

Constructing a tool to identify if two texts have the same meaning.

Authors: Thomas Guegan & Mickaël Bobovitch

Abstract

This report presents an approach to identify the semantic similarity between two texts using lexical analysis and cosine similarity. The goal is to develop a tool that can accurately determine if two text inputs have the same meaning or not. The solution leverages the NLTK library for text preprocessing, Spacy for lexical similarity checks, and cosine similarity for overall text comparison. The report describes the development process, highlights the key components of the solution, provides an analysis of the results, and proposes potential enhancements using BERT-based contextual embeddings for future improvements.

1. Introduction

The objective of this project is to create a tool that can compare the meaning of two text inputs and determine if they convey the same message or have distinct meanings. Traditional approaches often rely on lexical analysis or simple similarity metrics, which may not capture the nuances of language and context. Therefore, this project aims to enhance the accuracy of text comparison using semantic analysis, synonyms, and cosine similarity, taking into account both lexical and overall text similarity.

2. Development

The solution was implemented using Python and relied on the NLTK library for text preprocessing and analysis. The development process involved several key steps:

2.1 Preprocessing

Text preprocessing is an essential step in preparing the texts for analysis. It involves tokenization, lowercase conversion, removal of stopwords, and lemmatization. The NLTK library provides useful functions for these tasks. Here's an example code snippet presenting the preprocessing step:

```
def preprocess_text_nltk(text):
```

```

    tokens = word_tokenize(text.lower()) # Tokenize and convert to
lowercase
    tokens = [word for word in tokens if word.isalpha()] # Remove
non-alphabetic characters
    tokens = [word for word in tokens if word not in stop_words] #
Remove stopwords

    lemmatizer = nltk.WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word) for word in tokens] # Lemmatize
words

    return tokens

```

The output if we provide the text “The sun is shining brightly today” will be: ['sun', 'shining', 'brightly'].

2.2 Cosine Similarity

Cosine similarity is a metric commonly used to measure the similarity between two vectors. In the context of text analysis, we can represent texts as vectors using term frequency vectors. The scikit-learn library provides functions for calculating cosine similarity. We used different ways of processing the text before calculating this similarity so here are the two functions we used in the end to compute the similarity.

```

def calculate_soft_cosine_similarity(text1, text2):
    tfidf_vectorizer = TfidfVectorizer()
    tfidf_matrix = tfidf_vectorizer.fit_transform([text1, text2])
    soft_similarity = cosine_similarity(tfidf_matrix)[0][1]

    return soft_similarity

def calculate_similarity_nltk(text1, text2):
    tokens1 = preprocess_text_nltk(text1)
    tokens2 = preprocess_text_nltk(text2)

    # Create a set of unique words from both texts
    unique_words = list(set(tokens1 + tokens2))

    # Generate word vectors for the unique words
    vector1 = [tokens1.count(word) for word in unique_words]
    vector2 = [tokens2.count(word) for word in unique_words]

```

```
# Calculate cosine similarity
similarity = cosine_similarity([vector1], [vector2])[0][0]

return similarity
```

The first one uses a TfidfVectorizer while the second one uses the preprocess_text_nltk function we implemented before.

The output will be the cosine similarity score between the two texts. Then we use these results to compute the product of the two cosine distances and set a threshold that decides if the two texts have the same meaning based on this threshold.

We thought of this method to try to minimize the error due to one way of processing the text. Using two different methods we believe that we could achieve a better result than with one method alone.

2.3 Lexical Similarity

Lexical similarity allows us to compare the semantic relations between words. Spacy contains a lexical database that can be used to check the similarity between words based on their semantic relations. We tried to implement lexical similarity using Spacy by using the ".similarity()" methods that provide the lexical similarity between two words. At first we extracted the different nouns, adjectives and verbs of each sentence and then compared them to get the similarity value between the lexical field of each sentence. However this method was not really successful, we believe that our method of comparison could be improved more by changing the methods to compute the similarity of a set based on the similarity between each element. Still here is an example of the code we tried to set up for this method :

```
def word_similarity(sentence1,sentence2):
    # Extract POS tags
    nouns1, adjectives1, verbs1 = extract_pos_tags(sentence1)
    nouns2, adjectives2, verbs2 = extract_pos_tags(sentence2)

    noun_sim = []
    for i in nouns1:
        for j in nouns2:
            #print(i, " ",j, " ", i.similarity(j) )
            noun_sim.append(i.similarity(j))

    n_sim_value = 0
    for i in noun_sim:
        n_sim_value = n_sim_value+i
```

In this code snippet we use the “extract_pos_tags()” methods provided by Spacy to identify the nouns, verbs and adjectives of the two sentences. Then we compute the sum of all similarity scores inside the nouns to make an average of this similarity later on. Using those 3 averages we tried to compute a “score of similarity” between the two sentences but it wasn’t really successful. The best results of this methods were at most 50% of correct predictions which isn’t really good.

3. Summary

The implemented solution successfully addresses the task of identifying semantic similarity between two texts. However the score of 67.5% of correct prediction might not be enough and should be improved to be used as a great tool.

The preprocessing step standardizes the texts and removes noise, improving the accuracy of subsequent analyses. The lexical similarity check using Spacy allows for the detection of words in the same lexical field, capturing synonymous relationships. Incorporating cosine similarity provides an overall similarity assessment by considering the frequency of terms in the texts.

We believe that our approach to use the combination of lexical analysis and cosine similarity is robust for detecting similar meanings in sentences. However we could also improve the tool by implementing a “context checking” methods to analyze in which context each word is used because it usually have a big impact on the meaning of a sentence.

Based on our research, it seems that enhancing our tool by considering the surrounding context and meaning of words could be achieved using BERT-based contextual embeddings. This approach would provide a more nuanced understanding of the texts and potentially improve the accuracy of similarity detection.

4. Bibliography

- NLTK Documentation, by Steven Bird
(<https://buildmedia.readthedocs.org/media/pdf/nltk/latest/nltk.pdf>)
- Scikit-learn documentation
(https://scikit-learn.org/stable/user_guide.html)
- Tkinter Documentation
(<http://tkdocs.com/tutorial/index.html>)
- Ressource : WordNet, A Lexical Database for English
(<https://wordnet.princeton.edu/>)

- Spacy Documentation
(<https://spacy.io/api/doc>)
- BERT Explained: State of the art language model for NLP by Rany Horev
(<https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270>)