Word Embeddings

Contents

1 . Introduction

With the rise of the famed ChatGPT, it follows that many companies are investing their resources into Natural Language Processing (NLP), in a bid to develop a model stronger than ChatGPT. NLP is a technique that allows words to have meaningful representations in the context of the machine, and allows the computers to effectively understand language and communicate with humans (Stryker & Holdsworth, 2024).

The main machine learning technique behind the wonders of NLP is the implementation of word embedding vectors. Word embedding essentially represents words as numbers, to help computers understand the respective similarities and differences between words, as well as how they are used (Elastic, n.d.).

2. Implementation Process

In this section, we will discuss the methods behind implementing the word embeddings model. We will explore the various possible methods out there, weigh the advantages and disadvantages, before advancing forward with a feasible process.

2.1 Common Methods

We can consider three different ways to vectorize the words, and each method has its respective advantages and disadvantages (Tensorflow).

Firstly, we can consider implementing one-hot encodings. Imagine a sentence “I am so happy today”. Each word will have a respective zero vector of length equal to the number of unique words in the sentence, and place a ‘1’ in the index that corresponds to each word. For example, this could result in “I” being represented by [1, 0, 0, 0, 0], and “am” could be represented by [0, 1, 0, 0, 0]. The advantage of this method is that it is easy to implement, with no model implementation needed, and no training involved. However, it is computationally inefficient, as each vector will be sparse, and in a much longer dictionary of words, each vector could include thousands of 0s.

Secondly, each unique word could be encoded to a unique number. Using the same sentence as before, this could result in “I” being represented by 1, “am” being represented by 2, so on and so forth. Therefore, the number of unique encoded variables will be equivalent to the number of unique words in the sentence. The advantage of this method is that it is computationally more efficient than the previous method, as now there are no sparse vectors. However, the numerical encoding is usually arbitrary, and therefore, we are unable to capture any relationships between the various words.

Thirdly, word embeddings can be implemented to efficiently represent each word with a dense vector of numerical values. The dimension of the vector is a variable that can be chosen by the user, and each variable is essentially a weight / parameter that can be trained, similar to how the weights in a neural network can be trained and updated. The disadvantage is that by implementing a trainable model to represent this, we would need a lot of data, and the process behind accurately training the model would not be trivial. However, the advantage that this has over the other two methods would be that it can capture the different relationships and semantics between words. This can be tuned by experimenting with different dimensions, and it results in a much more accurate representation of words, as compared to the previous two methods.

2.2 Model Design

For this word embedding task, we have decided to progress with the Continuous Bag-Of-Words model. The benefits of this model is that it can effectively capture the semantics of words in the corpus, as well as understand the relative context of words. Let us consider this sentence: I am so hungry today that I could literally eat a whole cow, as well as a whole pig. What a CBOW model would do, is to then predict a word given W words before and W words after, where W = window size (Jeeva, 2023). For the implementation of this model, we have decided to go for a window size of 5. However, we have to consider that for words nearer to the start or the end of a sentence, the number of context words taken will be lower than the window size. These are some of the context-target word pairs that will be formed from this sentence: ([am, so, hungry, today, that], I), ([I, am, so, today, that, I, could, eat], hungry). Essentially, for each word *i*, the number of context words will be $min\{i-1, W\} + min\{n - i, W\}$, where n = total number of words in the sentence, and i = index of word in the sentence, starting from 1.

With this, we can create a co-occurrence matrix. A co-occurrence matrix represents the frequency that each element occurs together with another element (Van Otten, 2024). In our context, this would mean that given our specified window size of 5, for each target word, we can define a span of words which is the set of words that occur 5 words before and 5 words after the target word. All of these words in the span will therefore co-occur (Enozeren, 2024). An important thing to note is that the co-occurrence matrix will take into account repeated words, so words that appeared more than once in our sentence example, i.e. “I”, “whole”, would only have one entry in the rows and columns of this co-occurrence matrix. Each entry $cell_{ij}$ = 1 if there is a co-occurrence, and $cell_{ij}$ = 0 otherwise. Across multiple documents, if there are multiple co-occurrences, the entry will reflect the total number of co-occurrences.

	I	am	…	whole	pig
I	0	1	…	0	0
am	1	0	…	0	0

…	…	…	…	…	…
whole	0	0	…	0	1
pig	0	0	…	1	0

2.2.1 Dataset & Preprocessing

One of the more important things we have to consider when implementing word embeddings, is the size of trainable data. There are intuitively multiple relationships that could occur between words, and having a smaller dataset would limit the instances in which our model could catch the different relationships. However, we also have to consider the computational time, in that a large dataset would result in a much longer time to train (Blankenship et. al, 2024).

We can use a dataset that is already available on the internet, and we have decided to progress with the ‘book corpus’ dataset available on Hugging Face. The dataset involves texts from 7185 unique books in total, each coming from various genres. The initial number of rows adds up to 74004228.

Similar to other data science projects, it is naive to assume that the dataset is free from imperfections, and therefore, preprocessing is an important step of our model implementation (Vu, 2021).

In addition, due to the sheer size of the dataset, and our limited computing power, we have decided to randomly subsample 1% of the total dataset and consider that as our full dataset. This amounts to 740043 rows. This is done before the preprocessing steps to optimise the process.

The aforementioned preprocessing steps include:

- Expansion of contractions. It is difficult to preprocess words like “can’t” and “I’m” unless they are separated into their base words. “Can’t” would be separated into “can not” and “I’m” will be separated into “I am”.
- Changing all of capital letters to lowercase letters. It is counter-intuitive to represent the words “Machine” and “machine” as two different words with different vector representations, as they are exactly the same word. Hence, we uniformly apply all words to start with a lowercase letter to avoid this problem.
- Separating words that were accidentally concatenated together. Due to the nature of the dataset, there was prior preprocessing done, where some words were unintentionally concatenated together, for example “cameyesterday” which should be read as “came yesterday”.
- Lemmatization, which is the reduction of words to their base forms. This reduction can include stripping of prefixes and suffixes, as well as removing the tense of the word. The advantage of having lemmatization as a step in the preprocessing is to link words that

have similar meanings together (Babel Street, 2024). This will result in our model learning the similarities between words like “ask” and “asked”, which are intuitively very similar.

- Removing special characters and punctuation marks. It is difficult to include embeddings for all the special characters, and these characters rarely add information to the document. Hence, we can remove these special characters to improve the performance of our model.
- Removal of numerical characters. Similar to special characters, these are redundant to our model as there is not much meaning attached to a number. Additionally, a number that might appear in the corpus might only appear once, and many numbers might be occurring in the corpus, thus accumulating a lot of noise in our training data.

2.2.2 Model Training & Implementation

We train our model using a neural network. Firstly, we split our data into three, train_data, val_data and test_data. The train_data will be the data that we are using purely for training of the model. We validate our model during each epoch with val_data, and we only use test_data at the end to evaluate our model and show the final performance of our model. The input features are the context words, and the model will try to predictively output the target word. This is a simplified illustration of what the model looks like:

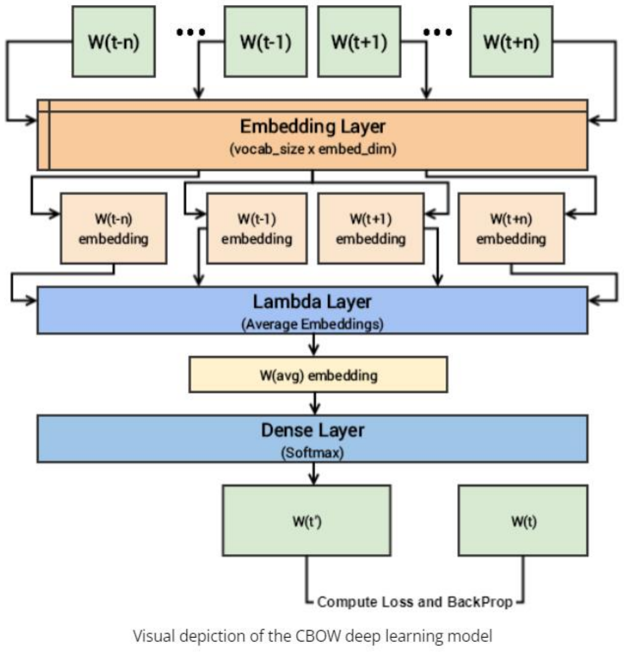

Figure 1: Illustration of Word Embedding Model (Sarkar, 2018)

When we consider the model illustrated in Figure 1, we modify it such that there are three layers in the projection. Let us run through one data point being passed through the model, where the objective is to predict the target word, so the target word will be the output. Firstly, the inputs are the respective context words corresponding to the target word. These context words are then passed through an embedding layer, where the initial weights are randomised. This embedding layer essentially maps each context word to its corresponding vector representation. The second layer is another layer where we average out the word embeddings, termed a lambda layer. The objective of this second layer is to combine our embeddings into a single representation to output a single target variable, and averaging out the embeddings helps to normalise the contribution of each word, whereas summing up the embeddings might favour more frequent words. There are other methods of combining the embeddings, and these explorations are discussed in the next section. The last layer is where we apply a softmax function, which will convert the output logits into probabilities, and the output is the word that has the highest probability. Since this is a multi-class classification problem, the loss function that we employ for the CBOW model is the categorical cross-entropy loss function. The loss function is as follows: $Loss = -\Sigma_{i=1}^{N} y_i \, log(p_i)$, where y_i is the true label and p_i is the predicted probability for the *i-th* class, which is the target word, and this is summed up over the total number of classes, which is equivalent to the vocabulary size.

The optimiser we have decided to employ for the training of our model is the ADAM optimiser. ADAM optimiser calculates and updates the moving averages of the first and second moment, and this results in a typically faster convergence rate than other optimisers, like Stochastic Gradient Descent (SGD) (Vishwakarma, 2024). It also has a low memory usage and is relatively robust, therefore making a good choice for our word embedding model.

The metric we have decided to use to train our word embedding model is accuracy, one of the most common metrics in machine learning models. The way to measure accuracy is the total number of correct predictions divided by the total number of predictions made. In this context, there would be one prediction for one target word in one epoch. However, due to the nature of the multi-class classification problem, it is possible that the standard accuracy metric would not perform well. We discuss other potential considerations in the following section.

2.2.3 Tuning of Different Parameters and Hyperparameters

In this section we discuss the tuning of different parameters and hyperparameters throughout the model training.

Firstly, we explore whether averaging of word embeddings in the lambda layer is the best method to implement this model. Besides implementing the standard mean as the technique, we also consider implementing weighted averaging, where we place larger weights on words that are closer to the target words. This method was observed to be performing better, as compared to taking the standard average.

Secondly, we implement a different metric as opposed to the standard accuracy metric. It is possible that our previous implementation could result in a low accuracy, but this does not necessarily mean that our model is not learning. This is where the Top-K accuracy metric can come in handy. Considering that in the context of a word embedding model, there could be more than one appropriate option for a target word. In that case, we want to have the model learn some diversity in predicting the target word, and measuring the Top-K accuracy could be more beneficial (Riva, 2021). This metric would count the model as predicting correctly if the actual target word is within the top K highest probabilities in the softmax prediction. We tried to implement it with K = 3 and K = 5, to explore if this metric could help the model predict better. Next, we also consider training the model on different sizes of the dataset. Given that our dataset is still quite big, we can take subsamples of it and still train a competent model. Therefore, we explore training of the model with 0.25 and 0.5 of the total dataset, as well as the full dataset. These subsets are randomly subsampled, and we explore the difference that the size of the dataset can make to the performance of the model, seeing that it is also possible that too large a dataset can result in overfitting.

Lastly, we also consider tuning the hyperparameters of the model:

- Number of epochs: It is difficult to know how many iterations it will take for the algorithm to converge, while also balancing the consideration of how long the model will take to run. Therefore, we experiment with different values for the total number of epochs, namely 10 and 20. This is because in our experimentation, the early stopping algorithm was stopping at about 12 to 13 epochs, therefore leading us to consider smaller values.
- Batch size: A smaller batch size might result in faster convergence, since there will be more parameter updates, however this might also cause an increase in computational time (Devansh, 2022). Thus, the optimal value for batch size has to be explored, and for our model, we have decided to test out the following values, 64 and 128.
- Dimension of embedding layer: The size of the embedding layer can potentially result in a major difference in the model performance. A smaller embedding dimension might lose out on information but can result in lesser noise, while a larger embedding dimension can provide more information but might face overfitting issues. As such, we have decided to explore 50 and 100 as the values to explore in our tuning.
- Learning rate: This hyperparameter could arguably be one of the most important hyperparameters to tune. A small learning rate could take too long to train and be stuck in local minima, while an overly huge learning rate could result in oscillations and in some cases, divergence. Thus, we consider two different values, 0.001, and 0.01.

The final results of this hyperparameter tuning will be explored in Section 4.1.

3. Evaluation Metrics

The goal of an evaluator is to compare characteristics of different word embedding models with a quantitative and representative metric. A good word embedding evaluator should be comprehensive, computationally efficient, have robust testing data and sufficient statistical significance to compare the performances of different word embedding models (Wang et al., 2019). Here, we employ a combination of intrinsic and extrinsic evaluation methods to assess the quality of the generated word embeddings. In this section, we introduce the metrics employed to test the performance of our model, and results will be discussed in 4.2 and 4.3.

3.1 Intrinsic Evaluation

Intrinsic evaluators measure the quality of word embeddings independent of specific natural language processing (NLP) tasks. Instead, they measure syntactic or semantic relationships like its ability to capture relationships and similarities between words (Wang et al., 2019). These evaluations are conducted in a controlled environment, often involving tasks that measure how well the embeddings align with human intuitions about word meanings.

3.1.1 Word Similarity

Word similarity tasks are conducted using subsets of word pairs, including synonyms and antonyms, carefully selected from existing datasets that contain human-judged similarity scores such as MEN (Bruni et al., 2014), SimLex (Hill et al., 2014) and WordSim (Finkelstein et al., 2001). However, we have to carefully select the words to compare similarity, since it could be possible that there are some words that do not exist in our models' vocabulary. The cosine similarity between the corresponding vectors of these word pairs is then calculated before utilising the Spearman's rank correlation coefficient to quantify the correlation between the computed similarities and human judgement (Ailem et al., 2018). This metric allows us to gauge how well the embeddings align with human intuitions about word meanings.

3.1.2 Concept Categorisation

Concept categorization is another intrinsic evaluation metric that examines how well word embeddings can capture and differentiate between distinct semantic categories. In this approach, sets of words belonging to predefined categories are selected and each word in the selected categories is represented by its corresponding embedding vector. The goal is to split a given set of words into different categorical subsets of words (Wang et al., 2019). By employing indicators such as V-Measure and Adjusted Rand Index (ARI), we can assess the degree to which the embeddings cluster similar concepts together. A successful embedding model should show shorter average distances among words within the same category and longer distances between words from different categories (Kozlowski et al., 2019).

3.2 Extrinsic Evaluation

Extrinsic evaluations measure the contribution of a word embedding model to a specific task. Word embeddings are used as input features to a downstream task and measure changes in performance metrics specific to that task (Schnabel et al., 2015). Based on the definition of extrinsic evaluators, any NLP downstream task can be chosen as an extrinsic evaluation method (Wang et al., 2019).

3.2.1 Sentiment Analysis

In sentiment analysis, word embeddings serve as input features which enable models to determine the sentiment expressed in text, such as reviews (Marreddy & Mamidi, 2023). By converting words into dense vector representations that capture semantic meanings and relationships, embeddings allow the model to recognize sentiment patterns more effectively. As such, we consider using the IMDB reviews dataset, and sample 20000 rows in total, 10000 positives and 10000 negatives. During training, a Logistic Regression classifier learns to associate specific patterns in the embeddings with sentiment labels using a labelled dataset followed by the evaluation of the performance of the model on a separate test set using metrics like the F1 score (Schnabel et al., 2015). The use of word embeddings significantly enhances the model's ability to understand nuances in language, leading to improved sentiment classification and highlighting their critical role in natural language processing applications.

3.2.2 Named-entity Recognition (NER)

Using NER as an extrinsic evaluator for word embeddings involves assessing how embeddings impact the performance of NER tasks. The NER task is widely used in NLP, focusing on recognising information units such as names including person, location, and organisation and numeric expressions such as time and percentage (Wang et al., 2019). This process begins with generating word embeddings using Bag of Words (BoW) (Qader et al., 2019), followed by preparing a labelled NER dataset like the CoNLL-2003 dataset containing annotated entities (Sang & De Meulder, 2003). Both baseline models using traditional features without the benefit of word embeddings and the word embedding model are trained on the same dataset. Performance is then evaluated and compared using metrics such as precision and recall.

4. Results and Discussion

In this section, we discuss the results of our hyperparameter tuning, as well as some intrinsic and extrinsic evaluation of our word embedding model. The intrinsic evaluators focus on the ability of the embeddings to capture semantic relationships and categorical distinctions while the extrinsic evaluators assess the model's performance in real-world applications, such as sentiment analysis and NER. We aim to provide a comprehensive understanding of the embeddings' effectiveness and their practical implications in NLP tasks through employing a combination of both evaluators (Hailu et al., 2020).

4.1 Hyperparameter Tuning Results

Now that we have conducted our hyperparameter tuning across a few hyperparameters, as well as different sizes of the dataset and model architecture, we can progress to the next step of our model testing. The models for different sizes of the dataset as well as different lambda layers were named as follows: model_full, model_half, and model_quarter for the models incorporating the standard averaging across the embeddings, with the suffixes denoting the size of dataset used. For the models with weighted averaging, they were named weighted_model_full, weighted_model_half and weighted_model_quarter. The results of this tuning are in the table below:

Dataset & Model	Hyperparameters				Validation Top K Accuracy				Time Taken (s)
	Batch Size	Embedding Dimension	Learning Rate	Number of Epochs	K = 1	K = 3	K = 5	K = 10	
100% Standard Average	64	50	0.001	10	0.0919	0.1341	0.163	0.2616	1131.51
	64	50	0.001	20	0.0997	0.1495	0.1729	0.2449	1125.44
	64	50	0.01	10	0.0956	0.154	0.1883	0.2142	1713.42
	64	50	0.01	20	0.1041	0.1699	0.1938	0.2285	1416.54
	64	100	0.001	10	0.0914	0.1362	0.1587	0.2593	1256.02
	64	100	0.001	20	0.1044	0.1587	0.1811	0.2525	1251.25
	64	100	0.01	10	0.0971	0.1534	0.1713	0.1805	1304.16
	64	100	0.01	20	0.0944	0.1428	0.1541	0.1605	1634.03
	128	50	0.001	10	0.0996	0.1496	0.1726	0.2611	1281.7
	128	50	0.001	20	0.1041	0.1582	0.1792	0.2554	1285.47
	128	50	0.01	10	0.1055	0.1688	0.194	0.2417	1087.37
	128	50	0.01	20	0.1043	0.1634	0.1838	0.2207	1532.36
	128	100	0.001	10	0.0769	0.1114	0.1574	0.2521	712.84
	128	100	0.001	20	0.0863	0.1265	0.1563	0.251	713.48
	128	100	0.01	10	0.0977	0.1581	0.1932	0.255	485.25
	128	100	0.01	20	0.1031	0.1692	0.202	0.2527	732.59
50% Standard Average	64	50	0.001	10	0.081	0.1185	0.1547	0.2476	526.79
	64	50	0.001	20	0.1018	0.1572	0.1788	0.2515	655.88
	64	50	0.01	10	0.0985	0.1565	0.1907	0.2535	393.29
	64	50	0.01	20	0.0996	0.1639	0.1934	0.2359	653.83
	64	100	0.001	10	0.102	0.1573	0.179	0.2513	587.24
	64	100	0.001	20	0.102	0.1569	0.1788	0.2487	585.83
	64	100	0.01	10	0.0934	0.1519	0.1788	0.219	443.2
	64	100	0.01	20	0.1004	0.1636	0.1865	0.2113	744.09
	128	50	0.001	10	0.0761	0.1096	0.1554	0.2498	398.13
	128	50	0.001	20	0.0977	0.1487	0.1715	0.2492	598.52
	128	50	0.01	10	0.0998	0.159	0.1906	0.2543	401.34
	128	50	0.01	20	0.098	0.1547	0.1787	0.2129	598.84
	128	100	0.001	10	0.1021	0.1574	0.181	0.2428	544.92
	128	100	0.001	20	0.0977	0.1492	0.1739	0.2497	544.06
	128	100	0.01	10	0.1006	0.1578	0.1826	0.2409	550.91
	128	100	0.01	20	0.1027	0.1695	0.1985	0.2429	666.04
25% Standard Average	64	50	0.001	10	0.095	0.1479	0.1736	0.2545	462.1
	64	50	0.001	20	0.096	0.1485	0.1737	0.2468	402.42
	64	50	0.01	10	0.0992	0.167	0.1978	0.2229	346.49
	64	50	0.01	20	0.0883	0.1405	0.1667	0.2317	232.96
	64	100	0.001	10	0.0881	0.1342	0.1564	0.2574	384.41
	64	100	0.001	20	0.1004	0.157	0.1815	0.2536	383
	64	100	0.01	10	0.0909	0.1496	0.1772	0.2223	258.07
	64	100	0.01	20	0.0846	0.1372	0.1559	0.189	323.46
	128	50	0.001	10	0.0985	0.1551	0.1789	0.2553	433.45
	128	50	0.001	20	0.0876	0.134	0.1559	0.2609	389.29
	128	50	0.01	10	0.0995	0.1614	0.1878	0.2293	215.16
	128	50	0.01	20	0.1015	0.1702	0.2005	0.2337	259.05
	128	100	0.001	10	0.1001	0.1563	0.1796	0.2572	328.06
	128	100	0.001	20	0.0999	0.1564	0.1822	0.2572	342.47
	128	100	0.01	10	0.1062	0.1779	0.2103	0.2497	358.37
	128	100	0.01	20	0.0967	0.1565	0.1957	0.2425	205.88

100% Weighted Average	64	50	0.001	10	0.1252	0.2321	0.2937	0.3861	1363.85
	64	50	0.001	20	0.1247	0.2316	0.2933	0.3857	1382.85
	64	50	0.01	10	0.1173	0.2202	0.2805	0.3711	559.36
	64	50	0.01	20	0.1159	0.2193	0.2805	0.3712	552.77
	64	100	0.001	10	0.1286	0.2384	0.302	0.3964	1526.19
	64	100	0.001	20	0.1291	0.2388	0.3023	0.3963	1539.94
	64	100	0.01	10	0.1162	0.2179	0.2781	0.3689	622.67
	64	100	0.01	20	0.116	0.2183	0.2787	0.3698	621.26
	128	50	0.001	10	0.1266	0.2345	0.2966	0.3893	2024.94
	128	50	0.001	20	0.1269	0.2349	0.2976	0.3903	2394.56
	128	50	0.01	10	0.1197	0.2244	0.2863	0.3782	400.4
	128	50	0.01	20	0.1201	0.2247	0.286	0.378	398.29
	128	100	0.001	10	0.1301	0.2406	0.3044	0.3987	1985.54
	128	100	0.001	20	0.1293	0.24	0.3043	0.3993	2213.34
	128	100	0.01	10	0.1191	0.2246	0.2863	0.3794	452.71
	128	100	0.01	20	0.1194	0.224	0.2861	0.3794	451.46
50% Weighted Average	64	50	0.001	10	0.1224	0.2276	0.2889	0.3805	1132.27
	64	50	0.001	20	0.1224	0.2274	0.2887	0.3802	1163.12
	64	50	0.01	10	0.1157	0.218	0.2779	0.3686	258.74
	64	50	0.01	20	0.1146	0.2161	0.2769	0.3684	259.86
	64	100	0.001	10	0.1255	0.233	0.2953	0.3888	1146.04
	64	100	0.001	20	0.1251	0.2328	0.2954	0.3887	1133.45
	64	100	0.01	10	0.1116	0.2132	0.2736	0.3653	287.61
	64	100	0.01	20	0.1124	0.2142	0.2752	0.3666	291.18
	128	50	0.001	10	0.1209	0.2252	0.2862	0.3773	960.43
	128	50	0.001	20	0.122	0.2278	0.2893	0.3809	1428.56
	128	50	0.01	10	0.1176	0.221	0.2814	0.3732	191.76
	128	50	0.01	20	0.1174	0.2208	0.2813	0.3724	287.61
	128	100	0.001	10	0.1239	0.2309	0.2929	0.3856	833.59
	128	100	0.001	20	0.1242	0.2307	0.2925	0.3857	835
	128	100	0.01	10	0.1173	0.221	0.2822	0.3744	206.78
	128	100	0.01	20	0.1177	0.2203	0.2815	0.3736	205.15
25% Weighted Average	64	50	0.001	10	0.1167	0.2179	0.2779	0.3684	638.04
	64	50	0.001	20	0.1175	0.2194	0.2793	0.3694	764.51
	64	50	0.01	10	0.1129	0.213	0.273	0.3632	125.98
	64	50	0.01	20	0.1128	0.2127	0.2724	0.3628	125.85
	64	100	0.001	10	0.1187	0.2202	0.2807	0.3724	401.87
	64	100	0.001	20	0.1184	0.2211	0.2816	0.3726	404.56
	64	100	0.01	10	0.1098	0.2091	0.269	0.3596	135.13
	64	100	0.01	20	0.1103	0.2094	0.2688	0.36	135.68
	128	50	0.001	10	0.1156	0.2157	0.2752	0.3656	446.58
	128	50	0.001	20	0.1155	0.2158	0.2753	0.3652	455.68
	128	50	0.01	10	0.1146	0.2145	0.2733	0.3629	141.15
	128	50	0.01	20	0.1146	0.2147	0.2738	0.3635	138.81
	128	100	0.001	10	0.1177	0.2198	0.2799	0.3718	349.77
	128	100	0.001	20	0.1179	0.2197	0.2796	0.3714	346.37
	128	100	0.01	10	0.1141	0.2146	0.2748	0.3657	98.79
	128	100	0.01	20	0.1142	0.2159	0.2757	0.3659	98.79

Although the results could be confusing, since there are four evaluation metrics, we can observe that there seems to be a positive correlation, where combinations that perform better in k = 1 accuracy tend to perform better when k increases. The total time taken is also something we have to take into account, as computational efficiency has to be a consideration. However, this is improved for a lot of the combinations, due to the early stopping algorithm. To further evaluate the models, we pick one hyperparameter combination for each of the 6 models and compare how these perform on our intrinsic and extrinsic evaluation tests.

- 100% Standard Average: We pick the following combination: batch_size = 128, embedding_dim = 100, learning_rate = 0.01, num_epochs = 20. This combination boasts the best accuracy for k = 5, as well as ranking highly for the other values of k.

However, this combination results in a relatively quick computation that increases its appeal.

- 50% Standard Average: The combination for this model is exactly the same hyperparameter combination as the model for 100% Standard Average. This combination for this model outperforms all other combinations when k = 1, k = 3 and k = 5, while marginally losing out when k = 10. The time taken is also not obscene, thus making it a good choice.
- 25% Standard Average: The optimal combination for this model has been decided as: batch_size = 128, embedding_dim = 100, learning_rate = 0.01, num_epochs = 10. This combination tops the ranking for k = 1, k = 3 and k = 5, but loses out slightly when k = 10. The time taken is also relatively fast, solidifying this combination as the best choice for this model.
- 100% Weighted Average: We pick the following combination: batch_size = 128, embedding_dim = 100, learning_rate = 0.001, num_epochs = 10. This has the best accuracy when k = 1, k = 3 and k = 5, but slightly loses out by 0.006 when it comes to k = 10. However, this combination is about 4 minutes faster than the other combination which performed the best for k = 10. Thus, we decide that the loss in accuracy for k = 10 is a worthwhile sacrifice for the increased computational efficiency.
- 50% Standard Average: This model's optimal combination is: batch_size = 64, embedding_dim = 100, learning_rate = 0.001, num_epochs = 10. This combination outperforms all the other combinations across all 4 metrics. Although the total time taken for this is on the longer side, it is a necessary downside we must accept.
- 25% Weighted Average: We pick the following combination: batch_size = 64, embedding_dim = 100, learning_rate = 0.001, num_epochs = 20. This combination was chosen as the best as it edged its counterpart, with num_epochs = 10, for k = 3, k = 5 and k = 10, while slightly losing out when k = 1. The time taken is also very similar, but the combination with num_epochs = 20 is slightly better due to performing better in more metrics.

It is interesting to note that for the standard average models, the best hyperparameter combinations are very similar, with only num_epochs varying slightly. The dimension of the embedding layer is consistently picking 100 as the optimal value. The learning rate is also consistently 0.01 for the standard average models while 0.001 for the weighted average models. Another interesting observation is that for the weighted average models, a learning rate of 0.01 almost always triggers the early stopping mechanism at the second epoch, meaning that the model is likely to be oscillating and will not reach the minima.

Model	K = 1 accuracy	K = 3 accuracy	K = 5 accuracy	K = 10 accuracy	Time taken (s)
model_full	0.0990	0.1574	0.1834	0.2398	776.29
model_half	0.1030	0.1720	0.2034	0.2502	436.35
model_quarter	0.1050	0.1780	0.2123	0.2514	271.45

weighted_model_full	0.1297	0.2403	0.3041	0.3989	1458.17
weighted_model_half	0.1253	0.2326	0.2953	0.3882	833.07
weighted_model_quarter	0.1166	0.2191	0.2788	0.3691	298.31

Surprisingly, the test results have shown an increase in test accuracies as the dataset decreases in size for the normal model, but the converse happens across the weighted models. The accuracies for the weighted models are also much higher than those of the normal models, which corroborates our intuition. We can conclude that the model can learn more when context words are given differing weights, and it remains to investigate the optimal dataset size for implementing such a model. The accuracies are higher for a larger dataset but computational inefficiencies are a drawback.

4.2 Intrinsic Evaluation Results

Since our model is trained on a less intensive and limited dataset, we can expect that it will not perform as well as other pre-trained models out there. Therefore, most of the cosine similarities are not evident of their true supposed relationship. However, we can see from the results that the models are learning some information. The model that performs the best is, surprisingly, model_quarter with a correlation of 0.48, with weighted_model_half performing the worst across the 6 models, having a correlation of 0.0486. This goes against intuition, as it was expected that the more information the model is trained on, the better it would perform. In addition, we would expect the weighted models to perform better, since the architecture of the weighted models are a better representation of word embeddings. There are some word pairs that are being predicted very poorly by all the models, such as ‘alligator’ and ‘viper’, which although do not seem very similar, are both reptiles. However, none of the models produced a cosine similarity exceeding 0.2, with some even producing negative similarities. This could be due to the words appearing scarcely throughout the training dataset, thus little information is learnt about these words.

However, our models perform better when we evaluate with concept categorisation. V-Measure evaluates homogeneity and completeness of clusters, with values ranging from 0 to 1. ARI compares similarity between predicted clusters and true categories, ranging from -1 to 1. The best performing model with this evaluator is weighted_model_full, boasting a V-Measure score of 0.703 and ARI of 0.45. The worst performing model is model_full, with a V-Measure score of 0.35 and ARI of -0.0922. This means that our well-performing models can cluster similar words together, although the predicted clusters are some distance off the true clusters. The poorer-performing models are not clustering the words well, and clustering is similar to random assignment, thus showing that insufficient information about the words is learnt during model training.

Overall, it shows that there is potential in our word embedding models, and it is likely that with more experimentation and understanding about our dataset, we can train and produce models that can perform consistently well across all the evaluators.

4.3 Extrinsic Evaluation Results

In our sentiment analysis task, we trained a logistic regression classifier using 20000 labelled reviews from the IMDB dataset, 10000 each pertaining to positive and negative sentiment. These reviews were randomly sampled and both of our models were trained and tested on the dataset, to see how it performed with sentiment analysis. From our results, the best performing model was weighted_model_full, achieving an accuracy of 71.2% and an F1 score of 71%, with model_full coming a close second with 70.5% accuracy and F1 score of 70%. The worst model was model_half, having an accuracy of 50.7%. An interesting observation was that model_half struggled with predicting positive sentiment, with the model only correctly predicting 6% of all positive reviews in the test dataset, while correctly predicting 95% of all negative reviews. This most likely represents an inherent bias in model_half, making it more likely to classify the review as negative rather than positive. However, there is still potential in our models, and this further highlights the advantages of word embeddings in capturing semantic nuances, its richer and more informative representations of words compared to traditional methods.

In our NER study using word embeddings, we trained a model on the CoNLL-2003 dataset and all of our models achieved good results, with model_full and model_half, the best performing models, achieving an accuracy of 82.5%. The worst performing model was weighted_model_quarter, with an accuracy of 82.1%. This demonstrates the embeddings' ability to capture semantic relationships and context. It is likely that with further analysis and improvement in our model training, we can achieve better scores, similar to other models out there.

However, error analysis for both sentiment analysis and NER tasks revealed challenges with sarcasm, mixed sentiments and ambiguous names that could belong to multiple categories, suggesting that context can still pose difficulties for the word embedding model. These results are useful in helping us understand the effectiveness of word embeddings in sentiment analysis while identifying potential areas for improvement through advanced techniques like contextual embeddings and domain-specific training.

5. Conclusion

Word embeddings have proven to be a transformative approach in NLP as it enables machines to not only understand but represent the nuances of human language more effectively. Their ability to represent words in a continuous vector space allows for rich, informative representations that significantly outperform traditional methods like one-hot encoding. By capturing semantic relationships and contextual meanings, word embeddings facilitate various applications, from sentiment analysis to information retrieval and machine translation (PingCAP, 2024).

There is much potential for further advancements in word embeddings. Researchers are actively exploring ways to incorporate additional context, including domain-specific knowledge, to enhance the quality of embeddings for specialised applications. Leveraging on contextual embeddings from models like BERT and GPT opens new avenues for nuanced language understanding, allowing embeddings to adapt based on surrounding context (Alshattnawi et al., 2024). This adaptation leads to improved performance and more accurate representations, paving the way for more effective applications in natural language processing.

As the field of NLP continues to evolve, word embeddings will play a crucial role in developing more sophisticated models that can better understand and generate human language (Alshattnawi et al., 2024). By integrating advanced techniques and utilising the vast amount of data, the future of word embeddings promises to yield even greater insights and capabilities in natural language understanding and generation.

6. References

Alshattnawi, S., Shatnawi, A., AlSobeh, A. M., & Magableh, A. A. (2024). Beyond Word-Based model embeddings: Contextualized representations for enhanced social media spam detection. Applied Sciences, 14(6), 2254. https://doi.org/10.3390/app14062254

Blankenship, A., Connell, S., Dombrowski, Q. (2024). "Understanding and Creating Word Embeddings," Programming Historian 13 https://doi.org/10.46430/phen0116

Bruni, E., Tran, N. K., & Baroni, M. (2014). Multimodal distributional semantics. Journal of Artificial Intelligence Research, 49, 1–47. https://doi.org/10.1613/jair.4135

Elastic (n.d.) What are Word Embeddings? https://www.elastic.co/what-is/word-embedding

Enozeren (2024). Word2Vec from Scratch. https://medium.com/@enozeren/word2vec-from-scratch-with-python-1bba88d9f221

Finkelstein, L., Evgeniy Gabrilovich, Matias, Y., Rivlin, E., Solan, Z., Wolfman, G., & Eytan Ruppin. (2001). Placing search in context. The Web Conference. https://doi.org/10.1145/371920.372094

Hailu, T. T., Yu, J., & Fantaye, T. G. (2020). Intrinsic and extrinsic automatic evaluation strategies for paraphrase generation systems. Journal of Computer and Communications, 08(02), 1–16. https://doi.org/10.4236/jcc.2020.82001

Hill, F., Reichart, R., & Korhonen, A. (2014). SimLex-999: Evaluating Semantic Models with (Genuine) Similarity Estimation. arXiv (Cornell University). https://doi.org/10.48550/arxiv.1408.3456

Jeeva, C. (2024). Continuous Bag of Words (CBOW) Model in NLP. https://www.scaler.com/topics/nlp/cbow/

Kozlowski, A. C., Taddy, M., & Evans, J. A. (2019). The Geometry of Culture: Analyzing the Meanings of Class through Word Embeddings. American Sociological Review, 84(5), 905–949. https://doi.org/10.1177/0003122419877135

Marreddy, M., & Mamidi, R. (2023). Learning sentiment analysis with word embeddings. In Elsevier eBooks (pp. 141–161). https://doi.org/10.1016/b978-0-32-390535-0.00011-2

PingCAP. (2024, July 16). Understanding embedding models for better machine learning. TiDB. https://www.pingcap.com/article/understanding-embedding-models-for-better-machine-learning/

Qader, W. A., Ameen, M. M., & Ahmed, B. I. (2019). An Overview of Bag of Words;Importance, Implementation, Applications, and Challenges. 2019 International Engineering Conference (IEC). https://doi.org/10.1109/iec47844.2019.8950616

Sang, E. F. T. K., & De Meulder, F. (2003). Introduction to the CoNLL-2003 shared task. CNTS - Language Technology Group University of Antwerp. https://doi.org/10.3115/1119176.1119195

Vu, K. (2021). Text preprocessing methods for deep learning. dzone.com. https://dzone.com/articles/text-preprocessing-methods-for-deep-learning

Babel Street (2024). Lemmatization in language processing. https://www.babelstreet.com/blog/what-is-lemmatization-learn-why-this-process-is-vital-to-language-processing

Sarkar, D. (2018). Implementing Deep Learning Methods and Feature Engineering for Text Data: The Continuous Bag of Words (CBOW) https://www.kdnuggets.com/2018/04/implementing-deep-learning-methods-feature-engineering-text-data-cbow.html

Simha, A. (2021). Understanding TF-IDF for Machine Learning | Capital One. https://www.capitalone.com/tech/machine-learning/understanding-tf-idf/

Riva, M. (2021). Top-N Accuracy Metrics. https://www.baeldung.com/cs/top-n-accuracy-metrics

Vishwakarma, N. (2024). What is Adam Optimizer? https://www.analyticsvidhya.com/blog/2023/09/what-is-adam-optimizer/#:~:text=Advantages%20of%20using%20Adam%20Optimizer,-Fast%20Convergence%3A%20Adam&text=Low%20Memory%20Usage%3A%20Low%20memory,without%20extensive%20hyperparameter%20tuning%20experience.

Devansh (2024). How does Batch Size impact your model learning? https://medium.com/geekculture/how-does-batch-size-impact-your-model-learning-2dd34d9fb1fa

Schnabel, T., Labutov, I., Mimno, D., & Joachims, T. (2015). Evaluation methods for unsupervised word embeddings. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. https://doi.org/10.18653/v1/d15-1036

Stryker, C., Holdsworth, J. (2024). What is NLP (natural language processing)? https://www.ibm.com/topics/natural-language-processing

Tensorflow. Word Embeddings. https://www.tensorflow.org/text/guide/word_embeddings

Texts and Images. Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, 1478–1487. https://doi.org/10.18653/v1/d18-1177

Van Otten, N. (2024). Co-occurrence Matrices Explained: How To Use Them In NLP, Computer Vision & Recommendation Systems [6 Tools]. https://spotintelligence.com/2024/04/04/co-occurrence-matrices/

Wang, B., Wang, A., Chen, F., Wang, Y., & Kuo, C.-C. J. (2019). Evaluating word embedding models: methods and experimental results. SIP, 8, 1–14. https://doi.org/10.1017/ATSIP.2019.12