

Comparative Analysis on the Text Representation Techniques for Natural Language Processing

Charles Aaron Sarmiento
ITE Elective IV – BSIT – 4011
Jose Rizal University
Mandaluying City, Philippines
charlesaaron.sarmiento@my.jru.edu

Abstract—This paper observed five text representation techniques, from simple ones like dictionary lookups and one-hot encoding, to more complex techniques, like Bag-of-Words (BOW) and word embeddings with Continuous Bag-of-Words (CBOW). The implementations were made using various Python built-in functionalities as well as third party libraries. Each representation's output was observed, finding similarities, as well as capabilities and limitations. Results show that several factors matter in picking the right text representation for a Natural Language Processing (NLP) application, including dataset size, how context-heavy should the application be, as well as preprocessing steps need to be done for the text.

Keywords—text representation, NLP, bag-of-words, TF-IDF, word embeddings, one-hot encoding, dictionary lookup

I. INTRODUCTION

Natural Language Processing (NLP), as a subfield under Artificial Intelligence (AI), looks into allowing computers to understand human text and speech, as well as the ability to generate responses to it. To look at it in a deeper level, NLP aims to transform raw, natural language into meaningful insights that can be applied into a given field or domain. This often involves multiple steps, one of which is generating a representation of the text, such that a computer can make use of it in statistical analysis and mathematical computations [1].

But human language is complex, requiring context and may often contain nuances that is hard for computers to make sense of. Often, there are words that are spelled the same, but can have a different meaning. An example of such situation is the word “train”, where it could mean “to teach a particular skill,” or it could refer to “a connected line of railroad cars.” It is the job of effective text representation to aid computers make sense of these text nuances so that it can provide effective data for NLP applications.

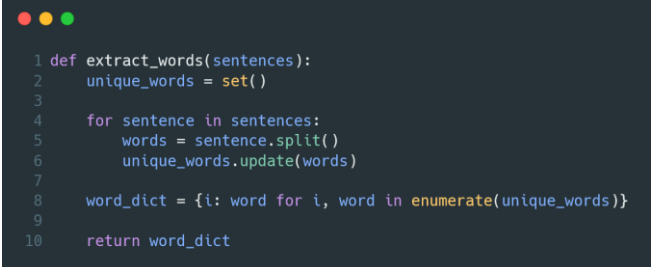
As such, this paper aims to observe the various techniques utilized to represent text data for use in NLP applications. From the simplest ones, like dictionary lookups and one-hot encoding, to more complex representations through bag-of-words and word embeddings. Each text representation methods would be implemented, and their results will be compared to each other.

II. OBSERVATIONS AND RESULTS

The implementation of each text representation technique is done in Python, utilizing various libraries including:

1. **scikit-learn** – to use its CountVectorizer, for Bag-of-Words (BOW) representation, and TfidfVectorizer modules.
2. **numpy** – to use its array data structure in one-hot encoding text data.
3. **Natural Language ToolKit (NLTK)** – to use its word_tokenize method.
4. **gensim** – to use its Word2Vec module for word embeddings using Continuous BOW (CBOW).

A. Dictionary Lookup



```
1 def extract_words(sentences):
2     unique_words = set()
3
4     for sentence in sentences:
5         words = sentence.split()
6         unique_words.update(words)
7
8     word_dict = {i: word for i, word in enumerate(unique_words)}
9
10    return word_dict
```

Fig. 1. Implementation of a dictionary lookup text representation.

The implementation to generate a dictionary of unique words in a given set of documents is done with the Set data structure built into the Python environment. A loop iterates through the sentences provided to the function and stores the words in each sentence into the unique words set.

A set in Python does not allow for duplicate values, so it is guaranteed that only unique words will appear in the final set. It is worth noting that words that have different cases (e.g. “The” and “the”) will be considered unique. As such, lowercasing of the whole dataset is required to avoid scenarios as previous described.

Once all sentences have been iterated over, a final dictionary is generated that assigns a numeric ID to each unique word. This is done with the help of the enumerate() method, a built-in function in the Python environment. This dictionary is then returned by the function.

```

Extracted words with their numeric ID
{0: 'over',
 1: 'with',
 2: 'jumps',
 3: 'or',
 4: 'lazy',
 5: 'quick',
 6: 'fox',
 7: 'journey',
 8: 'the',
 9: 'single',

```

Fig. 2. Snippet of the output of the generated unique words dictionary.

```

Enter look up key (or 'exit' to quit): 0
Result: The numeric id '0' is used to identify the word: quick

Enter look up key (or 'exit' to quit): 12
Result: The numeric id '12' is used to identify the word: question

Enter look up key (or 'exit' to quit): 50
Result: The ID '50' was not found in the dictionary.

```

Fig. 3. Example output when looking up the word associated to a numeric ID.

The output of the `extract_words()` function creates a dictionary of words, each of which are assigned a numeric ID. This dictionary can then be referenced when the need to find the ID of a word, or the word associated to an ID, arises. If there is no word associated to an ID, or vice versa, a message describing such is displayed.

While dictionary lookups may seem not ideal to use in NLP, the idea of collecting all unique words becomes the foundation that other text representation methods builds on to create a better way to represent text for use in NLP applications.

B. One-Hot Encoding

```

1 for i, sentence in enumerate(sentences):
2     for j, considered_word in list(enumerate(sentence.split())):
3         index = token_index.get(considered_word)
4         print(i,j, sentence, index, considered_word)
5         results[i, j, index] = 1
6         print(results[i,j])

```

Fig. 4. Implementation of one-hot encoding for text data.

For one-hot encoding, numpy arrays were utilized. An initial multi-dimensional array, filled with zeros, the dimension count depending on the amount of sentences, the max length of a sentence in the whole dataset, and the amount of unique words in the whole dataset.. This was made possible using numpy's `zeros()` method.

```

Final Vector Representations:
{'the': 0, 'cat': 1, 'in': 2, 'hat': 3, 'dog': 4, 'house': 5}
array([[1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0.],

       [[1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0.],
        [0., 0., 1., 0., 0., 0.],
        [1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0., 0.]])

```

Fig. 5. Example output of one-hot encoding a text dataset.

The output of one-hot encoding is a multi-dimensional array where the first dimension corresponds to a document in the dataset. Each document is then composed of multiple arrays, with child array corresponding to a word in the sentence. Each word is represented by an array of binary values, with each column corresponding to a unique word in the whole dataset.

Unlike dictionary lookups, a word is not represented by a numeric ID, but rather a binary array, which is more desirable to use in statistical analysis and model training.

C. Bag-of-Words (BOW)

```

1 vectorizer = CountVectorizer()
2 bow = vectorizer.fit_transform(sentences)
3 print(vectorizer.get_feature_names_out())
4 print(bow.toarray())

```

Fig. 6. Implementation of bag-of-words text representation.

The implementation of Bag-of-Words (BOW) text representation is made simple through scikit-learn's `CountVectorizer` module. Once it is initialized, it accepts a list of sentences that will be fitted into the vectorizer. This allows it to find the unique words in the dataset, as well as count how much each word occurs in each sentence.

	bird	cat	dog	hat	house	in	sky	the
0	0	1	0	1	0	1	0	2
1	0	0	1	0	1	1	0	2
2	1	0	0	0	0	1	1	2

Fig. 7. Example output of bag-of-words text representation.

The output of bag-of-words text representation is a collection of arrays, one for each sentence in the dataset, with each column corresponding to a unique word. Each word is assigned a value which corresponds to how much they occur in each sentence.

From observation, this text representation can suffer from spikes of word occurrences for words commonly found in an English sentence, which can be referred to as stop-words. Words like "the" often appear frequently in a sentence which

might confuse models training on these data that stop-words may have significant impact on the meaning of a sentence, when in fact they do not.

D. Term Frequency – Inverse Document Frequency (TF-IDF)

```
1 tfidf_vectorizer = TfidfVectorizer()
2 X_tfidf = tfidf_vectorizer.fit_transform(documents)
3 print("Feature Names:", tfidf_vectorizer.get_feature_names_out())
4 print("TF-IDF Matrix:\n", X_tfidf.toarray())
```

Fig. 8. Implementation of TF-IDF text representation.

	and	document	first	is	one	second	the	third	this
0	0.00000	0.469417	0.617227	0.364544	0.00000	0.000000	0.364544	0.00000	0.364544
1	0.00000	0.728445	0.000000	0.282851	0.00000	0.478909	0.282851	0.00000	0.282851
2	0.49712	0.000000	0.000000	0.293607	0.49712	0.000000	0.293607	0.49712	0.293607

Fig. 9. Example output of TF-IDF text representation.

The implementation of Term Frequency – Inverse Document Frequency (TF-IDF) is made possible with scikit-learn’s `TfidfVectorizer` module. Like the `CountVectorizer` module, it accepts a list of documents and fits them to the vectorizer, finding the unique words in the dataset. Unlike the previous vectorizer, this one looks at how much a term appears in a particular document, and as well as how rare it is to appear in the whole dataset.

TF-IDF, from observation, aims to lower down the score of words commonly found across the whole dataset, highlighting rarer, more significant words in a document. In the example output, words with a higher value can be said to be more significant and affects the meaning of a document more than those with values lower than it.

E. Word Embeddings

```
1 data = list(map(lambda sentence: word_tokenize(sentence), sentences))
2 word2vec_model = Word2Vec(
3     data,
4     min_count=1,
5     vector_size=100,
6     window=5,
7 )
```

Fig. 10. Implementation of word embeddings using continuous bag-of-words.

The implementation of word embeddings is made possible with gensim’s `Word2Vec` module, where it takes in a list of sentences that has been tokenized. In this case, the sentences were tokenized using NLTK’s `word_tokenize` method. Continuous Bag-of-Words, which is the underlying technique used by this implementation, looks at the surrounding words of a token and uses it as context.

The `Word2Vec` initialization takes a few parameters that can be adjusted to fine tune the accuracy of the produced results. These are:

- **min_count** – The minimum frequency of a word for it to be included in the model.
- **vector_size** – The size of the word vector result.

- **window** – The number of words “around” a particular that will be looked at by the model for context.

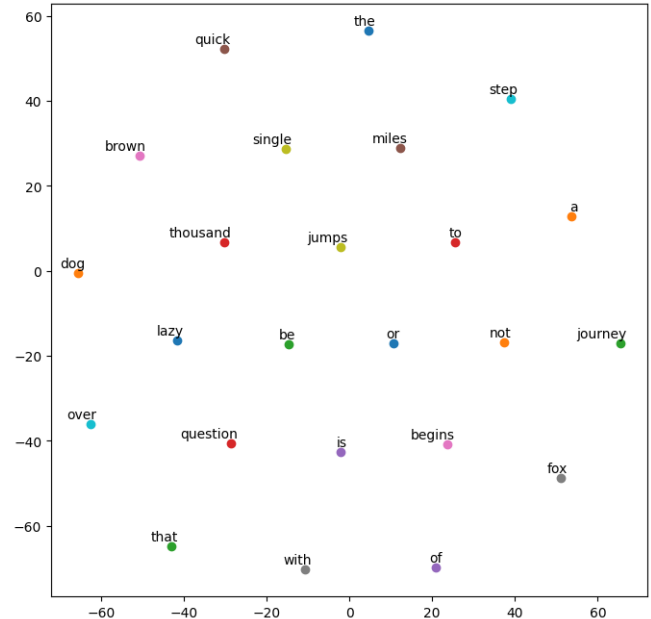


Fig. 11. Example visualization of word embeddings using continuous bag-of-words.

```
Enter first word:the
Enter second word:of
Similarity: -0.027850127
```

Fig. 12. Example output for word similarity with a word embeddings model.

The model, once fitted with data and fine-tuned with the desired parameters, is used to measure the similarity between two words. The given value corresponds to how similar the words provided, with positive values approaching one (1) can be said to be alike, while negative values approaching negative one (-1) is said to be different.

In this scenario, the fitted dataset is too small and the results of the model can be unreliable. As such, this technique should be used with larger sets of data, allowing the model to be trained on more words and bigger context for said words.

III. DISCUSSION

From the various implementations described and observed, several key insights were found:

A. Importance of Preprocessing

While there are various way to represent text into a data format understandable for computers, they all highlight an important fact, the importance of proper preprocessing of text data. While it may look like the implementation of advanced text representation techniques may not need preprocessing, internally, it still performs basic preprocessing, like lowercasing and special character removal, to ensure that it outputs an acceptable result. Simpler techniques will need to implement these manually to ensure it outputs correct results.

B. Simplicity vs. Context-rich Representation

Simple text representations are easier to implement, but these representations often lack context that is needed to extract higher quality insights from these text data. Advanced representations can capture these context and semantic relationships, with the cost of more complex calculations that is harder to reproduce by hand.

C. Influence of Dataset Size

The size of a dataset can affect the reliability of a text representation. Context-reliant representation, like word embeddings, need large datasets to accurately define context for a word, leading to more reliable results.

On the flip side, larger datasets can exponentially increase the resource needed to load a representation. BOW or one-hot encoding increases in size for every unique word it encounters.

D. Using Existing Libraries for Efficiency

While it may seem good to manually implement a text representation technique yourself, it is often better to use a library that offers that functionality, highlighting the concept of “not reinventing the wheel.” Libraries, like those mentioned in a previous section, are often tested for reliability before being released. This leads to more robust implementations that you can expect to produce the result that you want.

REFERENCES

- [1] H. S. Muslim, “Unpacking text representation in NLP: A Comparative Study of models and methods,” *COJ Robotics & Artificial Intelligence*, vol. 4, no. 3, Mar. 2025. doi:10.31031/cojra.2025.04.000589J. Clerk Maxwell, *A Treatise on Electricity and Magnetism*, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.