Term Paper for Natural Language Processing

Sharon Laurance Muthipeedika
Universität Hildesheim
Hildesheim, Lower Saxony, Germany
laurance@uni-hildesheim.de

Abstract

Words can mean different things depending upon the context of the sentence. To identify the correct meaning of the word depending upon the context, we use Word Sense Disambiguation (WSD). The paper describes three methods which perform this. Most Common Sense Algorithm, Plain Lesk [3], Implementation of Oele and van Noord [5]. These methods are evaluated on SemCor, SenseEval2 and SenseEval3 annotated datasets. The final results for English data are provided in the results section. Six extensions have been performed and the impact on accuracy is studied.

1 Introduction

Word Sense Disambiguation solves an important problem in NLP. It helps us identify the correct meaning of the word by removing ambiguity. The aim of this paper is to implement Distributional Lesk [5] and understand how modifications can help improve the accuracy. For e.g we see two sentences

1. The bank is in grave danger.

2. He was sitting by the river bank.

In the two sentences bank can mean different things but we understand this by reading as we look at the context. Distributional Lesk is a knowledge-based system. It does not require a large labelled corpora and also can be efficiently translated to other languages. The authors of the paper [5] talk about the effects of sorting the words according to the number of synsets. If a word has been disambiguated, we can utilize the sense embedding in place of the word embedding. This helps improve the accuracy. Further experiments are carried out by removing stop words, using our own word embeddings and using SBERT to see whether these factors help improve accuracy.

2 Baselines

The concept of Word Sense Disambiguation using Lesk Distance was introduced in 1986 [3]. It counts the number of overlaps between the sentence and given senses of a word in the dictionary to predict the correct sense of a word in that context.

The most common sense algorithm says the first synset of the word corresponds to the most frequent sense. The algorithm takes the sentence as the input and iterates over each word (w_i,..,w_n). For each word w_i, the first sense is taken as the correct synset and stored in "predicted_synset". The synsets are taken from Wordnet 1.7.1 in the nltk.corpus. Evaluation is done for Semcor, SenseEval2 and SenseEval3. The ground-truth values of the sense are compared with the values predicted by the algorithm. Evaluation function checks if the lemma is the synset and compares the POS-tag and the wnsn number. Even if this is a very simple algorithm, the accuracy of the algorithm is high.

The Plain Lesk algorithm [3] by Michael Lesk performs Automatic Sense Disambiguation. The algorithm takes the particular word for which the correct sense needs to be determined and the sentence in which the word occurs as the input. For each word, w_i it iterates over all the synsets of the word and finds the overlap between the sentence and the dictionary meaning (i.e gloss). For constructing the gloss, the definition and the examples of a synset are considered. The number of words that overlap are counted and returned by the function. This overlap is compared with all hyponyms and the synset with the highest overlap is considered the meaning of the word in the context. The best sense is taken as the "predicted_synset". The ground-truth values of the word sense are compared with the values predicted by the algorithm. Evaluation is done for Semcor, SenseEval2 and SenseEval3. The accuracy of the Plain Lesk (PL) algorithms for the three datasets is discussed in the

Results section.

3 Methodology

In this paper, the Distributional Lesk algorithm [5] is implemented along with some extensions which help improve the accuracy. It is a knowledge-based disambiguation system which uses pre-trained word embeddings from Word2Vec and also lexeme embeddings from AutoExtend [6]. The entire procedure for implementation is discussed below.

First, we load the pre-trained word embeddings from the Word2Vec model trained on the "GoogleNews-vectors-negative300" dataset. This will be used to find word embeddings. The model contains 300-dimensional vectors for 3 million words and phrases [4].

Implementation of Oele and van Noord [5] algorithm is done in the function "calculate_meaning()". The function takes a sentence as an input and outputs the preferred sense of each polysemous word in the sentence depending on the context. Prior to disambiguation, we sort the words based on the number of synsets they have [2]. The words with the fewest synsets gets disambiguated first. The algorithm iterates over the sorted words (w_i,...,w_n). For each word w_i, it iterates over all the synsets of the word and calculates a score. The synset which has the maximum score is taken as the correct sense for the word in that sentence. This is known as the "predicted_synset". To calculate the score, we need two components as seen in Equation 1. For this, we need to calculate the "gloss-vector" G_s, "context-vector" C_w and "lexeme-vector" $L_{s,w}$.

First component is the cosine similarity between the "gloss-vector" G_s and the "context-vector" C_w. To calculate the "gloss-vector" the function "find_gloss_embedding()" is used. The function takes the sense of the word as an input and returns an array containing word embeddings of all the words in the gloss. For calculating the gloss, we take the definition and all the examples of the sense. Both are included to increase accuracy [1]. All the words are tokenized using "word_tokenize" from nltk library. After this, the function words are removed as they do not provide any useful information. The array containing embeddings of all words in the gloss is averaged to find the final gloss embedding. This is a 300 dimensional vector.

To find the "context-vector" C_w, the entire sentence is taken by removing the word whose sense is to be found. Now, embeddings for all words in the context are computed and averaged. As the words in the sentence are sorted, some ambiguous words have already been disambiguated. When we encounter these words in the context, we can use the sense embedding instead of the word embedding. This helps us find the context embedding which is a 300-dimensional vector. Now we calculate the first component of the score by taking the cosine similarity between the "gloss-vector" G_s and the "context-vector" C_w using the "scipy" library.

The second component of the score requires "context-vector" C_w and "lexeme-vector" $L_{s,w}$. For computing the "lexeme-vector" we use the lexemes.txt file from AutoExtend embeddings [6]. We read the file lexemes.txt and store it in a dictionary called "dict1". The dictionary has the key as lemma with pos tag (e.g. entity-wn-2.1-00001742-n) and the value as 300-dimensional vector. In order to find the lexeme for the word we construct the same format for the word. We have the lemma and POS-tag from the dataset. We calculate the offset for the particular sense using sense.offset() and construct the key to be searched [lexeme_key= lemma+"-wn-2.1-"+str(offset).zfill(8)+"-"+pos]. This entire thing becomes the search key. If it exists in the lexeme file, it returns a 300-dimensional lexeme embedding. Now we have the "lexeme-vector" from the above step. So, we find the second component of the score using the cosine similarity between the "lexeme-vector" and "context-vector". We add both components to find the final score. Scores for all synsets of a word are appended in an array "final-score-arr" and the maximum value of the array gives the best predicted sense of the word.

$$Score\,(s,w) = \cos\,(G_s, C_w) + \cos\,(L_{s,w}, C_w) \tag{1}$$

To evaluate the accuracy, predicted sense is compared with the true sense of the word. If they are the same, 1 is appended to the "final-arr" array. Finally, the number of correct predictions are counted. This is divided by the length of the array to get the accuracy percentage for the particular dataset. Thus, Distributional Lesk helps to achieve Word Sense Disambiguation as seen in algorithm 1.

4 Datasets

Three datasets are used for evaluation. SemCor, SenseEval2 and SenseEval3.

- **SemCor1.7.1**: It has three folders which contain brown1,brown2 and brownv. brown1

Algorithm 1 : Distributional Lesk

Require: The pre-trained word embeddings from Word2Vec and lexeme embeddings from AutoExtend

Ensure: The predicted sense for each polysemous word.

1: Sort the words in the sentence according to the number of synsets.
2: **for all** $w_i \in sorted\_list$ **do**
3: **for all** $sense \in synsets(word)$ **do**
4: Calculate Gloss embedding G_s
5: Calculate Context vector C_w
6: Calculate Lexeme vector $L_{s,w}$
7: Calculate the score of each sense. Score(s,w)=cos$(G_s, C_w) + (L_{s,w}, C_w)$
8: Append score in an final_array
9: **end for**
10: The highest score from the final_arr gives the predicted sense of the word
11: Compare with True Sense to calculate accuracy
12: **end for**

and brown2 have all content words tagged. brownv has only verbs tagged. It has 352 manually annotated documents. It has 34,374 sentences. Out of this, 5000 sentences are selected at random.

- **SenseEval2**: It has three manually annotated documents selected from folder wordnet1.7.1. It has 238 sentences.
- **SenseEval3**: It has three manually annotated documents selected from folder wordnet1.7.1. It has 300 sentences.

All sentences are used from SenseEval2 and SenseEval3. The files are in html format and have tags associated with them. They contain sentences and each word has additional information regarding POS-tags, lemma, wnsn and lexsn. They also say which words and punctuation marks should be ignored. This is indicated by "cmd=ignore". All sentences are marked with s tag and have snum associated with them.

For parsing html files and extracting information BeautifulSoup library is used. Pos-Tag mapping is carried out to map different Pos-Tags to noun, verb, adjective and adverb (denoted by n, v, a, r respectively). All data is stored in a nested dictionary. The key of the dictionary is the sentence number (e.g. 0,1,2) and it has two values. The first value is the sentence and second value is every word of the sentence as key and the value as "lemma-POS-Tag-wnsn". This structure is useful for evaluation.

5 Experiments

The experiments are carried out only with English Data and Dutch Dataset is not considered for evaluation. Wordnet version used is 1.7.1. As the embeddings file also have the same version, mapping can be performed. This is done for version compatibility.

5.1 Extension 1: Removal of Stop words

Stopwords are high-frequency words that do not contribute much to the context of a sentence. We can remove these frequently occurring words (e.g. I, me, he, him) and be left with words that contribute to the meaning of the sentence. Stopwords for the English language are loaded from the nltk library. They are removed from the sentence which is processed. Also, they are removed from the synset definition and examples. So, the gloss doesn't have any stopwords before computing the gloss vector. Similarly, stopwords are excluded from the context vector.

5.2 Extension 2: Evaluation of different categories of Brown Corpus

Three categories are chosen for evaluation. As the Brown Corpus has tagged data (data with POS tags) and additional information of synset is required while evaluation, Brown Corpus categories are mapped with Semcor Data. This mapping of Brown Corpus categories with Semcor Data is done from the official nltk site https://www.nltk.org/ . In total brown corpus has 15 categories. Three cateories with most number of documents are selected. Files are read with the help of BeautifulSoup and html.parser.

Three categories chosen are:

- **br-h** – Genre Government – Here dict-for-government is a nested dictionary that stores sentences from all files. It has a total of 1489 sentences. In addition, the sub-dictionary stores word as the key and the "lemma-postag-wnsn" as the value. This is useful while evaluating the category.

Algorithm	Dataset		
	SemCor	**SenseEval2**	**SenseEval3**
Distributional Lesk	42.01%	39.94%	39.41%
W/o Stopwords	**42.28**%	39.64%	**39.57**%
Embedding	41.15%	40.62%	36.26%
Parameter 1	41.22%	41.67%	37.14%
Parameter 2	40.68%	42.05%	36.21%
FastText	40.77%	**43.48**%	37.09%
S-BERT	41.53%	38.93%	38.22%

Table 1: Comparison of Distributional Lesk and all other Extension except Brown Corpus for three Datasets. W/o Stopwords indicate stopwords are removed from gloss, context and sentences. Embedding indicates training with our dataset. Parameter 1 and Parameter 2 indicate some parameters are modified. FastText indicates using FastText embeddings and for the final extension SBERT is used for sentence embeddings.

- **br-e**– Genre Hobbies - Here dict-for-hobbies is a nested dictionary that stores sentences from all files.It has a total of 2962 sentences.
- **br-n**– Genre Adventure - Here dict-for-adventure is a nested dictionary that stores sentences from all files.It has a total of 2591 sentences.

5.3 Extension 3: Training my own Word Embedding

For this extension, I have trained my own word embedding using the Word2Vec from genism.models. The parameters used while training are:

- **vector_size**: It is the dimensionality of the word vector. Here, it is chosen as 300.
- **min_count**: It ignores all words with frequency less than min_count. Here it is chosen as 1.
- **window**: It is the maximum distance between the current and predicted word within a sentence. Here, it is chosen as 3.
- **workers**: Here workers are taken as the cpu_count. Number of threads to be taken. Here it is considered as 8.
- **sg**: It is the training algorithm. Here, sg=1 which means skip-gram is used. Otherwise, it is Continuous Bag-of-Words (CBOW).
- **sentences**: It is a list of tokens or for larger corpora, it is iterable that streams directly from the disk/network.

After initializing the model, build_vocab() and train() methods are called on the model. Once we train the model, keyed vectors are accessed through model.wv attributes. This newly trained model is used to find word embedding. The trained vectors are stored in KeyedVectors. The reason for separating the vectors is efficient loading and processing vectors in RAM between processes. The whole model need not be stored and instead only the keys and vectors can be stored. After evaluation of Semcor, we can see that the accuracy decreases to 41.15%. Similarly, for SenseEval2 there is an decrease to 40.62%. For SenseEval3, the accuracy decreases to 36.26%.

5.4 Extension 4: Different Parameter Changes

To understand how parameters affect the accuracy better, we experiment with different parameter settings. Here we keep the vector_size as 300. It is kept the same as other vectors are of 300-dimension, and we need the same size for computation. The number of workers are also kept the same. I have experimented by changing min_count. Higher min_count leads to ignoring words that have lower frequency. This affects the model in a negative way. Different window sizes have been tried. Increasing the window_size leads to adding distance between current and predicted word. After a point, increasing the window size does not affect the algorithm much and the accuracy drops slightly. We can observe window size of upto 30 leads to an accuracy of 40.68%. sg is taken as 1 which means skip gram is used. Big changes have been made to parameters to see observable changes. But change in accuracy is not much. It can be said that accuracy also depends on dataset. For SenseEval2, accuracy increases from 41.67% to 42.05%. Thus, we can say increasing the window size is beneficial.

5.5 Extension 5: FastText Embedding

Now, we use FastText and evaluate the word representations. It is loaded from genism.models. It enables to compute embeddings for unseen words (i.e., the words that are not present in the vocabulary) as a sum of n-grams included in the word. For training this, we have included all the sentences from three datasets. vector_size is kept the same. Here, the window_size is changed to 5. And the down_sampling rate is 1e-2. "sample" controls the down_sampling of the more frequent words. For these parameters, we can see a significant change in SenseEval2. For other datasets, change in accuracy is not much.

5.6 Extension 6: SBERT Pre-trained Model

SBERT is a Sentence Transformer that maps a variable sentence input to a fixed-size dense vector. Here a pre-trained model "all-MiniLM-L6-v2" is used.

The transformer network like BERT takes a sentence as input and produces contextualized word embedding for each token. We consider a Pre-trained SBERT model. There are several options to choose from (e.g., all-mpnet-base-v2, multi-qa-distilbert-cos-v1etc.). Parameters while training SBERT are maximal sequence length which denotes that sequences longer than this will be truncated. It is chosen as 256. The output of SBERT is a 384 dimensional vector. Different pre-trained model can be chosen based on applications. Applications include semantic search or an all-round model tuned for many use-cases.

In SBERT algorithm, gloss and context vectors are computed using model_sbert.encode() function. For computing the gloss vector, definition and examples sentences are considered. Entire sentences are passed to the encode function and the output is a 384 dimensional vector. The gloss embedding is computed by averaging the embedding of all sentences in the gloss. Similarly, the entire context of the word is passed to the model_sbert to compute the context embedding. Here cosine similarity is taken to identify the similarity between Gloss and Context embeddings. The second component of the score is computed same as before. In this way, SBERT is used for Distributional Lesk.

6 Results

In this section, we discuss the results of the evaluation for 3 datasets. In addition, we look at six extensions and how they contribute to improving accuracy.

	SemCor	SenseEval2	SenseEval3
PlainLesk	38.58%	43.26%	37.77%
D-Lesk	42.01%	39.94%	39.41%
MCS	50.01%	49.38%	47.29%

Table 2: Comparison of Baselines for three Datasets. We can see Most Common Sense (MCS) has the highest accuracy across all datasets.

The Plain Lesk algorithm gives the best accuracy on SenseEval2 dataset. Most Common Sense algorithm gives the best accuracy on SemCor dataset. We can understand from this, accuracy is dataset specific and the larger the data better the evaluation results. We can observe from Table 2, Distributional Lesk (D-Lesk) performs better than Plain Lesk for SemCor and SenseEval3 [5]. There is a **4%** increase in SemCor and a **2%** increase for SenseEval3. Taking more number of sentences become computationally expensive and keeping this factor in mind evaluation is carried on subset of 5000 sentences in SemCor.

We now see the accuracy of Distributional Lesk and how it performs along with various extensions from Table 1.

- The accuracy for Distributional Lesk is the highest for SemCor Dataset at 42.01%. Here, we observe that it exceeds the Plain Lesk algorithm for SemCor and SenseEval3. We can infer the use of embeddings for computing sense of a word is helpful.

- After removing stopwords from the sentence, gloss and context vectors we see slight improvement in acccuracy for SemCor and SenseEval3. We can understand that removing stopwords also help speed up the algorithm. Out of all the extensions, removing stopwords performs best for SemCor and SenseEval3 dataset.

- By training own embeddings, there is a slight decrease in accuracy for SemCor and SenseEval3. This could be due to the fact that training parameters are not optimal. Still, we see an increase of 1% accuracy for SenseEval2.

- Many parameter changes have been performed, but only two are included in the table. I have even changed sg = 0, but changes

are not observable. There is an increase for SenseEval2, but for other datasets the accuracy does not increase.

- We can see for FastText Embedding the accuracy of all three datasets surpasses all the baselines. Highest accuracy is obtained for SenseEval2 which is **43.48%**.
- For SBERT we can see SemCor has the highest accuracy. This could also be due to the high number of sentences.

	Government	**Hobbies**	**Adventure**
Accuracy	**42.46%**	40.43%	41.53%

Table 3: Categories of Brown Corpus.

We also see how different categories of Brown Corpus are evaluated against each other from Table 3. As they have nearly 2000 sentences in each category, we can see that it performs best for Government category with an accuracy of nearly 42.46%.

7 Discussion

Distributional Lesk appears to have the ability to perform well in the task of Word Sense Disambiguation. Using FastText embeddings instead of Word2Vec helps improve accuracy for SenseEval2. It gives the best performance on all three datasets. Evaluating on the whole corpus SemCor would also lead to improved accuracy.

In this approach, we average the word embeddings for gloss and context embedding. This does not tell us anything about the importance of certain words. As further improvement, a weighted approach would be more beneficial as there would be a multiplication factor for each word before averaging the embeddings. This factor tells us which words are not that important and can be ignored. This can also be extended to other languages given we have an sense inventory.

8 Conclusion

The paper discusses the implementation of Distributional Lesk in detail. The baseline models of Plain Lesk and Most Common Sense also have been implemented. The accuracy of Most Common Sense is highest. Evaluation for three datasets in English language has been carried out. The advantage of this approach is its application to other languages. Gloss embeddings have been computed by taking both definition and examples of the senses. This helps in improving accuracy. The context vector is computed by averaging the word embedding of all words except whose sense is to be computed. Lexeme vector is calculated with the help of AutoExtend embeddings. To further improve accuracy many extensions have been tried. After seeing the results, we can conclude that the extensions help in improving the accuracy of the model. As Distributional Lesk does not require manually annotated data and only requires sentences along with their senses, it can be extended to other domains.

Acknowledgements

The equations and algorithm used in the paper are taken from the authors Oele and Van Noord [5].

References

[1] Satanjeev Banerjee and Ted Pedersen. 2002. An adapted lesk algorithm for word sense disambiguation using wordnet. In *International conference on intelligent text processing and computational linguistics*, pages 136–145. Springer.

[2] Xinxiong Chen, Zhiyuan Liu, and Maosong Sun. 2014. A unified model for word sense representation and disambiguation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1025–1035.

[3] Michael Lesk. 1986. Automatic sense disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone. In *Proceedings of the 5th annual international conference on Systems documentation*, pages 24–26.

[4] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

[5] Dieke Oele and Gertjan Van Noord. 2017. Distributional lesk: Effective knowledge-based word sense disambiguation. In *IWCS 2017—12th International Conference on Computational Semantics—Short papers*.

[6] Sascha Rothe and Hinrich Schütze. 2015. Autoextend: Extending word embeddings to embeddings for synsets and lexemes. *arXiv preprint arXiv:1507.01127*.