7-5-2019 11:00 AM

Measuring Enrichment Of Word Embeddings With Subword And Dictionary Information

Felipe Urra, *The University of Western Ontario*

Supervisor: Mercer, Robert E., *The University of Western Ontario*
A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science
© Felipe Urra 2019

Abstract

Word embedding models have been an important contribution to natural language processing. Following the distributional hypothesis, that is "words used in similar contexts tend to have similar meanings", these models are trained on large corpora of natural text. By analyzing how words are used in context, the mapping of a word to a vector of real numbers is learned. Some later models have tried to incorporate external information to enhance the vectors. The goal of this thesis is to evaluate three models of word embeddings and measure how these enhancements improve the word embeddings. These models, along with a hybrid variant, are evaluated on several tasks of word similarity and a task of word analogy. Results show that fine-tuning the vectors with semantic information dramatically improves performance in word similarity; conversely, enriching word vectors with subword information increases performance in word analogy tasks, with the hybrid approach finding a solid middle ground.

Keywords: Word embedding, word2vec, fastText, dict2vec, semantic relatedness, word similarity, dictionary information

Summary for Lay Audience

Word embedding models have been an exploding field in Natural Language Processing in the last few years. After several improvements in computational power and the development of neural networks, previously proposed models are now possible to implement in modern computers. In simple terms, a word embedding is a mapping of words to vectors of real numbers, while a word embedding model is a structure or a program that builds said mapping. The advantage of being able to represent words as vectors of a hundred or so dimensions makes these models ideal to use in real applications. Moreover, word embeddings have demonstrated to have several interesting properties, such as making words with similar meanings also similar in the vector space and the ability to represent word analogies through vector operations, for example, $king - man + woman \approx queen$.

The way these models learn this information is based on word frequency and analyzing contexts to learn which words are used together. The learning is founded on the distributional hypothesis: "words that occur in similar contexts tend to have similar meanings", also known as "a word is characterized by the company it keeps". In addition to using this hypothesis some models derive extra information from the data or incorporate external information from other sources. One thing that makes the models difficult to analyze is the fact that all of them use different values of hyperparameters to do the learning, such as the number of vector dimensions, or the size of the contexts to analyze, among many others. This situation is also worsened by the fact that the values of the hyperparameters are not universal, and the performance of the model changes depending on the type or size of the data and the problem to solve.

This thesis aims to evaluate several word embedding models based on a popular one called word2vec. Two of these models are novel: a simplified variant of an existing model and a hybrid combination of two other models. Word similarity and word analogy tasks are used in this evaluation. This thesis also aims to replicate some previously published results and hyperparameter values and to check whether the novel models can obtain competent results on both types of tasks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Word embeddings are one of the most used representations of vocabulary in NLP. Essentially, a word embedding is a particular mapping of words to vectors (collection of real numbers); additionally, in a confusing way, word embedding is also the collective name of several models that can build said mapping. The possibility to transform words to vectors of real numbers makes them an ideal model to explore and apply several mathematical operations and transformations to hopefully represent some of the features that languages have.

There has been an exploding interest in these kinds of techniques after word2vec [16] was proposed in 2013. A shallow neural network (composed of 2 layers) is used to produce a word embedding, i.e., a *vector space* where some interesting properties can be observed. One of such properties is vector similarity (in terms of semantics). Given that words are now vectors, several similarity functions can be applied as a quantifiable way of comparing such words. A good word embedding would map similar words to similar vectors, and would map not so similar words to not so similar vectors. For example: the vectors for *cat* and *dog* should be more similar than say, *cat* and *flower*. Another property is *analogies*. This aspect of word embeddings has gained a lot of popularity after word2vec successfully modeled this property via addition and substraction operations on the word vectors: $king - man + woman \approx queen$.

The real power of the word embedding approach is that it is an *unsupervised learning model*, meaning that it works on unannotated data (and text data is widely available nowadays). The way these models learn these numbers is based on analyzing the contexts between different words and using a prediction approach based on the idea of the Neural Probabilistic Language Model [3] which, in turn, is based on the idea of statistical language models: predict words given a certain sequence of pre-occurring words. For example, given the phrase "read the New York", there are some words (like "Times") that are more likely to occur than others. The idea is that if in the training data there are enough examples of this phrase, then it's possible to calculate the probability of the next word through the frequency of these word sequences.

There are many models and algorithms that create word embeddings, with different approaches. In the last few years there's been a resurgence in deep learning with several architectures and hyperparameters being proposed; some of them can even use pre-trained word embeddings from other models as a starting point. Moreover, with of all these different approaches, different numbers are obtained for the embeddings, making it really important to have a solid way of evaluating the different results. However, there's no one universal standard way of evaluating the word vectors and additionally, there are different properties of the word embeddings that can be evaluated, which makes the evaluation and comparison between models inconsistent across different studies.

1.1 Problem Statement

Adding to the situation of the different evaluation methods, all models are implemented in different programming languages, frameworks, and platforms. While this is natural, given the preferences and capabilities of the different options, the fact that they're all implemented differently can make some comparisons inaccurate. The main motivation for this thesis is both to try different combinations of hyperparameters and also training the models using the same base language and program; several word embedding models share the same base architecture which makes them ideal to use in this scenario. Additional goals are to confirm the usual recommended values for the hyperparameters and also to be able to measure the improvement of the modules or enhancements of the later models to the baseline in a more accurate way.

1.2 Contributions

First, experiments are run to show how much tuning hyperparameters impacts word relatedness and word similarity tasks for five models based on the same base architecture. Graphs show results for some values of hyperparameters using grid search for individual models, and then all together in the same graph; the best five results of the individual models are shown in tables as well. Finally, for the best result of each model, results of the individual tasks are displayed in a table alongside the average scores.

In addition, an extra evaluation dataset is tested. Given that word embeddings have shown some analogy properties in the form of vector differences, it is desirable to study if embeddings that have high performance over word similarity tasks can transfer some of that performance to vector differences. Experiments show an inverse relationship over these two kinds of tasks, meaning that a model with higher performance in word similarity will have a lower performance in analogies and viceversa. A hybrid model combining the particular approaches from two models which shows a solid middle ground between these tasks is proposed, albeit with a tradeoff with the training time.

1.3 Vectors

As previously mentioned, vectors are supposed to represent words (at least in word embeddings; there are other types of embeddings at sentence-level, paragraph-level, character-level, etc.), but certain issues can occur. For example, things like: polysemy, word senses, and inflections can make it difficult to interpret the mapping from the word to the vector; additionally, the way to learn these vectors is through examples and usage, not by grammar or rules, making them dependent on the quality of the text or type of discourse. On top of that, it's also important to consider that these embeddings are used in different fields; for example, engineers may use them as inputs for apps or other models, whereas linguists may be more interested in exploring properties related to semantics or other related topics (for more specific details about word embeddings, see Appendix A).

Regarding the size or dimension of the vectors, it's not quite understood what the dimensions mean in the different models; moreover, the number of dimensions is a hyperparameter in most, if not all, with some tasks having better results with some specific numbers. In addition, the same number of dimensions could have different results depending on the training data. This makes it more difficult to understand what the vectors are or represent.

Finally, different tasks have different requirements for "word relatedness"; for example, when dealing with *part-of-speech* tagging, two words with the same POS tag should be more related than two words that do not, independent of their individual meanings, whereas in other tasks, there's no such requirement. In the same vein, tasks like plagiarism detection need finer tuning to define what word or phrases could mean, due to things like paraphrasing.

1.3.1 Syntactic Information

In linguistics, syntax refers to how words in a sentence (clause) relate to each other. It has to do with word order, sentence structure, and grammar. Usually, word vectors don't encode syntactic information in them (mainly due to not encoding word order), but sentence or paragraph vectors do. However, when employing context windows, some word order information is used when learning the representations, so it can be argued that there is some syntactic information; nevertheless, it's hard to know *how much*, if any, information is encoded.

1.3.2 Semantic Information

Semantics refers to the meaning of lexical items, like words or phrases. Word embeddings base their representations on meaning, which is *inferred* by the usage of the word, according to the *distributional hypothesis* "Words used in similar context, *tend* to have similar meanings". There is one problem though: **word senses**.

Languages tend to "like" polysemous words. One theory is that it makes languages more efficient (recycling of signs, i.e., words and phrases)[1]. Independently of the reason, strictly speaking, all words are *polysemous* to one degree or another[2] (although this is a point of contention among linguists). Some languages are also more open to the use of words to describe certain concepts that are related but not the same thing; in these cases, the word can "bleed" over time in the lexicon and become a new word sense. For example the word "eat", even though as a verb it describes the act of putting, chewing and swallowing food in the mouth, it also means to have a meal, and additionally consuming a lot of something. The degree on which all of this is defined is not easy to determine and, on top of that, there's the complication of differentiate between *homonymy* and *polysemy*. Word embeddings have some problems with this because the only thing that is used in the vocabulary is the word itself, which depending on the data, could have multiple usages in the context data (which would make the representation too general) or, have some word senses totally absent, which would make the word embedding to not capture those meanings.

Lastly, synonyms and antonyms are an important mention. Synonyms are usually well encoded in word embeddings (given that the basis for learning the word is based on semantic information); however, antonyms are not: They are not *explicitly* used when training so there's no clear encoding in the embedding itself. Additionally, *similarity* between words is not well defined (conceptually speaking), so it's not clear what the difference is between words that are not related at all with words that are antonyms.

1.3.3 Analogical Information

Simply stated, an analogy is an inference of a relation between two or more lexical items that can be transferred to the same number of other items. Example: ***"Annoying like nails on a chalkboard"***. In NLP, analogies are treated more broadly, i.e., as similarities, for example: *"New York is to the United States as Paris is to France"*.

There are *pair-based* and *set-based* methods[3] and two types of analogies that are commonly

[1] http://news.mit.edu/2012/ambiguity-in-language-0119
[2] http://anglisztika.ektf.hu/new/content/tudomany/ejes/ejesdokumentumok/2011/Kovacs_2011.pdf
[3] https://aclweb.org/aclwiki/Analogy_(State_of_the_art)

used:

- Syntactic Analogy: plurals, verb gerunds, prefixes, suffixes (morphology).
- Semantic Analogy: collective nouns, hypernym, hyponym, meronyms, or simple meaning.

1.4 Evaluation

One of the problems with word embeddings is that it's not fully clear how to quantify the quality of them; considering that there are a lot of tasks where word embeddings are used (for example, part-of-speech tagging, semantic, syntactic, paraphrasing tasks, etc.), to evaluate them is not straightforward. Currently, two popular categories exist [2]:

- Extrinsic Evaluation: Using the vectors as features of a downstream task (outside of the word embedding itself).
- Instrinsic Evaluation: Evaluating the vectors by comparing them to human judgement (innate properties of the vectors or the vector space).

1.4.1 Extrinsic Tasks

Some evaluation tasks [2] include:

- Noun Phrase Chunking: Extracting "compact" noun phrases (not containing other NP-chunks) from a sentence (Bigrams *noun* + *dependent word*. Example[4]: *The yellow dog barked at the cat*-> (the yellow dog), (the cat).
- Named Entity Recognition: Identify names of organizations, brands, people.
- Sentiment Analysis: Text classification task where a piece of text is marked with a label that reflects level of positive, negative or neutral polarity. Examples: customer reviews, tweets.
- Text classification: Mark a piece of text with a label depending on the content. Examples: tag text by topics (sports, news, film).

[4]https://www.nltk.org/book/ch07.html

- Input for Neural Networks: Using pre-trained word vectors as inputs for other models, for example, using GloVe vectors as a starting point for an LSTM.

1.4.2 Intrinsic Tasks

Some examples of intrinsic tasks [2] are:

- Word semantic similarity: word vectors with similar meanings should be closer than word vectors that are not; the more closer, the more similar the words are.
- Word analogy: as mentioned previously in Section 1.3.3, it's based on the idea that arithmetic operations over the vectors can model certain relations or concepts by association, e.g., a is to $a*$ as b is to ?.

1.4.3 Datasets

Some evaluation tasks and datasets, which will also be used in this thesis are:

- MC-30[5]: 30 pairs with a scale of 0 to 4 (the higher the "similarity of meaning", the higher the number).
- MEN[6]: 3000 pairs with a score scale from 1-50 of human-assigned similarity judgments.
- MTurk-287[7] 287 pairs and a similarity rating with a scale of 1 to 5.
- Mturk-771[8]: 771 word pairs with mean relatedness scores. Scores were collected on Amazon Mechanical Turk, with at least 20 ratings collected for each word pair. Rating consists of a 1-5 scale, where 5 stands for "highly related" and 1 stands for "not related".
- RG-65[9]: 65 pairs with a scale of 0 to 4 (the higher the "similarity of meaning," the higher the number).
- RW[10] 2034 pairs of "rare words" and a similarity rating with a scale of 0 to 10.
- SimLex999[11]: 999 pairs with a scale of 0 to 10.

[5]https://aclweb.org/aclwiki/MC-28_Test_Collection_(State_of_the_art)
[6]https://staff.fnwi.uva.nl/e.bruni/MEN
[7]http://www.cs.technion.ac.il/~shaulm/papers/pdf/Radinsky-WWW2011.pdf
[8]http://www2.mta.ac.il/~gideon/mturk771.html
[9]https://aclweb.org/aclwiki/RG-65_Test_Collection_(State_of_the_art)
[10]https://nlp.stanford.edu/~lmthang/morphoNLM/
[11]https://fh295.github.io/simlex.html

- SimVerb-3500[12]: 3500 verb pairs and a similarity rating on a scale of 0 to 10.
- WordSimilarity-353[13]: 353 word pairs with similarity score from 0 (totally unrelated words) to 10 (very much related or identical words). The dataset used in the evaluation task is based on a further subdivision[14] which has 2 extra subsets:
    1. ALL: This is the full 353 pairs set.
    2. REL: Goldstandard for measuring relatedness (252 pairs), e.g., clothes-closet 8.0.
    3. SIM: Goldstandard for measuring similarity (203 pairs), e.g., clothes-closet 1.9.
- YP-130[15] 130 pairs with similarity rating on a scale of 0 to 4.

1.5 Format of the Thesis

This chapter has introduced the problem, where some background for word information was given, explained the types of evaluation tasks, and finally described the evaluation tasks that will be used in this thesis.

Chapter 2 talks about related works, specifically, how the models work in theory. The base architecture is explained, mentioning the modules that will be relevant for the later models.

Chapter 3 is about the models' code implementations, where a detailed overview is given, due to the difference between a model's theory and its implementation. Additionally, the proposed approach along with the evaluation process are explained as well.

Results of the experiments are shown in Chapter 4, with several graphs for all the different hyperparameters for all the different models for the average scores. The highest values for the models are shown in a table along with the scores for each task for two averages.

Chapter 5 talks about conclusions over the results and Chapter 6 outlines some potential future work.

[12]http://people.ds.cam.ac.uk/dsg40/simverb.html
[13]http://www.cs.technion.ac.il/~gabr/resources/data/wordsim353/wordsim353.html
[14]http://alfonseca.org/eng/research/wordsim353.html
[15]http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.119.1196

Chapter 2

Related Work

This chapter will be an overview on several word embedding and related embedding models. Detailed descriptions on the word embedding models relevant to this thesis are given on later sections, with special emphasis on word2vec's skip-gram; this is because all the models that are going to be analyzed are based on this architecture. The first two section are a general overview of word embedding models and some sentence embedding models; the third section describes word2vec, defining the basic concepts and ideas, and describing some optimizations over the initial proposal. Later sections deal with the changes and/or enhancements applied to this base model that fastText and dict2vec propose.

2.1 Word Embedding Models

Although word embedding models have been increasedly cited in the last few years, the concept has been discussed for several years before. The idea of vector spaces representing words were discussed previously in computational linguistics as *distributional semantic models*; they were eventually realized after some early models such as Latent Semantic Analysis [7] and Hyperspace Analogous to Language [13] appeared. In particular, LSA was very influential in demonstrating that information from contexts was enough to produce representative word spaces with interesting properties, such as representing semantic similarity for word vectors and relating words that don't co-occur together. On the other hand, HAL had more success in the field of cognitive science as a model representative of semantic memory [6].

In parallel, neural models such as simple recurrent neural networks eventually started to develop. Once computational power increased and the curse of dimensionality was solved by the Neural Language Model [3], neural word embedding models really started to develop. Starting with word2vec [16], and in conjunction with several libraries and frameworks being developed to treat text data, word embedding models made a big impact in the field. Word2vec is based on training word pairs taken from contexts found in text corpora and it learns to relate words that occur together. Afterwards, several other models were being proposed as improvements of word2vec. Among them, GloVe [18] focuses on incorporating co-occurrence information from counts instead of training the network using probability, arguing that the order on which words are mentioned is important. Although GloVe has been highly cited, there are some people that have criticized the method of evaluation of GloVe vs word2vec for being "not fair"[1].

Other examples of later models include: wang2vec [12], which includes the positional information of the context words to improve the embeddings; sense2vec [24], which proposes adding part-of-speech tags to the words in the training data to improve the vectors and differentiate word senses; fastText [4], which represents the word vectors as a normalized sum of the word vector plus its n-gram subword vectors, partially solving the problem of out-of-vocabulary words and incorporating syntactic properties; dict2vec [23], that proposes to incorporate semantic information for the words found in dictionary definitions to fine-tune the vectors.

Later, more complex architectures were developed for neural networks (called deep neural networks, unlike the word2vec model, which is *shallow*, i.e., 2 layers), with models such as Deep Structured Semantic Model [22], originally designed for information retrieval, that uses a deep network and bag-of-words inputs to learn low-dimensional representations, and CoVe [15] that employs an encoder-decoder architecture with 2-layered LSTMs in conjunction with *attention* (a network sub-module to help the encoder part of the network remember long sentences). Other models trained as language models like ELMo [19] that learns contextualized words (that is, words as a function of their context) which are also character-based, with 2 biLSTM stacked together, and BERT [8] which gets its word representations from learning bidirectional contexts with the *transformer* [26] encoder, are some of the latest state-of-the-art methods to be proposed.

2.2 Sentence Embedding Models

Alternatively, other complex models have been proposed, where, instead of word embeddings, these new models can represent whole sentences and even paragraphs. Notably, doc2vec [11], which is also based on the architectures of word2vec, uses a parameter, a *document id*, along with the sentence as an input so the model learns both the sentence and its unique id as a vector. On another note, the Deep Average Network or DAN [10], which consists of two fully-connected layers and a softmax layer, can use pre-trained word embeddings by averaging the

[1]https://rare-technologies.com/making-sense-of-word2vec/#glove_vs_word2vec

word vectors that make up the input sentence to learn the sentence vector.

Recent developments such as the Universal Sentence Encoder [5], which can use DAN or the transformer as part of its internal architecture, and LaSeR[2] [1], which uses bidirectional LSTMs, are based on machine translation and an encoder-decoder architecture to learn the representations; the former learns one language, while the latter is language-agnostic.

On the dictionary side, this model [9] also uses an LSTM but it incorporates dictionary definitions to map the word to its definition. By also using pre-trained word embeddings, the authors also built a reverse dictionary, where the input is a phrase or description, and the output is a word; this way, the model learns the definition of the word by making its representation closer to its word vector representation.

2.3 Word2vec

Word2vec [16], consists of a predictive model applied to text data, specifically, co-occurrence data within a sliding symmetric context window; this window is made from the middle word (called the *target word*) and its surrounding words (*context words*):

Example: **"The snow is white"**

Context 1:

- Target word: *snow*
- Context words: *The*, *is*

Context 2:

- Target word: *is*
- Context words: *snow*, *white*

The main idea is to feed the model different *contexts* and learn which words co-occur in the same context; in other words, it aims to find good representations of words that can predict other words that occur in similar contexts. This is fairly similar to Probabilistic Language Models, where given a sequence of words, the next word is predicted. The main problems with these kinds of models are the "Curse of Dimensionality" (where the number of variables for the sequences explodes as the vocabulary grows) and that once the model is trained, it's not possible to calculate the probability over a previously not seen sequence (because there's no training data, it's impossible to calculate the probability over counts).

[2]Although this is the name of the released framework, the reference is the study where its learning architecture is proposed. More details on: https://github.com/facebookresearch/LASER

Both of these problems were minimized by the Neural Probabilistic Language Model [3] through the use of a shallow neural network; because this model doesn't use the frequency of words (it tries to fit the known data with neuron weights and it tries to generalize from that). Word2vec uses a similar architecture and the same base idea.

Word2vec has 2 different "versions" of training:

- Continuous Bag-of-Words: Predicts a target word given the context words.
- Skip-Gram: Given a target word, predicts the context words.

For both of these cases, the model tries to maximize the log probability of the output word(s) given the input word(s). That means predicting through probability the context words given the target word in the case of Skip-Gram and the target word given a context word in the case of Continuous Bag-of-Words.

Figure 2.1 shows the architecture of CBOW. Note that V is the size of the vocabulary (the number of words that are going to be trained in the model). Words x_{ci} represent the context words in one-hot encoding; this means that the current i-th value (which is the position of the word in the vocabulary) of the input vector is 1, and 0 for all other values. h_i is the vector representation of word x_{ci}, C the size of the context, and j the position of the target word in the vocabulary. Skip-Gram, depicted in figure 2.2, follows the same notation as CBOW, with the difference that now the input (x_i) is the target word and the outputs y_{cj} will be the context words.

One important variable is the dimension of the vectors (which is N and is a hyperparameter of the model); while the model is originally trained for finding good representation of words that can predict co-occurring words, the ultimate objective is to learn low-dimensional representations of the words in the vocabulary (which are originally represented with dimension V, the size of the vocabulary). Notice that because the model has words in both its input(s) and output(s), the dimensions of the weight matrices are inverted in order to match the multiplication of the vectors.

Cost function (Continuous Bag-of-Words Model):

$$J = \max(\sum_{t=1}^{C} \sum_{k=-n}^{n} \log p(w_t|w_{t+k})) \tag{2.1}$$

$$p(w_t|w_{t+k}) = \frac{e^{v_t \cdot v_{t+k}}}{\sum\limits_{w \in V} e^{v \cdot v_{t+k}}} \tag{2.2}$$

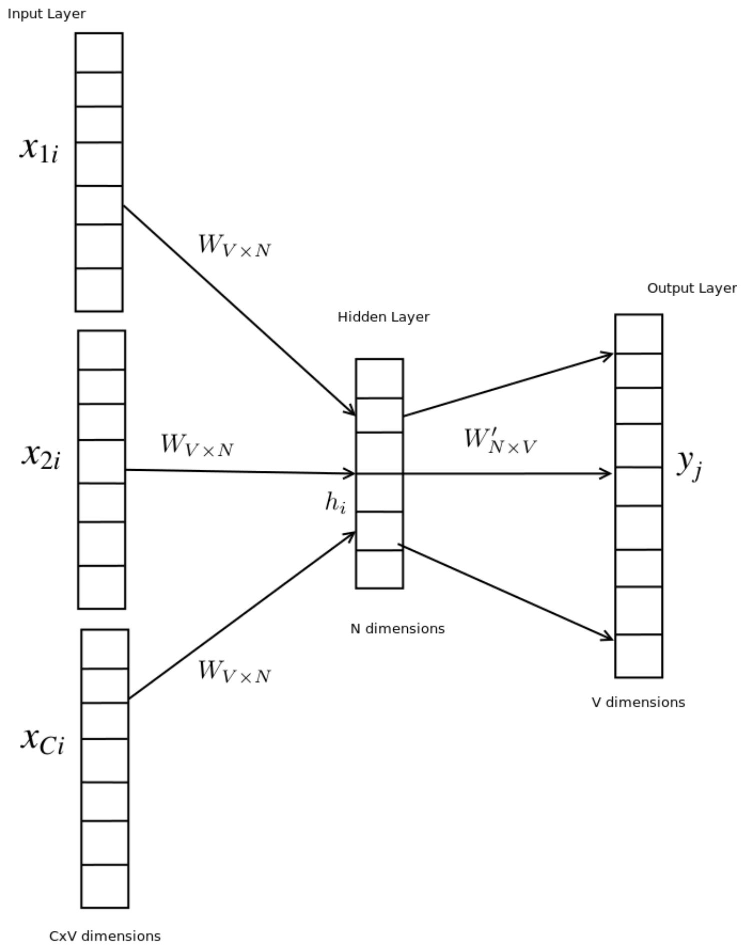


Figure 2.1: continuous bag of words model;
inputs are *C context words* and output is the *target word*.

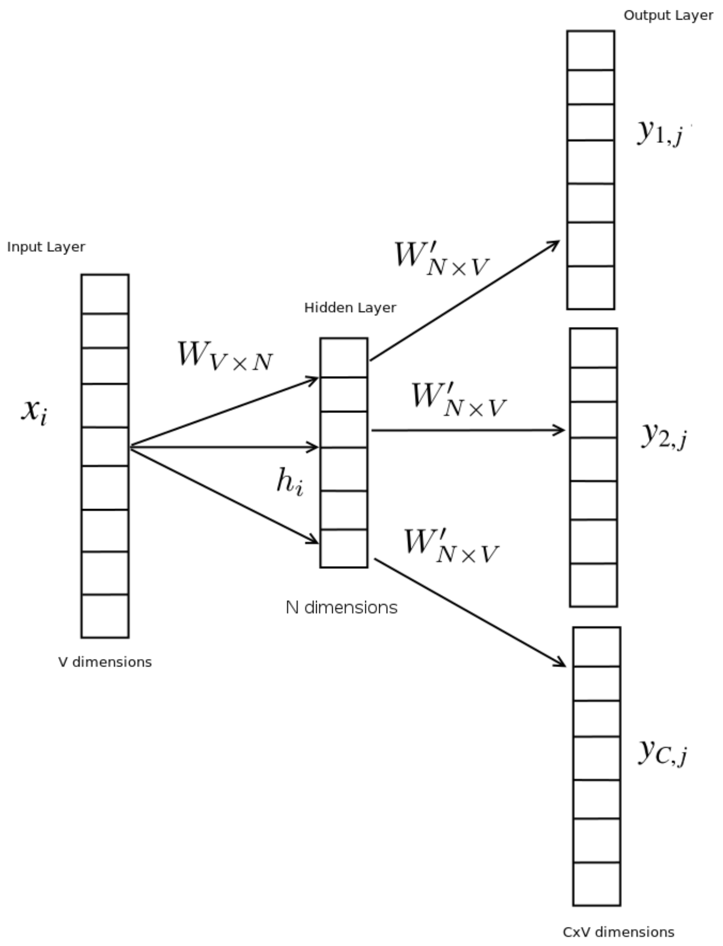


Figure 2.2: Skip-gram model; input is the *target word*, outputs are *C context words*.

And for Skip-Gram:

$$J = \max(\sum_{t=1}^{C}\sum_{k=-n}^{n}\log p(w_{t+k}|w_t)) \tag{2.3}$$

$$p(w_{t+k}|w_t) = \frac{e^{v_{t+k}\cdot v_t}}{\sum_{w\in V} e^{v\cdot v_t}} \tag{2.4}$$

where, for both cases, w is a word in the vocabulary, w_t is the target word, w_{t+k} a context word, v is the vector for word w, and v_i is the vector for word w_i. For CBOW, context word vectors are taken from their corresponding rows of W and the target word vector is taken from its corresponding column of W'; conversely, for skip-gram, the target word vector is taken its corresponding row of W and context word vectors from their corresponding columns of W'.

In both models, the input layer is also called *the embedding layer*, the second layer is called *the hidden layer* and the third layer is the output or *softmax layer*. It is important to note that, ultimately, the main task of the model is to represent words as vectors; so once the model has finished training, the hidden and output layers are removed. This causes the output to now be the low-dimensional vector representation of the input word (this process is common in learning models, where the goal is to train the model for a certain task and the objective is to get a good representation instead of the predictive model itself[3]).

Additionally, while the embeddings of both architectures are comparable, according to Mikolov[4] that skip-gram needs more data and represents infrequent words well; conversely, CBOW is faster to train and it represents frequent words slightly better. This is probably because for CBOW, more input data points with the same output are generated[5].

2.3.1 Skip-Gram

In this section, Skip-gram will be explained over CBOW because the other models analyzed are based on this architecture; for more in-depth analysis of the formulas specifically to word2vec, read this document[6]).

Vocabulary and Weights

The architecture, as previously stated, consists of 2 layers of neuron weights plus an output layer. The number of weights in each layers is the same as words in the vocabulary (so every word has a set of weights), meaning that some pre-processing must be made beforehand to determine the vocabulary. Because too rare words are not useful (because they don't appear in enough contexts) a cut-off frequency is commonly used where words below a certain frequency are deleted (or ignored) from the input data.

Embedding

A training input will be a word pair (*target word*-*context word*) obtained from a context; a context is obtained by moving a sliding window over the text. The context size (how many words from a context), from now on called window size, is a hyperparameter. Because the data is read as this moving window, all punctuation is removed and all text is put in one line (separated by spaces) so the sliding window can move through all words, not taken the structure of text into account (titles, paragraphs, etc.).

For example, given the following sentence, with a window size of 5 (some sources use the number of context words at either side of the target word; in this work, the word2vec

[3]http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

[4]https://groups.google.com/forum/#!searchin/word2vec-toolkit/c-bow/word2vec-toolkit/NLvYXU99cAM/E5ld8LcDxlAJ

[5]https://groups.google.com/forum/#!msg/word2vec-toolkit/LNPeC5gyhmQ/p8683JkD6LoJ

[6]http://www.1-4-5.net/~dmm/ml/how_does_word2vec_work.pdf

convention is used, which is all words in the window) **"The waves were crashing on the shore"**, generates the folowing target word, context word pairs (edges ignored):

- **"the waves were crashing on":** (were, the), (were, waves), (were, crashing), (were, on)
- **"waves were crashing on the":** (crashing, waves), (crashing, were), (crashing, on), (crashing, the)
- **"were crashing on the shore":** (on, were), (on, crashing), (on, the), (on, shore)

Now, given that the neural network weights can be modeled as a matrix (one for each layer), with each row of the matrix corresponding to each word in the vocabulary and the input is a vector (using one-hot encoding), then the embedding h_i for input word w_i (the target word) can be defined as:

$$h_i = w_i^T \cdot x_i \tag{2.5}$$

where x_i is the *i-th* word in the vocabulary. Because this input is in one-hot encoding, then this is the same as just taking the *i-th* row of weight matrix W. This works for both layers of the model.

Error Function

The objective function is the log probability of the context word w_c given the target word w_t:

$$-\log p(w_c|w_t) \tag{2.6}$$

In other words, the desire is to maximize the probability of the model outputting the context word given the target word. Because several contexts exist, then given a target there's going to be a big range of likely words and these probabilities are adjusted (or learned) by modifying the weights. According to this angle, this is really similar to a multi-classification problem (having the output words w_c as the classes), and given that the outputs are probabilities, then softmax (Section 2.3.1) is suitable.

Output

As previously stated, the output layer uses the so-called *softmax* function 2.7 which outputs probabilities of one item over all possible items (essentially a normalized sum), with all its calculated terms summing 1. In the case of word2vec, the term is calculated for every training word pair, so it has to iterate over all words in the vocabulary and it can become computationally demanding. What the model does is to maximize the objective function through the softmax function, and then changing the weights of the two previous layers via back propagation (Section 2.3.1).

$$F(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}} \tag{2.7}$$

Updating Weights

After making a prediction in a training pair, the error is calculated and the weights are modified according to the product of the error and the gradient of the output of said layer. The idea is to apply gradient descent to find the optimum weight value which would minimize the error of both sets of weights.

After modifying the weights of the hidden layer, then the same process is made for the previous layer but instead taking the gradient of the activation function of the next layer. Because each gradient depends on the output of the layer, this process is called back propagation because the errors *propagate* through each layer (to put it differently, the output of a layer depends on the output of the previous layer). For more details on how to calculate the gradients, see [20].

2.3.2 Optimizations

In a follow-up paper [17], the authors mention several implementation details to speed up the training process. The most important (and cited) are:

- Hierarchical Softmax
- Negative sampling

Hierarchical Softmax

Because calculating the softmax denominator output is expensive (going through **all** words in the vocabulary), a binary tree can be built with the leaves of the tree being the words in the vocabulary. Then, the probability of a word is given by the path from the root to the leaf node. This way, it is only necessary to evaluate $\log_2(vocab\_size)$ at most.

When navigating the path, going a to certain direction in any non-leaf node (left or right) has a probability p, then going in the other direction has a probability $1 - p$; then the word probability is going to be the multiplication at each non-terminal node of the path. Given that both options at each node sum to one, then the summation of all leaf-nodes probabilities is also one.

Sigmoid can be used as a function to get probabilities at each node, which is defined as the similarity between the representation of the word and a *coefficient vector*[7] which can be defined and learned in a multitude of ways; they can be seen as weights of the hidden layer in the neural network model.

$$P(left) = \sigma(v'T_n v_w) \tag{2.8}$$
$$P(right) = 1 - P(left) \tag{2.9}$$

with $v'T_n$ being the coefficient vector for node n.

Example: "The yellow dog barked at the cat".

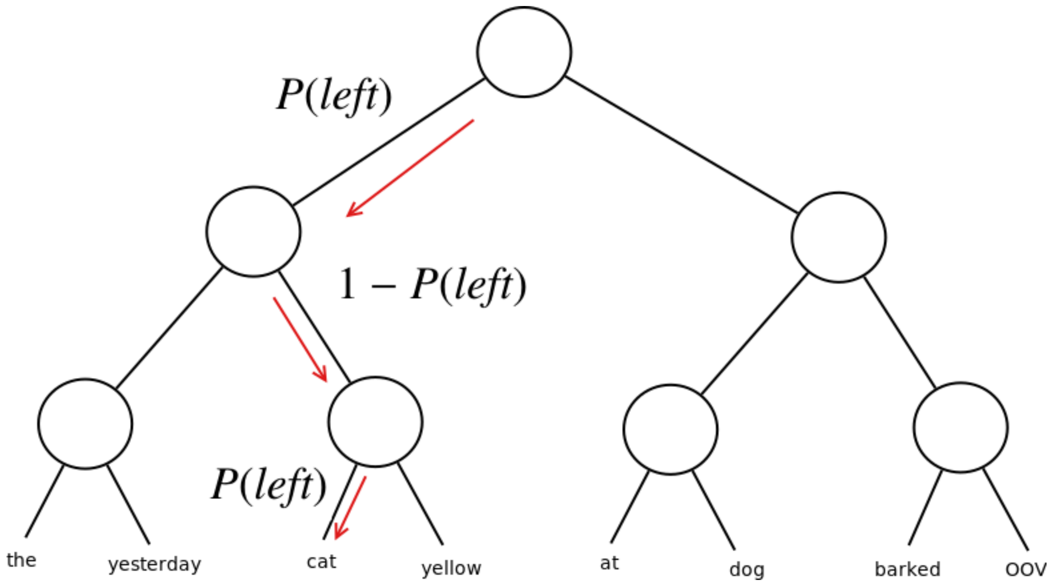


Figure 2.3: Hierarchical Softmax

Also, the way in which the tree is built, impacts the performance directly. The authors also proposed to use Huffman encoding so that more common words are closer to the root, further improving performance.

[7]http://building-babylon.net/2017/08/01/hierarchical-softmax/

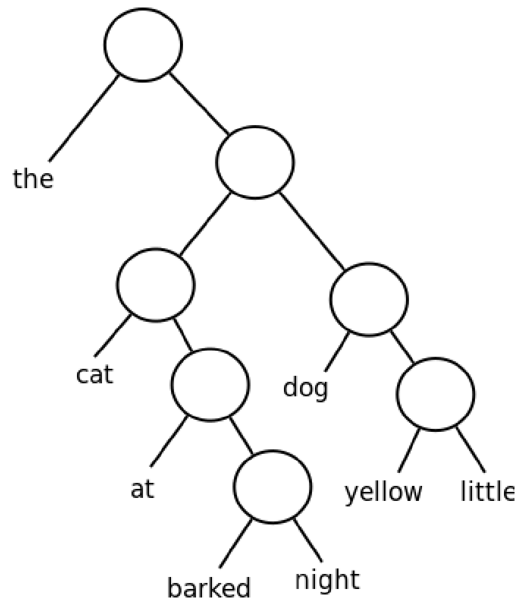


Figure 2.4: Huffman encoding; words that are more common will be closer to the root.

Negative Sampling

An alternative to softmax (based on Noise Contrastive Estimation, or NCE). Given that the model is only interested in learning high-quality vector representations, the objective function can be changed to use pairs of words, instead of a probability distribution. In a simplified way, only "good pairs of words" (or good examples) are needed to train the model; in this vein, it can be argued that "negative examples" make the algorithm learn a generalized representation and "positive examples" make the algorithm learn a specific representation. These negative samples are sampled randomly from the vocabulary given a noise distribution. A good distribution, according to the authors, is the unigram ditribution:

$$\frac{U(w)^{\alpha}}{\sum_{i=1}^{n} U(w_i)^{\alpha}} \tag{2.10}$$

Figures 2.5 and 2.6 show how this distribution looks for different values of α; the authors mentioned that $\alpha = \frac{3}{4}$ gave the best results for the model. It was also suggested that a good number of negative samples is between 5-20 for small datasets and 2-5 for large datasets.

Using these labeled samples, with the actual word pair as a positive label and the word and its negative samples as a negative label, it's possible to model this problem as binary classifiction problem. This way, logistic regression can be applied in conjunction with the cross-entropy error as its loss function:

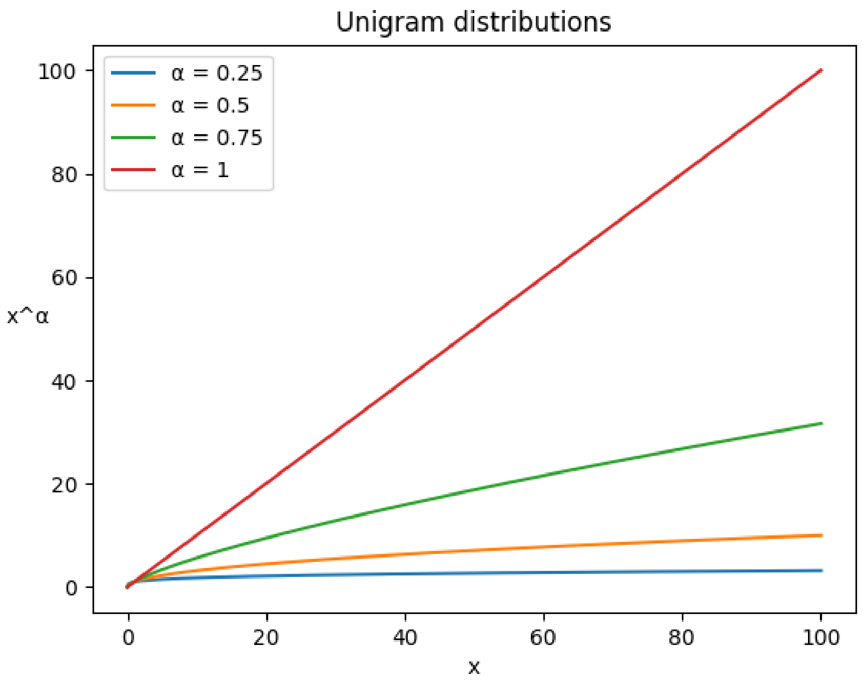


Figure 2.5: Unigram distribution

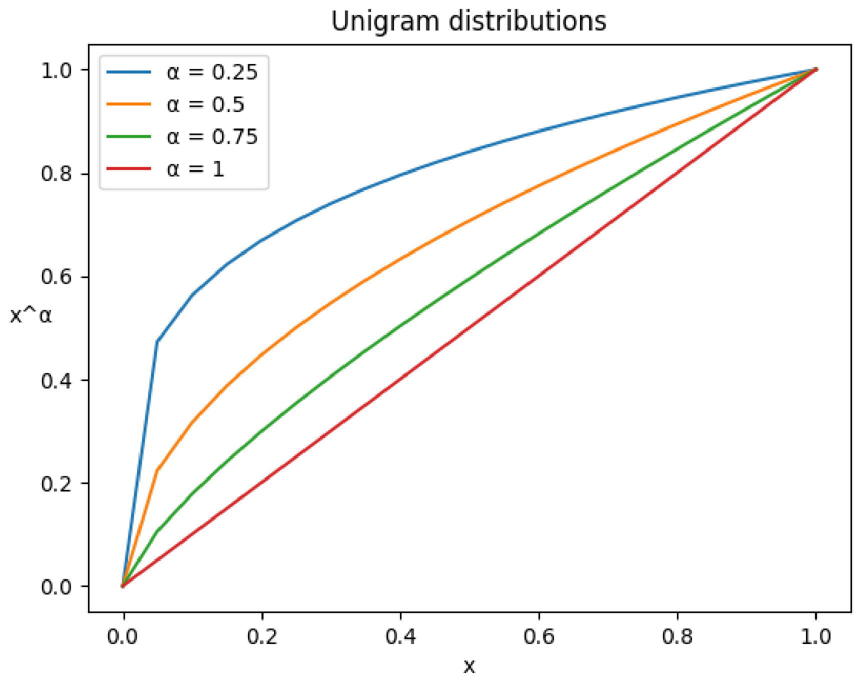


Figure 2.6: Unigram distribution zoomed up to $x = 1$

$$J = -\sum_{i=1}^{|C|}\left(L\ \log(\sigma(w_t \cdot w_{ci})) - \sum_{k=1}^{K}(1-L)\log(1-\sigma(w_t \cdot w_{nk}))\right) \tag{2.11}$$

with K being the number of words sampled from the noise distribution (and a hyperparameter), L the label of the word pair, S the sigmoid function: $S(x) = \frac{1}{1+e^{-x}}$, w_t the vector for the target word taken from the embedding layer weights W, w_{ci} the context word i from the hidden layer weights W', and w_{nk} the word k from the noise distribution (the negative sample).

The resulting training process is now changed to a positive sampling operation plus K negative sampling operations *for each target word and context word pair*.

2.4 FastText

Proposed in [4], it uses the same architecture as skip-gram of word2vec, but makes a small but important change. Instead of just using words in the vocabulary, subword character n-grams are also learnt; a word representation is now the normalized sum between the representation of its subwords and the whole word. This means that for a word like ***apple***, the generated tokens will be: <***ap, app, ppl, ple, le***>***, apple*** (having < and > as the start and end delimiters for the word, respectively). This also slightly increases the vocabulary size and a function has to be used to find the subword n-grams or they can be pre-computed.

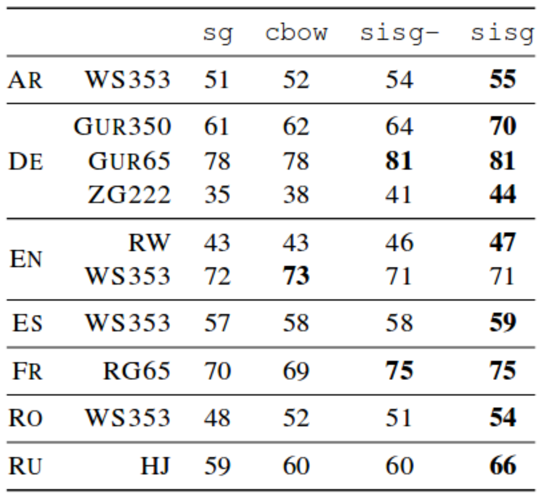
The objective function is updated to maximize the summation of the representations:

$$J = \sum_{t=1}^{C}\sum_{k=-n}^{n}\sum_{w \in G_{w_t}} \log p(w_{t+k}|w) \tag{2.12}$$

One strength about fastText in contrast to word2vec, is that these word vectors give the possibility of representing out of vocabulary words (OOV) by just using their subword vectors.

Figure 2.7 shows results of evaluating[8] the word vectors on several tasks of word similarity and word relatedness in languages such as: Arabic (AR), German (DE), English (EN), Spanish (ES), French (FR), Romanian (RO), and Russian (RU). The two english tasks are also evaluated in this thesis and are described on Section 1.4.3.

[8]The score is the Spearman's rank correlation coefficient between human judgement and cosine similarity over the word vectors; results are multiplied by 100.



		sg	cbow	sisg-	sisg
AR	WS353	51	52	54	**55**
DE	GUR350	61	62	64	**70**
	GUR65	78	78	**81**	**81**
	ZG222	35	38	41	**44**
EN	RW	43	43	46	**47**
	WS353	72	**73**	71	71
ES	WS353	57	58	58	**59**
FR	RG65	70	69	**75**	**75**
RO	WS353	48	52	51	**54**
RU	HJ	59	60	60	**66**

Figure 2.7: Spearman's rank correlation between human judgement and cosine similarity of fastText vectors for different tasks in different languages (results are multiplied by 100). Columns *sg* and *cbow* are both of word2vec's implementations; *sisg-* is the Subword Information Skip Gram (fastText) when representing OOV words as *null* and *sisg* when subword vectors are used to model OOV words. The languages evaluated are: Arabic (AR), German (DE), English (EN), Spanish (ES), French (FR), Romanian (RO), and Russian (RU) (image taken from source paper [4]).

2.5 Dict2vec

Dict2vec [23] also uses the same architecture as skip-gram but tries to add semantic information from word definitions to the training process. Additional sampling steps are added: sampling new words from sets defined as either *strong* or *weak* depending on their mutual appearance on dictionary definitions (which are taken from different dictionaries online). For example: if the word *road* appears in the definition of *car*, and viceversa, then they form a *strong pair*; if only one of the words appear in the other's definition, then they form a *weak pair*. Additionally, pre-trained word embeddings can be used to help define if words can be a *strong* or *weak* pair depending if they mutually appear in their k-nearest neighbors. Additionally, this model also uses negative sampling and the same weight initialization as word2vec.

The objective function is changed to this:

$$J(w_t, w_c) = \ell(v_t \cdot v_c) + J_{pos}(w_t) + J_{neg}(w_t) \tag{2.13}$$

where:

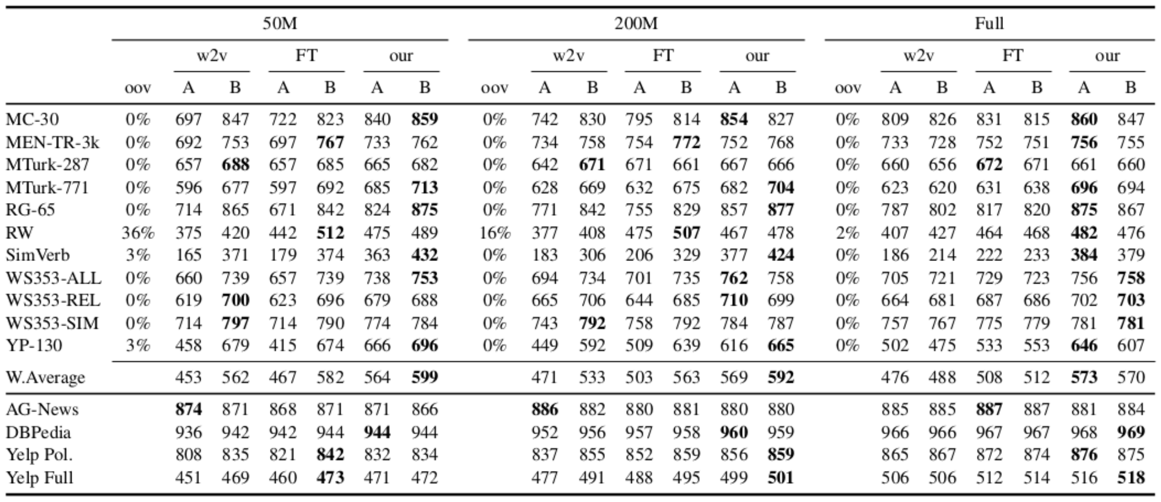
$$J_{pos}(w_t) = \beta_s \sum_{w_i \in V_s(w_t)} \ell(v_t \cdot v_i) + \beta_w \sum_{w_j \in V_w(w_t)} \ell(v_t \cdot v_j) \tag{2.14}$$

$$J_{neg}(w_t) = \sum_{\substack{w_i \in F(w_t) \\ w_i \notin S(w_t) \\ w_i \notin W(w_t)}} \ell(-v_t \cdot v_i) \tag{2.15}$$

One additional constraint is that the negative samples can't belong to the strong or weak sets of the target word, as shown in equation 2.15.

According to the authors, this model gets better results than word2vec and fastText for both word similarity and text classification evaluations and it's also claimed that it has less training time (it takes less time to reach the optimum); however, it uses more hyperparameters (the *strong* and *weak* sampling and beta values (for the strong and weak sampling).

Table 2.8 shows the published results of evaluating several word similarity tasks on the dict2vec embeddings using the Spearman's rank correlation coefficient (scores are multiplied by 1000); this method of evaluation is going to be reutilized for this thesis, hoping to replicate results.



	50M							200M							Full						
		w2v		FT		our			w2v		FT		our			w2v		FT		our	
	oov	A	B	A	B	A	B	oov	A	B	A	B	A	B	oov	A	B	A	B	A	B
MC-30	0%	697	847	722	823	840	**859**	0%	742	830	795	814	**854**	827	0%	809	826	831	815	**860**	847
MEN-TR-3k	0%	692	753	697	**767**	733	762	0%	734	758	754	**772**	752	768	0%	733	728	752	751	**756**	755
MTurk-287	0%	657	**688**	657	685	665	682	0%	642	**671**	671	661	667	666	0%	660	656	**672**	671	661	660
MTurk-771	0%	596	677	597	692	685	**713**	0%	628	669	632	675	682	**704**	0%	623	620	631	638	**696**	694
RG-65	0%	714	865	671	842	824	**875**	0%	771	842	755	829	857	**877**	0%	787	802	817	820	**875**	867
RW	36%	375	420	442	**512**	475	489	16%	377	408	475	**507**	467	478	2%	407	427	464	468	**482**	476
SimVerb	3%	165	371	179	374	363	**432**	0%	183	306	206	329	377	**424**	0%	186	214	222	233	**384**	379
WS353-ALL	0%	660	739	657	739	738	**753**	0%	694	734	701	735	**762**	758	0%	705	721	729	723	756	**758**
WS353-REL	0%	619	**700**	623	696	679	688	0%	665	706	644	685	**710**	699	0%	664	681	687	686	702	**703**
WS353-SIM	0%	714	**797**	714	790	774	784	0%	743	**792**	758	792	784	787	0%	757	767	775	779	781	**781**
YP-130	3%	458	679	415	674	666	**696**	0%	449	592	509	639	616	**665**	0%	502	475	533	553	**646**	607
W.Average		453	562	467	582	564	**599**		471	533	503	563	569	**592**		476	488	508	512	**573**	570
AG-News		**874**	871	868	871	871	866		**886**	882	880	881	880	880		885	885	**887**	887	881	884
DBPedia		936	942	942	944	**944**	944		952	956	957	958	**960**	959		966	966	967	967	968	**969**
Yelp Pol.		808	835	821	**842**	832	834		837	855	852	859	856	**859**		865	867	872	874	**876**	875
Yelp Full		451	469	460	**473**	471	472		477	491	488	495	499	**501**		506	506	512	514	516	**518**

Figure 2.8: Spearman's rank correlation coefficient (scores multiplied by 1000) between human judgement and cosine similarity of word vectors (top) and accuracies on text classification tasks (bottom). Models analyzed are word2vec, fastText, and dict2vec (labelled as *ours*). Corpus A is Wikipedia, whereas corpus B is Wikipedia + the dictionary definitions concatenated; the 50M & 200M refer to the first 50 million and the first 200 million words from Wikipedia, respectively (image taken from source paper [23]).

Lastly, as a side note, the authors also tried the model using semantic information taken from WordNet pairs; however, no finer details were mentioned.

Chapter 3

Model Implementations and Evaluation

In this chapter, technical and specific details about the implementations are discussed. Given that the theory of how the models work differs slightly from the actual implementations (mostly due to inherent limitations of the medium, such as hardware or memory), special attention is given to these aspects of the models, which are sometimes not covered. As mentioned in previous chapters, given that both fastText and dict2vec are based on skip-gram, only this model will be discussed; this chapter also presents this thesis' main contribution: a hybrid model of fastText and dict2vec.

In addition, the method of evaluation of the models is presented along with descriptions of the experiments such as: training data and preprocessing, hyperparameter discussion, and details about extra information needed for both dict2vec and the proposed model.

3.1 Word2vec - Skip Gram

The code is written in C and it's available on GitHub[1]. It is important to note that all the multiplication routines are done in basic C with no external libraries; it's also highly optimized for several other tasks such as random values, shuffling and the activation function. The repository also contains other programs related to word2vec and scripts with demos and basic evaluation tasks.

In terms of how it works, the program reads the input, which is a text file of words written in a single line separated by spaces, builds the vocabulary and creates the sampling table. The sampling table is a big array of 10 million values where each value in the array is going to be the index of a word. The amount of times each value appears in the table is given by:

[1]https://github.com/tmikolov/word2vec

$$frequency = \frac{f^{\frac{3}{4}}}{\sum_{i=1}^{V} f_i^{\frac{3}{4}}} \times ts \tag{3.1}$$

with f_i being the sum of all counts to the 0.75 power and *ts* is the size of the table (10^7).

After the sampling table is filled with values, it gets shuffled so the same values are not contiguous. Then, the input file is divided into chunks which are then given to a set of threads that go through every word (a thread per chunk). For every word, the context window is calculated (edges are ignored) and for each pair of target and context words, negative words are sampled from the sampling table. Because of the sampling table shuffling, an index at the start of the array that goes through it is enough to simulate the random sampling.

An additional hyperparameter called *sampling* is used: it is used to calculate the probability of discarding a word when going through the input. The probability of discarding the word is defined as:
$p(w) = 1 - \sqrt{\frac{t}{f(w)}}$, where t is the treshold (the *sampling* hyperparameter) and $f(w)$ is the proportion of the word w in the corpus, i.e., $f(w) = \frac{count(w)}{N}$, where N is the total number of words. The intention of this is to train the model with words that are not so frequent because words that are too frequent tend to have no meaning (stopwords). The usual recommended value for sampling is 0.0001.

After the positive and negative samples are obtained (1 positive + n negatives), logistic regression is applied to the dot product of the vectors, where the error is obtained. That value gets multiplied by the learning rate to obtain the gradient, which will be used to modify the weights of both layers. Through backpropagation, the output weights get updated first by the gradient multiplied by the input word vector, and then the input weights are modified after all samples are processed by the accumulated sum (of all samples) of the gradient multiplied by the output word vector.

After the epochs have finalized, then the program goes through every word in the vocabulary and writes a .vec file with the vector representation of every word (one for each line, where first comes the word and the weight values, all separated by space). Notably, this implementation also has the option of using either hierarchical softmax or negative sampling for the training.

3.2 FastText

The main difference with word2vec is that for getting the word representation, the subword vectors are needed. To accomplish this, when the vocabulary is being built, the program calculates the subwords and adds them to a *subword vocabulary* with their own weights and the list of indices of the subwords are stored in the structure of the word. Then, at training, when reading a word, the program gets the weights of the subwords (via the list of indices) and adds them. Afterwards, the vector of the full word is added to this sum and then it's averaged. Note that this sum is only calculated for context words (the input weights). The rest remains the same.

The process of updating the weights is mostly the same, the only change is that each of the subwords weights are also going to be modified by the same amount as the input weights. This implementation also gives the option to choose between hierarchical softmax and negative sampling. The original code[2] is in C++, although there's a fairly popular python implementation in gensim[3]. It's also important to mention that the published program also allows one to use the word embedding for text classification problems.

3.3 Dict2vec

This code is a re-implementation of the skip-gram base code, it's also written in C and it doesn't have hierarchical softmax available. The big change is the additional sampling of the dictionary words, from both the strong and weak sets; basically two extra steps after updating the output weights (the normal positive and negative sampling) but before updating the input weights (because this uses the sum of all the output words multiplied by the gradient). In this model, the number of words sampled from each set as well as the beta (learning rate) of each sampling are hyperparameters.

The process itself is a form of positive sampling but from semantically related sets. The idea is that when training the context word, it's possible to randomly sample the set of definition-related words (both the strong and weak sets) to add semantic information (just like negative sampling, but in this case the label is positive). In other words, normal positive sampling finds word pairs that co-occur in a given context in writing, and dict2vec positive sampling finds word pairs that co-appear in word definitions.

3.4 Proposed Model (dictFastText)

The implementation of this proposed model is based on the Dict2vec C code with the subword embedding process added, and was dubbed *dictFastText*. This means that training is the same as it is in fastText (with a word being the normalized sum of the internal subwords that make

[2]https://github.com/facebookresearch/fastText
[3]https://github.com/RaRe-Technologies/gensim

the word), but after the positive and negative sampling training is done for a word pair, then the strong and weak sampling from dictionary word pairs is applied.

To sum up, for a given word pair (w_i, w_j), the process is as follows:

- build the word embedding sw_i with subwords of the target word w_i and apply logistic regression over this positive sample.
- sample negative pairs of the context word w_j (word2vec negative sampling) and apply logistic regression.
- sample pairs from the strong set of words for w_j and apply logistic regression.
- sample pairs from the weak set of words for w_j and apply logistic regression.

The hypothesis is that this hybrid model can have some of the syntactic properties and the capability of representing words not in the vocabulary of fastText and the high performance over word similarities demonstrated by dict2vec.

3.5 Evaluation

3.5.1 Experiments

Word Semantic Similarity

This task is a replication of the process of evaluation applied in dict2vec [23]. It consists of comparing the tasks' evaluation scores (described in Section 1.4.3) against a similarity metric score applied on the embeddings. The chosen similarity metric was cosine similarity (same as the dict2vec evaluation process):

$$similarity = \cos(\theta) = \frac{A \cdot B}{||A|| \cdot ||B||} = \frac{\sum_{i=1}^{n} A_i \cdot B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{3.2}$$

Given that the human judgements scores and the cosine similarity are in different scales, the Spearman's rank correlation coefficient (See appendix B for more information) is used as the score of each task. With all of these tasks evaluated, an average score is calculated to measure this overall evaluation, which will be the metric to compare the different models. The range of values of this average score is from -1 to 1, where closer to 1 is better (which implies a positive correlation).

Said average score is the micro average, which is calculated by:

$$\frac{\sum_{i=1}^{T} S_i \cdot N_i}{\sum_{i=1}^{T} N_i} \tag{3.3}$$

where T is the number of tasks, S_i the score (the Spearman's rank correlation coefficient) of task i, and N_i the number of samples (word pairs) in task i.

The reason to use this metric was not backed by the authors, but it's usually employed when there's a significant difference in the number of samples of each task; this is because the score of each task is proportional to its number of samples. In this case, the lowest number of samples is 30 while the maximum is 3500, so it's highly likely that this was the reason to do so. On top of this, because each task has a different focus (for example, a task contains only verbs, another one only rare words) it can happen that the score can be skewed to certain tasks. To give a better view of performance, a macro average (the average of the task scores) was added to show how the models perform across all tasks.

Additionally, training time is also analyzed for this task, with lower being better and measured in minutes. Lastly, as a note, the evaluation script was slightly modified to add support for the models with n-grams.

Word analogy

As previously mentioned, the word analogy property can be generalized to pairs of words that display *some kind* of relation. Based on this view, Diffvec [27] is a dataset built to evaluate several lexical relations that can be represented by vector differences (see Appendix C for more information). Essentially, there are 9 classes such as: *collective noun*, *hypernym*, *meronym*, and *singular-plural* that represent said relation between words; the idea is to evaluate if the word embedding can produce vectors that show these relations through vector differences (Section 1.3.3).

As the previous section dealt with the Spearman's rank correlation coefficient for evaluation, in this case, spectral clustering (expanded in Appendix C) is employed. Spectral clustering measures how well the dataset represents or fits a given number of clusters (which is a hyperparameter). The way to evaluate this is through a score called V-measure, which is the harmonic mean between two metrics: *homogeneity* and *completeness*, both of these metrics measure how well the clusters contain samples: the first is maximized when clusters contain samples from the least amount of different classes as possible; the latter is maximized when all samples from a class belong to a single cluster. Therefore, when the V-measure is high, it means that the dataset is grouping the data properly in the given amount of clusters. There is some discussion to be had in what the cluster are or represent, but just as the number of dimensions in word embeddings, they're an abstract concept.

3.5.2 Training Data

The same corpus of dict2vec is used (Wikipedia dump from July 2017[4]), however, due to time constraints and the number of hyperparameters, only the first file (the first 50 million words) is going to be used for training.

The preprocessing is the same as the one applied by dict2vec[5] which is quite simple: all text is lowercased, and only characters in the english alphabet are kept, except digits which are translated to words, e.g., 1 turns to *one*. All text is then written in one line with words separated by space.

3.5.3 Web Scraping

To obtain the strong and weak pairs data, the words from the vocabulary (which is made from all words with a frequency of 5 and more) are searched in these oline dictionaries: Cambridge[6], Collins[7], Dictionary.com, and Oxford[8]; all pages were downloaded and only the HTML element with the definitions' texts were saved to disk. This method was preferred because if there's any IP blocking or disconnect midway, the files are saved.

Because the way to access these webpages is by a direct URL, some websites return a 404 response and those webpages can be ignored; however, there's other websites where accessing a non-existing URL triggers a search page, so the only way to check if the dictionary has the definitions for the word is to actually analyze the webpage. The base program was modified to read these files from disk instead of downloading them. An additional script was made to delete files with no definitions.

3.5.4 Hyperparameters

As mentioned before, one objective is to reproduce and confirm the usual recommended values for hyperparameters of the models. After the values were selected, the models were then trained using grid search over seven epochs and word embeddings were saved for the third, fifth, and seventh epoch. Experiments were run with 8 threads each, on a linux machine with its CPU being an Intel i7-7700K @ 4.20GHz.

[4]Obtained from https://dumps.wikimedia.org/enwiki/, although the used corpus is no longer available.

[5]Although the perl script to preprocess the data is available in the dict2vec's github page, the original can be found here http://www.mattmahoney.net/dc/textdata.html

[6]https://dictionary.cambridge.org/dictionary/english/

[7]https://www.collinsdictionary.com/dictionary/english

[8]https://en.oxforddictionaries.com

The hyperparameters are:

- Vector size: Size of the vectors. Usual recommended values are 100 and 300.
- Window size: Number of words in each context. Usual recommended value is 5.
- Sampling: Threshold factor to estimate the probability of discarding a word (as described in Section 3.1). Usual recommended value is 0.0001.
- Negative sampling: Number of words sampled from the noise distribution (described in Section 2.3.2). Usual recommended values are 2-5 for big datasets and 5-20 for small datasets.
- Ngrams: size of the subwords (described in Section 2.4). Recommended values are 5 and 6.
- Strong draws: Number of words sampled from the strong set (described in Section 2.5). Recommended values are 2-5.
- Beta strong: Factor that multiplies the positive strong sampling (described in Section 2.5). Recommended value are 0.5-1.2.
- Weak draws: Number of words sampled from the weak set (described in Section 2.5). Recommended values are 2-5.
- Beta weak: Factor that multiplies the positive weak sampling (described in Section 2.5). Recommended values are 0.2-0.6.

The selected hyperparameters for word2vec were:

- Vector size: 10, 25, 50, 100, 200, 300, and 400.
- Window size: 3, 5, and 7.
- Sampling: 0.1, 0.01, 0.001, and 0.0001.
- Negative sampling: 5, 10, and 20.

For FastText the hyperparameters chosen were:

- Vector size: 10, 50, 100, 200, 300.
- Window size: 3, 5, and 7.

- Sampling: 0.0001.
- Negative sampling: 5.
- Ngrams: 2, 3, 4, and 5.

For dict2vec the hyperparameters chosen were:

- Vector size: 10, 50, 100, 200, 300.
- Window size: 3, 5, and 7.
- Sampling: 0.0001.
- Negative sampling: 5.
- Strong draws: 2, 4, and 8.
- Beta strong: 0.2, 0.4, and 0.8.
- Weak draws: 2, 5, and 8.
- Beta weak: 0.3, 0.45, and 0.9.

For dictFastText, the hyperparameters chosen were:

- Vector size: 100, 200, and 300.
- Window size: 3, 5, and 7.
- Sampling: 0.0001.
- Negative sampling: 5.
- Strong draws: 4 and 8.
- Beta strong: 0.4 and 0.8.
- Weak draws: 5 and 8.
- Beta weak: 0.45 and 0.9.
- Ngrams: 3, 4, and 5.

These values were selected in a way to be somewhat extensive enough without increasing the number of experiments too much; this is a bigger problem for dictFastText and dict2vec given the high number of hyperparameters. Additionally, all models were trained with a learning rate of 0.025, which is the factor that multiplies the gradient for backpropagation (see Section 2.3.1) to change the neural network weights.

Chapter 4

Results

This chapter shows and discusses the results over the experiments defined in the previous chapter, with the first section dealing with the hyperparameter tuning for each model individually. In general, all results are shown in graphs for each model for qualitative results, and tables for quantitative ones. The reader should note how the different combination of hyperparameter values affect the models; for this effect, some additional type of graphs (such as box plots) are shown to try to highlight different views of the results. Towards the end of the section, results of all models are shown in the same graph to have a better idea how they compare to each other.

At the end of the chapter, the highest performances are selected for each model and shown in a summary table including the performance over each task. On top of this, a less-known additional evaluation is applied to the models to evaluate a different aspect of the embeddings and show that it appears there's some indirect relationship between the two properties.

4.1 Analysis

It's important to remember that some hyperparameters of the models were mantained constant in all experiments such as the learning rate (locked at 0.025) and the cutoff frequency of words for the vocabulary (5, meaning that words with frequency of 4 and less were ignored). Also, as mentioned before, the micro average of the Spearman's rank correlation coefficient is going to be considered as the metric for evaluation and, to a small extent, the training time as well. Each experiment was run once, saving the vectors for epochs 3, 5, and 7; that means that one experiment appears three times on any given graph (unless plotting separate epochs).

Regarding the graphs, given the high number of experiments in some of the models and that hyperparameter values are discrete, swarmplots were preferred; swarmplots, in essence, work just as scatterplots. The main difference is that swarmplots try to show or display more data points when there are many of them by exploiting the horizontal space (which is possible because of the discrete values in the x-axis), whereas in scatterplots, those data points would not be visible because most of them would be superimposed, making it harder to analyze the results. Boxplots were also added to this end, hoping that information from both types of graphs can give a good view of the results.

4.1.1 Word2vec

As a note, the plotting of the sampling hyperparameter was changed to its exponential value to make it easier to graph results; also, time values in plots are in minutes.

Figures 4.1 and 4.2 show different experiments with different epochs colored. With the exception of vector size 10 and 25, there's a clear tendency of lower epochs having lower scores. This is offset by the fact that all these experiments have different values for their hyperparameters, which will be shown as well. One interesting thing is that window size 3 seems better to train for more epochs (meaning that hyperparameters don't have that much influence); however, window sizes 5 and 7, paint a different situation: in general, increasing the epochs increases the range of possible results, in other words, training for more epochs could lead to better or worse results, meaning that the choice of hyperparameters have more influence in these cases.

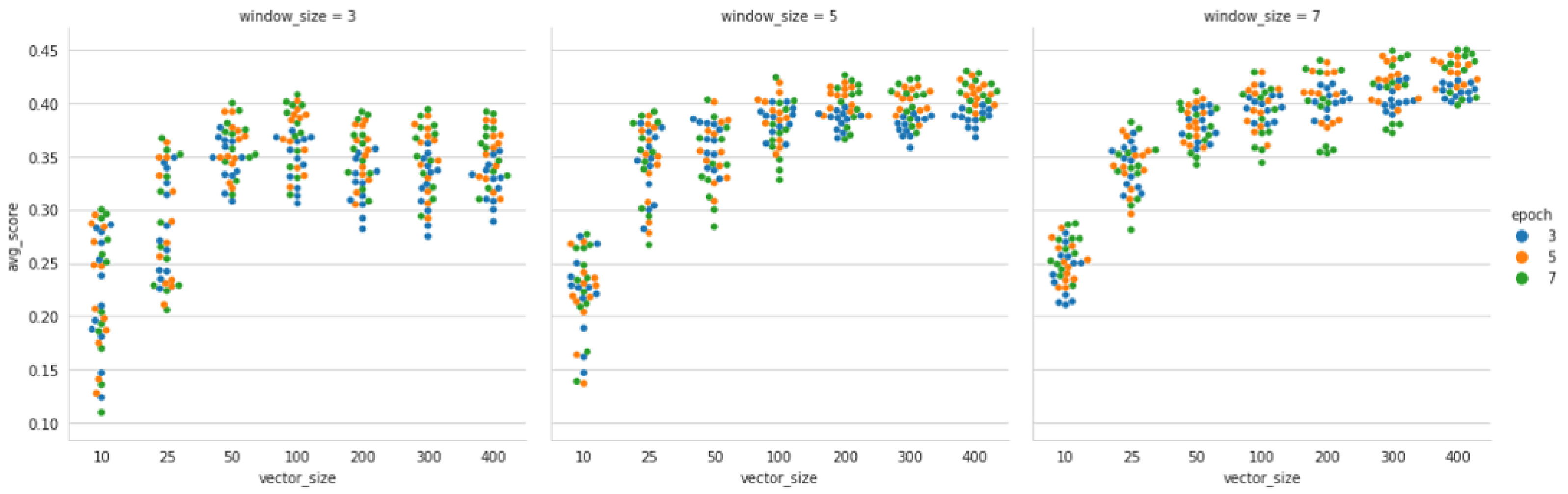


Figure 4.1: Word2vec performance by epoch, note how higher results are associated to higher epochs.

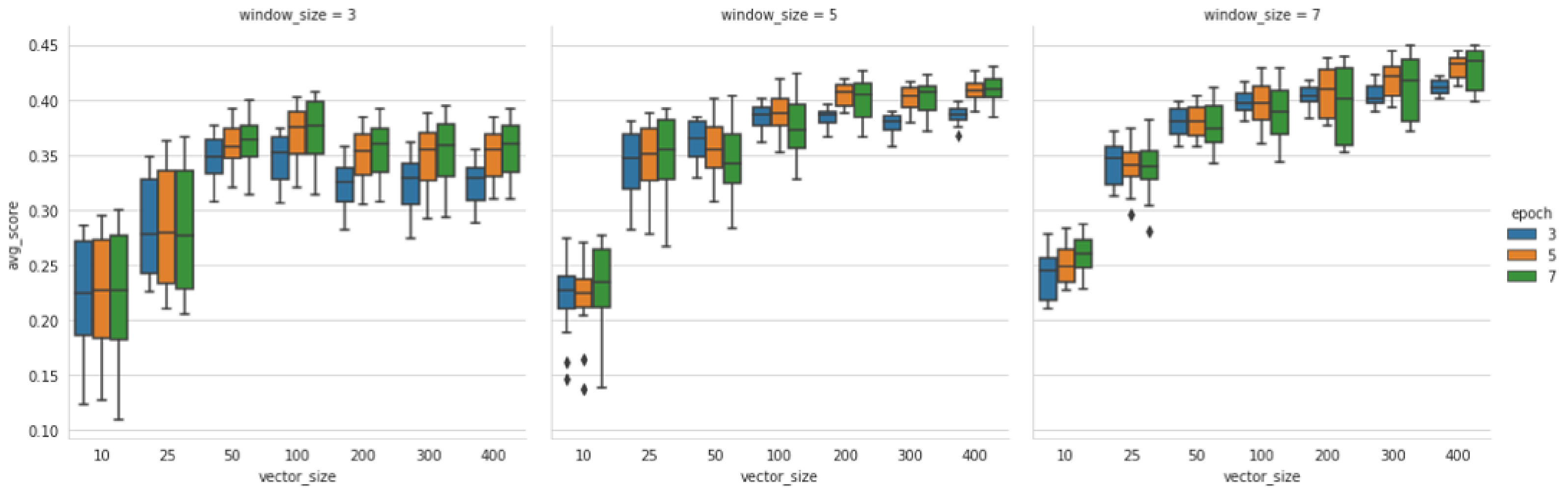


Figure 4.2: Word2vec performance by epoch in boxplot.

Figure 4.3 shows that for window sizes 5 and 7, lower negative sampling values tend to have higher results (for window size 3 it's similar but it seems to be less clear). This also confirms that the usual recommended value for negative sampling (5); apparently it seems that trying to learn too much from negative examples disrupts the learning from the positive sampling. It can be seen in Figure 4.4 how adding more samples also makes it much slower. Finally, Figure 4.5 shows results divided by epoch and window size in order to see them more granularly.

Figures 4.6 and 4.7 show some interesting results: the usual recommended value of 0.0001 (or -4 in the log scale) has consistent high scores for windows sizes 3 and 5 for vector sizes 100 and lower (for greater values -4 and -3 are on par). With window size 7 there's a less clear scenario in the higher scores with a mix between all values for the hyperparameter; looking at the box plot, it seems that -3 seemes to have a higher max and median, although the difference looks minimal. Window size 7 seems to have better results than other windows, specially at higher values of vector size; this seems to indicate that the decision of window size could be more important than the sampling value.

Regarding training time, Figure 4.8 shows how increasing the sampling hyperparameter affects its training time; increasing the value means training more words because the probability of discarding a word is lower. It's important to note that although increasing this hyperparameter makes the model to have *somewhat* better results, the training time is considerable higher; hence, a sampling of 0.0001 is good enough because the performance increase is negligible.

epoch	vector size	window size	sampling	negative sampling	average score	time
7	400	7	-2	5	0.450	14.295
7	400	7	-3	5	0.450	11.349
7	300	7	-3	5	0.449	9.468
7	400	7	-4	5	0.446	8.110
7	300	7	-2	5	0.445	11.964

Table 4.1: Top 5 score results for word2vec.

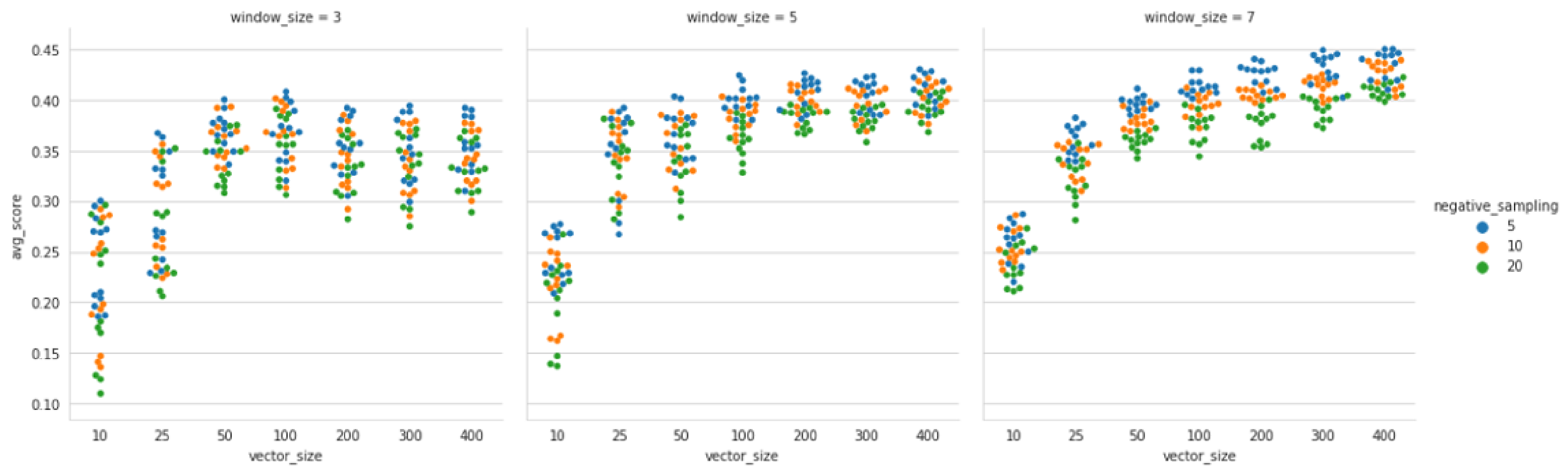


Figure 4.3: Word2vec performance by negative sampling.

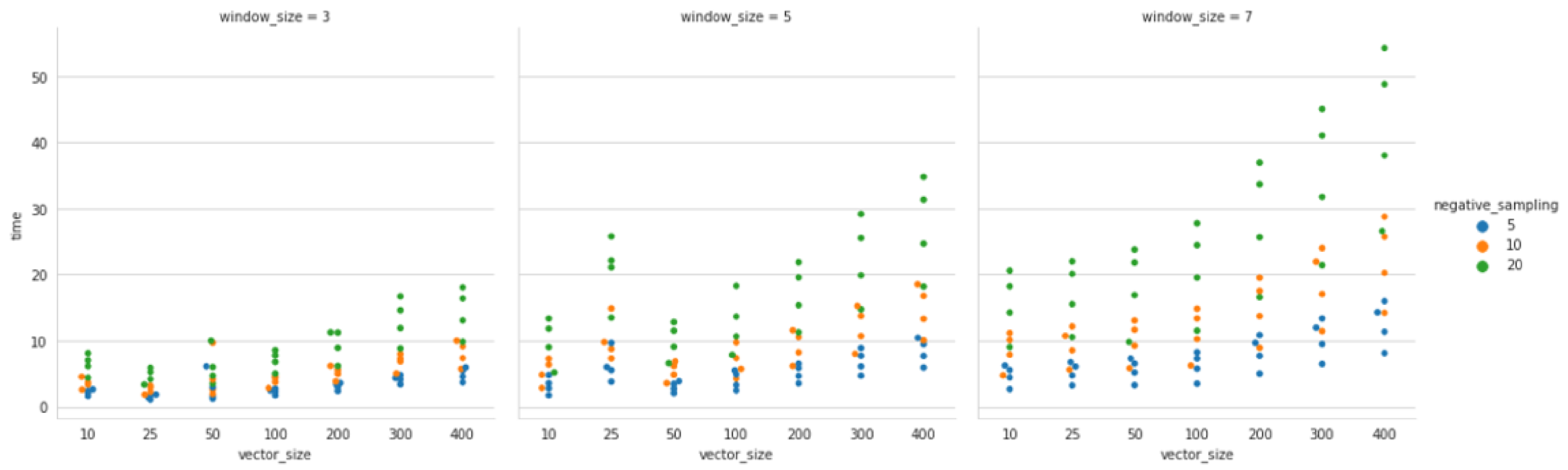


Figure 4.4: Word2vec training time by negative sampling.

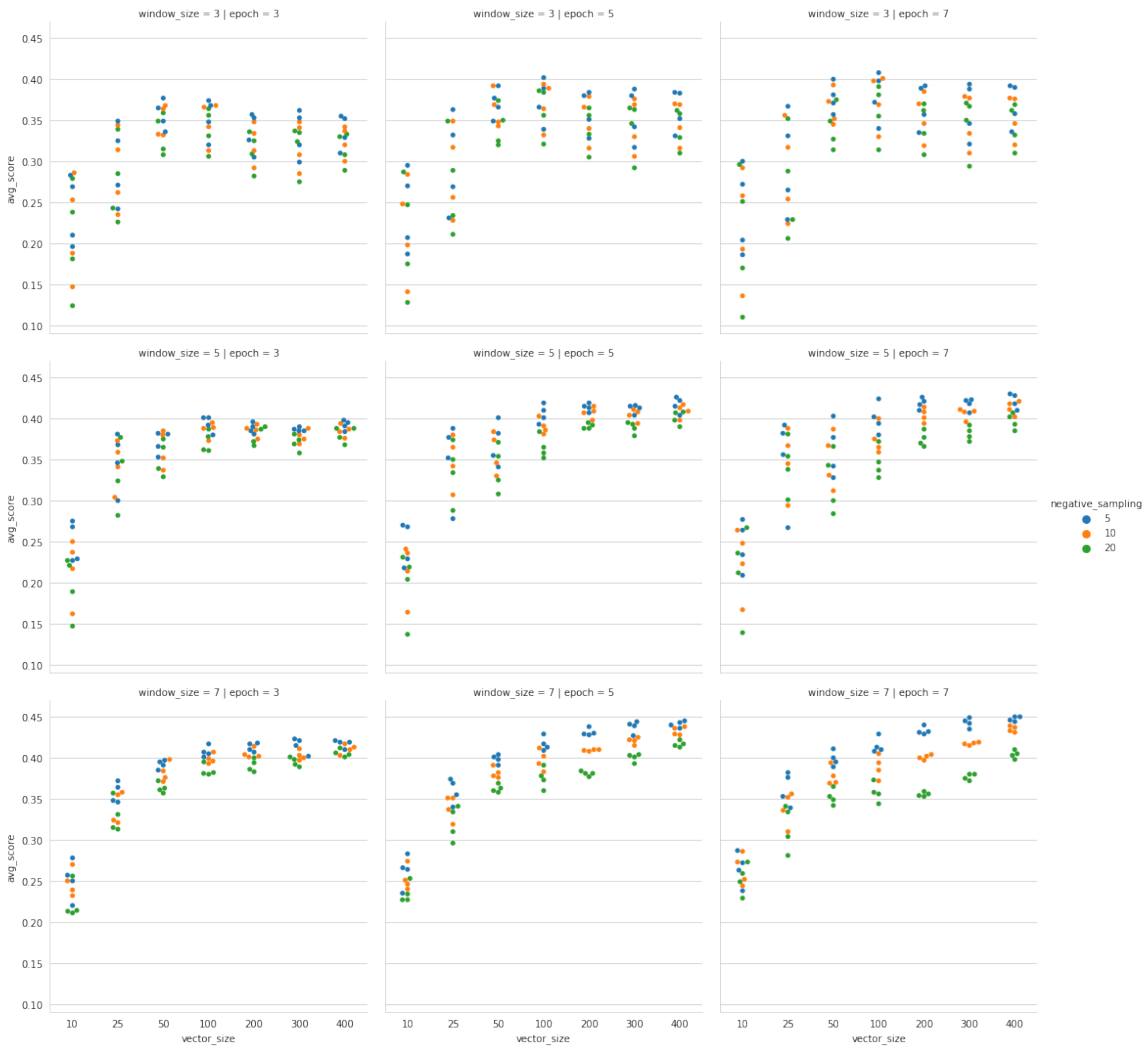


Figure 4.5: Word2vec performance by epoch and negative sampling.

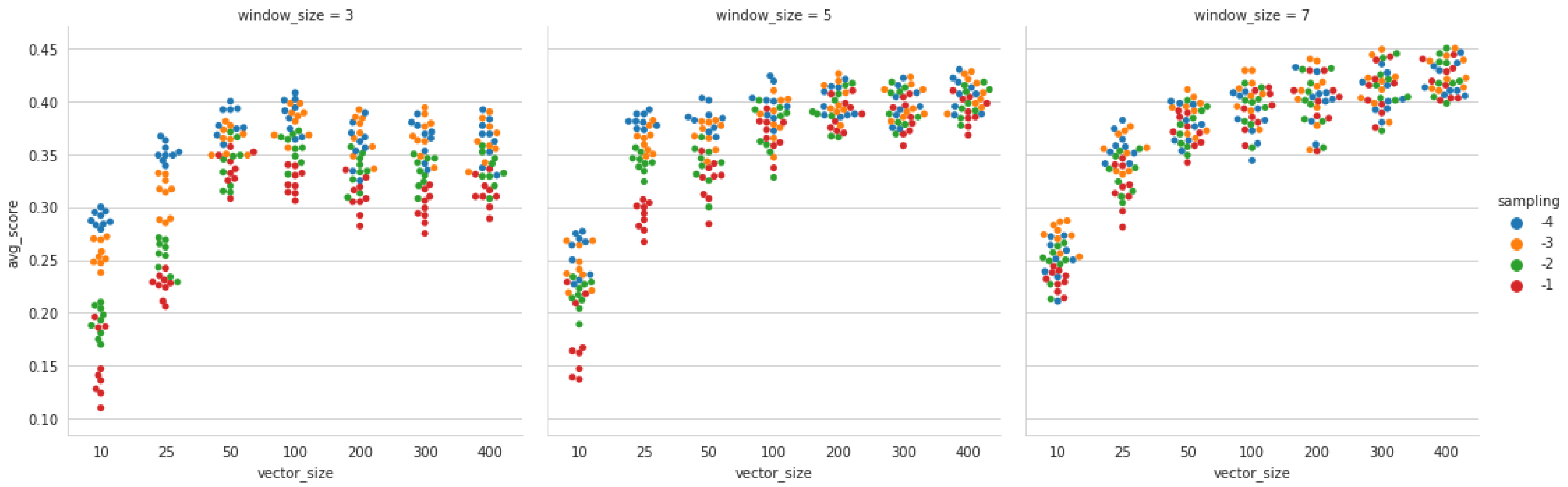


Figure 4.6: Word2vec performance by sampling.

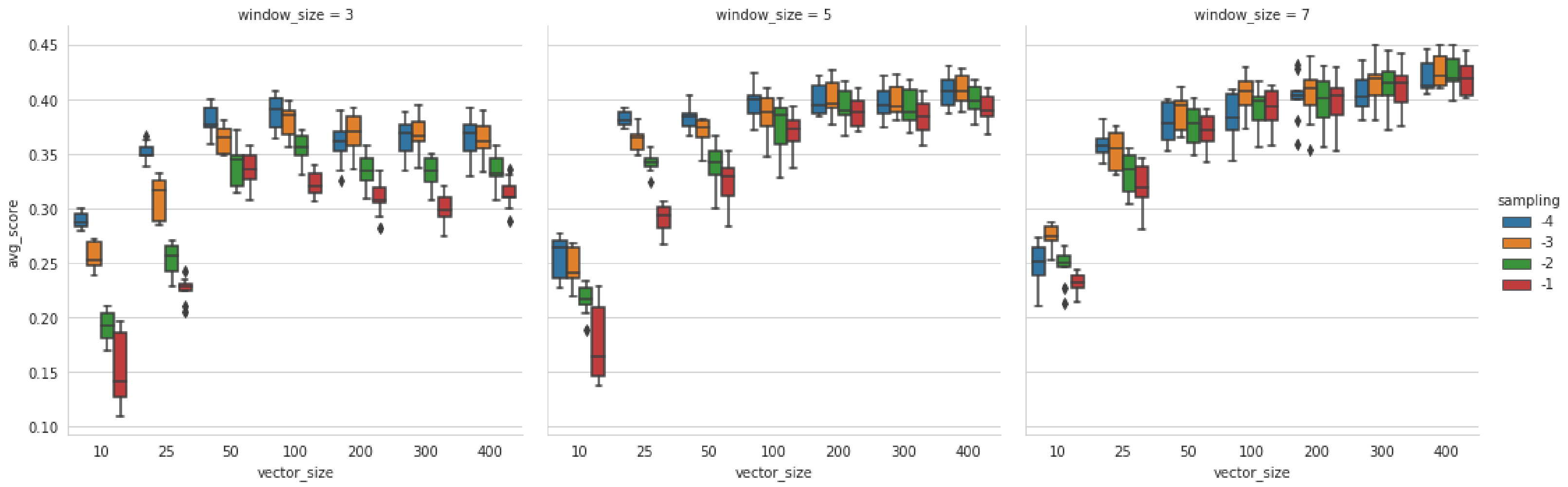


Figure 4.7: Word2vec performance by sampling in boxplot.

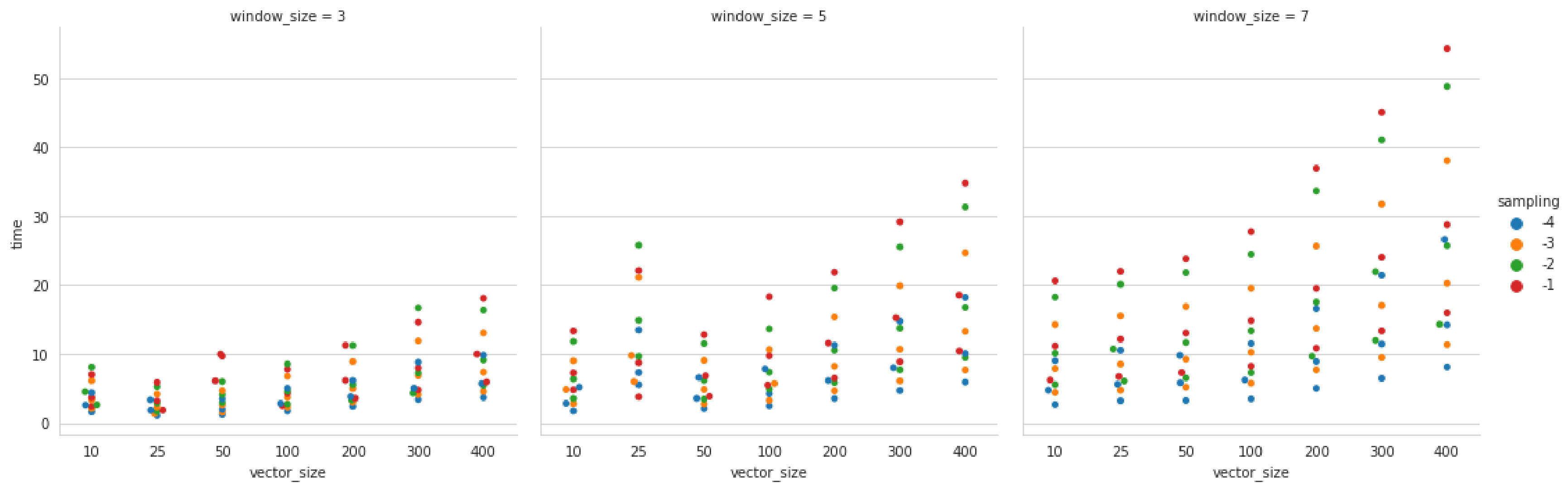


Figure 4.8: Word2vec training time by sampling.

4.1.2 FastText

For fastText, some interesting results were obtained. Figure 4.9 shows how increasing the number of n-grams raises performance; this makes sense because 4-grams and 5-grams are getting closer to actual words. It was surprising to see that 3-grams had very low scores, because one good strength about 3-grams is that it's possible to reconstruct (almost) all words.

Additionally, extra experiments were defined to check the performance of fastText using only subword n-grams and skipping the addition of the whole word; this implementation was named *subwords*. Figure 4.10 shows the subwords embeddings; results are somewhat similar to fastText and also shows better performance as n-grams increase. One key difference between fastText and *subwords* is that words shorter than $ngram\_length - 2$ can't be represented in *subwords*[1]. This can be important if shorter words are needed and it's obviously dependant on n-gram length, with only being a significant problem for n-grams greater than 4. There are some possible solutions to this: for example, using the whole word only in these cases, or adjust the number of n-grams on the fly depending on the length of the word; in this work however, these options were not tried.

Figure 4.11 shows both variants in the same grid for different epochs. Surprisingly, it seems that adding the word doesn't have that much impact in the embeddings, although maybe this doesn't occur when using more training data, bigger vector sizes, or bigger window sizes. In any case, if this holds, then smaller embedding files can be used without too much of a loss in performance.

[1]For example, given 5-grams, the word *dog* can be represented by: *'<dog>'*; on the contrary, a word like *'do'* has the following n-gram: *'<do>'*, which is shorter than a 5 gram and thus, can't be represented. In the case of fastText, the whole word can be used in absence of n-grams.

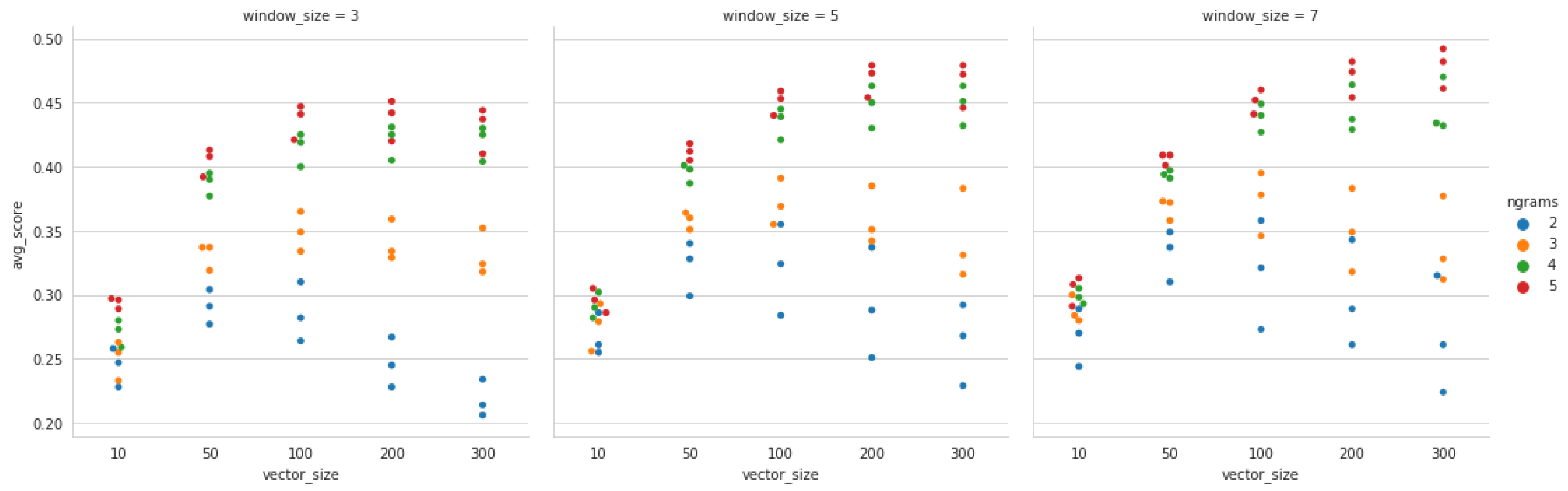


Figure 4.9: FastText performance by n-grams.

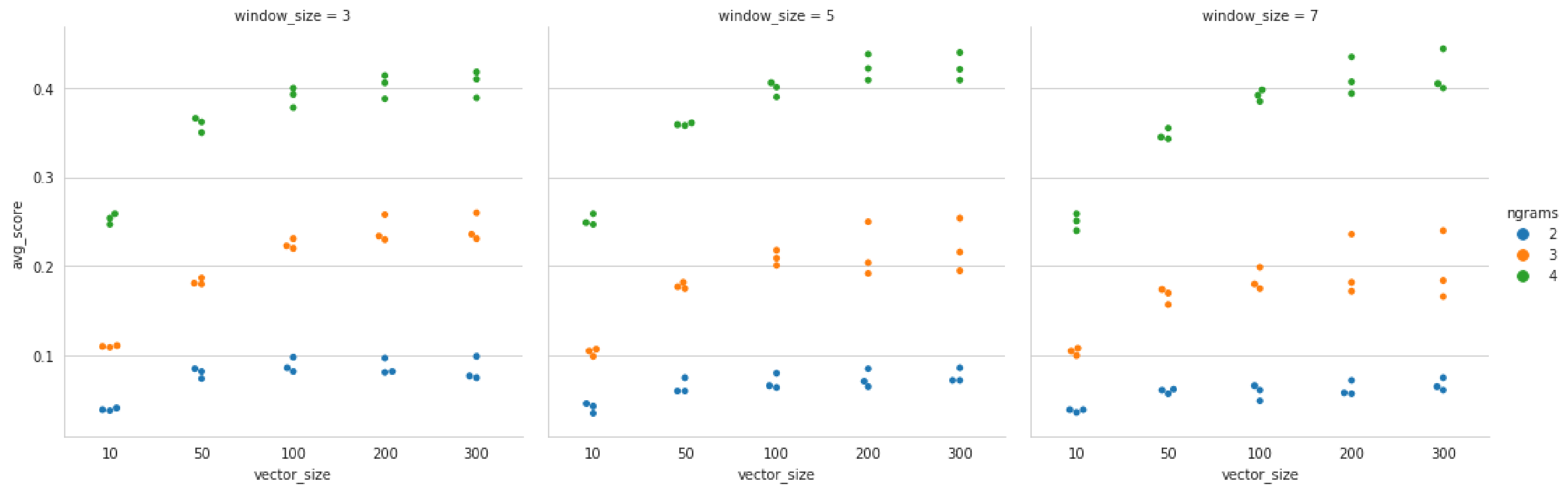


Figure 4.10: Subwords performance by n-grams.

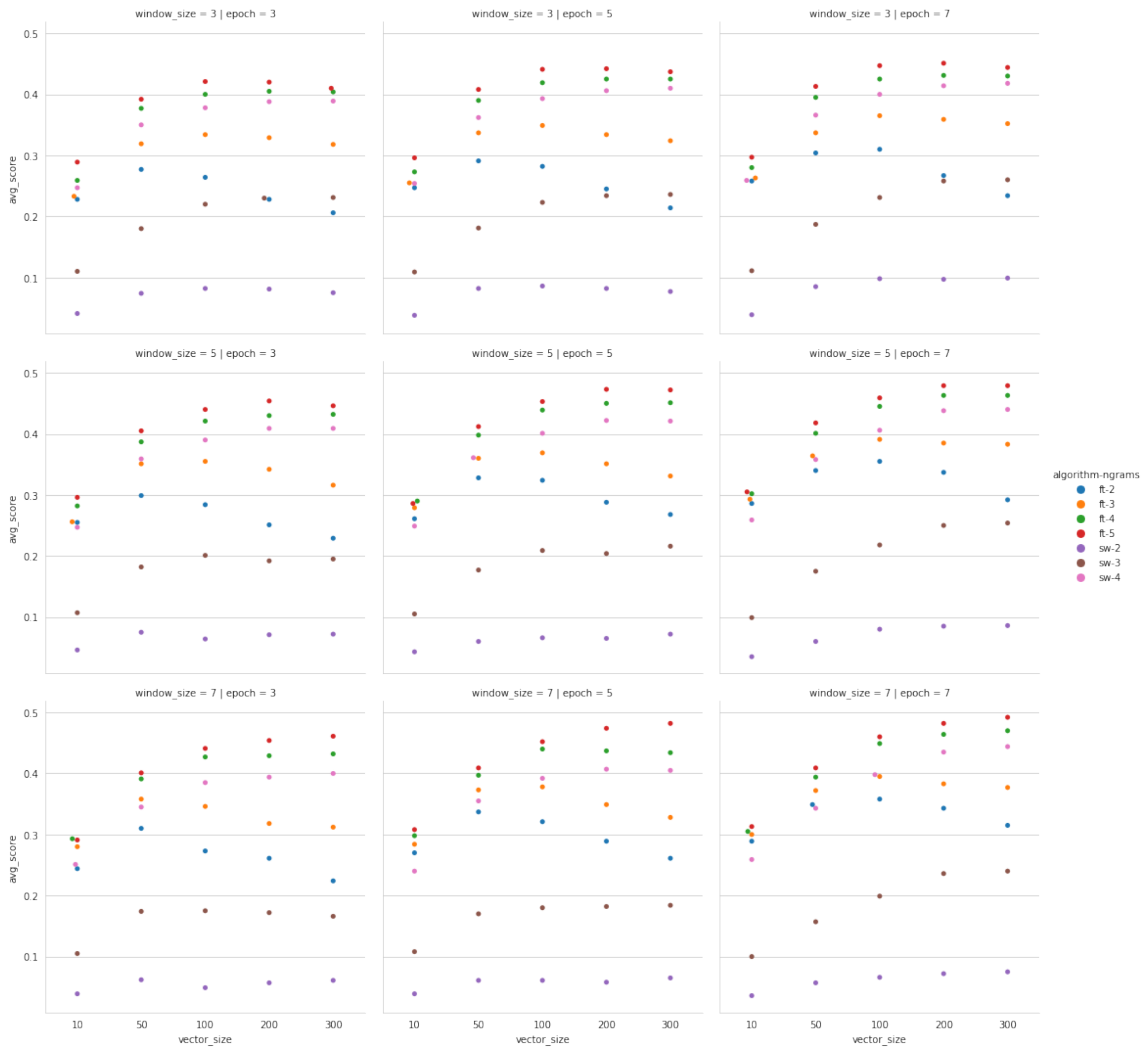


Figure 4.11: FastText and Subwords performance by epoch and n-grams.

algorithm	epoch	vector size	window size	n-grams	average score	time
ft	7	300	7	5	0.492	17.731
ft	5	300	7	5	0.482	17.731
ft	7	200	7	5	0.482	13.161
ft	7	300	5	5	0.479	12.786
ft	7	200	5	5	0.479	9.514
sw	7	300	7	4	0.444	11.336
sw	7	300	5	4	0.440	7.821
sw	7	200	5	4	0.438	5.831
sw	7	200	7	4	0.435	8.429
sw	5	200	5	4	0.422	5.831

Table 4.2: Top 5 score results for fastText (top 5) and fastText with no word added (bottom 5). Time is measured in minutes.

4.1.3 Dict2vec

Figures 4.12 and 4.13 show some interesting results: almost all top scores have values of 4 and 8 for the strong draws and 0.4 and 0.8 for the beta strong; conversely, 2 strong draws are almost at the bottom with a handful at the top. This implies that higher values of the strong set variables have an important role in higher score values. This is slightly different from the results proposed by the authors where 4 strong draws and strong beta 0.4 were mentioned as the optimum; however, the authors did give some intervals where all these variables would be better, so it could depend on the dataset.

Regarding the weak set hyperparameters, there's a similar scenario; however, it's not as clear-cut. Figure 4.14 shows more of a mixture between 5 and 8 weak draws, where there's no clear winner; additionally, there's also some results with 2 weak draws at the top of some of them. In any case, the lower scores seem to be mostly 2 weak draws. Regarding the beta, there's quite a mix too: some results have the lowest value at the top. This seems to indicate that the weak beta is not as important as the other variables.

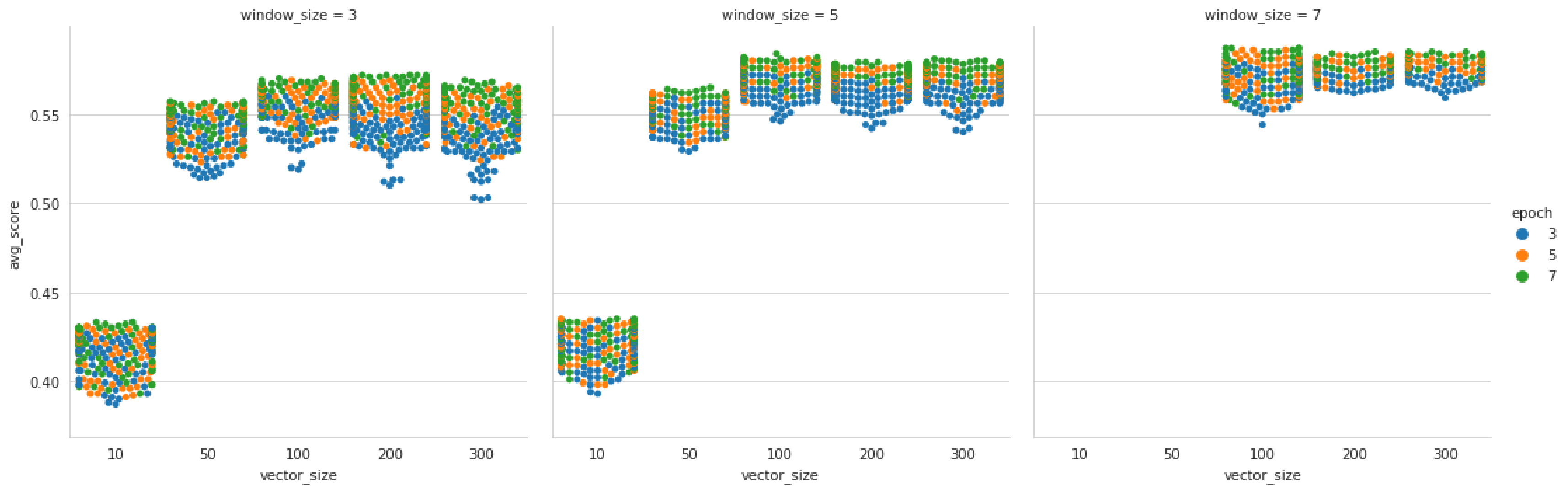


Figure 4.12: Dict2vec performance by epoch

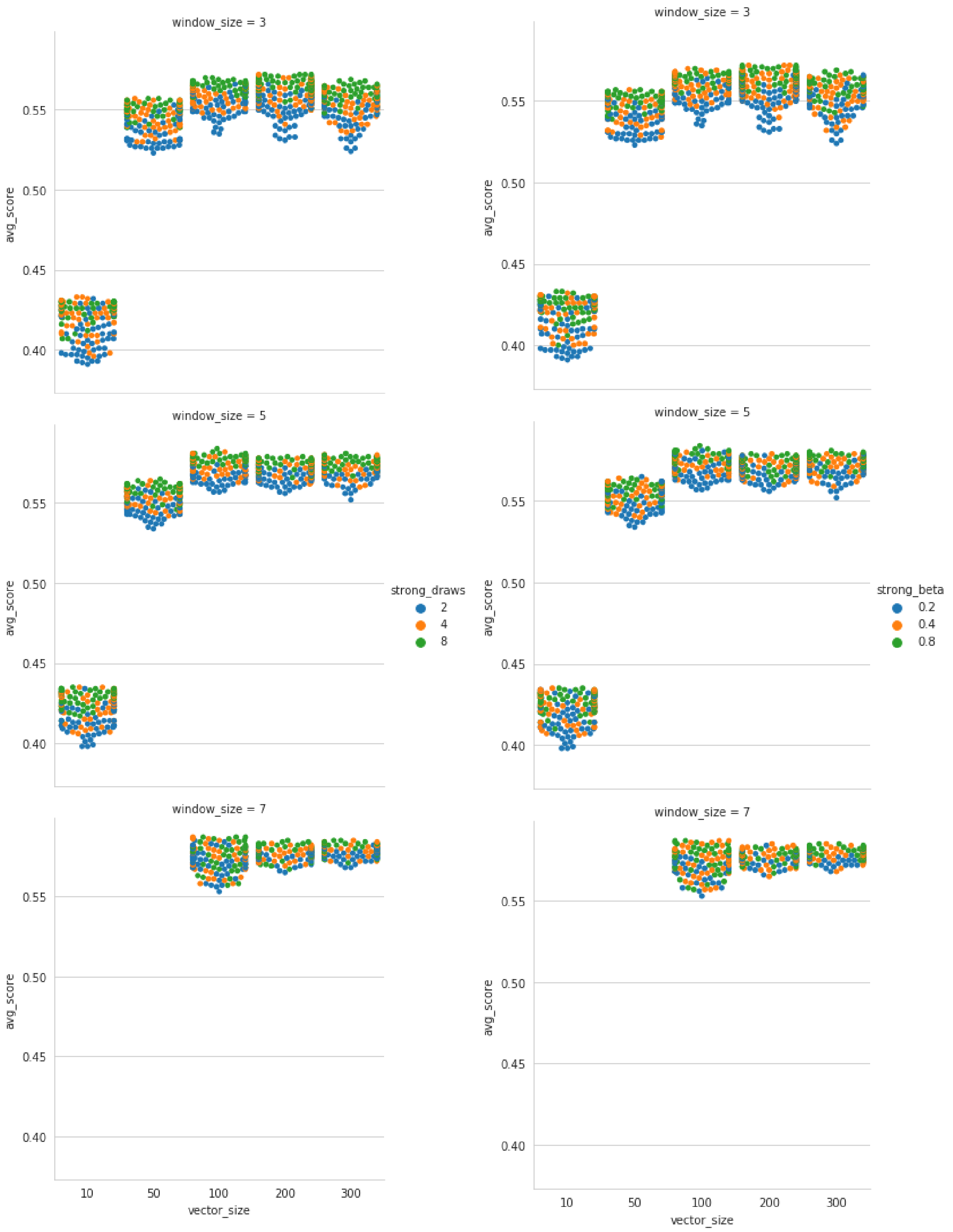


Figure 4.13: Dict2vec performance by strong set draws (left) and beta of strong set sampling (right)

Figure 4.15 shows results by some high hyperparameters (strong draws: 4 and 8, strong

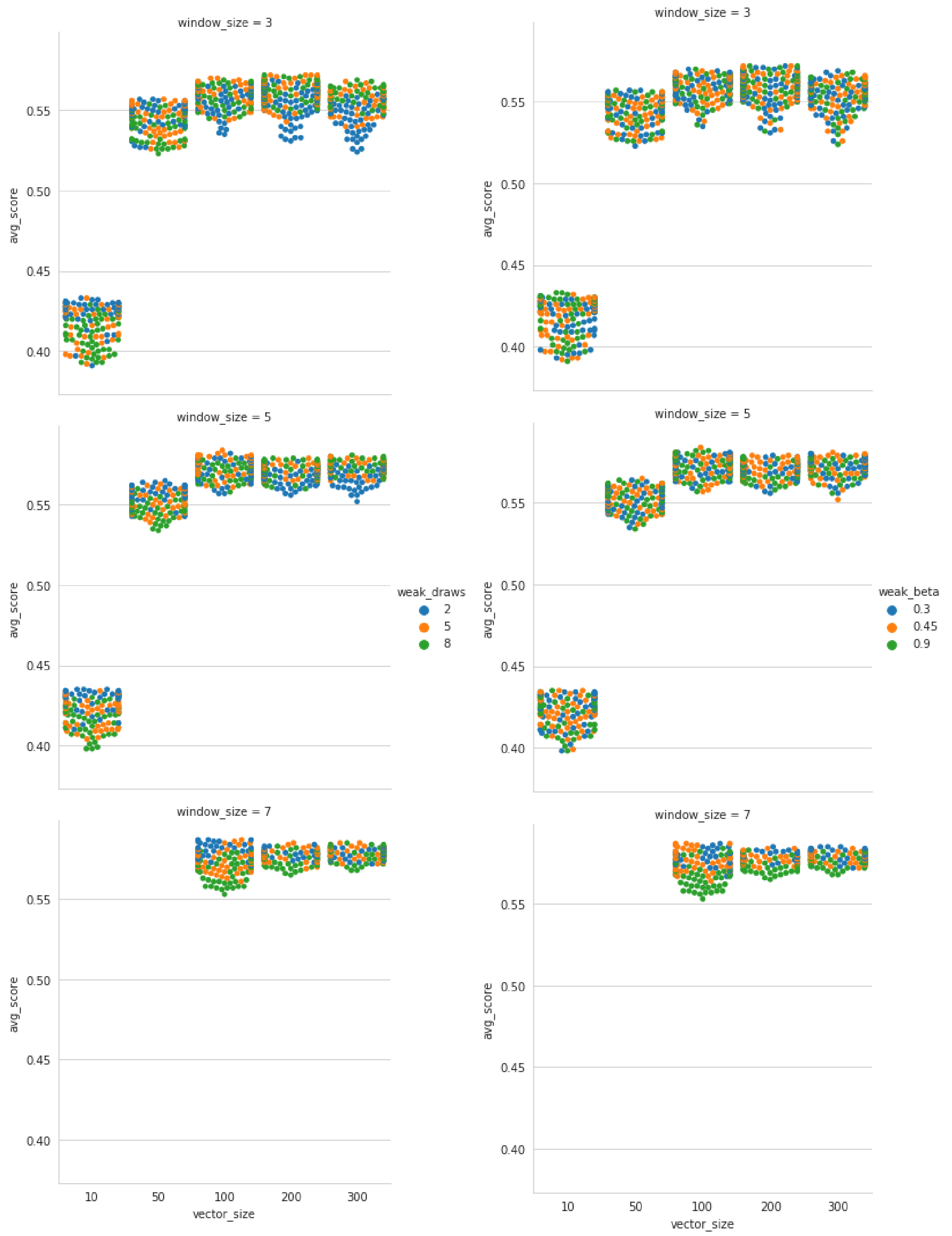


Figure 4.14: Dict2vec performance by weak set draws (left) and by beta of weak set sampling (right)

beta: 0.4 and 0.8, weak draws: 5 and 8, and weak beta: 0.45 and 0.9; vector sizes 100, 200, and 300. Looking at both hyperparameters side by side, it can be noticed that for window size 7, it seems to be there's some correlation for the weak hyperparameters (particulary, the weak beta; lower values have higher scores and vice-versa). For the other window sizes, there's more mixture; like the previous figures, lower values for both the strong draws and strong beta have lower scores. At the top, it could be said that strong beta of 0.8 (orange and red) has higher scores, although there are plenty points of 0.4 with 8 strong draws (green). For the weak hyperparameters is hard to discern some clear pattern.

epoch	vector size	window size	strong draws	strong beta	weak draws	weak beta	average score	time
7	100	7	8	0.8	2	0.45	0.587	11.210
7	100	7	8	0.4	5	0.30	0.587	13.864
7	100	7	8	0.4	2	0.45	0.587	11.410
7	100	7	4	0.8	2	0.45	0.587	10.211
5	100	7	8	0.8	2	0.45	0.586	11.210

Table 4.3: Top 5 score results for dict2vec.

4.1.4 DictFastText

Just like in fastText, Figure 4.16 shows that increasing the number of n-grams also increases the score; however, it has much better overall performance than fastText. Given that 4-grams and 5-grams have much better performance and there's more hyperparameters in this model than the rest of them, only 4-grams and 5-grams will be analyzed for the rest of the hyperparameters. Figure 4.17 shows results by the strong set hyperparameters. Interestingly enough, it's hard to conclude anything, because they look evenly distributed. This could mean that the performance from these hyperparameters is somewhat comparable. One important thing to note is that window size 3 has a lower interval in the low end than the others.

epoch	vec-tor size	win-dow size	n-grams	strong draws	strong beta	weak draws	weak beta	average score	time
7	200	7	5	4	0.8	5	0.45	0.587	22.657
7	200	7	5	4	0.4	5	0.90	0.586	24.335
7	200	7	5	8	0.4	5	0.90	0.586	23.514
7	200	7	5	8	0.4	5	0.45	0.585	24.517
7	200	7	5	4	0.4	5	0.45	0.585	23.260

Table 4.4: Top 5 score results for dictFastText.

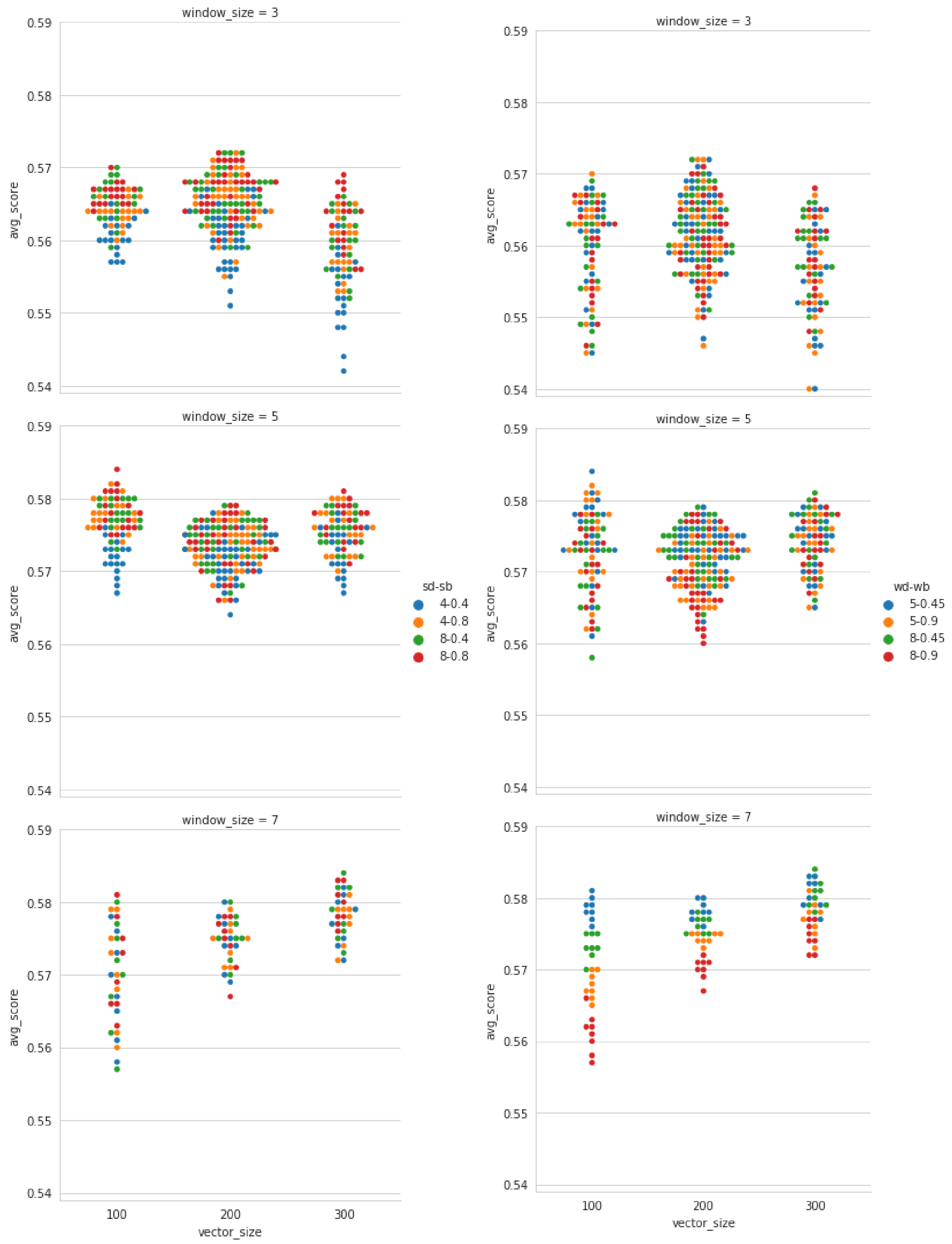


Figure 4.15: Dict2vec performance by strong set hyperparameters (left) and weak set hyperparameters (right)

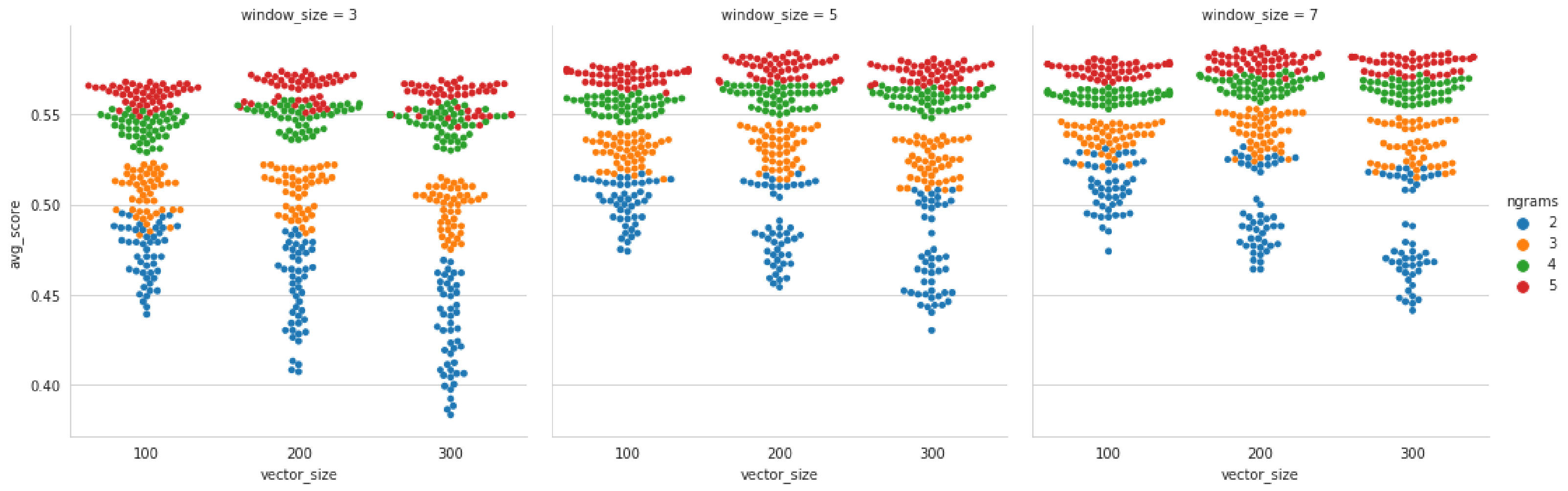


Figure 4.16: DictFastText performance by n-grams.

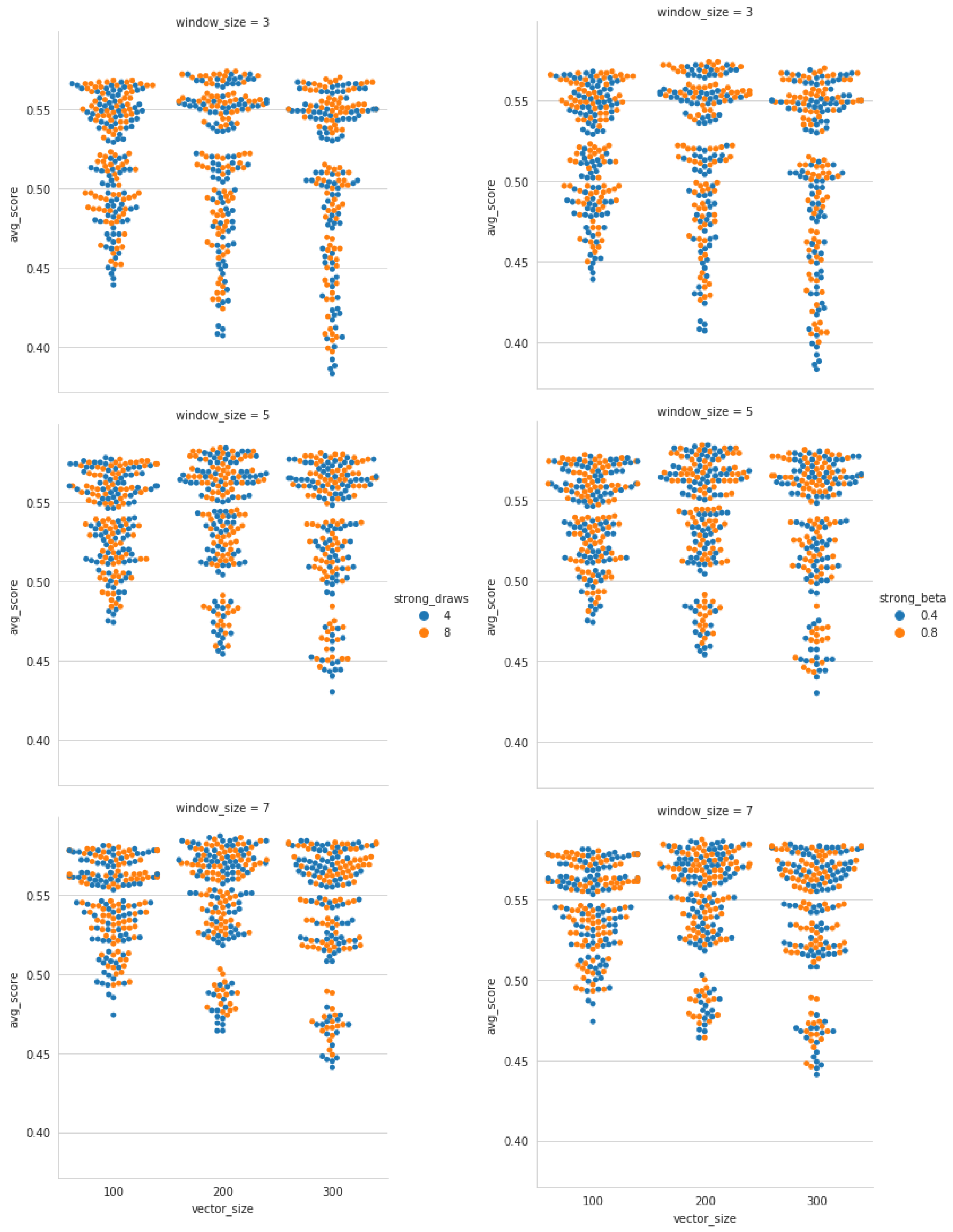


Figure 4.17: DictFastText performance by strong set draws (left) and by beta of strong set sampling (right).

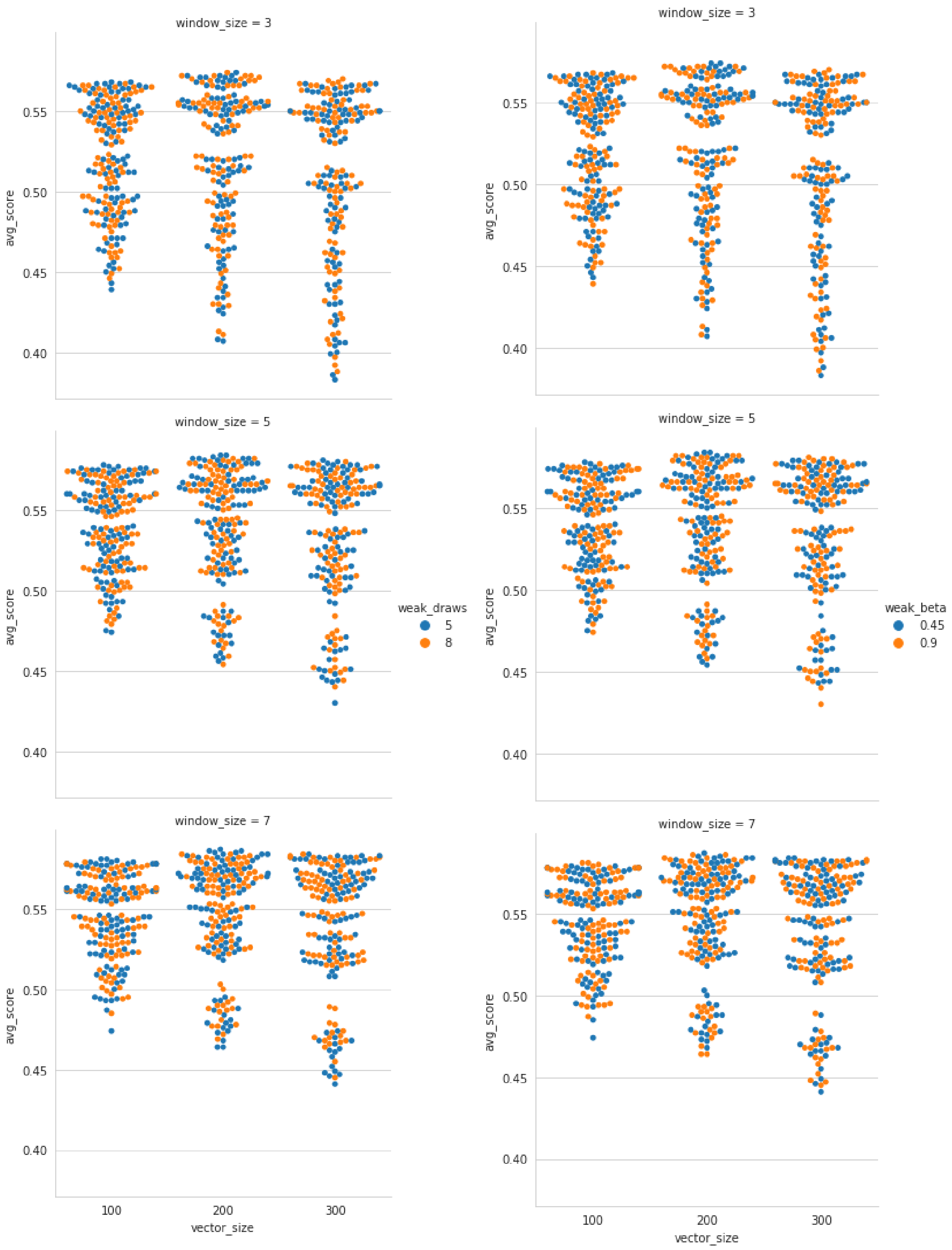


Figure 4.18: DictFastText performance by weak set draws (left) and by beta of weak set sampling (right).

4.1.5 All Models

For this section, only results with epochs 3, 5, and 7; n-grams 3, 4, and 5; and negative sampling 5 and 10 are shown[2]. For referece, the model names are shortened: *d2v* is dict2vec, *dft* is dictFastText, *ft* is fastText, *sw* is subwords, and *w2v* is word2vec. Figure 4.19 shows all models with different vector and window sizes. At first glance, it can be seen that the best ones are dict2vec and dictFastText. DictFastText seems to show better results at vector sizes 200 (in all window sizes) and 300 (the exception being in window size 7, although there's a couple of points at the top). Dict2vec is predominant near the top across all window sizes. Next to them is fastText with 5 n-grams which is slightly over word2vec but under the dictionary based ones. Finally, subword embeddings (fastText with no added word) and word2vec come lower.

Figure 4.20 shows the same graphic but zoomed in. DictFastText has comparable results to dict2vec, but comes with a big trade-off: Figure 4.23 shows training time for dict2vec and dict-FastText (5-grams). DictFastText's training time goes up dramatically as vector and window size are increased; at the most extreme instance, dictFastText takes almost double the time for vector size 300 and window size 7. This means that for bigger datasets, it could take a considerable amount of time. Finally, it's important to emphasize that even if there are improvements over the other implementations, the difference is minimal (in the order of thousandths) for this type of evaluation. With this in mind, the task discussed in the next section shows results that are suggestive of dictFastText's importance as a word embedding model.

[2]It's important to note that not all models share all hyperparameter values; for example, some vector sizes weren't included in dictFastText because of time constraints and lower performance on other models. Please refer to Section 3.5.4 for the exact values.

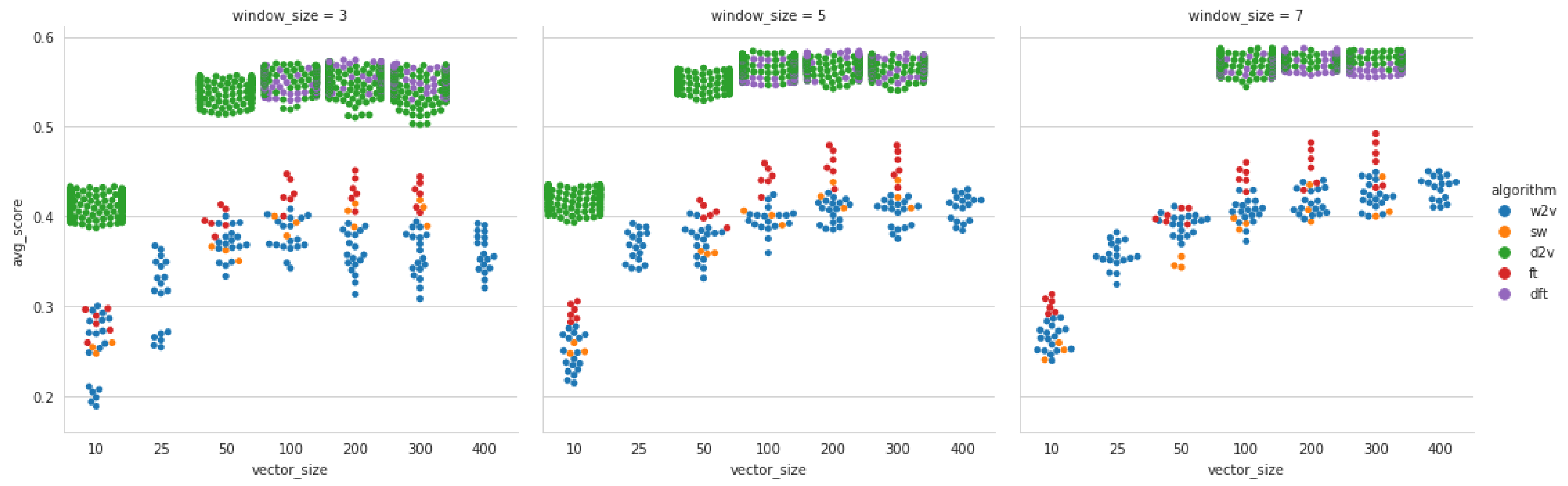


Figure 4.19: Results for all algorithms (4-grams and 5-grams only).

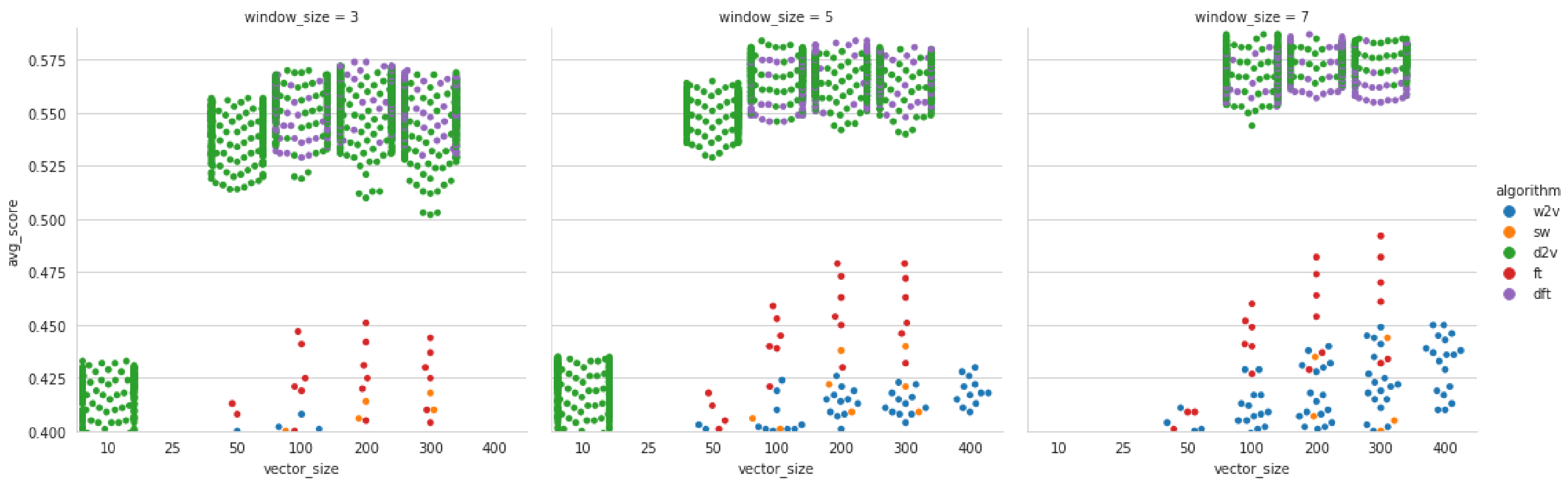


Figure 4.20: Results for all algorithms (zoomed).

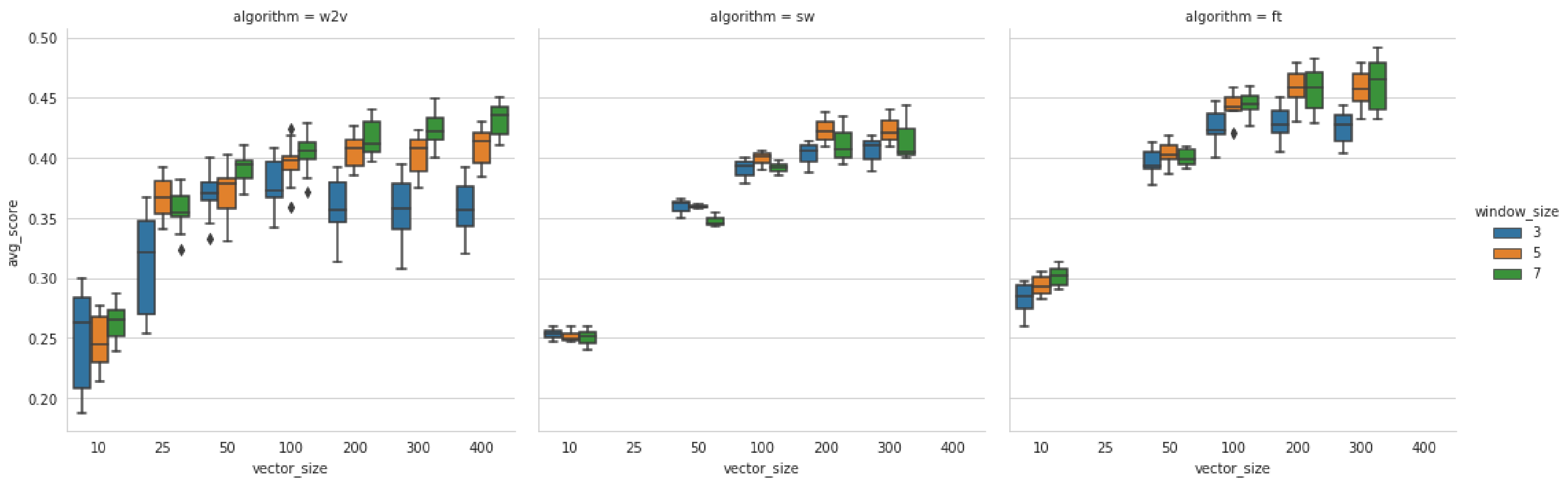


Figure 4.21: Box plot for results of word2vec, fastText and subwords (4-grams and 5-grams only).

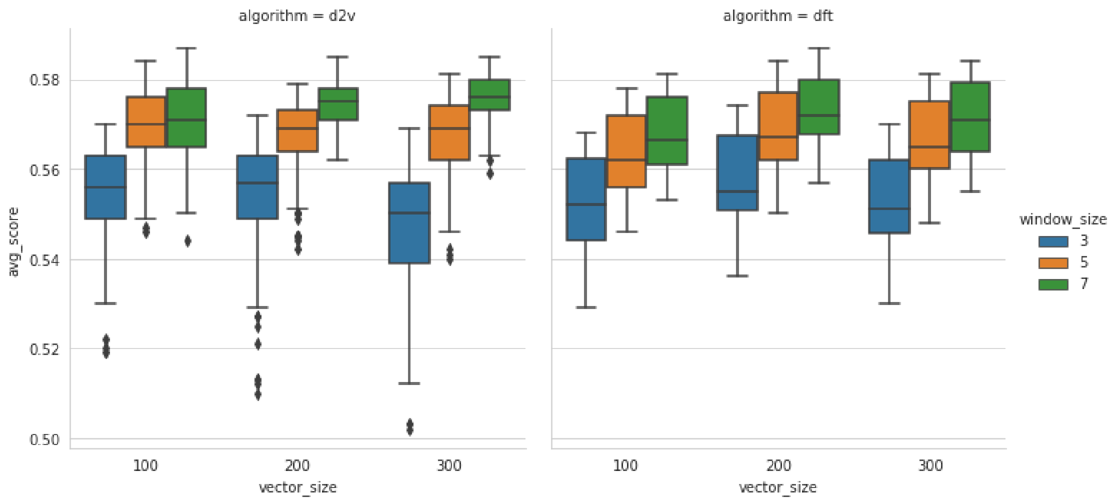


Figure 4.22: Box plot for results of dict2vec and dictFastText (4-grams and 5-grams only).

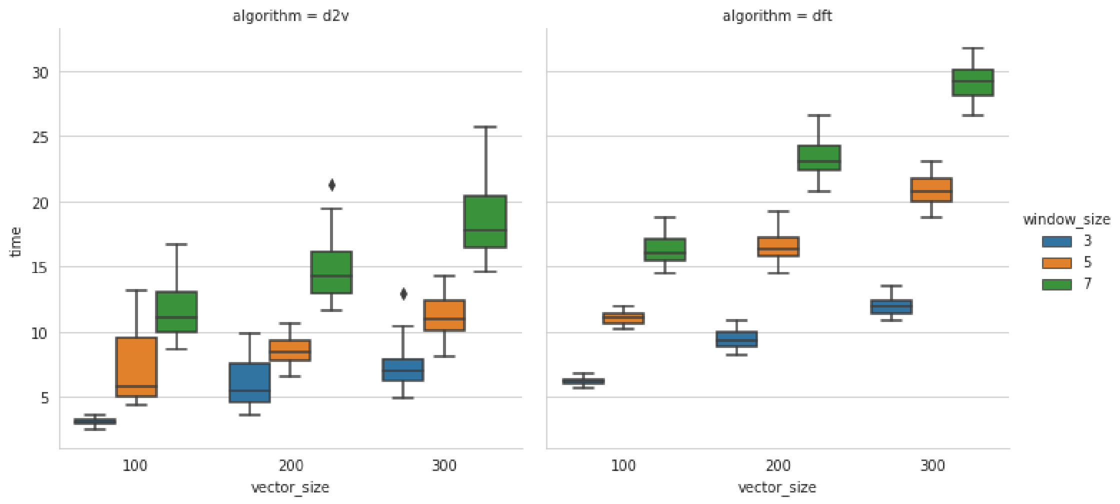


Figure 4.23: Training time of dict2vec and dictFastText (5-grams). Time is measured in minutes.

4.2 Evaluation

4.2.1 Word similarity

It's important to remember that the average score is calculated as a micro average; because all tasks have different number of word pairs, each score is normalized by the amount of word pairs found in the embedding. Table 4.5 shows the top average scores of all algorithms along with their hyperparameters. Notably, dict2vec and dictFastText show very similar results but dict2vec needs lower dimensional vectors and much less training time to reach those scores.

algorithm	epoch	vec-tor size	win-dow size	n-grams	strong draws	strong beta	weak draws	weak beta	avg score	time
d2v	7	100	7	-	8	0.8	2	0.45	0.587	10.211
dft	7	200	7	5	4	0.8	5	0.45	0.587	22.657
ft	7	300	7	5	-	-	-	-	0.492	17.731
w2v	7	400	7	-	-	-	-	-	0.450	14.295
sw	7	300	7	4	-	-	-	-	0.444	11.336

Table 4.5: Top scores and hyperparameters for each algorithm. Time is measured in minutes.

For a more detailed analysis, Table 4.6 shows a breakdown of scores for each evaluation task (detailed in Section 1.4.3). As mentioned previously, both the micro and macro average were included to show how well the models perform across all tasks. It can be noted that the macro average is higher; the main reason for this is due to the tasks *simlex999* and *simverb-3500* which drag the micro average down due to their low scores in all models.

For the best performing algorithms (in almost all cases, dict2vec and dictFastText), the results are fairly similar (the highest score of each row is bolded). There are some tasks that one of them performs slightly better than the other, but the difference is negligible. However, dict2vec has the advantage of having a smaller vector size and a lower training time; one advantage about dictFastText is that there are fewer out of vocabulary words because of the use of subwords, however if a high number of n-grams is needed for better performance then this advantage can be rendered moot.

Notably, fastText has the highest score in one of the tasks (word similarity) and, in general, performs slightly better than word2vec. It's also interesting to note that the scores of fastText for *ws-353* and *rw* tasks are comparable with the results claimed by the authors in the paper where the model was proposed [4]; however, for word2vec, it's not the case. This could be due to the difference in training data, where in this case it was a fraction of Wikipedia, whereas in the paper, the full Wikipedia is trained. It's still interesting that results are somewhat similar (with the exception of *rw-stanford*).

	task type	model				
	-	d2v	dft	ft	w2v	sw
micro average	-	**0.587**	**0.587**	0.492	0.45	0.444
macro average	-	0.669	**0.67**	0.603	0.552	0.532
mc-30	similarity	0.814	**0.821**	0.769	0.759	0.634
men-tr-3k	similarity	0.735	**0.741**	0.71	0.651	0.68
mturk-287	relatedness	0.63	0.633	**0.653**	0.622	0.609
mturk-771	relatedness	0.683	**0.694**	0.616	0.586	0.569
rg-65	similarity	0.836	**0.871**	0.716	0.686	0.616
rw-stanford-2034	similarity	0.528	**0.543**	0.468	0.375	0.372
rw-stanford-2034 OOV	-	37%	8%	8%	37%	**2%**
simlex999	similarity	**0.477**	0.473	0.369	0.365	0.342
simverb-3500	similarity	0.438	**0.439**	0.252	0.225	0.222
simverb-3500 OOV	-	2%	1%	1%	2%	**0%**
ws-353-all	-	**0.755**	0.748	0.704	0.675	0.634
ws-353-rel	relatedness	**0.694**	0.684	0.666	0.604	0.612
ws-353-sim	similarity	**0.766**	0.751	0.751	0.739	0.666
yp-130	similarity	**0.672**	0.64	0.567	0.343	0.429
yp-130 OOV	-	3%	3%	3%	3%	**0%**

Table 4.6: Task score breakdown for each algorithm's best performance. The number in the name of each task corresponds to the number of samples in said task. Model names have been shortened due to space reasons: *d2v* is dict2vec, *dft* is dictFastText, *ft* is fastText, *sw* is subwords, and *w2v* is word2vec.

On another note, subwords has the fewest OOV words of all the algorithms and its performance is slightly lower than word2vec, with the exception of *yp-130* and *men-tr-3k* where the difference is considerable higher in favor of subwords. Notably, the 2% in the stanford dataset is due to hyphenated compound words; as mentioned before, the preprocessing of the training data eliminated all hyphens, so the model didn't learn those kind of subwords.

4.2.2 Word analogy

Figure 4.24 shows spectral clustering (for more specific details check Appendix C.2) applied to the models (subwords was excluded). Interestingly, fastText has the best performance of all algorithms, with the only exception at 20 clusters, where dictFastText surpasses it. Moreover, as the number of clusters increase, dictFastText shows a steady decrease, where fastText shows some sort of increasing tendency.

Also, in a surprising manner, dict2vec shows the worst performance in this task in relation to the others. Apparently the semantic fine-tuning given by the dictionary definitions makes the embedding to lose some properties in the vector difference area (which makes sense, because

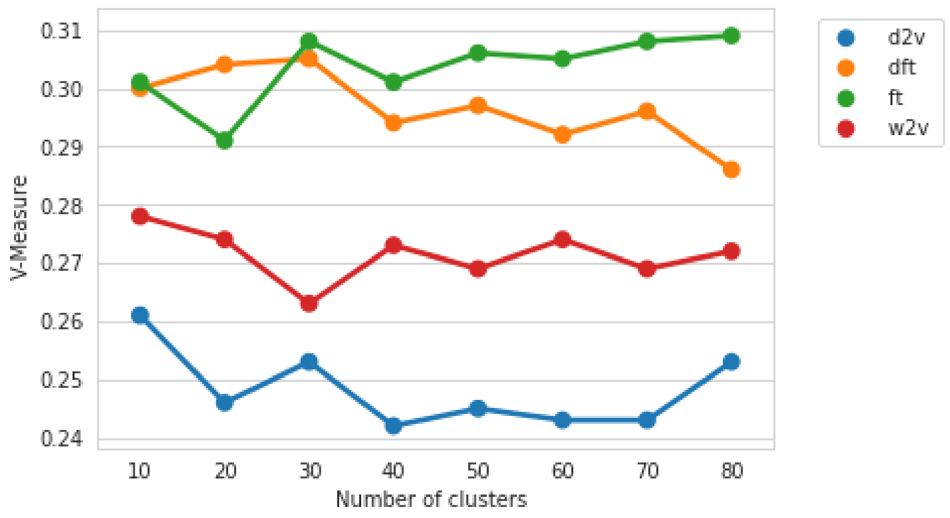


Figure 4.24: Spectral clustering applied to the models on the diffvec dataset.

this fine-tuning is based on definitions, which is a type of description and not usage of the term, where analogies are commonly seen). This seems to indicate that there's an inverse relationship between the two tasks (word similarity and word analogies); however, more research is needed to confirm this hypothesis. Lastly, this also pinpoints that these different effects should be considered when expecting to deploy the embeddings to a real application, because improvement over one task at expense of the other could have unexpected side-effects.

Finally, these results show a different view of the models; dictFastText is on par with dict2vec in the word similarity task and does better than the baseline in the word analogy task. This means that when considering all tasks, dictFastText does better overall than the rest at the expense of higher vector size, a bigger vocabulary, and more training time.

Chapter 5

Conclusions

This chapter is about commenting the results shown in the last chapter and try to derive meaningful conclusions. It's important to note that these are early conclusions based on this particular dataset and may not hold true in a different scenario (which in itself is an interesting situation), meaning that more experimentation is needed to confirm these indications. Additionally, the fact that grid search in itself is an extensive endeavor, limiting the number of values that went into the experiments, makes the experiments not exhaustive. Moreover, following one of the objectives of this thesis, evaluating the models under the same implementation, in order to measure how much a small improvement (such as normalizing subwords or fine-tuning) over the vectors affects results, the models were evaluated for the properties of word similarity and word analogy to check if there's a correlation between the results of the two tasks.

Regarding results, both dict2vec and dictFastText are considerable better than word2vec and fastText on the word similarity and word relatedness tasks. Some of the recommended values for the hyperparameters were confirmed for dict2vec while others weren't (as previously mentioned, it's important to remember that not all ranges for the hyperparameters were explored). It could also mean that the suggested values are good in general, but for every specific case there could be a better value, being dependant on the size or type of training data, or even other hyperparameters. In this regard, we can safely conclude that adding up semantic information taken from word associations given by their dictionary definitions provides a considerable increase in the word similarity and word relatedness tasks while not adding a substantial training time. In general, lower values of window and vector sizes perform worse, while performance gets better from window size 5 and vector size 100, although this depends on the model being used.

In the case of the subword-based models, adding full words to fastText doesn't seem to add that much information; disregarding the cases where n-grams are much bigger than the shortest words, not adding the word produces somewhat similar results (performance-wise) which makes it an interesting trade-off. The main drawback of n-grams is slightly more training time, and this is obvious because now training a word takes more cycles due to the normalized sum of the grams. Surprisingly, even though that fastText does indeed perform better than word2vec in the word similarity tasks, the difference is not that much, however, it's still important. In spite of this, the word analogy task shows a different strength of fastText.

Going back to the word similarity tasks, although combining the dictionary information with the subwords modelling in dictFastText provides a generous increase to performance, it comes with the penalty of a significantly higher training time. As mentioned before, this is because of the n-grams word construction (which is the only difference between dict2vec and dictFastText). An additional important thing to consider (and this also counts for dict2vec) is that extracting the dictionary data is extremely time-consuming and is subjected to some external factors (API access, IP blocking). In comparison with dict2vec, its average performance is on par, with some small differences in some tasks.

That being said, in the word analogy task the scenario is quite different: fastText shows the highest V-measure of all models with dictFastText following next. Surprisingly, dict2vec has the lowest V-measure of all models, which seems to indicate that fine-tuning the vectors with the dictionary information somehow affects this property, even though that word similarity score goes higher. It also seems that the subword embeddings represent word analogies better. One potential reason for this, is that several of the lexical relations defined in *diffvec* have syntactic patterns (for example, prefix rule, verb conjugations, pluralization, etc.), which this approach can address.

In conclusion, if training time is not an issue and the dictionary information is available, dictFastText looks like a reasonable alternative for having good scores on both types of tasks, doing just as good as dict2vec in the word similarity task and slightly worse than fastText in the word analogy task. Nevertheless, more experiments and more tasks are needed to confirm if this pattern is consistent for other scenarios.

Chapter 6

Future Work

Following the conclusions detailed in the last chapter, this chapter mentions some aspects about the experiments that can be improved or worked on in order to strengthen or confirm the ideas presented in this thesis.

Bigger corpus

In this work, only the smallest version (the first 50 million words from Wikipedia) was used as training data. Trying the whole Wikipedia corpus could show interesting results (although the authors already proved that it doesn't change the results too much). To have a perspective, the 50 million words set is 284 megabytes, the first 200 million words set is 1.1 gigabytes, and the full set is 24 gigabytes.

Additionally, other corpora can be explored.

Other Languages

FastText has made available several embeddings of other languages; while this task needs further work (specially to normalize the data in other languages, and also because Wikipedia in other languages can be different to the english one), it can result in notable insights.

One costly operation would be to find dictionary information in other languages, which could be a hindrance.

Additional Hyperparameter Values

Several hyperparameters weren't thoroughly explored; for example: number of n-grams and some ranges & values in the dict2vec hyperparameters.

Dictionary Information

While it's clear that the semantic information from dictionary definitions have rich content for the embeddings, different approaches can be taken to try to include this information on the embeddings (for example, not using word pairs but instead defining sentences, context windows or using part of speech tags). Additionally the cost of extracting the definitions is quite high and it could not necessarily work for certain types of words (compound words for example).

While the authors of dict2vec already analyzed resources such as WordNet, it would be interesting to see if there's an easier or less-expensive way to get related word pairs (not necessarily semantic); for example, words related by a Thesaurus, or grouping from a hand-made dataset.

Bibliography

[1] Mikel Artetxe and Holger Schwenk. Massively multilingual sentence embeddings for zero-shot cross-lingual transfer and beyond. *CoRR*, abs/1812.10464, 2018.

[2] Amir Bakarov. A survey of word embeddings evaluation methods. *CoRR*, abs/1801.09536, 2018.

[3] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[5] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder. *CoRR*, abs/1803.11175, 2018.

[6] Patrick Conley and Curt Burgess. A computational approach to modeling population differences. *Behavior Research Methods, Instruments, & Computers*, 32(2):274–279, Jun 2000.

[7] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 41(6):391–407, 1990.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[9] Felix Hill, Kyunghyun Cho, Anna Korhonen, and Yoshua Bengio. Learning to understand phrases by embedding the dictionary. *CoRR*, abs/1504.00548, 2015.

[10] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. Deep unordered composition rivals syntactic methods for text classification. pages 1681–1691, 01 2015.

[11] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[12] Wang Ling, Chris Dyer, Alan Black, and Isabel Trancoso. Two/too simple adaptations of word2vec for syntax problems. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2015.

[13] Kevin Lund and Curt Burgess. Producing high-dimensional semantic spaces from lexical co-occurrence. *Behavior Research Methods, Instruments, & Computers*, 28(2):203–208, Jun 1996.

[14] Ulrike Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, December 2007.

[15] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. Learned in translation: Contextualized word vectors. *CoRR*, abs/1708.00107, 2017.

[16] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[17] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119. Curran Associates, Inc., 2013.

[18] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *In EMNLP*, 2014.

[19] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

[20] Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[21] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning(EMNLP-CoNLL)*, pages 410–420, 2007.

[22] Xinying Song, Xiaodong He, Jianfeng Gao, and Li Deng. Unsupervised learning of word semantic embedding using the deep structured semantic model. Technical Report MSR-TR-2014-109, August 2014.

[23] Julien Tissier, Christopher Gravier, and Amaury Habrard. Dict2vec : Learning word embeddings using lexical dictionaries. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 254–263. Association for Computational Linguistics, 2017.

[24] Andrew Trask, Phil Michalak, and John Liu. sense2vec - A fast and accurate method for word sense disambiguation in neural word embeddings. *CoRR*, abs/1511.06388, 2015.

[25] Peter D. Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *J. Artif. Int. Res.*, 37(1):141–188, January 2010.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[27] Ekaterina Vylomova, Laura Rimell, Trevor Cohn, and Timothy Baldwin. Take and took, gaggle and goose, book and read: Evaluating the utility of vector differences for lexical relation learning, 2015. cite arxiv:1509.01692.

Appendix A

Word embeddings

Word embeddings is the collective name of several models used in NLP where features (such as word or phrases) are represented as vectors of real numbers. Word embedding is also known as the Distributional Semantic Model in the Computational Linguistics community, where the main idea is the Distributional Hypothesis: *"linguistic items with similar distributions tend to have similar meanings"* [25].

The first model to use vectors as word representations was the Vector Space Model, which led eventually to the development of Latent Semantic Analysis by means of applying the Singular Value Decomposition to the original model.

There's different types of models nowadays, the most known are the ones that spawned from the Probabilistc Neural Language Model proposed by Bengio et al. [3], namely word2vec.

A.1 Word representation

In word embedding models, a word is represented as an n-dimensional vector of real numbers. The number of dimensions will depend on which model is used:

A.1.1 Bag of words

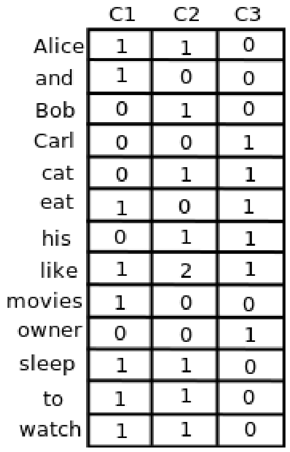
In this representation, the number of dimensions will be the number of *documents*[1] and the value of each dimension will be the number of times the word appears in said document.

Example of 3 contexts:

- Alice likes to watch movies, eat and sleep.

[1]An instance or context where a word appears; it can be a sentence, paragraph, section or even a full text-document. It's named *document* because it first spawned from the vector space model where a term-document matrix is used.

- Bob likes to watch his cat sleep like Alice.
- Carl's cat eats like his owner.



	C1	C2	C3
Alice	1	1	0
and	1	0	0
Bob	0	1	0
Carl	0	0	1
cat	0	1	1
eat	1	0	1
his	0	1	1
like	1	2	1
movies	1	0	0
owner	0	0	1
sleep	1	1	0
to	1	1	0
watch	1	1	0

Figure A.1: Example of a term-document matrix, with the columns being the 3 context examples. Keep in mind that conjugated verbs are considered in their infinitive form for simplicity purposes.

In summary, this model represents the words by their frequency of appearance across all contexts known by the model regardless of word order.

A.1.2 One hot encoding

In this representation, the frequency of the values of the n-dimensional vector are not important, so if a word appears in a context (this context being, for example, the vocabulary or a row or column in a co-occurrence matrix) it's just marked as a one.

A.2 Word similarity

The Distributional Hypothesis (from the distributional semantics field) states that: "linguistic items with similar distributions tend to have similar meanings"[2]. This is often interpreted in practice as "words that appear in similar contexts, *tend* to have similar meanings"; intuitively, if 2 words can be interchangeable in a given piece of text without losing general meaning, then those words can be considered similar.

Now, how to *quantify* that similarity is a problem that has been tackled for a long time with several approaches. Because word embeddings allow for words to be compared as vectors, a lot of options are available. Some of them are:

[2]https://en.wikipedia.org/wiki/Distributional_semantics

- Dot product
- Cosine similarity
- Euclidean distance

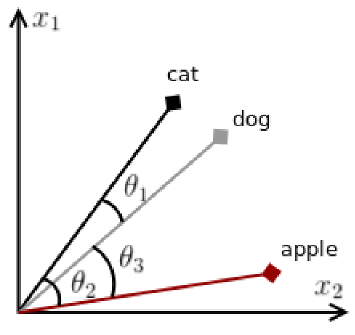


Figure A.2: Example of vector visualization

It's important to note that cosine similarity is one of the most used because it doesn't get affected by the magnitude of the vectors (as opposed with the other 2 distances); also, cosine similarity can be interpreted as the normalized dot product. There are more other metrics to compare vectors, but there hasn't been any substantial use in NLP works.

Appendix B

Spearman's Rank Correlation Coefficient

The Spearman's rank correlation coefficient[1], normally noted as ρ or r_s, is a measurement of correlation between the ranked values of two variables through the use of a monotonic function; in other words, the Spearman correlation determines if the variables show a monotonic relationship. A monotonic relationship occurs when the values of both variables are non-decreasing (positive correlation) or non-increasing (negative correlation); roughly speaking, if one variable increases or doesn't change when the other increases, or if the variable decreases or doesn't change when the other decreases, then they are monotonic. Figure B.1 shows examples of monotonic and non-monotic functions.

For two variables X and Y, with their ranks calculated as $R(x)$ and $R(y)$ respectively, the Spearman's coefficient is:

$$\rho_s = \frac{\sum_i (R(x_i) - \overline{R(x)})(R(y_i) - \overline{R(y)})}{\sqrt{\sum_i (x_i - \overline{x})^2 \sum_i (y_i - \overline{y})^2}} \tag{B.1}$$

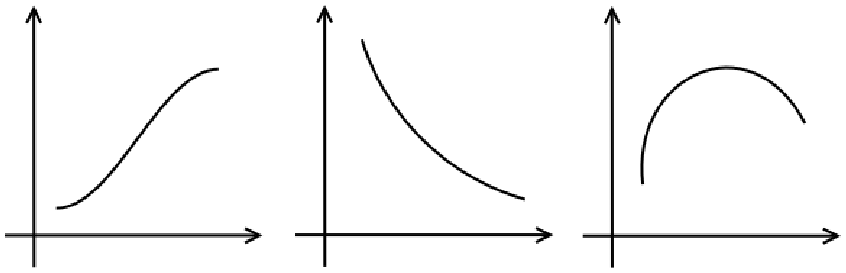


Figure B.1: Examples of monotonic and non-monotic functions. The left and the middle image are monotonic functions, whereas the right one is non-monotonic.

[1]https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

Appendix C

Diffvec

This paper [27] evaluates several types of word embeddings, measuring how good the embeddings are to model *lexical relations* by analyzing the vector differences produced by them. The authors argue that because these vector differences (***diffvecs***) are so prevalent in *analogical reasoning*[1], then the "diffvec themselves must be meaningful".

C.1 Lexical Relations

A lexical relation is a relation r between a pair of words (w_i, w_j). For example, (*light*, *illumination*) belong to the relation CAUSE-PURPOSE. Some tasks of NLP where this is used include: relation learning, where a word pair is classified as a certain relation; relation extraction, where relations are learned from text given a corpus; and analogical reasoning, where there's a need to complete analogies in the form of $A : B :: C :?$.

Image C.1 describes the dataset; it shows the relations, the name of the dataset where it comes from, and some examples. It's important to mention that while the image shows 15 types of relations, there are actually more (36 to be precise); this is because some of the relations have sub-types, such as CAUSE-PURPOSE having both **Instrument:Goal** and **Instrument:IntendedAction** as sub-types.

C.2 Spectral Clustering

Spectral clustering is a technique that employs the eigenvectors of a similarity matrix to reduce the dimensionality of the data, where then a clustering algorithm is applied (like k-means). First, a similarity matrix must be constructed. To construct it, a similarity graph is defined;

[1] $king - man + woman = queen$

Relation	Description	Pairs	Source	Example
LexSem$_{\text{Hyper}}$	hypernym	1173	SemEval'12 + BLESS	(*animal*, *dog*)
LexSem$_{\text{Mero}}$	meronym	2825	SemEval'12 + BLESS	(*airplane*, *cockpit*)
LexSem$_{\text{Attr}}$	characteristic quality, action	71	SemEval'12	(*cloud*, *rain*)
LexSem$_{\text{Cause}}$	cause, purpose, or goal	249	SemEval'12	(*cook*, *eat*)
LexSem$_{\text{Space}}$	location or time association	235	SemEval'12	(*aquarium*, *fish*)
LexSem$_{\text{Ref}}$	expression or representation	187	SemEval'12	(*song*, *emotion*)
LexSem$_{\text{Event}}$	object's action	3583	BLESS	(*zip*, *coat*)
Noun$_{\text{SP}}$	plural form of a noun	100	MSR	(*year*, *years*)
Verb$_{3}$	first to third person verb present-tense form	99	MSR	(*accept*, *accepts*)
Verb$_{\text{Past}}$	present-tense to past-tense verb form	100	MSR	(*know*, *knew*)
Verb$_{\text{3Past}}$	third person present-tense to past-tense verb form	100	MSR	(*creates*, *created*)
LVC	light verb construction	58	Tan et al. (2006b)	(*give*, *approval*)
VerbNoun	nominalisation of a verb	3303	WordNet	(*approve*, *approval*)
Prefix	prefixing with *re* morpheme	118	Wiktionary	(*vote*, *revote*)
Noun$_{\text{Coll}}$	collective noun	257	Web source	(*army*, *ants*)

Figure C.1: Details of lexical relations dataset

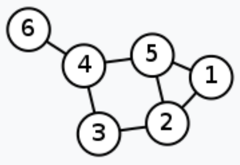
Labeled graph	Degree matrix	Adjacency matrix	Laplacian matrix
	$\begin{pmatrix} 2&0&0&0&0&0\\ 0&3&0&0&0&0\\ 0&0&2&0&0&0\\ 0&0&0&3&0&0\\ 0&0&0&0&3&0\\ 0&0&0&0&0&1 \end{pmatrix}$	$\begin{pmatrix} 0&1&0&0&1&0\\ 1&0&1&0&1&0\\ 0&1&0&1&0&0\\ 0&0&1&0&1&1\\ 1&1&0&1&0&0\\ 0&0&0&1&0&0 \end{pmatrix}$	$\begin{pmatrix} 2&-1&0&0&-1&0\\ -1&3&-1&0&-1&0\\ 0&-1&2&-1&0&0\\ 0&0&-1&3&-1&-1\\ -1&-1&0&-1&3&0\\ 0&0&0&-1&0&1 \end{pmatrix}$

Figure C.2: Example of an unnormalized Laplacian matrix

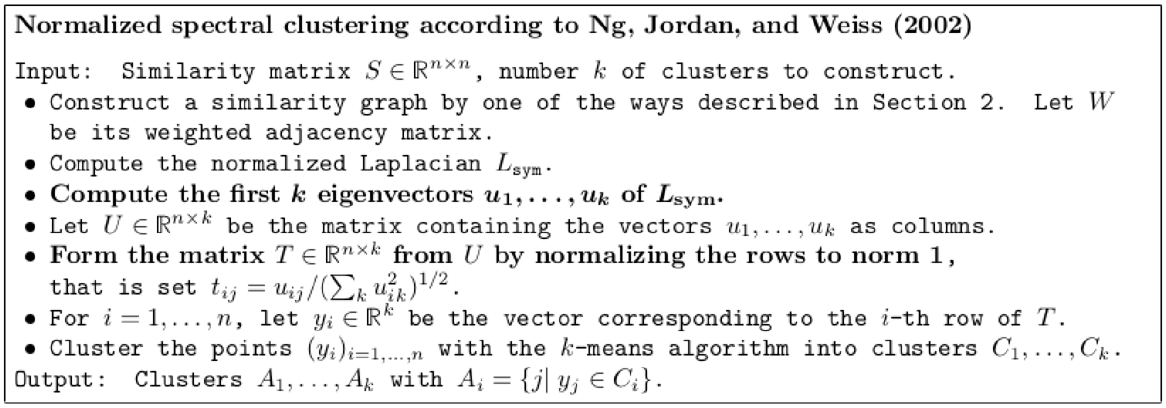
each vertex of the graph is a data point (in this case a *diffvec*) and an edge occurs if a pair of vertices are similar (the definition of similarity depends on the similarity function and its threshold). With the graph defined, then the similarity matrix can be obtained by calculating the Laplacian matrix, which there are several types of them. Image C.2 shows an example of an *unnormalized* Laplacian matrix[2], while image C.3 shows the *normalized* version of spectral clustering.

C.3 V-measure

V-measure is a score to evaluate the quality of a clustering model. In the *diffvec* paper, the authors employed v-measure as proposed by Rosenberg [21], where V-measure is defined as the harmonic mean of two metrics:

$$v = 2 * \frac{homogeneity * completeness}{homogeneity + completeness} \tag{C.1}$$

[2]taken from https://en.wikipedia.org/wiki/Laplacian_matrix



Normalized spectral clustering according to Ng, Jordan, and Weiss (2002)

Input: Similarity matrix $S \in \mathbb{R}^{n \times n}$, number k of clusters to construct.

- Construct a similarity graph by one of the ways described in Section 2. Let W be its weighted adjacency matrix.
- Compute the normalized Laplacian L_{sym}.
- **Compute the first k eigenvectors $u_1, \ldots, u_k$ of L_{sym}.**
- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \ldots, u_k$ as columns.
- **Form the matrix $T \in \mathbb{R}^{n \times k}$ from U by normalizing the rows to norm 1,** that is set $t_{ij} = u_{ij} / (\sum_k u_{ik}^2)^{1/2}$.
- For $i = 1, \ldots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of T.
- Cluster the points $(y_i)_{i=1,\ldots,n}$ with the k-means algorithm into clusters $C_1, \ldots, C_k$.

Output: Clusters $A_1, \ldots, A_k$ with $A_i = \{j \mid y_j \in C_i\}$.

Figure C.3: Normalized Spectral clustering algorithm, taken from source paper [14]

Homogeneity is maximized when each cluster contains elements of as few different classes as possible and it's calculated by:

$$h = 1 - \frac{H(C|K)}{H(C)} \; if \; H(C|K) \neq 0; \; 1 \; otherwise. \tag{C.2}$$

where:

$$H(C|K) = -\sum_{k=1}^{|K|} \sum_{c=1}^{|C|} \frac{a_{ck}}{N} \log \frac{a_{ck}}{\sum_{c=1}^{|C|} a_{ck}} \tag{C.3}$$

$$H(C) = -\sum_{c=1}^{|C|} \frac{\sum_{k=1}^{|K|} a_{ck}}{n} \log \frac{\sum_{k=1}^{|K|} a_{ck}}{n} \tag{C.4}$$

Completeness aims to put all elements of each class in a single cluster and it's calculation is given by:

$$c = 1 - \frac{H(K|C)}{H(K)} \; if \; H(K|C) \neq 0; \; 1 \; otherwise. \tag{C.5}$$

where:

$$H(K|C) = -\sum_{c=1}^{|C|} \sum_{k=1}^{|K|} \frac{a_{ck}}{N} \log \frac{a_{ck}}{\sum_{k=1}^{|K|} a_{ck}} \tag{C.6}$$

$$H(K) = -\sum_{k=1}^{|K|} \frac{\sum_{c=1}^{|C|} a_{ck}}{n} \log \frac{\sum_{c=1}^{|C|} a_{ck}}{n} \tag{C.7}$$

For both cases, C is the set of classes $\{c_i | i = 1, ..., n\}$, K is the set of clusters $\{k_i | 1, ..., m\}$, and

a_{ij} is the number of samples with class c_i in cluster k_j.

Curriculum Vitae

Name:	Felipe Urra
Post-Secondary Education and Degrees:	Universidad Técnica Federico Santa María, Valparaíso, Chile 2005 - 2013 Telematics Engineer B.A.
Related Work Experience:	Engineer Foris, Santiago, Chile 2013-2014
	Developer Lemontech, Santiago, Chile 2014-2016
	Teaching Assistant The University of Western Ontario, London, Canada 2017 - 2018