considering their surrounding context in a sentence or document. This contextual understanding allows BERT to capture nuances and polysemy better than GloVe, which generates static embeddings based solely on word co-occurrence statistics.

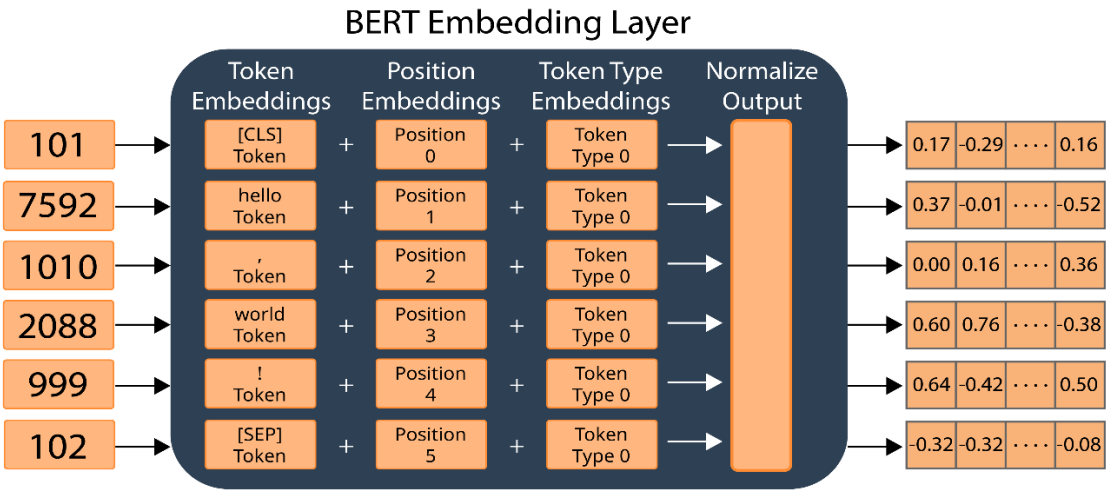


Figure 5. BERT Embedding

```
# Load BERT tokenizer and model
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Tokenize the text
tokenized_texts = [tokenizer.encode(text, add_special_tokens=True) for text in data_abstract['abstract']]

# Pad sequences to a fixed length
max_length = max(len(tokens) for tokens in tokenized_texts)
padded_tokenized_texts = [tokens + [0] * (max_length - len(tokens)) for tokens in tokenized_texts]

# Convert token IDs to tensors
input_ids = torch.tensor(padded_tokenized_texts)

# Obtain BERT embeddings
with torch.no_grad():
    outputs = model(input_ids)
    bert_embeddings = outputs.last_hidden_state  # Extract embeddings from the last layer
```

Semantic Processing

- **Named Entity Recognition(NER)**

  Named Entity Recognition (NER) is a natural language processing (NLP) technique used to identify and classify named entities within a text into predefined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc.

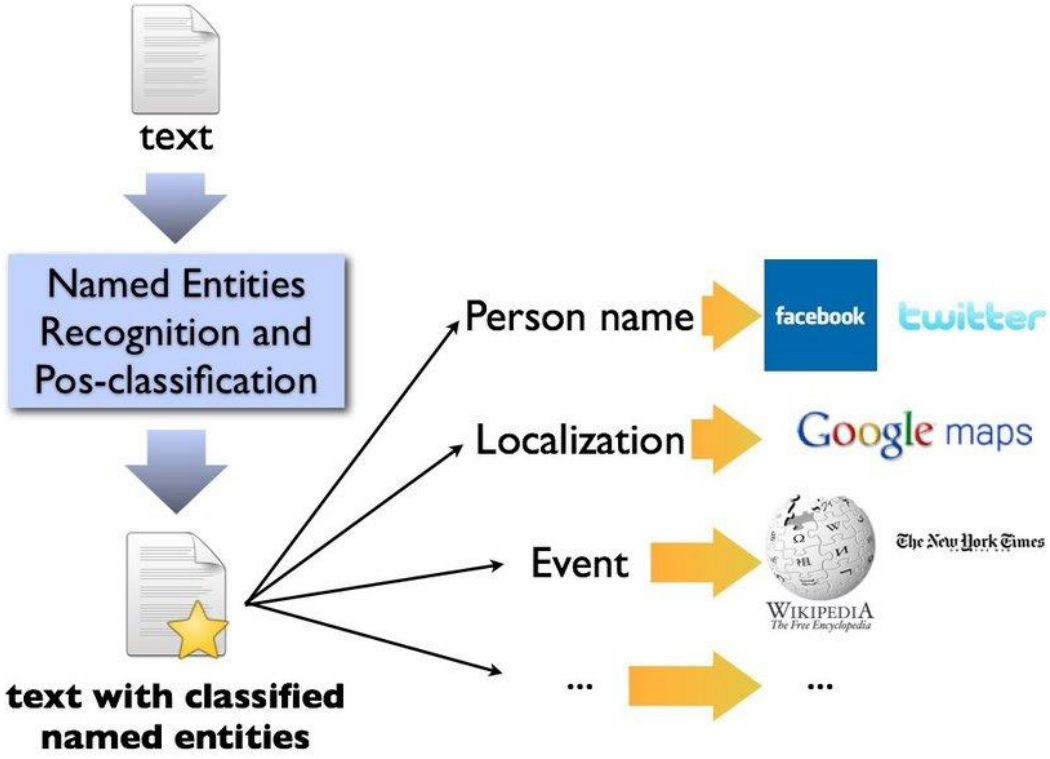


Figure 6: Named Entity Recognition

Named entities can be quite valuable in keyword extraction tasks, as they often represent important concepts or entities within a document:

➔ Filtering: Since, we are analyzing scientific articles, we want to exclude named entities like name and dates

```
# Named Entity Recognition (NER)
import spacy

# Load the SpaCy model
nlp = spacy.load("en_core_web_sm")
def extract_named_entities(text, filter_entities=True):
    doc = nlp(text)
    entities = []
    for ent in doc.ents:
        if filter_entities:
            # Exclude named entities that are person names or dates
            if ent.label_ not in ['PERSON', 'DATE']:
                entities.append(ent.text)
        else:
            entities.append(ent.text)
    return entities

data_abstract['abstract'] = data_abstract['abstract'].apply(lambda x: ' '.join(x))
data_summaries['abstract'] = data_summaries['abstract'].apply(lambda x: ' '.join(x))

# Apply the modified function to filter out specific named entities
data_abstract['abstract'] = data_abstract['abstract'].apply(lambda x: extract_named_entities(x, filter_entities=True))
data_summaries['abstract'] = data_summaries['abstract'].apply(lambda x: extract_named_entities(x, filter_entities=True))
```

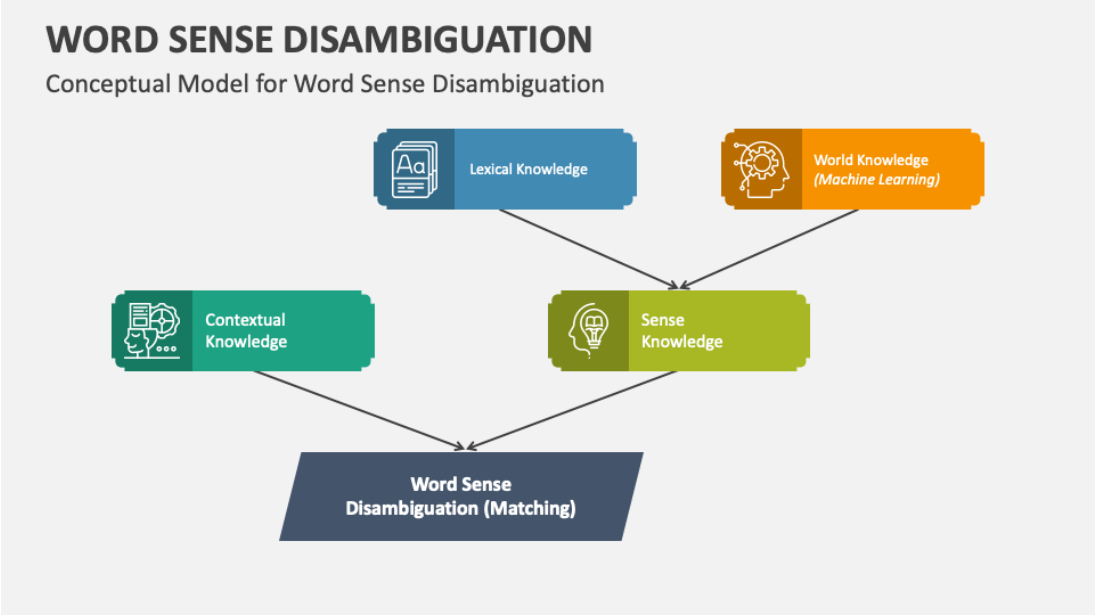
➔ Expansion of keyphrases:

We expand keyphrases by including named entities that are related to the extracted phrases. This can help in generating more comprehensive summaries or analyses by capturing additional relevant information.

```python
# ============================================================================
# Named Entity Recognition (NER)
# ============================================================================
def column_named_entities(text):
    doc = nlp(text)
    entities = [ent.text for ent in doc.ents]
    return entities
# Ensure the 'abstract' column contains strings
data_abstract['abstract'] = data_abstract['abstract'].apply(lambda x: ' '.join(x))
data_summaries['abstract'] = data_summaries['abstract'].apply(lambda x: ' '.join(x))

# Apply the modified function to filter out specific named entities
data_abstract['named_entities'] = data_abstract['abstract'].apply(column_named_entities)
data_summaries['named_entities'] = data_summaries['abstract'].apply(column_named_entities)
```

- Word Sense Disambiguation:
  Word sense disambiguation (WSD) is a natural language processing task that aims to determine the correct meaning (sense) of a word in context, particularly when a word has multiple possible meanings. While WSD is typically used in tasks such as machine translation, information retrieval, and question answering, it can also be applied to keyphrase extraction to improve the quality and specificity of extracted keyphrases.



Since we are dealing with computer science articles we will try to mention the re-occuring ambiguous terms that appear in such papers. For example: Stream, Object, python, table, Ruby, Java, Shell etc.

```
from nltk.wsd import lesk
from nltk.tokenize import word_tokenize
# Tokenize the research paper text
def word_disambiguation(text):
    tokens = word_tokenize(text)

    # Remove stop words
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word.lower() not in stop_words]

    # Apply WSD to disambiguate ambiguous terms
    disambiguated_tokens = []
    for token in filtered_tokens:
        # Apply WSD only to domain-specific terms or ambiguous terms
        if token in ['Java', 'algorithm','python', 'ruby', 'shell', 'kernel', 'address', 'query', 'table', 'object', 'stream']:
            synset = lesk(filtered_tokens, token)
            if synset:
                disambiguated_tokens.append((token, synset.definition()))
            else:
                disambiguated_tokens.append((token, None))
        else:
            disambiguated_tokens.append((token, None))

    # Extract keyphrases based on disambiguated terms
    keyphrases = []
    for token, definition in disambiguated_tokens:
        if definition:
            keyphrase = f"{token} ({definition})"
        else:
            keyphrase = token
        keyphrases.append(keyphrase)
```

- Topic Modelling:
  Topic modeling algorithms, such as Latent Dirichlet Allocation (LDA), can segment documents into topics based on the distribution of words. Each topic represents a set of words that frequently co-occur in documents. Keyphrases can then be extracted from each topic, as they are likely to represent the main themes or concepts discussed within that topic.
  After segmenting documents into topics, the most

representative terms within each topic can be identified. These terms often serve as keyphrases that summarize the main ideas discussed within the topic.

```
# ============================================================================================================
# Topic Modeling
# ============================================================================================================
def perform_topic_modeling(texts):
    tfidf_vectorizer = TfidfVectorizer(max_df=0.95, min_df=2, stop_words='english')
    tfidf_matrix = tfidf_vectorizer.fit_transform(texts)
    lda_model = LatentDirichletAllocation(n_components=5, random_state=42)
    lda_model.fit(tfidf_matrix)
    return lda_model

# Fit LDA model on combined abstracts and summaries
lda_model = perform_topic_modeling(combined_texts)
```

- Semantic Similarity Matrix:
  In TextRank-based keyphrase extraction, the semantic similarity matrix is used to calculate the similarity between different pairs of sentences or text units within the document. the semantic similarity matrix is computed based on some measure of similarity between sentences. Commonly used similarity measures include cosine similarity, Jaccard similarity, or even more sophisticated methods like word embeddings-based similarity. We have used cosine similarity.

```
# ============================================================================================================
# Semantic Similarity
# ============================================================================================================
def calculate_semantic_similarity(texts):
    tfidf_vectorizer = TfidfVectorizer()
    tfidf_matrix = tfidf_vectorizer.fit_transform(texts)
    similarity_matrix = cosine_similarity(tfidf_matrix, tfidf_matrix)
    return similarity_matrix

# Combine abstracts and summaries for semantic similarity calculation
combined_texts = data_abstract['abstract'].tolist() + data_summaries['abstract'].tolist()
semantic_similarity_matrix = calculate_semantic_similarity(combined_texts)
```

Keyphrase Extraction

- **MultipartiteRank:**

  We've utilized the MultipartiteRank algorithm from the pke library for unsupervised keyphrase extraction.
  MultipartiteRank begins by selecting potential keyphrase candidates from the input text. These candidates are typically nouns, proper nouns, or adjectives, as they often carry significant information in a document.

  Once the candidates are chosen, MultipartiteRank constructs a graph representation of the document. In this graph, nodes represent the selected keyphrase candidates, and edges denote relationships between them. These relationships can be based on co-occurrence statistics or syntactic dependencies observed in the text.

  Each node (or candidate keyphrase) in the graph is assigned a weight based on various factors, such as its frequency in the document, its position, and its relevance to the overall content.

  Edges in the graph are also assigned weights, reflecting the strength of the relationship between connected nodes. This relationship may be determined by measures of co-occurrence or semantic similarity.

MultipartiteRank partitions the graph into disjoint sets or partitions based on the weights of both nodes and edges. These partitions represent groups of closely related keyphrases that likely convey coherent topics or themes.
Finally, MultipartiteRank selects keyphrases from each partition based on their importance scores within the partition. The top-ranked keyphrases from each partition are returned as the final extracted keyphrases for the document.

```
def extract_keyphrases(data, tfidf_matrix, glove_model):
    pred_keyphrases = []
    for indx, abstract_document in enumerate(data['abstract']):
        abstract_document = preprocessing(abstract_document)



        extractor = pke.unsupervised.MultipartiteRank()
        extractor.load_document(input=abstract_document, normalization="stemming")
        pos = {'NOUN', 'PROPN', 'ADJ'}
        stoplist = list(string.punctuation)
        stoplist += ['-lrb-', '-rrb-', '-lcb-', '-rcb-', '-lsb-', '-rsb-']
        stoplist += stopwords.words('english')
        extractor.candidate_selection(pos=pos)

        # Compute similarity scores between candidate phrases and document using GloVe embeddings
        candidate_scores = []
        for phrase in extractor.candidates:
            candidate_embedding = np.mean([glove_model.get(word, np.zeros((100,))) for word in phrase[0].split()], axis=0)
            document_embedding = np.mean([glove_model.get(word, np.zeros((100,))) for word in abstract_document.split()], axis=0)
            similarity_score = np.dot(candidate_embedding, document_embedding) / (np.linalg.norm(candidate_embedding) * np.linalg.norm(document_embedding))
            candidate_scores.append(similarity_score)  # Keep similarity scores directly

        # Using similarity scores for candidate phrases weighting
        extractor.candidate_weighting(method='average')  # Set method only
        pred_kps = extractor.get_n_best(n=10)
        pred_keyphrases.append([kp[0].split() for kp in pred_kps])
    return pred_keyphrases
```

- **TextRank Algorithm:**

  TextRank is an algorithm based on PageRank, originally developed by Google for ranking web pages in search engine results. It has been adapted for text summarization and keyword/keyphrase extraction tasks.

  Here's how TextRank works and how it's used in the provided code:

  TextRank views text as a graph, where words are nodes and relationships between words are edges. In the code, each sentence is processed separately, and a graph is constructed for each sentence.

  Each word in the sentence becomes a node in the graph.

  Edges are created between words based on their semantic similarity. In the provided code, the semantic similarity matrix calculated earlier is used to determine if an edge should be created between two words. If the similarity between two words exceeds a predefined threshold (0.2 in this case), an edge is added between them.

  Once the graph is constructed, the PageRank algorithm is applied to assign importance scores to each word (node) in the graph. PageRank considers both the number and quality of links (edges) to a node when determining its score.

  Finally, the top-ranked words are selected as keyphrases for each sentence. The number of keyphrases to extract

can be adjusted as needed. In the provided code, the top 5 words with the highest PageRank scores are chosen as keyphrases for each sentence.

```
# ============================================================================================================
# Graph-based Methods (TextRank)
# ============================================================================================================
def textrank_keyphrases(text):
    doc = nlp(text)
    sentences = [sent.text for sent in doc.sents]
    keyword_phrases = []

    for sentence in sentences:
        words = sentence.split()
        graph = nx.Graph()

        # Add nodes to the graph
        for word in words:
            graph.add_node(word)

        # Add edges to the graph
        for i in range(len(words)):
            for j in range(i+1, len(words)):
                similarity = semantic_similarity_matrix[i][j]
                if similarity > 0.2:  # Threshold for edge creation
                    graph.add_edge(words[i], words[j])

        # Calculate TextRank scores
        scores = nx.pagerank(graph)

        # Select top keywords based on TextRank scores
        keywords = sorted(scores, key=scores.get, reverse=True)[:5]
        keyword_phrases.append(keywords)

    return keyword_phrases

pred_keyphrases_textrank = data_abstract['abstract'].apply(textrank_keyphrases)
```

- Averaged Embeddings using BERT:

```
# Load BERT tokenizer and model
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Tokenize the text
tokenized_texts = [tokenizer.encode(text, add_special_tokens=True) for text in data_abstract['abstract']]

# Pad sequences to a fixed length
max_length = max(len(tokens) for tokens in tokenized_texts)
padded_tokenized_texts = [tokens + [0] * (max_length - len(tokens)) for tokens in tokenized_texts]

# Convert token IDs to tensors
input_ids = torch.tensor(padded_tokenized_texts)

# Obtain BERT embeddings
with torch.no_grad():
    outputs = model(input_ids)
    bert_embeddings = outputs.last_hidden_state  # Extract embeddings from the last layer


def extract_keywords_bert(embeddings):

    avg_embeddings = np.mean(embeddings.numpy(), axis=1)

    # Perform keyword extraction using the averaged embeddings
    keywords = []
    for emb in avg_embeddings:
        # Example: Find top words based on the highest values in the embedding vector
        keywords.append(" ".join([str(token) for token in emb.argsort()[-5:]]))  # Extract top 5 tokens as keywords

    return keywords

# Apply keyword extraction function to BERT embeddings
extracted_keywords = extract_keywords_bert(bert_embeddings)
```

We have used all the above mentioned methods to extract the keyphrases, and then merged them.

Evaluation

We have used the traditional evaluation function for evaluating performance metrics.
Although we have experimented and evaluated different code, we are here providing the best ones.

```
Semi-exact Match - Extraction
Precision: 0.7923
Recall: 0.8643
F1-score: 0.8267
```

```
Semi-exact Match
Precision: 0.7923
Recall: 0.6578
F1-score: 0.7188
```

```
Partial Match - Extraction
Precision: 0.3194
Recall: 0.5839
F1-score: 0.4129
```

```
Partial Match
Precision: 0.3877
Recall: 0.3477
F1-score: 0.3666
```

Challenges

- **Managing Noisy Text Data:**
  Leveraging three datasets, we enhance reliability by tapping into diverse information sources, crucial for robust analysis.

- **Dependency and Environment Issues:**
  While dealing with transformer, pytorch-lightning and necessary libraries for bi-LSTM-CRF we faced many dependencies issues.

- **Addressing Dimension Compatibility Issues:**

  Dealing with dimension compatibility issues, especially with high-dimensional embeddings, posed challenges during model tuning, such as with the BiLSTM and BERT model. To ensure seamless operation within the same dimensional space, we utilized MultipartiteRank and TextRank algorithms, providing effective solutions.

- **Evaluation Challenges:**
  Developing evaluation metrics for our model was challenging. The model's output format differed, and achieving a perfect match was difficult.

Conclusion

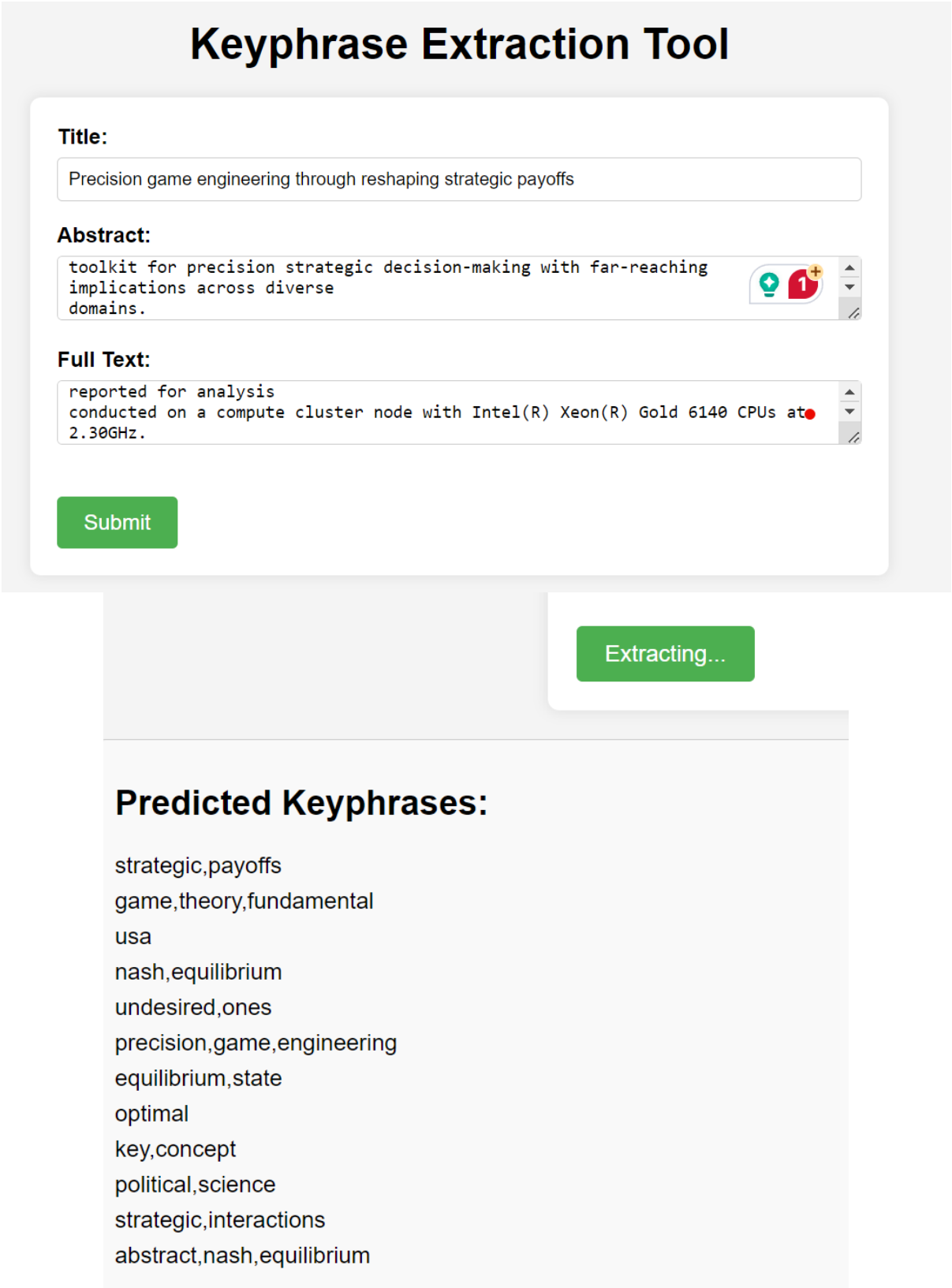
- **NLP Advancements:** Integration of DistilRoBERTa, GloVe embeddings, and Named Entity Recognition drives significant progress in text analysis and summarization.
- **Efficient Summarization:** DistilRoBERTa streamlines extractive summarization, extracting crucial information swiftly and accurately.
- **Enhanced Accessibility:** Our project empowers users to navigate complex text data effortlessly, fostering better understanding in computer science research.
- **Future Focus:** Continuous refinement of NLP models promises ongoing advancements in text analysis and summarization.
- **Wide-ranging Impact:** Insights from our project extend beyond summarization, influencing domains like information retrieval and knowledge management in academia and industry.

Working Project

(https://github.com/simrath-kaur/Keyphrase_Extraction_)

- For running this project, clone the github repo.
- Install dependencies specified in environment.
- Download the distilRoberta from transformerSum library(refer the references), and place it under models\
- Download glove embedding (find the link in references), and place it under GloVe\glove.6B directory.
- Then, go ahead and place the files in the src\ folder in root folder.
- Then, you may run python app.py in the root folder.
- If you want to see all the NLP methods that we used, checkout the Final_integration notebook
- If you want to run the files under experiments, first place them in the root folder, and also place the datasets(find link in references) and place them in datasets\summarization_experiment.

Snapshots of the working project:



References

- Keyphrase Extraction from Scientific Articles via Extractive Summarization Chrysovalantis-Giorgos Kontoulis and Eirini Papagiannopoulou and Grigorios Tsoumakas paper: https://aclanthology.org/2021.sdp-1.6.pdf

- Github Reference: https://github.com/liamca/keyphrase_extraction_and_summarization_over_custom_content

- TransformerSum : https://github.com/HHousen/TransformerSum

- DistilRoBERTa:https://huggingface.co/distilbert/distilroberta-base

- Datasets:
  SemEval: https://github.com/boudinfl/ake-datasets/tree/master/datasets/SemEval-2010

  ACM: https://github.com/boudinfl/ake-datasets/tree/master/datasets/ACM

  NUS: https://paperswithcode.com/dataset/nus

- Pytorch-lightning Library: https://lightning.ai/docs/pytorch/stable/

- NLTK: https://www.nltk.org/

- Glove Embeddings:https://www.kaggle.com/datasets/danielwillgeorge/glove6b100dtxt

- spaCy: https://pypi.org/project/spacy/
- Flask: https://flask.palletsprojects.com/en/3.0.x/