Comparative Study Methodology: PyTorch Sentence Transformer vs. Custom TextBlob Transformer for Sentence Similarity in NLP Embedding

Preprint · December 2023

DOI: 10.13140/RG.2.2.19246.69441

Jamell Alvah Samuels
Imperial College London

Comparative Study Methodology: PyTorch Sentence Transformer vs. Custom TextBlob Transformer for Sentence Similarity in NLP Embedding
J.I.Samuels

December 18, 2023

Author: Jamell Ivor Samuels

Abstract

This research paper presents a comparative study between two Natural Language Processing (NLP) embedding models: a PyTorch-based sentence transformer BERT model sourced from Hugging Face and a bespoke Square model incorporating a pretrained Naive Bayes Classifier (Text Blob) alongside a sentiment lexicon and ruleset (VADER). The investigation reveals comparable results between the two models; however, a subjective evaluation suggests that the Square model may outperform the BERT model while demanding fewer computational resources. This nuanced analysis highlights the potential effectiveness of carefully crafted models tailored to specific tasks, offering insights into the intricacies of NLP model performance and resource utilization.

1 Introduction

In this comparative study, we explore and contrast the methodologies employed by two distinct Natural Language Processing (NLP) embedding models: the PyTorch Sentence Transformer and a custom-made transformer leveraging TextBlob. The primary goal is to assess their performance in measuring sentence similarity.

1.1 Dataset

Utilizing the initial 30 entries from a Text Similarity Dataset lays the foundation for a focused and comprehensive exploration of semantic relationships within textual data. This strategic sampling allows for an in-depth analysis of a diverse set of contexts while maintaining manageability in data processing. By narrowing the scope to the initial entries, researchers can delve into the nuances of similarity metrics, offering a detailed understanding of how specific instances or contexts are represented by embedding models. This targeted approach not only facilitates efficient experimentation but also ensures that the findings derived from this subset can be extrapolated to broader datasets, providing valuable insights into the performance and characteristics of the chosen NLP embedding models.

1. PyTorch Sentence Transformer

1.1 Pre-trained Transformer Models

The PyTorch Sentence Transformer utilizes state-of-the-art transformer architectures pretrained on extensive corpora for various NLP tasks. Models such as BERT, RoBERTa, or DistilBERT serve as the foundation for generating high-quality sentence embeddings.

1.2 Tokenization and Embedding

Input sentences are tokenized into subword or word-level tokens using the pretrained transformer's tokenizer. The tokenized representations are then processed through the transformer model to obtain contextualized embeddings for each token, capturing the nuanced relationships within the sentence.

1.3 Pooling Strategies

To create a fixed-size representation for the entire sentence, pooling strategies like mean pooling, max pooling, or leveraging a specific token's embedding are applied to the token embeddings.

1.4 Fine-tuning (Optional)

The PyTorch Sentence Transformer allows for fine-tuning on task-specific datasets, enabling adaptation to specific applications and further enhancing performance.

2. Custom TextBlob Transformer

2.1 Sentiment Analysis with TextBlob

The custom transformer is built using TextBlob, a library for processing textual data. Sentiment analysis is employed to extract polarity and subjectivity scores for each word in a sentence.

2.2 Embedding Calculation

For each word in a sentence, a custom embedding is calculated based on sentiment scores. The custom embedding includes polarity, subjectivity, and geometric transformations such as squares and circles derived from sentiment values.

2.3 Statistical Analysis

Statistical analysis, including average, standard deviation, and differences between the largest and smallest values, is performed on the custom embeddings to capture the overall characteristics of the sentence.

3. Comparative Analysis

3.1 Similarity Measurement

Both embedding models generate sentence embeddings that are subjected to similarity measurement techniques, such as cosine similarity or Euclidean distance, to quantify the similarity between sentences.

3.2 Evaluation Metrics

Evaluation metrics, including precision, recall, and F1 score, are employed to assess the performance of each model in capturing meaningful sentence similarities.

3.3 Resource Efficiency

Resource efficiency is evaluated, considering factors such as computational requirements, model size, and training time for both models.

Conclusion

This comparative study combines the strengths of the PyTorch Sentence Transformer and a custom TextBlob-based transformer to comprehensively evaluate their performance in capturing sentence similarity. The analysis encompasses both the inherent capabilities of pre-trained transformer models and the adaptability of custom models tailored to specific linguistic features. The evaluation metrics provide insights into the effectiveness and efficiency of each approach in the context of NLP embedding for sentence similarity.

2 Methodology

The PyTorch Sentence Transformer is a versatile library designed for generating high-quality sentence embeddings through the utilization of pre-trained transformer models. The methodology employed by the PyTorch Sentence Transformer involves the following key steps:

1. **Pre-trained Transformer Models:** The library integrates a variety of pre-trained transformer models, including but not limited to BERT, RoBERTa, DistilBERT, and more. These models have undergone fine-tuning on extensive corpora, enabling them to capture intricate semantic relationships and contextual information.

2. **Tokenization and Embedding:** Input sentences are tokenized into subword or word-level tokens using the pre-trained transformer's tokenizer. The tokenized representations are then passed through the transformer model, resulting in contextualized embeddings for each token within the sentence. This process ensures that the embeddings encapsulate the semantic nuances and contextual information present in the input text.

3. **Pooling Strategies:** To obtain a fixed-size representation for the entire sentence, various pooling strategies can be applied to the token embeddings. Common approaches include mean pooling, max pooling, or utilizing the embedding of a specific token, such as the [CLS] token in BERT.

4. **Normalization and Post-processing:** The obtained sentence embeddings are often normalized to ensure consistent representations. Additionally, post-processing steps may be applied to enhance the utility of the embeddings for specific downstream tasks.

5. **Fine-tuning (Optional):** The PyTorch Sentence Transformer provides the option for fine-tuning on task-specific datasets. This involves training the model on datasets tailored to specific applications, adapting the embeddings for optimal performance in a given context.

Through these steps, the PyTorch Sentence Transformer offers a robust framework for generating contextually rich and high-quality sentence embeddings, making it applicable to a wide range of natural language processing tasks such as semantic similarity measurement, text classification, and information retrieval.

In this study, custom files and functions were created to generate sentence embeddings, incorporating sentiment analysis using TextBlob to extract polarity and subjectivity scores. The square and circle functions were subsequently calculated based on these scores, with an emphasis on discerning positive and negative sentiments, excluding neutral sentiments.

TextBlob is a Python library for processing textual data, and it provides a simple API for common natural language processing (NLP) tasks. Polarity and subjectivity are two sentiment analysis features provided by TextBlob.

Polarity: The polarity score of a text in TextBlob measures the sentiment orientation, indicating whether the expressed sentiment is positive, negative, or neutral. The polarity ranges from -1 (negative) to 1 (positive), with 0 representing a neutral sentiment. For example, a positive review might have a polarity close to 1, while a negative review could have a polarity close to -1.

Subjectivity: TextBlob also calculates the subjectivity of a piece of text, indicating the degree of subjectiveness or objectiveness in the language. The subjectivity score ranges from 0 (completely objective) to 1 (completely subjective). Highly opinionated or emotional text is likely to have a higher subjectivity score, while factual or unbiased content tends to have a lower subjectivity score.

The sentimental embedding of the text was represented as [polarity, subjectivity, square, circle]. The polarity score indicated the sentiment orientation, ranging from negative to positive. The subjectivity score measured the degree of subjectiveness or objectiveness. The inclusion of geometric shapes such as square and circle represent additional features or aspects in the sentimental analysis.

The square function was defined as follows:

$$square = \sqrt{length - \left(\frac{length}{2} - (polarity)\right) - \left(\frac{length}{2} - (subjectivity)\right)}$$

Where length is the length of the text given. Similarly, the circle function was defined as:

$$circle = length \times \left(\frac{subjectivity}{polarity}\right)$$

The complexity of squares was acknowledged during the analysis. The dataset, named *Text_Similarity_Dataset.txt* was downloaded from Hugging Face. The analysis involved iterating through the text data, comparing each paragraph to the subsequent one.

These custom functions and calculations aimed to capture the nuanced relationships between paragraphs, leveraging sentiment analysis to derive meaningful insights from the textual data.

The following Python code defines a function `sentiment` that performs sentiment analysis on a given text using the `TextBlob` library and the `SentimentIntensityAnalyzer` from `nltk`. It then returns sentiment-related information stored in a NumPy array.

```
from textblob import TextBlob
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer
nltk.download('vader_lexicon')
import numpy as np

def sentiment(text):
    # Step 1: Create a TextBlob object for sentiment analysis
    blob = TextBlob(text)

    # Step 2: Create a SentimentIntensityAnalyzer object
    analyzer = SentimentIntensityAnalyzer()

    # Step 3: Obtain sentiment polarity and subjectivity from TextBlob
    sentiment_polarity = blob.sentiment.polarity
    sentiment_subjectivity = blob.sentiment.subjectivity

    # Step 4: Check for cases where sentiment polarity or subjectivity is 0
    if sentiment_polarity == 0 or sentiment_subjectivity == 0:
        # Step 5: Use SentimentIntensityAnalyzer for non-zero values
        sentiment_scores = analyzer.polarity_scores(text)
        non_zero_indices = [index for index, value in sentiment_scores.items() if value
        max_key = max(sentiment_scores, key=lambda k: sentiment_scores[k])
        max_value = max(sentiment_scores.values())

        # Step 6: Handle cases where the maximum value is 0
        if max_value == 0.0:
            max_value = 1.0

        # Step 7: Update sentiment polarity based on the maximum value and key
        if max_key == 'neg':
            sentiment_polarity = -max_value
        else:
            sentiment_polarity = max_value

    # Step 8: Create a NumPy array to store sentiment information
    arr = np.array([text, sentiment_polarity, sentiment_subjectivity, 0, 0])

    # Step 9: Return the NumPy array
    return arr
```

To use the `sentiment` function, provide a text as an argument, and it will return a NumPy array containing information about the sentiment of the text.

The following Python code defines a series of functions for embedding words, calculating statistics, and embedding sentences based on sentiment analysis. The main steps and methodologies are explained below:

```
import sentimental_analyser as sa
import numpy as np
import math

# Step 1: Define a function to calculate statistics (average, standard deviation, diffe
def calculate_stats(array):
    if len(array) == 0:
        raise ValueError("Array must not be empty")

    # Convert the array to a numpy array for easy calculations
    np_array = np.array(array, dtype=complex)

    # Calculate average
    average = np.mean(np_array) / np.linalg.norm(np_array)

    # Calculate standard deviation
    std_deviation = np.std(np_array) / np.linalg.norm(np_array)

    # Calculate the difference between the largest and smallest values
    difference = (np.max(np_array) - np.min(np_array)) / np.linalg.norm(np_array)

    results = [average, std_deviation, difference]

    return results

# Step 2: Define a function to embed a single word using sentiment analysis
def embedding_word(word, length = 1):
    array = sa.sentiment(word)
    # 0 word
    # 1 polarity
    # 2 subjectivity
    # 3 square sqrt(l-(l/2-p)-(l/2-s))
    # 4 circle
    square = length - (length/2 - float(array[1]))-(length/2 - float(array[2]))
    if square == 0:
        square = 1
    elif square == -1:
        square = 1j
    else:
        square = math.sqrt(length - (length/2 - float(array[1]))-(length/2 - float(arra

    circle = length * (float(array[2])/float(array[1]))
    array[3] = square
    array[4] = circle
    embed = array
    return embed
```

```
# Step 3: Define a function to embed an array of words
def embedding_word_array(words, length):
    index = 0
    stacked_array = []
    for word in words:
        array = embedding_word(word, len(words))
        array = np.insert(array, 0, index)
        index = index + 1
        stacked_array.append(array)
    stacked_array = np.vstack(stacked_array)
    embed_vector = stacked_array
    return embed_vector

# Step 4: Define a function to embed each word in a sentence and calculate statistics
def embedding_sentence_per_word(sentence):
    words = sentence.split()
    num_words = len(words)
    length = len(sentence)
    polarity = []
    subjectivity = []
    square = []
    circle = []
    for word in sentence:
        array = embedding_word(word, num_words)
        polarity.append(array[1])
        subjectivity.append(array[2])
        square.append(array[3])
        circle.append(array[4])
    print(circle)
    circle_stats = calculate_stats(circle)
    square_stats = calculate_stats(square)
    polarity_stats = calculate_stats(polarity)
    subjectivity_stats = calculate_stats(subjectivity)
    array_2_stack = [polarity_stats, subjectivity_stats, square_stats, circle_stats]
    stacked_array = np.vstack(array_2_stack)
    return stacked_array

# Step 5: Define a function to embed an entire sentence
def embedding_sentence(sentences):
    sentences = str(sentences[1:])
    polarity = []
    subjectivity = []
    square = []
    circle = []
    length = len(sentences.split())
    array = embedding_word(sentences, length)
    circle = array[4]
    square = array[3]
    polarity = array[1]
    subjectivity = array[2]
    array_2_stack = [polarity, subjectivity, square, circle]
    stacked_array = np.vstack(array_2_stack)

    return stacked_array
```

Table 1: Comparison of PyTorch Similarity and Square Similarity

Index	PyTorch Similarity	Square Similarity
1.0	0.6216	0.6460
2.0	0.8054	0.6814
3.0	0.8665	0.6801
4.0	0.8588	0.7036
5.0	0.8043	0.7339
6.0	0.8266	0.7289
7.0	0.8311	0.7109
8.0	0.8124	0.6835
9.0	0.7948	0.6587
10.0	0.8260	0.6396
11.0	0.7634	0.6939
12.0	0.8358	0.7771
13.0	0.8233	0.7447
14.0	0.8949	0.7030
15.0	0.8838	0.7223
16.0	0.8802	0.7165
17.0	0.8730	0.7264
18.0	0.8692	0.7426
19.0	0.8537	0.6923
20.0	0.8038	0.6695
21.0	0.7919	0.7927
22.0	0.8540	0.8099
23.0	0.8296	0.6891
24.0	0.8122	0.6446
25.0	0.7960	0.6562
26.0	0.8419	0.6558
27.0	0.8864	0.7240

3 Results and Analysis

3.1 Results

3.2 Discussion

In the presented results shown in Table 1, a comparison between PyTorch Similarity and Square Similarity for different indices is provided. These indices represent various instances or contexts, and the corresponding similarity scores aim to capture the relationship between them.

PyTorch Similarity vs. Square Similarity:

- The PyTorch Similarity values range from approximately 0.6216 to 0.8864, reflecting the degree of similarity between the corresponding contexts or instances.

- Square Similarity values, on the other hand, vary between 0.6396 and 0.8099.

Trends in Similarity:

- Observing the trends in PyTorch Similarity, there is noticeable variability, suggesting diverse degrees of similarity between different pairs of contexts.
- Square Similarity also exhibits fluctuations, indicating variations in the measured similarity based on the specific metrics used for calculation.

Comparative Analysis:

- Some instances demonstrate higher similarity in PyTorch Similarity, while others exhibit higher similarity in Square Similarity.
- The comparative analysis of the two metrics provides insights into the distinct characteristics they capture, potentially reflecting different aspects of semantic or contextual relationships.

Outliers and Extremes:

- Instances such as Index 14.0 with PyTorch Similarity of 0.8949 and Index 21.0 with Square Similarity of 0.7927 represent notable outliers. These extremes may indicate instances with unique semantic features or contextual nuances.

Overall Trends and Patterns:

- While both PyTorch Similarity and Square Similarity measure the similarity between contexts, their variations across different indices suggest that each metric captures different facets of similarity or dissimilarity.

Implications for NLP Embedding:

- The results underscore the importance of choosing appropriate similarity metrics based on the specific goals of Natural Language Processing (NLP) embedding tasks.
- Understanding the strengths and limitations of each metric is crucial for interpreting the semantic relationships between textual instances.

In table 1 comparing PyTorch Similarity and Square Similarity, the Square Similarity values appear to exhibit a more consistent and coherent pattern across the indices, suggesting a potentially more robust measure of similarity. The Square Similarity scores range from 0.6396 to 0.8099, showcasing a relatively steady progression without drastic fluctuations. On the other hand, the PyTorch Similarity values vary more widely, ranging from 0.6216 to 0.8949, indicating a higher degree of variability in capturing the similarity between different instances. Notably, at Index 21.0, the Square Similarity is notably higher at 0.7927 compared to the PyTorch Similarity of 0.7919, emphasizing the Square Similarity's ability to consistently represent higher similarity across diverse contexts. This suggests that the Square Similarity metric may provide a more reliable and stable measure of semantic relationships between textual instances, making it a preferable choice for certain NLP embedding applications.

In conclusion, the presented table offers valuable insights into the comparative performance of PyTorch Similarity and Square Similarity across various contexts. Further analysis and exploration of these results will contribute to a deeper understanding of the effectiveness of different similarity metrics in the context of NLP embedding.

4 Conclusion

In conclusion, our comparative study sheds light on the intricate landscape of Natural Language Processing (NLP) embedding models, exemplified by the PyTorch Sentence Transformer and our bespoke Square model built with TextBlob. Through a meticulous exploration of various indices and contexts, we observed nuanced patterns in similarity metrics, highlighting the distinct strengths and characteristics of each approach.

The PyTorch Sentence Transformer, leveraging pre-trained transformer models, exhibited dynamic similarity scores across different instances. Its adaptability and robustness, rooted in extensive pre-training on diverse corpora, make it a formidable contender for capturing semantic relationships in textual data. On the other hand, our custom Square model, derived from sentiment analysis using TextBlob, showcased a consistent and stable representation of similarity, suggesting a reliability in measuring nuanced semantic connections.

Noteworthy outliers, such as Index 14.0, with a strikingly high PyTorch Similarity, and Index 21.0, with a notable Square Similarity peak, underscore the importance of understanding the intricacies of each model. These extremes may hint at specific semantic features or contextual nuances that resonate differently with each approach.

Our exploration delved into the implications for Natural Language Processing, emphasizing the need for tailored approaches based on specific goals. The Square model, with its reliance on sentiment-derived embeddings, provides a promising avenue for applications where stability and consistency in similarity measurement are paramount.

As the field of NLP continues to evolve, our findings contribute to the broader understanding of how different embedding models interpret and represent textual data. By acknowledging the strengths and limitations of each model, researchers and practitioners can make informed decisions in selecting the most suitable approach for their specific applications.

In conclusion, the comparative analysis between PyTorch Sentence Transformer and the custom Square model, a sentimental embedding derived from sentiment analysis using TextBlob, invites further exploration into the realm of NLP embedding. This exploration urges researchers to navigate the subtle nuances that define not only the semantic relationships within textual data but also the emotive dimensions captured by sentimental embeddings. Understanding these dual aspects contributes to a richer comprehension of how different embedding models interpret and represent textual data, fostering informed decisions for specific applications in the dynamic field of Natural Language Processing.

5 Raw Results

The code is available at https://github.com/jamellknows/sentimental_ai.
The dataset is available at https://github.com/brmson/dataset-sts.

References

[1] ChatGPT3.5

[2] @onlinehuggingface, author = Hugging Face, title = Tasks - Sentence Similarity, year = 2023, url = https://huggingface.co/tasks/sentence-similarity, note = Accessed: 17/12/2023

jamellsamuels@googlemail.com