TF-IDF Implementation Report

Tuan Hung Nguyen

Swinburne University of Technolology

27/10/2023

Executive Summary

This Report aim to explain step-by-step the implementation of the TF-IDF computing in the Legal Textual Entailment Recognition task of VLSP. As all the instructions are informed in the previous report, the implementation will have a small change in the actual code.

Pre-Processing Code Implementation of TF-IDF

Tokenizing Text into Words

In Vietnamese, the construction of words is different because a definition of words in vietnamese could be single word or compound words. Therefore, I decided to use an external library of vietnamese word-tokenize which is "underthesea".

```
In [4]: from underthesea import word_tokenize
        import re
        job_1 = {}
        for i, row in enumerate(df.articles):
            for text_data in row:
                text = text_data["text"]

                tokens = word_tokenize(text)

                for token in tokens:
```

Figure 1. Underthesea library

Stopwords Removing

While Stopwords are as known as the words that does not majorly contribute to the meaning of the sentence evenly when they are being removed. After tokenzing, each token would be compared and removed in case of matching with the Vietnamese stopwords list.

```
stopwords_file = open("stopwords.txt", "r", encoding="utf-8")
stopwords_list = []

for line in stopwords_file:
    stopwords_list.append(line.strip())

stopwords_file.close()
```

Figure 2. Stopwords list importing

Before comparing with stopwords list, token is also needed to be cleaned from special character and removing space in head and tail.

```
        for token in tokens:
            token = re.sub(r'[_!@#]', '', token).lower().strip()
             Check if the token is a number (digits only) or a word

            if not re.search(r'[^\w\s\d]+', token):
                if not token in stopwords_list:
```

Figure 3. Filtering and cleaning token

TF-IDF Caculating

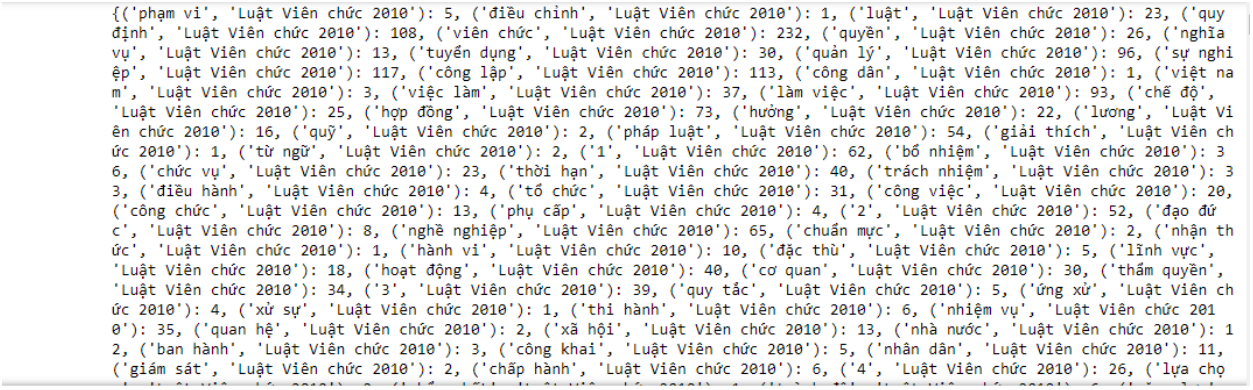
Job 1: Compute Term-Frequency (TF)

After tokenizing text, a key including term and document name that contain that term would be pass in a hash map with the value is the count of the term.

```
from underthesea import word_tokenize
import re
job_1 = {}
for i, row in enumerate(df.articles):
    for text_data in row:
        text = text_data["text"]

        tokens = word_tokenize(text)

        for token in tokens:
            token = re.sub(r'[_!@#]', '', token).lower().strip()
#            Check if the token is a number (digits only) or a word

            if not re.search(r'[^\w\s\d]+', token):
                if not token in stopwords_list:
                    key = (token, doc_dict[i])

                    if job_1.get(key):
                        job_1[key] += 1
                    else:
                        job_1[key] = 1
print(job_1)
```

Figure 4. Job 1 full code

The output would be look like below:



Job 2: Compute Document Inverted Frequency (IDF)

Figure 5. Job 1 output

In job 2, it would be divided into 2 smaller job including job 2 mapper and job 2 reducer.

Job 2 Mapper:

The input key which is belong to job 1 would be re-called in job 2 mapper to get tf. From here, the key of the hash map in job 2 mapper would be terms, and the output is doc_id, tf of the term in key, and 1 (Sums 1s to compute the number of documents containing term).

```
In [5]: job_2_mapper = {}
        for term, doc_id in job_1:
            input_key = (term, doc_id)
            output_key = (doc_id, job_1[input_key], 1)
            if job_2_mapper.get(term):
                job_2_mapper[term].append(output_key)
            else:
                job_2_mapper[term] = [output_key]
```

Figure 6. Job 2 mapper full code

Job 2 Reducer:

After mapper, we sum 1s to get n (number of documents containing term) in job 2 reducer, then we will take N (the total number of document) divided for n (which is known above) and we get the final IDF.

```
job_2_reducer = {}
for term in job_2_mapper:
    count = 0
    for article in job_2_mapper[term]:
        count += article[2]
    job_2_reducer[term] = len(doc_dict) / count
print(job_2_reducer)
```

Figure 7. Job 2 reducer full code

The Output would look like this:

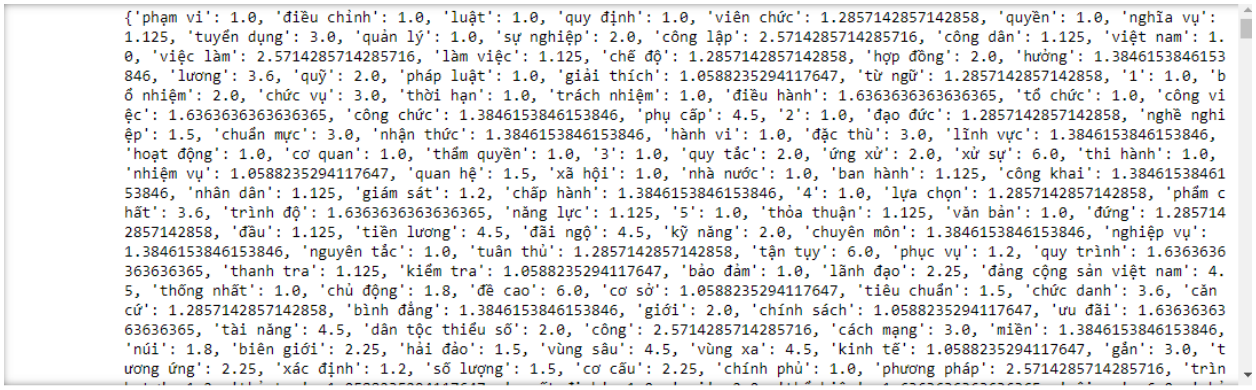


Figure 8. Job 2 reducer output

Job 3: Compute TF-IDF

After getting both TF and IDF, we just simply take the job 1 hash map (TF) and multiple by job 2 reducer output (IDF) to get the final TF-IDF calculation.

```
In [7]: job_3 = {}
        for term, doc_id in job_1:
            input_key = (term, doc_id)
            job_3[input_key] = job_2_reducer[term] * job_1[input_key]
        print(job_3)
```

Figure 9. Job 3 full code

The output would look like this:

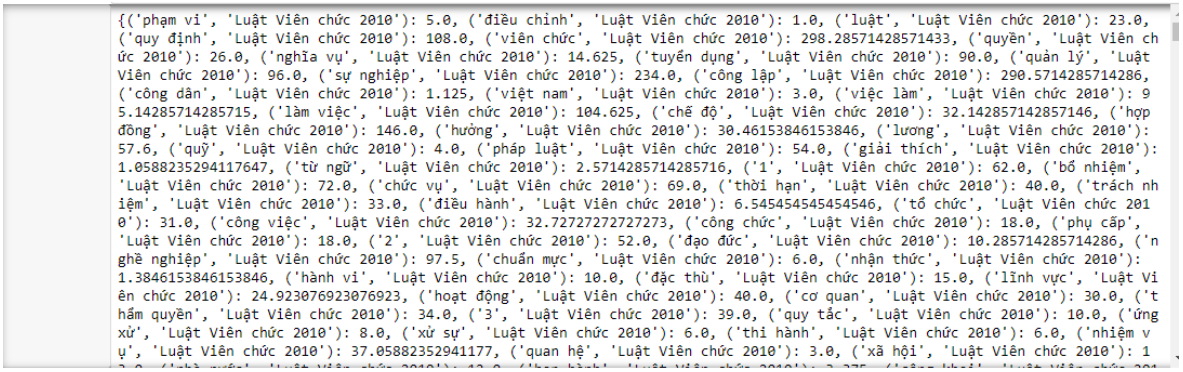


Figure 10. Job 3 output

Dicussion

Above is my final implementation to calculating the TF-IDF for the task of checking the legalness of a passage which deployed by the hadoop abstract code. In term of enhancing the algorithm to calculating TF-IDF, I think it could be enhanced by integrate the job 2 mapper and reducer together to get a cleaner code.