## [Jax Implementation of Word2Vec](https://pantelis.github.io/data-science/aiml-common/projects/_index.html)

[Religious and Philosophical Texts](https://www.kaggle.com/datasets/tentotheminus9/religious-and-philosophical-texts?select=pg2680.txt)

Robert Diasio

Dhruvil Kansara

Ryan Kitson

ABSTRACT

This project focuses on creating unique word embeddings with Word2Vec, a technology developed to turn words into unique vectors with semantic components. With the help of the Jax library, the following lays out a strategy where one can implement this algorithm themselves in the Python programming language. Also, a discussion of the advantages of using this approach as compared to TF-IDF vectors highlights the potency of this method to turn a difficult semantic problem into a tractable one, thereby illuminating the critical differences between this, and other popular methodologies.

INTRODUCTION

In this project we are trying to solve the US Patent Phrase to Phrase Matching problem as stated on Kaggle. This is a problem where you are given large amounts of text information and attempt to draw conclusions and meaning in order to match different currently existing information with existing patented products. Achieving this requires an implementation of the Word2Vec algorithm using the Jax library, with a focus on the Continuous Bag of Words (CBOW) architecture. Doing so would create a series of word-embedded vectors, which are necessary for quantifying the semantic similarity of differing phrases and contexts. The goal then would be to fit a model which would accurately predict the similarity of unseen data. However, due to the complexity of producing these word-embeddings with Word2Vec, we decided that this problem is itself sufficiently difficult for the scope of this project.

Word2Vec is a technology which can leverage the ability of neural networks to fit a model based on unsupervised learning techniques. Its goal is to produce the ‘meaning’ of words by processing large amounts of unlabeled data in the form of text. It does this by inferring the context of a given word by looking at those around it - this is its ‘window.’ Predicting what words come before or after are the goal of the learning algorithm, and the size of the window effects both the accuracy and the time it takes to generate a word vector. The output of this model is considered to be the semantic characterization of a given word, which can then be used as any normal mathematical vector. In other words, it is now able to make inferences and predictions in conjunction with other known classification and/or regression methods. However, depending on the size of the data set, generating these embeddings can be computationally expensive and time consuming, therefore it is crucial that one is aware of how the window size and other hyperparameters alter the outcome of the Word2Vec embedding process.

DATA / RELATED METHODS

In order to showcase this method appropriately, we decided to use a Kaggle dataset called [Religious and Philosophical Texts](https://www.kaggle.com/datasets/tentotheminus9/religious-and-philosophical-texts?select=pg2680.txt), which contains a handful of publically available texts from [Project Gutenberg](https://www.gutenberg.org/). These texts are not too large, and therefore help mitigate any possible computational issues stated above. The advantage of using these documents is that there is very little in terms of preprocessing that needed to be done, and therefore provide a quality demonstration of how a non-trivial implementation of this algorithm may be achieved. Although, as will be noted later, there is always going to be some level or processing of text that needs to be done in order to properly prepare it for use in a Natural Language Processing (NLP) problem.

Before moving on, it is worthwhile to discuss why such an approach is an attractive alternative to other methods. The benefit of Word2Vec is that it is able to uniquely generate an abstract datatype that is not only useful for computation, but conserves the semantic quality inherent within the data that one uses. This is a powerful method, one that exceeds others in use and flexibility. Consider a Term Frequency-Inverse Document Frequency (TF-IDF) embedding: This is a straightforward way of representing a document in terms of the words within it. It uses as its components the relative frequency of the terms found within a document, and weights each component with the inverse of how often a word appears within a corpus. This is great for discovering topic generalization, but it is unable to capture the essential meaning of the words it is processing. By discovering which words may have meaning depending only on their frequency, the subtlety of context-dependency is wholly avoided. Topic scores, cosine similarity, and any other method of using these vectors ultimately fall victim to this fundamental issue; since the words themselves are treated as vector-components, any kind of semantic meaning will have little depth. So while using TF-IDF vectors makes discovering certain semantic relationships possible, it is a method which focuses on the forest instead of the trees. Word2Vec is able to avoid such an overgeneralization by treating each word as a vector itself; providing a comprehensive inference-based approach to deriving the semantic similarity of given textual data is reduced to finding a vector whose components are relative-semantic scores.

Other approaches, such as Transformers, are powerful tools used frequently in NLP models. These are certainly noteworthy in that they provide a unique and often more suitable alternative to Word2Vec embeddings, and while they are interesting in their own right, they were decided to be beyond the scope of this project.

RELATED WORK / METHODS / TUTORIAL

There is a plethora of work related to the topic of word embeddings. The approach of all of these is fairly common and does not differ much. The works we have found to be the greatest help for this project are: Efficient Estimation of Word Representations in Vector Space, and Distributed Representations of Words and Phrases and their Compositionality, both authored by Tomas Mikolov of Google Inc. and others. Another published work related to our project is Word2Vec Model Analysis for Semantic Similarities in English Words by Derry Jatnika. The approach mentioned in the above work is a more advanced breakdown of our process. While our approach is similar to these references, the tutorial nature of this project means it is more rudimentary in terms of how robust the overall algorithm is.

In order to create our implementation of Word2Vec, we will be utilizing the Jax library. This library provides one with all the necessary classes and methods needed to properly carry out the required mathematical and computational operations for a neural network architecture as well as those for many other data related tasks, and is very helpful in terms of making the entire coding process more efficient.

To begin with, our raw data in this case is in the form of a text file. In order to properly use this text file we need to trim the file in such a way so that there is no punctuation, uppercase letters, or other undesirable characters that would impede our ability to properly parse the text. This is accomplished by taking the text file, reading it line by line, and inserting each line into a list. We then need to go back and check our list to make sure that every sentence in it is not blank, as this could happen if our file contained excessive leading or trailing spaces. We do this because we do not want our initial parsing to allow any of these instances to slip through, otherwise we may incur erratic results. Once this initial formatting is complete, we can proceed to do work on our data. This process of taking raw text and breaking it down into its component parts is called tokenization, and it is a critical step in the processing stage of the algorithm. This is because the tokenization step ensures that text of any kind will be properly formatted and does not take into account any repeated and/or case-differing spellings. Had we allowed any of these things into our workable data, we may incorrectly generate an embedding for it that does not optimally fit the context.

The next step is to create a method that will create a vocabulary for our corpus; this is a collection of all the words found within the tokenized data. We only need to keep one copy of each word as repeated instances would interfere with the later processing stages. After this vocabulary is built, we are ready to apply the final processing step for our training cycle. This is achieved by a method where we define our window size and pass in our tokenized corpus and vocabulary, the output of this method is a pair of numpy arrays that act as CBOW vectors. These will act as the formatted input to our final training cycle for the Word2Vec embeddings.

Creating the embeddings requires a class definition for Word2Vec that provides the necessary functionality for accepting the formatted input, selecting a window size, and defining various class methods for generating our unique word vectors. This includes a method for calculating loss, propagating this loss, generating different batches for training, and the training method itself. Most of the heavy-lifting within the class is done with the help of Jax, which assists with all the various numerical computations and neural network architecture.

Once this class is completed we are then able to use our code to create our word vectors. One simply needs to pass the text data to the tokenizer, the tokenized text to the vocabulary builder, and pass both to the generating method along with a specification for the window size. After this it is as simple as creating a Word2Vec object, training it, and analyzing the output data.

CONCLUSION

For our tutorial, we found that this implementation of Word2Vec to be highly effective. Our testing showed that generating these embeddings with ‘Meditations’ By Marcus Aurelius was successful. It is likely that this success is predictive of any other properly formatted text data, as there is no viable reason as to why an embedding procedure should fail to produce adequate results. Furthermore, the adjustable parameters allow a user to fine-tune this procedure to their liking, giving even more flexibility than other possible methods.

The key takeaway from this project is that if one is to approach a NLP problem with semantic analysis in mind, then a Word2Vec implementation is a highly attractive option. It is a very powerful tool to create unique solutions to otherwise seemingly intractable problems. Other methods may give adequate results, but they will fail in key areas if your problem demands a more subtle approach to semantic analysis. Therefore it seems likely that although our implementation is successful, it would be better to research more robust ones as they will most likely have better optimization and greater precision for a wider application of tasks.

The final thing to note is that it has become clear that this method bears with it a great potential to enhance already existing methodologies. Being able to generate word vectors is a highly useful tool if one has any potential application to study or analyze the meanings of words themselves; being able to turn a word into a quantifiable object, one capable of doing computations and calculations, is a great asset to the field of Data Science.