Natural Language Processing : Term Paper

Sruthy Annie Santhosh
Matriculation ID: 312213
MSc Data Analytics
University of Hildesheim
santhosh@uni-hildesheim.de

Abstract

This paper contains a detailed explanation of the implementation of various algorithms for word sense disambiguation. The baseline algorithms discussed are the Most Common Sense and the Plain Lesk algorithms. The main focus of the paper is on the implementation of the Distributed Lesk algorithm and its evaluation on three datasets: Senseval-2, Senseval-3 and SemCor. Further extensions on this algorithm are also experimented and the results are discussed and analysed thoroughly.

1 Introduction

Word Sense Disambiguation (WSD) is a highly researched topic in Natural Language Processing (NLP). WSD refers to finding the correct meaning of a word depending on the context. Each word may be associated with multiple meanings or senses. There are mainly two approaches to tackle this problem : Supervised Systems and Knowledge based systems. Supervised systems use supervised learning methods while knowledge based systems do not require manually tagged data and can be used across domains(Escudero et al., 2000). In this paper we mainly deal with the algorithms based on Knowledge based systems.

Word embeddings represent meanings of words as contextual feature vectors. Word embeddings can be used in algebraic operations and can help find the similarity between words in a vector space, but they only assign one embedding for a word. Hence different meanings of the word are not considered. Thus arises the need to use sense embeddings also. Lesk Algorithm is one method to help identify the correct sense of a word in a context, by checking the number of overlapping words between the context of the word and the definition of each sense of the word(Lesk, 1986). An extension to this implementation is the Distributional Lesk, which exploits the definitions of the senses of the word (referred to as gloss), the context of the word and the lexeme embeddings of the word for the corresponding sense.

2 Contributions of Paper

In this paper, I have described the methods by which the Distributional Lesk mentioned in paper written by Oele and van Noord (2017) is implemented. The baseline algorithms and some of the extensions given in the question paper are also implemented and evaluated. All the algorithms and extensions are evaluated on three datasets - Senseval2, Senseval3, SemCor. The different algorithms implemented are as follows:

1. Two baseline methods - most common sense and plain lesk algorithms.
2. Implementation of the method proposed by Oele and van Noord (2017) using the pre-trained word embeddings
3. Evaluation on all three datasets

Extensions implemented:

1. Experiment with removing stop words and punctuation from the dictionary glosses, sense descriptions and contexts in the occurrences of the words before measuring the distance.
2. SemCor data come from the Brown corpus. The Brown corpus consists of texts from different text categories (see e.g. https://www1.essex.ac.uk/linguistics/external/clmt/w3c/corpus_ling/content/corpora/list/private/brown/brown.html). Evaluate the results for individual categories.
3. Train your own word embeddings for this task
4. Train embeddings with various parameter settings (you probably need to make big changes

so you actually get significantly different results for WSD) and study the influence of the used embeddings on the disambiguation task.

5. Extend the word embedding model to also use character-based representations,i.e,. fastText embeddings.

6. Use transformers and sentence embeddings to compare a sentence and a gloss using the SBERT pre-trained models.

This paper contains the description of the methods and evaluation of the results on datasets. The steps taken and reasoning behind them are also given in a detailed manner.

3 Datasets

In this term paper, three datasets were given for evaluating the algorithms and the extensions. The datasets were all downloaded, extracted and preprocessed to retrieve the sentences. Each of these datasets contain English sentences in XML format. Each word in the sentence is also annotated with postags, lemma, the true sense of the word (stored as a number in attribute wnsn) and other information. All the information regarding a word is stored in a <wf> tag, and the text is stored inside the <wf> tags. Each sentence starts and ends with a <s> tag. The datasets also contain unwanted words and tags which need to be ignored. We only consider those words which have attribute cmd as 'done', have lemma attribute and do not have 'ot' attribute.

First the datasets were all downloaded and extracted from their corresponding urls:

1. SemCor : `http://web.eecs.umich.edu/~mihalcea/downloads/semcor/semcor1.7.tar.gz`

2. Senseval-2 : `http://web.eecs.umich.edu/~mihalcea/downloads/senseval.semcor/senseval2.semcor.tar.gz`

3. Senseval-3 :`http://web.eecs.umich.edu/~mihalcea/downloads/senseval.semcor/senseval3.semcor.tar.gz`

Each dataset is a directory with files in many subfolders. We retrieve the files corresponding to wordnet1.7.1 as we use the word embeddings and lexeme embeddings of this version. The file paths of the required files are stored in an array for each dataset.

Here I have used BeautifulSoup to read and access the words in the datasets as it is in XML format. For each file in each dataset, the xml content is read. I am storing the sentences of a dataset in a dictionary. Along with the sentence, I am also storing a sub dictionary which contains each of the words in the sentence as the key and a phrase as its value. This phrase contains the lemma, pos-tag and wnsn values of the word. We need the lemma and pos-tag to retrieve the lexeme embeddings for the word from AutoExtend (Rothe and Schütze, 2015). and we require the wnsn value to get the ground truth value of the synset of the word in the sentence. We calculate the accuracy based on this wnsn value and the predicted synset value of the algorithm. The key of the dataset_dictionary will be an index value and its value will be an array whose first element is the sentence and second element is the sub dictionary.

Hence if the sentence is " I have a dog". The dictionary will be of the form: { 0: [" I have dog", { "I" : lemma,postag,wnsn , "have" : lemma,postag,wnsn , "dog": lemma,postag,wnsn }] }

I have made a comma separated phrase. The lemma, postag and wnsn values can be easily retrieved by splitting on ','. As mentioned before, each sentence is taken after filtering the tags. In BeautifulSoup, tags can be easily filtered using findAll().

First we select all <s> tags, and then the <wf> tags having 'cmd':'done'. Then I checked whether these selected tags have a lemma attribute with string value and that they don't have the attribute 'ot'. After this filtering , we obtain the tags which contain the required words. The words are joined together to form sentences and the attribute values of lemme, postag and wnsn values are taken to form phrases.

In the given lexeme embeddings in AutoExtend (Rothe and Schütze, 2015)., the postags are different from the postags in wordnet. Hence I created a dictionary for mapping. All the postags in the dataset are mapped to their corresponding postag, before forming the phrase. Eg: NN in the dataset is mapped to n as all noun words have an n tag in the embeddings file.

These steps are followed to create dictionaries for all the datasets. I used the dictionary data structure for storage as it is easy to retrieve values based on the keys.

It was observed that the Semcor dataset has 34374 sentences, Senseval-2 has 238 sentences and Senseval-3 has 300 sentences. As mentioned in the question paper, I have randomly selected 5000 sentences from Semcor for the evaluation process.

4 AutoExtend

Distributional lesk requires an embedding called Lexeme embedding $L_{s,w}$ for each sense s of the word w. Autoextend is used to get this additional sense embedding from WordNet based on the word embeddings. We use the version 1.7.1. As given in the question, I downloaded the pre-trained embeddings from Autoextend. The downloaded directory contains three files : lexeme.txt, synset.txt and mapping.txt . After observing and evaluating on simple examples, I found out that the file lexeme.txt contains the Lexeme embeddings for each sense of each word. These embeddings are of 300 vector size. Each embedding corresponds to a tag which forms a single line. We read the lexeme file and store each tag as the key and the embedding as a value in a dictionary. This dictionary can be used to retrieve Lexeme embeddings (Rothe and Schütze, 2015).

The tag in lexeme file is of the form: lemma+"-wn-2.1-"+str(sense.offset())+"-"+pos-tag

Here we get the sense embedding as we use the offset value of the corresponding synset and the lemma and the pos-tag of the corresponding word. The lemma and pos-tag of the word is derived from the phrase we store for each word in the sub dictionary of the dataset. Thus each embedding corresponds to not just the word but also the synset.

5 Word Embeddings

As mentioned in the method proposed by Oele and van Noord (2017), I have used word embeddings from the word2vec model trained on the Google News corpora. It consists of 300-dimensional vectors for 3 million words and phrases. These word embeddings are also the same size as the lexeme embeddings, hence algebraic operations like cosine similarity can be done. These word embeddings are used to create the Gloss Embeddings and Context Embeddings for Distributional Lesk implementation which will be discussed in section 7.2 (Oele and van Noord (2017)).

6 Accuracy Calculation

For accuracy calculation, the true synset is needed for comparison. The true synset of the word is formed as : lemma + '.'+pos+'.'+'0'+wnsn and is retrieved from the sub dictionary of the word.

In some cases, if more than one true synset exists for a word, wnsn will be of the form index_1;index_2. In such scenarios, a list of true synsets is made with all the wnsn values given. Then we check if the pos tags of predicted synsets and true synsets are equal and if the index values of both synsets are equal (wnsn values). Only if both are equal, a correct prediction is made. The lemma value is checked to be a synonym of the true lemma value, equality does not need to be checked.

Accuracy = (Number of correct predictions / Total number of predictions) * 100

Accuracy is calculated for the whole dataset (all the words in all the sentences of the dataset).

7 Methodology

7.1 Baseline Implementation

7.1.1 Most Common Sense algorithm

Most Common Sense algorithm or the Most Frequent Sense Algorithm is usually used as a baseline for any WSD experiment. This is a very high performing algorithm as it has a highly skewed distribution of word senses (Agirre and Edmonds, 2007). Hence this is taken as an upper bound due to its semi supervised nature and is very difficult to beat.

In this algorithm, the predicted sunset is the one which is most frequent for the word. This is the first synset of the word as they are ordered according to their frequency. Hence the synset in the index 0 is returned as the predicted synset of the word.

- Input - sentence and the sub dictionary of the sentence
- Output - accuracy of the algorithm

Steps :

1. Tokenize the sentence and remove the words which do not have any word embeddings or are not polysemous
2. For each word in the tokenized list, prediction is made and then iterated to the next word. Once prediction is made for all words accuracy is returned. This is done for all sentences in the dataset.

3. Prediction is made as follows wordnet.synsets(lemma)[0].Accuracy is calculated as mentioned in section 6.

This algorithm was evaluated on all sentences of Senseval-2 and Senseval-3 datasets and on the 5000 random sentences selected from the SemCor dataset. I have passed the dictionary created for each dataset which contains the sentences and a sub dictionary with the words and their phrases. SemCor dataset took the longest to run(8 mins) as it has the most number of sentences. Results are discussed in Section 8.

7.1.2 Plain Lesk Algorithm

Plain lesk algorithm is based on the idea that the overlap between the definition of senses of the word and the context of the word gives information regarding its meaning. We find the synset for which the maximum number of words overlap. This becomes the predicted synset. Here we do not use word embeddings to find similarity (Lesk, 1986).

- Input - sentence and the sub dictionary
- Output - accuracy

First the sentence is tokenized and for each token in the sentence which is not a function word (word which does not have word embedding or synset) I found the predicted synset. Finding the predicted synset is as follows:

1. For each word, we find the number of overlapping words for all its senses.
2. First I found the Gloss of the sense. Gloss contains the words which occur in the definition of that synset and in the examples of that synset. Function words are removed from the gloss.
3. Then I found the words which occur in both the gloss and in the sentence of the word. The number of words which occur in both the sets constitutes the overlapping words. I also found the number of words which overlap between each of the hyponyms of the corresponding synset and the sentence and that is also added to the overlap number.
4. Then I found out the synset of the word having the highest overlapping words. That synset becomes the predicted synset.

This algorithm of plain Lesk is taken from wikipedia and (Lesk, 1986). Then accuracy is found as explained in the section 6. I did the evaluation for all sentences of all the three datasets (Only 5000 sentences for Semcor). As in the case of the Most Common Sense algorithm, evaluation on Semcor data took the longest time.

7.2 Distributional Lesk Implementation

Now let us discuss the implementation of the algorithm proposed by Oele and van Noord (2017) using the pre-trained word embeddings. This method is similar to that of Plain Lesk but instead of computing the number of overlapping words, we compute the similarity between the gloss of a sense and context of the word using word embeddings to get the best synset of the word in the sentence.

- Input: Sentence and the sub dictionary
- Output : Accuracy

For a word w and sense s, we find the cosine similarity between the lexeme(combination of sense and word) with the context of w and the cosine similarity between the gloss of sense and the context of w.

If Gloss vector - G_s , Context vector - C_w and Lexeme vector - $L_{s,w}$, then

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

The sense with the highest score is chosen as the preferred sense of the word. Then accuracy is calculated by comparison with the true sense of the word as discussed in section 6. If any of the vectors is 0 (i.e, either gloss is empty or word embedding is not present for the word) the score in that side becomes 0.

Procedure:

Firstly I tokenized the sentence to words and sorted them in the increasing order of the number of synsets present. Only words having more than 1 synset is considered. Now the words having the least number of synsets will be disambiguated first. Such words are easier to disambiguate as written in the paper(Chen et al., 2014). Once a word is disambiguated, then when we consider that word as part of the context for another word, we take the predicted sense embedding. Hence we use this disambiguation for further iterations.

Now we take each sense of the word and find its score. The scores of all the senses of the word are appended to an array and then the maximum score is found out. The synset corresponding to the maximum score will be the predicted synset. For calculating the score we need the Gloss Vector, Context Vector and Lexeme Vector.

1. **Gloss vector** - Gloss vector is found for each sense of the word. It is a 300 size vector. Gloss consists of all the words in the definition of sense considered. I also added the words in the examples of the synset. Then I removed the function words from the gloss as it will not have word embeddings. Then for all the words in the gloss, its corresponding word embeddings are found from the word2vec model. Then the average of the embeddings is taken to get the Gloss vector for that particular sense. I have also added checks to make sure that no null exceptions arise.

2. **Context Vector** - Context vector is a 300 size vector found for each word. The context of a word contains all the words in the sentence except the word being considered. Firstly, I tokenized and removed function words from the sentence. Then I removed the word being considered to get the context words. Now for each word in the context, its corresponding word embedding is taken and stored in an array. If the word in the context is already disambiguated, then I have taken the gloss embedding of the predicted sense of that word. Otherwise I took the word embedding itself. For this scenario, after finding the predicted synset, they are stored in a dictionary, where the key is the word and value is its predicted synset. This dictionary is created for each sentence. After getting the context embeddings for all words in context, average is taken to get the context vector. Checks are added to make sure that the embeddings exist for the words.

3. **Lexeme Vector** - Lexeme vector is a 300 size vector found for each sense of each word. It is retrieved from AutoExtend. I have already saved all the tags and embeddings in the lexeme.txt file in a dictionary. To retrieve the lexeme embedding from the dictionary we need to form the corresponding tag. For this I get the lemma and postag of the word from the sub dictionary of the dataset. As previously mentioned, I had stored each word with a phrase containing its lemma and pos-tag. Then for the sense being considered, I find its offset and form the tag as :

   lemma+"-wn-2.1-"+str(sense.offset()).zfill(8)+"-"+pos.

   Then the corresponding lexeme embedding is retrieved.

If both the gloss vector and context vector are non-empty, then we find the cosine similarity between them. This can be referred to as sim1. Then I check whether both the context vector and lexeme vector are non empty and find their cosine similarity as sim2. To find the score, I add both these values. If any vector is of length 0, then the corresponding similarity becomes 0.

Now after finding the scores for all senses of the word, I found the maximum score and its corresponding synset. This becomes the predicted synset. True synset is also found as mentioned earlier and accuracy is calculated (Section 6).

This algorithm is done for all the sentences in datasets Senseval-2 and Senseval-3 and for 5000 sentences in Semcor. The Semcor dataset took the longest time to complete evaluation(10 mins).

7.3 Extensions Implementation

I have experimented with 6 of the given extensions of Distributional Lesk. For each extension I experimented with various parameters and the results show the optimal values received. For the Semcor dataset, 5000 sentences are randomly taken. The accuracies do not show much change when we change the selected sentences from SemCor.

7.3.1 Removing stop words and punctuation from the dictionary glosses, sense descriptions and contexts

In this extension, I added some changes to the Distributional Lesk function. First the stopwords english list was downloaded from nltk corpus. For this extension, after tokenizing the sentence, I checked and removed all stop words from the sentence before sorting them. Then in the gloss, I removed the stop words from the sense definition and sense examples. Only then the word embeddings are taken. Since stop words are removed from the sentence, they are not present in the context of the word also. In the scenario where the word is already disambiguated, for context embeddings, we find the gloss of the predicted sense. There I also removed the stopwords from the gloss and then the word embeddings were taken.

These were the only changes made. Rest all steps are the same as that of Distributional Lesk. Punctuations were already removed from glosses and context in Distributional Lesk itself. The evaluation was also done in the same manner.

7.3.2 Evaluating the Distributional Lesk algorithm on individual categories of Semcor Data

As mentioned in one of the queries in learnweb, I used the brown categories list from the nltk corpus. The categories were found and then fileids were found for each category. After many trial and error attempts, I found the top 4 categories with the most number of sentences. They were ‘news’, ‘hobbies’, ‘adventure’ and ’belles_lettres’.

Dictionaries are created for each of these categories. Earlier for Semcor, we had considered all the fileids in the dataset, while here we will take the fileids of only the required category. The sentences are read and stored in the same manner as before using BeautifulSoup. It was found that the ‘news’ category has 3819 sentences, ‘hobbies’ category has 2963 sentences, ‘adventure’ category has 2592 sentences and ’belles_lettres’ category has 2340 sentences.

Now the Distributional Lesk algorithm is implemented and evaluated on each of these individual data dictionaries. The function remains the same, only the dataset is different. Earlier, we had taken random 5000 sentences from Semcor, while here we selected sentences from each category.

7.3.3 Train your own word embeddings

Till now we used the word embeddings trained on Google news. In this extension, I trained word embeddings on the given datasets. Since training the model is an extensive task, I ran it on 8 cores. Word2Vec model is initialized from gensim with attributes min_count=1, window=3, vector_size=300, workers=8, sg=1. Min-count is kept as 1 because we need word embeddings for all the words in the dataset. We are not checking the number of occurrences of the word. In this experiment, I kept the window size as 3. The vector_size needs to be 300 as it is the size of the lexeme embeddings.

Then I built the vocabulary on all the three given datasets. I extracted the sentences from all three datasets and tokenized and stored them in an array. Then the vocabulary is built using the build_vocab function. After that the model was trained on this array for 10 epochs. Now we have a model which provides word embeddings based on our given datasets.

The Distributional Lesk function is extended to use this new model to retrieve word embeddings for gloss and context vectors. All other operations remain the same. The word embeddings can be accessed using model_new.wv[word]. The evaluation is done as we have done for distributional lesk.

7.3.4 Train embeddings with various parameter settings

For this extension, I experimented with different window sizes for the word2vec model which was created in section 7.3.3. Then the model was trained on our 3 datasets and evaluated in a similar manner. Here, min count and vector size cannot be changed as we need word embeddings of size 300 for all words present in the dataset. Hence I only tried on window size as other parameters did not show any impact on the model. Window size determines the length of the context of the word. For training we send tokenized sentences and the model only considers the context according to the window size specified. In section 7.3.3 I had used a window size of 3. Here I experimented with various window sizes ranging from 4 - 10. The results of the same are discussed in section 8.

7.3.5 Extending the word embedding model to use fastText embeddings

In this extension, I used fastText embeddings in place of word embeddings for gloss and context vectors. Lexeme embeddings are still taken from AutoExtend. All other operations in Distributional Lesk remain the same. FastText is another word embedding method that is an extension of the word2vec model. Instead of learning vectors for words directly, fastText represents each word as an n-gram of characters.

I imported the FastText embeddings from gensim. Then the model is initialized with vector size of 300, window size of 6 and minimum count 1. I took window size 6 because window size 6 yielded good results in section 7.3.4. The minimum word count is taken 1 since we require embeddings for all words. Vector size is specified as 300 as lexeme embeddings are of that size. This model is trained on all the sentences of the three given datasets.

Then distributional lesk is implemented and this model is used instead of word2vec model to get the word embeddings for gloss and context vectors. All other steps in the function remain the same. Then evaluation is done on the three datasets in the same manner.

7.3.6 Using sentence embeddings to compare a sentence and a gloss (SBERT pre-trained models).

In this extension, I experimented with various SBERT pre-trained models to obtain sentence embeddings for gloss and context vectors. Till now we have been dealing with only word embeddings. But sometimes, it can be beneficial to use one embedding for the whole sentence known as sentence embedding. SBERT provides many pretrained models which can be used easily. Sentence embeddings of a sentence and gloss can be compared using cosine similarity. This represents the first part of the score calculation. The next part remains the same as that of Distributional Lesk (cosine similarity of context and lexeme embeddings of the sense and the word).

Here I have used the model *all-MiniLM-L6-v2* and imported it using the Sentence Transformer. Then the Distributional Lesk function is modified to handle sentence embeddings as follows:

- *Gloss vector* - For finding the gloss vector a sense of a word, I found the sentence embedding for the sense definition and appended that to an array. Then I found the sentence embeddings for all the example sentences of that sense and appended them also. Finally the average of these embeddings are taken as a gloss vector. Here the size of the gloss vector is 768 as that is the size of the sentence embeddings.

- *Context vector* - For finding the context vector of the word, first a sentence is formed with only the context tokens (after removing function words and the given word). Then the sentence embedding is taken for this context sentence.

- Then cosine similarity is found between the gloss vector and context vector. This forms the first part of the score equation of distributional lesk.

Rest all operations remain the same. The evaluation is done on all the sentences in Senseval-2 and Senseval-3 datasets and on 500 random sentences of SemCor dataset. Here I selected only 500 sentences as the evaluation was taking almost 24 hours for 5000 sentences. The accuracy calculation is still done in the same way.

	Senseval-2	Senseval-3	SemCor
Most Common Sense	47.116%	45.969%	48.235%
Plain Lesk	43.444%	37.912%	38.916%

Table 1: This table shows the accuracy values for the baseline methods on all three datasets. We can see that Most Common Sense algorithm shows the best performance.

	Senseval-2	Senseval-3	SemCor
Distributional Lesk	36.463%	37.197%	36.908%

Table 2: This table shows the accuracy values for the Distributional Lesk algorithm on all three datasets.

8 Results and Discussion

In this section, the results of all the methods implemented are discussed. Most of the evaluations are done on all sentences in Senseval-2 and Senseval-3 and on 5000 sentences in SemCor. I tried with different sets of random 5000 sentences from SemCor and they all yielded the same accuracy.

First let us look into the accuracies of the baseline methods. From Table 1, we can see that the **Most Common Sense** algorithm has the best performance across all datasets (48%). The accuracy is also consistent across domains (different datasets). This is as expected. In the paper by Oele and van Noord (2017) also the Most Common Sense algorithm performed the best (60%). But the accuracy I got is lesser than that in the paper, this can be due to the preprocessing of data or coding efficiency. I had not removed stopwords and punctuations. For **Plain Lesk** implementation, we can see that the accuracy is comparatively lesser for all datasets. It has the worst accuracies for Sense-

	Senseval-2	Senseval-3	SemCor
Distributional Lesk	36.463%	37.197%	36.908%
Ext 1. Stopwords	35.698%	36.348%	36.707%
Ext 3. Trained Embeddings	37.684%	32.045%	36.242%
Ext 4. Parameter Change	47.116%	45.699%	48.359%
Ext 5. FastText Emb.	39.789%	33.694%	37.159%
Ext 6. SBERT Emb.	35.453%	34.827%	36.197%

Table 3: This table shows the accuracy values for the Distributional Lesk algorithm and all its extensions. We can see that the best performance is achieved using extension 4.

	News	Hobbies	Adventure	Belles_Lettres
Ext 2. Brown categories	34.036%	35.789%	37.455%	37.426%

Table 4: This table shows the accuracy values for individual categories in Brown Corpus (SemCor). This is part of extension 2.

val3 and Semcor datasets. We know that the Most common sense algorithm is difficult to beat, hence this result is also as expected. The time taken for both algorithms is almost the same. For datasets, SemCor takes the longest time to run due to its larger size.

Now let us discuss the results of **Distributional Lesk** implementation. Table 2 shows the accuracy of the method on all three datasets. We can see that the performance is around 37%. This is lesser than that of the baseline methods which is as expected. Here I have not removed stopwords. Also pre trained word embeddings are being used, they are not trained on the datasets. Hence accuracy will be lesser. But it performs consistently across domains, so this method is portable. By using more fine tuned word embeddings, accuracy can match that of plain lesk as they do not show large differences. Here Semcor took the longest time to run due to its larger size.

Table 3 shows the comparison of accuracy between Distributional Lesk and each of the extensions I implemented. As discussed the accuracies of Distributional Lesk on all the three datasets are around 37%. For **extension 1**, which consisted of removing stop words and punctuations from glosses, sentences and context, the accuracy was found to be almost the same in value. This is due to the fact that I had already removed words which do not have word embeddings initially. Punctuations were also removed in the base implementation itself. Hence we do not see any huge impact due to this change. Here also, the accuracies remain consistent across domains. The time taken is also the same.

For **extension 2**, I evaluated the Distributional Lesk algorithm on individual categories in the Brown Corpus. Table 4 depicts the results of this extension. I had considered 4 categories having the highest number of sentences. It is seen that the categories 'adventure' and 'belles_lettres' which have comparatively lesser number of sentences, shows more accuracy. It shows higher accuracy than what was obtained on the whole SemCor dataset. 'News' and 'hobbies' categories show slightly lesser accuracies. But here also the differences between the accuracies of individual categories are very small. There is not much difference between accuracy on individual categories and accuracy on the whole dataset. I had expected the model to perform better for individual categories as most of the sentences in a category will have similar context. Hence finding the sense could be easier. But from this experiment we can see that, overall context of sentences does not play an important role. Only the context of each sentence is useful in predicting the sense of the word.

Extension 3 deals with training our own word embeddings instead of using the pre-trained word embeddings from Google News corpora. After training the model on all the sentences in the three datasets and with the parameters as mentioned in Section 7.3.3, we can observe from Table 3 that, accuracy slightly increases for Senseval-2, decreases for Senseval-3 and remains the same for SemCor data. These inconsistencies may be due to the parameters selected (like window size). But overall we can see an improvement in the model. The time taken is slightly higher, as training is needed for the model.

I experimented with different parameter settings for **extension 4** and found that the accuracy increases for higher window sizes. Other parameter changes did not have much impact on the model. The model is still trained on all the sentences in all the datasets. For window size 3, the accuracies were in range 36%, but for window size 6 it increased to around 47% across all datasets. For window size higher than 8, the accuracy decreases slightly. Window size specifies the context length of the word. Lesser size gives very less contextual information. Hence by increasing the parameter value, more contextual information can be retrieved and hence the chances of predicting the true sense is higher. After a point, increasing the value, does not give any useful information as then it can contain different contexts. The results of this experiment denote that the model performs better than normal distributional and plain lesk algorithms. It is almost on par with the Most common sense algorithm. Here also time taken is similar to the time taken for extension 3.

For **extension 5**, I modified the Distributional Lesk algorithm to use fastText embeddings instead of word embeddings. We can observe from Table 3 that the model performs better for Senseval- 2

(39%) and Semcor datasets, while the accuracy decreases for Senseval3 dataset. FastText embeddings make use of n-gram embedding instead of word embeddings. Hence it can give better results for out of vocabulary words. This may be why it shows better performance for Senseval2 and SemCor. In the case of Senseval3, character representations may not be useful. It is highly dependent on the dataset. FastText embeddings were faster to train and evaluate than our own trained embeddings used in extension 3 and 4.

The last extension I tried was the **SBERT embeddings** for comparison between sentence and gloss. From table 3, we can see that using sentence embeddings actually decreases the performance of the model across all datasets. Hence we can conclude that for Distributional Lesk using embedding for a whole sentence in gloss and context do not yield any meaningful information. It is better to use embeddings for each word. I had tried with various pre-trained sentence embeddings and they all showed similar results. This extension also took much longer to run as each sentence needed to be encoded. For Semcor I evaluated 500 sentences as 5000 sentences were taking almost 24 hours.

I have not tried the extension to find a lexicon with glosses in another language and construct a (small) annotated dataset. It is due to the fact that constructing a dictionary with ambiguous words in another language was getting too time consuming and finding sense annotations for them were difficult.

I have tried to experiment with each extension thoroughly and make meaningful choices so as to improve the model. While some extensions increased the accuracy, some had no or negative impact on the model.

9 Conclusion

From the different implementations done, we can conclude that the Distributional Lesk method which uses sense and word embeddings is fast and easy to learn and is consistent across different English sentences datasets. It was seen to almost match the performance of the Most Common Sense Algorithm by using word embeddings trained on the domains with a suitable window size. Using fastText embeddings instead of word embeddings were also found to be beneficial for some datasets. But these results need to be checked in other language datasets also to ensure compatibility.

Some further steps which can be taken are to pre select the context words using wordnet hierarchy (Vasilescu et al. (2004)). In this implementation, we have taken the average of the embeddings of all the words in gloss and context to create gloss and context vectors. But in reality, some words may provide better contextual information than others. Hence one can maybe try a weighted average method. Another approach could be to check whether this algorithm can be evaluated on data which are not manually tagged. Also in this algorithm, we have selected the sense embeddings from AutoExtend. Other alternatives to this can also be explored.

Natural Language Processing is a highly evolving field with a wide scope for exploration and research. Through this paper, we have experimented with the various existing approaches along with some extensions for word sense disambiguation.

10 References

1. Agirre, E. and P. Edmonds (2007). Word Sense Disambiguation: Algorithms and Applications (1st ed.). Springer Publishing Company, Incorporated.

2. Chen, X., Z. Liu, and M. Sun (2014). A unified model for word sense representation and disambiguation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, pp. 1025–1035.

3. Escudero, G., L. Ma‘rquez, and G. Rigau (2000). An empirical study of the domain dependence of supervised word sense disambiguation systems. In Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora, pp. 172–180.

4. Lesk, M.. 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, SIGDOC ’86, page 24–26, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 0897912241. doi:10.1145/318723.318728. https://doi.org/10.1145/318723.318728.

5. Oele, D., and G. van Noord. Distributional Lesk: Effective knowledge-based word sense

disambiguation. In IWCS 2017 — 12th International Conference on Computational Semantics — Short papers,2017.https://aclanthology.org/W17-6931.

6. Sascha Rothe and Hinrich Schütze. AutoExtend: Extending word embeddings to embeddings for synsets and lexemes. In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 1793–1803, Beijing, China, July 2015. Association for Computational Linguistics.doi: 10.3115/v1/P15-1173.https://aclanthology.org/P15-1173.

7. Vasilescu, F., P. Langlais, and G. Lapalme (2004). Evaluating variants of the Lesk approach for disam- biguating words. In Proceedings of the Fourth International Conference on Language Resources and Evaluation.