DETECTING THE ODD-ONE-OUT AMONG WORDS WITH EMBEDDINGS

Sükrü Han Sahin, Carlos Parra Marcelo

Technical University of Denmark
s192136@student.dtu.dk, s192770@student.dtu.dk

ABSTRACT

The present research is motivated by the need of technical analysis of the word-embedding models' performance when facing word intrusion tasks. We compare the results of different models with the aim of figure out which one is the most adequate for detecting outliers within a small group of words in English. We try to detect the problems they have to face during this task, from different perspectives, and we implement and propose possible solutions to these problems. We found that gensim FastText model performs very well due to its process for building word vectors based on n-grams, and that it can be improved taking a closer look at some special cases like double words.

1. INTRODUCTION

The semantic analysis is very present in our lives. When we are writing to a friend by the chat the spelling checker suggests to us some words semantically related to the sentence we are writing. This is just one example, but there are plenty of them.

In this report we analyze and compare the performance of different word-embedding models in order to detect an outlier within a small group of words. The main aim of the project is to provide an overview of which are the advantages and disadvantages of each of the models when they try to detect which word in a group is the most different one taking into account the semantics, providing a good source to improve the current methods or create new ones.

2. DATASETS

At the beginning of the project, it was necessary to create a dataset in English composed by cases of 4 words, where the fourth is an outlier and the other three are semantically related words. Later, during the development of the project, it was useful to specify the type of odd-one-out word case in order to extract some characteristics from the different models. We classified the cases as homographs (each of two or more words spelled the same but not necessarily pronounced the same and having different meanings and origins), proper nouns (words referring to an entity), and double words (two words expressing one single meaning). As a result of this process, we got four different datasets: the *Main Dataset*, which is a data frame containing all the cases from the rest of the datasets and other ordinary examples; the *Homographs* data frame, in where each case has one word which has semantic relation with the outlier; the *Proper Nouns* dataset, containing at least one word starting with a capital letter for each of the cases; and the *Double Words* dataset, with cases containing at least one double word. **Table 1** depicts the shapes of the mentioned datasets.

3. WORD-EMBEDDING MODELS

We have been working with three different word-embedding models for this project: FastText, Wembedder and BERT.

First, we started working with FastText, an open-source, free, lightweight library that allows users to learn text representations and text classifiers. Each word is represented as a bag of character n-grams in addition to the word itself, so for example, for the word matter, with n = 3, the fastText representations for the character n-grams are <ma, mat, att, tte, ter, er>. < and > are added as boundary symbols to distinguish the ngram of a word from a word itself, so for example, if the word mat is part of the vocabulary, it is represented as <mat>. This helps preserve the meaning of shorter words that may show up as ngrams of other words. Inherently, this also allows you to capture meaning for suffixes/prefixes [1].

Gensim is an open-source library for unsupervised topic modeling and natural language processing [2]. We have used Gensim to work with the FastText model in Python, one of the pre-trained word vector models distributed by Facebook [3] with data from *Common Crawl* and *Wikipedia*. This English model was trained using CBOW (Continuous Bag Of Words) with position-weights, in dimension 300, with character n-grams of length 5, a window of size 5 and 10 negatives.

Wembedder [4] uses the Word2Vec model in the Gensim program, which was trained with the CBOW training algorithm, an embedding dimension on 100, a window of 1 and a minimum count of 20 (any "word" must appear 20 times or more in the training dump to be included in the model). But the main characteristic of this model is that it is trained with a dump from Wikidata (property of The Wikimedia Foundation), which contains the triples (sentences of three "words" which can be treated by standard word embedding implementations) [5].

In addition to the previous models, we made some unsuccessful tries with BERT model. We had several problems with the *bert-serving-client* library [6] for Python, but we also tried with a different BERT implementation using Pytorch [7]. Finally, after some research, we found that word-level similarity comparisons are not appropriate with BERT embeddings because these embeddings are contextually dependent, meaning that the word vector changes depending on the sentence it appears in. This fact discarded the possibility of using BERT model for our case of study in order to get good results (but also for a proper implementation).

4. DESCRIPTION OF METHODS

Before explaining the performance of each of the models in the case of study, it is necessary to provide some information about the methods mentioned in **Table 2** (in the appendix).

'*Fasttext*' model is a direct implementation of the `doesnt_match` function from Gensim library and FastText model. '*Wembedder*' model is a direct implementation of the `doesnt_match` function from the Wembedder model, but we first look for the first Q-identifier associated to the word with a query to the Wikidata API and we use it to get the word vector. '*Fasttext + Wembedder*' model is the result of '*Fasttext*' and '*Wembedder*' combined together. '*Upper Case*' indicates the number of words starting with a capital letter within a row and implies that row is evaluated with the '*Wembedder*' model if it has that specific number of words starting with capital letters. Otherwise, the row is evaluated by '*Fasttext*' model. '*Our Doesn't Match (Cosine)*' has the same calculations of FastText's `'doesnt_match'` function, but it is implemented by ourselves with a different logic. With this implementation, for each word, the cosine distance between the word vector and the mean vector resulting from all word vectors in the row are calculated. The word which has the highest value of cosine distance will be identified as the outlier. If we take a look at the logic behind the FastText `doesnt_match` function we find that, for each word, the cosine similarity between the word vector and the mean vector resulting from all word vectors in the row are calculated. The word which has the lowest value of cosine similarity is identified as the outlier. From that explanation, the equation below reveals itself [8]:

$$CosineDistance = 1 - CosineSimilarity$$

$$CosineSimilarity = \cos(\alpha) = [\frac{A.B}{\|A\|\|B\|}]$$

'*Our Doesn't Match (Correlation)*' is also implemented by ourselves. In this case, the logic behind the function is that, for each word, the correlation distance between the word vector and the mean vector resulting from all word vectors in the row is calculated. The word with the highest correlation distance will be identified as the outlier. We obtain the correlation distance with the formula mentioned below:

$$CorrelationDistance = 1 - [\frac{(A-\hat{A}).(B-\hat{B})}{\|A\|\|B\|}]$$

where Â is the mean of the elements of A and B̂ is the mean of the elements of B. If we normalize the attribute vectors by subtracting their mean (e.g., $A - \hat{A}$), the measure is called the centered cosine similarity and it is equivalent to the Pearson correlation coefficient [9]. Check **Figure 1** and **Figure 2** to see the differences between the normalized and the not normalized vector representations. If they are normalized, they are more uniformly distributed rather than grouped in the middle.

In all of our models, the vectors are obtained after being normalized. While the correlation distance is calculated, the cosine distance is also being calculated. Therefore, it is perfectly appropriate to have the same results for both '*Our Doesn't Match (Correlation)*' and '*Our Doesn't Match (Cosine)*' functions.

If we continue taking a look to the **Table 2**, the '*Our Doesn't Match for Double Words (with Averaging)*' function is implemented by ourselves one more time, using the same logic of the '*Our Doesn't Match (Cosine)*' function. The only difference between them is: if there is any word in a row composed of two words (as we call double words), then the word vectors of each word within the double word are summed and divided by two, which means we compute their average vector. Then, the same principles of '*Our Doesn't Match (Cosine)*' function are applied.

The '*Our Doesn't Match for Double Words (with Summing)*' function is implemented by ourselves with the same logic of the '*Our Doesn't Match (Cosine)*' function. But there is one difference between both: if there is any double word in a row, then the word vectors of each word being part of the double word are summed.

One of the last functions, the '*Our Doesn't Match (with Averaging)*' one, is implemented by ourselves again with the closely related logic of the '*Our Doesn't Match (Cosine)*' function. Via using Wikidata lexemes [10], all forms of each word in all rows are obtained. After that, an average word vector is created for each word by averaging word vectors of all word forms. Then, the cosine distance between the average word vector and the mean vector of all average word vectors are calculated. The word which has the highest value of cosine distance will be identified as the outlier.

Finally, the '*Dimesion Reduction (PCA)*' and the '*Dimesion Reduction (t-SNE)*' functions use the same logic but with one big difference: the dimension reduction application method. The process can be described like this: an average word vector is created for each word by averaging the word vectors of all word forms; then, the dimension reduction is applied to each average word vector getting a two-dimensional array for each vector (usually with FastText each word vector has a shape of (300,1) array); finally, the cosine distance is calculated and the word having the highest value will be identified as the outlier.

Before concluding this section, it is necessary to introduce the two-dimension reduction application methods we have just mentioned: t-SNE is the abbreviation for t-distributed stochastic neighbor embedding. It is a nonlinear dimension reduction technique well-suited for embedding high-dimensional data to visualize it in a low-dimensional space of two or three dimensions. [11]. Principal Component Analysis (commonly abbreviated as PCA) is one of the most popular dimension reduction techniques applied. PCA is a search for the best fitting line, which can be explained as: "Given a collection of points in two, three, or higher-dimensional space, a best-fitting line can be defined as one that minimizes the average squared distance from a point to the line" [12].

These aforementioned techniques are applied to reduce dimension and visualize the word vectors within the plots for each of the models. PCA did a better job of visualizing the vectors since t-SNE fits better for non-linear problems. These techniques have been used in '*Dimesion Reduction (PCA)*' and '*Dimesion Reduction (t-SNE)*' functions to find out the outliers, but the results are not as satisfactory as others (we will check it in next sections). Check **Figure 3** and **Figure 4** to see the difference between PCA and t-SNE representations. PCA is a better technique to reduce dimensions and represent the word vectors. For *father*, *brother*, *sister* and *iron*, the average vectors of among all of its forms are calculated and represented. For example, *father_mean* is the mean vector of all forms of *father*.

5. PERFORMANCE AND RESULTS

This is the most important section of the report and to understand it properly is very important to check the **Table 2**, because it contains the results mentioned. We can get the best performance detecting the outliers using the '*Our Doesn't Match for Double Words (with Summing)*' function for all the datasets, with an accuracy of 89,05% for the *Main Dataset*. As we mentioned previously, this function uses the cosine distance to check how far is each word vector from the mean vector resulting from all word vectors within the 4 words group. But it improves the 83.08% accuracy of the '*Our Doesn't Match (Cosine)*' by almost 5% more because of the special case of the double words. In fact, we can check this just taking a look at the accuracy of both models with the *Double Words* dataset (where the new implementation for the double word cases allows us to detect the outlier always instead of never). Moreover, it is interesting to see how the '*Our Doesn't Match for Double Words (with Summing)*' function slightly improves the results of the '*Our Doesn't Match for Double Words (with Averaging)*' function (by almost 1% more). This is due to how we calculate the final word vector using the two ones coming from the single words composing the double words: just making the sum of both vectors instead averaging them is more effective (remember that these vectors have 300 dimensions).

About the rest of the results, we can find some interesting facts as follows: We have improved the accuracy of FastText (the model reporting better results without modifications) by almost a 6%. This is due to the methods we created to handle the calculation of the distances between the word vectors; Wembedder performs better with proper nouns (61,9%) than with the rest of special cases (24,39% with homographs and 33,33% with double words). Its performance is very discreet compared to FastText; When using FastText and Wembedder combined it is better to switch to Wembedder if the row contains more than one proper nouns; It is necessary to manage the double word cases in a different way with the word vectors because only the methods doing it are getting good results with these cases; Applying dimension reduction to the word vectors affects significantly to the method's performance. We loose a lot of relevant information using this process since we reduce dimensions. '*Our Doesn't Match (with Averaging)*' yielded a worse performance by roughly 5% when compared with Fasttext.

6. CONCLUSIONS

After reviewing the results in the previous section, we can make some conclusions about the different models and methods we have used in this report. FastText word vectors are built from vectors of substrings of characters contained in it, so the overall word embedding is a sum of these character n-grams. The advantage of this approach generates better embeddings for rare and out-of-corpus words [13], so that is why we got a very good performance with this model from the beginning.

Wembedder is trained with the Wikidata lexemes and the method we used looks for the first Q-identifier associated with the word with a simple query to the Wikidata API. That is the main reason to explain why this model behaves well with proper nouns but very bad with the homographs and **Figure 5** and **Figure 6** explain it. In **Figure 6**, the month-related words are on the same axis and close together but the outlier word *Juliette* is far away from them. In **Figure 5**, *train* is close to plane but far from *exercise*. This also explains why the combination FastText+Wembedder works slightly better with the cases with more than one proper noun. Moreover, the pre-trained model sometimes does not include all the existing words in its vocabulary (it needs to maintain a balance between the performance and the memory requirements). When this occurs, the method runs the `doesn_match` function without the missing Q-identifiers and the accuracy drops significantly (or completely if the Q-identifier missing is the outlier).

If we check again **Table 2**, it is easy to see that FastText by itself is having problems to build a proper vector for double words. This is because the model is trained only with single words (although some double words can be used while training the model with an underscore, like *in_front*, becoming a single token) [14]. In **Figure 7**, the representations of *in*, *front* and the average vector of *in* and *front* (which is named as *in front* in the figure) are created with PCA. When we added a different implementation for the double words the accuracy with the *Double Words* dataset increased from 0% to 100%.

7. FUTURE WORK

From the conclusions of the report we can extract some interesting hints for future developments. It would be interesting to check with proper results that the BERT model is not adequate for this word intrusion task, but also look for other word-embedding models and analyze their characteristics. In the case of Wembedder, one of the big problems (as we mentioned in the conclusions) is its limited vocabulary. It would be interesting to train a model with different parameters, allowing more words to be included in the vocabulary and check how it affects the final performance. This can also be applied in similar ways to the rest of the models.

8. IMPLEMENTATION OF METHODS

All of the aforementioned models are implemented in Python on the Jupyter Notebook environment. Link to the Jupyter Notebook: Code Implementation

9. REFERENCES

[1] Nishan Subedi, "Fasttext: Under the hood," Available at *https://towardsdatascience.com/fasttext-under-the-hood-11efc57b2b3.*

[2] Selva Prabhakaran, "Gensim tutorial," Available at *https://www.machinelearningplus.com/nlp/gensim-tutorial.*

[3] Facebook Inc., "Word vectors for 157 languages," Available at *https://fasttext.cc/docs/en/crawl-vectors.html.*

[4] Finn Årup Nielsen, "Wembedder," Available at *https://github.com/fnielsen/wembedder.*

[5] Finn Årup Nielsen, "Wembedder: Wikidata entity embedding web service," pp. 1 – 2.

[6] Han Xiao, "bert-serving-client 1.10.0," Available at *https://pypi.org/project/bert-serving-client.*

[7] Chris McCormick and Nick Ryan, "Bert word embeddings tutorial," Available at *https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial.*

[8] Selva Prabhakaran, "Cosine similarity," Available at *https://www.machinelearningplus.com/nlp/cosine-similarity.*

[9] Alboukadel Kassambara, "Clustering distance measures," Available at *https://www.datanovia.com/en/lessons/clustering-distance-measures.*

[10] Wikimedia Foundation, "Wikidata query service," Available at *https://query.wikidata.org.*

[11] scikit-learn developers, "sklearn.manifold.tsne," Available at *https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html.*

[12] scikit-learn developers, "sklearn.decomposition.pca," Available at *https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.*

[13] Jerry Liu, "How does fasttext (facebook) work? is there any research paper or blog post explaining the theory behind its working?," Available at *https://www.quora.com/How-does-fastText-Facebook-work-Is-there-any-research-paper-or-blog-post-explaining-the-theory-behind-its-working.*

[14] Facebook Inc., "Fasttext faq," Available at *https://fasttext.cc/docs/en/faqs.html.*

A. APPENDIX

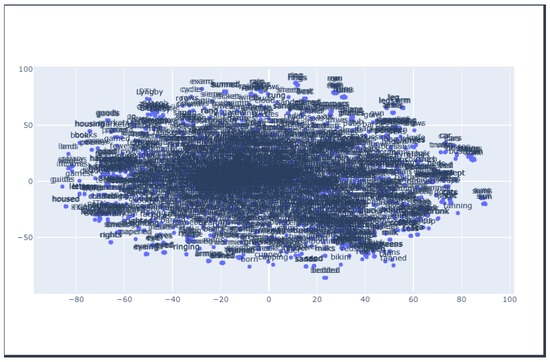

Figure 1: Not Normalized Word Vector Representation

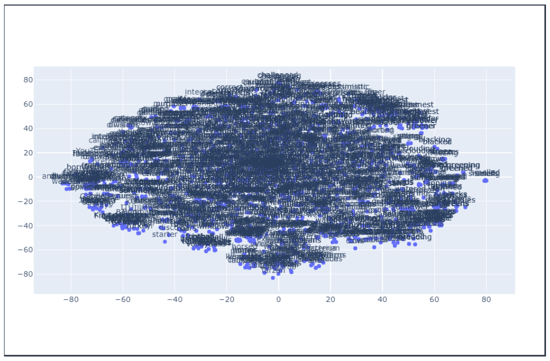

Figure 2: Normalized Word Vector Representation

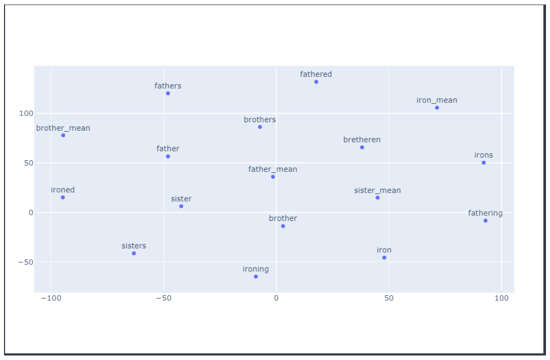

Figure 3: Dimension Reduction with t-SNE for Fasttext

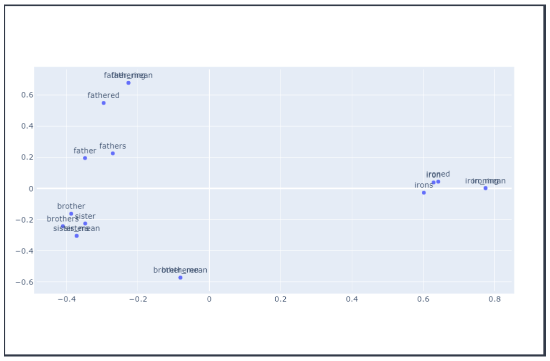

Figure 4: Dimension Reduction with PCA for Fasttext

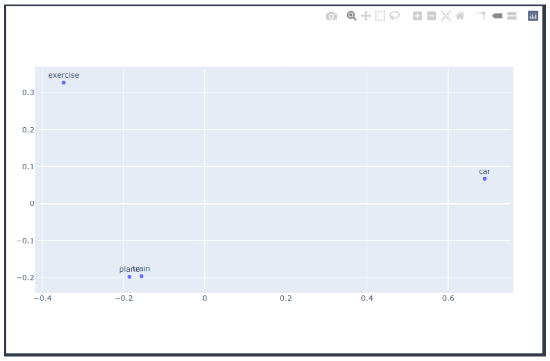

Figure 5: Dimension Reduction with PCA for Wembedder

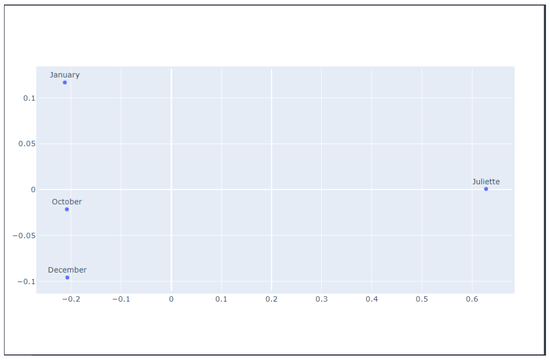

Figure 6: Dimension Reduction with PCA for Wembedder (2)

Datasets	Shapes
Main Dataset	(201, 4)
Homographs	(41, 4)
Proper Nouns	(21, 4)
Double Words	(12, 4)

Table 1. Dataset Shapes

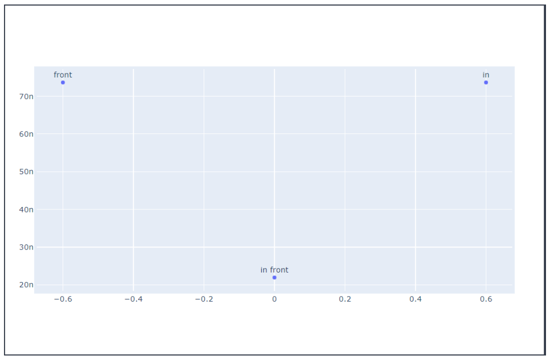

Figure 7: Dimension Reduction with PCA for Double Words

Models / Datasets	Main Dataset	Homographs	Proper Nouns	Double Words
Fasttext	*83.08%*	*75.61%*	*90.48%*	*0.00%*
Wembedder	*38.81%*	*24.39%*	*61.90%*	*33.33%*
Fasttext+Wembedder (Upper Case = 1)	*80.10%*	*75.61%*	*61.90%*	*0.00%*
Fasttext+Wembedder (Upper Case = 2)	*80.60%*	*75.61%*	*66.67%*	*0.00%*
Fasttext+Wembedder (Upper Case = 3)	*80.60%*	*75.61%*	*66.67%*	*0.00%*
Fasttext+Wembedder (Upper Case = 4)	*80.60%*	*75.61%*	*66.67%*	*0.00%*
Our Doesn't Match (Cosine)	*83.08%*	*75.61%*	*90.48%*	*0.00%*
Our Doesn't Match (Correlation)	*83.08%*	*75.61%*	*90.48%*	*0.00%*
Our Doesn't Match for Double Words (with Averaging)	*88.06%*	*78.05%*	*90.48%*	*83.33%*
Our Doesn't Match for Double Words (with Summing)	*89.05%*	*78.05%*	*90.48%*	*100.00%*
Our Doesn't Match (with Averaging)	*78.61%*	*N/A*	*N/A*	*N/A*
Dimesion Reduction (PCA)	*47.76%*	*N/A*	*N/A*	*N/A*
Dimesion Reduction (t-SNE)	*26.37%*	*N/A*	*N/A*	*N/A*

Table 2. Datasets used for empirical evaluation