Word Sense Disambiguation

Jaishree Janu, 311119

September 2022

1 Introduction

The term paper is based on the implementation of the research - *"Distributional Lesk: Effective Knowledge-Based Word Sense Disambiguation"*. It first starts by discussing the problem in hand and then implementing the baselines namely – the most common sense and the plain lesk algorithms.
Further it explains the implementation of distributional lesk.
The distributional lesk is extended by using embeddings from sentence transformer, employing other word embeddings like flair and removing stopwords from the context and glosses.
The source code can be found at my github repository `https://github.com/JaishreeJanu/Word-Sense-Disambiguation`

2 A brief on Word Sense Disambiguation problem

The goal of the WSD is to find the correct meaning of a word in the context. Words can have several meanings and those meanings are correct in different contexts. The input of a WSD algorithm is the **lemma** (or word which needs to be disambiguated) and the **context** (or the sentence in which it occurs). The output of WSD is the synset key of the meaning. For example:

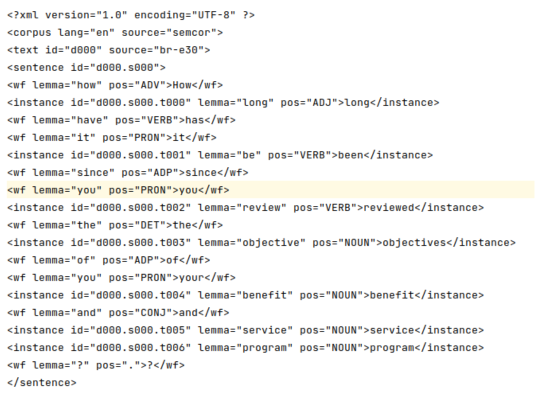
WSD Example

Sentence 1: The water is important for living.
Sentence 2: The important people will sit in the Royal box to watch Tennis.
The lemma which needs to be disambiguated in both sentences is **important**.
For sentence 1, WSD should return the correct key (also called *sense key*): *important%3:00:00::* or *of_import%3:00:00::*.

For sentence 2, WSD should return the correct key (*sense key*): *authoritative%5:00:00:influential:00* or *important%5:00:00:influential:00.*

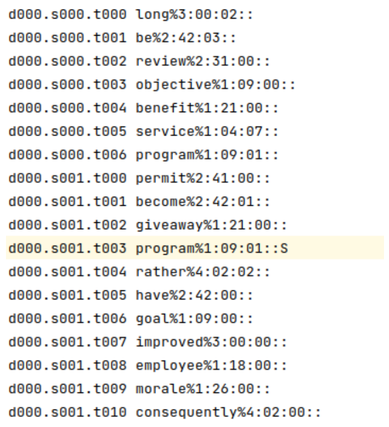
3 Introducing the dataset

I have downloaded semcor, senseval2 and senseval 3 datasets available on this website `http://lcl.uniroma1.it/wsdeval/evaluation-data`. The primary reason behind using the particular source is that it provides the lemma IDs in the dataset file (*xml file*) and also provides the correct keys/labels (*key file*) corresponding to each lemma id (for evaluation). Below snapshots will help understand the data more.



```xml
<?xml version="1.0" encoding="UTF-8" ?>
<corpus lang="en" source="semcor">
<text id="d000" source="br-e30">
<sentence id="d000.s000">
<wf lemma="how" pos="ADV">How</wf>
<instance id="d000.s000.t000" lemma="long" pos="ADJ">long</instance>
<wf lemma="have" pos="VERB">has</wf>
<wf lemma="it" pos="PRON">it</wf>
<instance id="d000.s000.t001" lemma="be" pos="VERB">been</instance>
<wf lemma="since" pos="ADP">since</wf>
<wf lemma="you" pos="PRON">you</wf>
<instance id="d000.s000.t002" lemma="review" pos="VERB">reviewed</instance>
<wf lemma="the" pos="DET">the</wf>
<instance id="d000.s000.t003" lemma="objective" pos="NOUN">objectives</instance>
<wf lemma="of" pos="ADP">of</wf>
<wf lemma="you" pos="PRON">your</wf>
<instance id="d000.s000.t004" lemma="benefit" pos="NOUN">benefit</instance>
<wf lemma="and" pos="CONJ">and</wf>
<instance id="d000.s000.t005" lemma="service" pos="NOUN">service</instance>
<instance id="d000.s000.t006" lemma="program" pos="NOUN">program</instance>
<wf lemma="?" pos=".">?</wf>
</sentence>
```

Figure 1: Snapshot of a sentence in xml file



```
d000.s000.t000 long%3:00:02::
d000.s000.t001 be%2:42:03::
d000.s000.t002 review%2:31:00::
d000.s000.t003 objective%1:09:00::
d000.s000.t004 benefit%1:21:00::
d000.s000.t005 service%1:04:07::
d000.s000.t006 program%1:09:01::
d000.s001.t000 permit%2:41:00::
d000.s001.t001 become%2:42:01::
d000.s001.t002 giveaway%1:21:00::
d000.s001.t003 program%1:09:01::S
d000.s001.t004 rather%4:02:02::
d000.s001.t005 have%2:42:00::
d000.s001.t006 goal%1:09:00::
d000.s001.t007 improved%3:00:00::
d000.s001.t008 employee%1:18:00::
d000.s001.t009 morale%1:26:00::
d000.s001.t010 consequently%4:02:00::
```

Figure 2: Snapshot of correct labels corresponding to each lemma id

4 Baselines

In this section, we discuss the two baselines: the most common sense and the plain lesk [1].

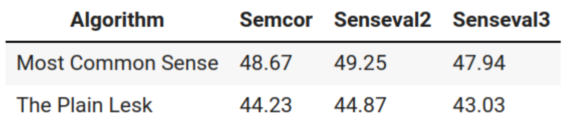
Since there was no concrete definition found for the **most common sense** algorithm, I have used the first synset as the correct label/key of the lemma.
The plain lesk algorithm finds the distance (**Lesk Distance**) between context (list of strings) and gloss/definitions (list of strings). It does so by computing the overlap between the gloss and the context. It returns the keys of the synset with whose definition the context overlaps the most. The source code for both the baselines in present in the notebook named **lesk_baselines.ipynb**

NOTE: The evaluation results of all WSD algorithms are presented in accuracy metric (in percentage). Finally, let's discuss the **evaluation** of these two baselines. The correct labels of all three datasets is found in *.gold.key.txt* files. To compute accuracy, we increase the *correct_count* variable whenever the label matches one of the keys from the predicted synset. Here is the code snippet for the evaluation of two baselines:

```
def eval_baselines(lemmas, labels):
  """
  Finds the accuracy of the baselines algorithms.
  It can take labels from any dataset: semcor, senseval2, senseval3
  """
  correct_count = 0
  total = len(labels)

  for lemma_id, label in labels.items():
    ## Get the predicted label
    ## Call the baselines
    pred_label = most_common_sense(lemmas[lemma_id].lemma)
    pred_label = plain_lesk(lemmas[lemma_id].lemma, lemmas[lemma_id].context)
    ## Get the correct label
    correct_label = labels[lemma_id][0]

    for prediction in pred_label:
      if correct_label == prediction:
        correct_count += 1
        break

  ## return the accuracy
  return (correct_count/total)*100
```

5 Distributional Lesk

Distributional lesk [3] is based on the idea of the plain lesk algorithm. It takes into account the embeddings of context, gloss and lexeme in computing the lesk distance [4]. Similarity measure is employed to find the correct gloss in the given context. The crucial part of the distributional lesk is to replace word_embeds



Algorithm	Semcor	Senseval2	Senseval3
Most Common Sense	48.67	49.25	47.94
The Plain Lesk	44.23	44.87	43.03

Figure 3: Comparison of baseline accuracies on three datasets

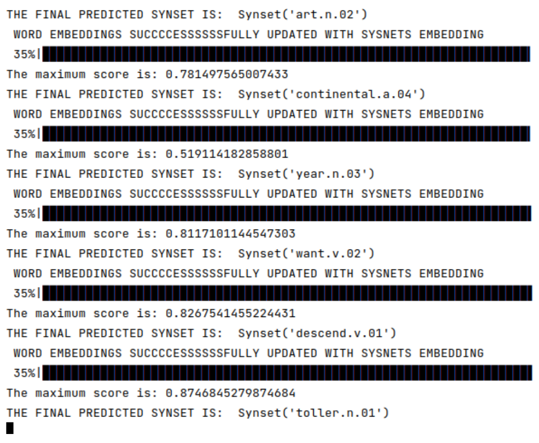
with synset_embeds for the lemmas which have been disambiguated. The pseudocode for the distributional lesk algorithm :

Algorithm 1 Dist lesk: semcor_lemmas, mapping, word_embeds, lexeme_embeds, synset_embeds

```
for $lemma = 1, 2, \ldots$ do
    context_embed ← $get\_embedding(lemma.context)$
    $synset = 1, 2, \ldots, N$
    gloss_embed ← $get\_embedding(synset.definition())$
        $synset\_lemma 1, 2, \ldots, N$
        this_synset_key ← $synset.key()$
            wn_synset_id ← $mapping[this\_synset\_key]$            end for
                lexeme_embed ← $lexeme\_dict[wn\_synset\_id]$
                    score ← $sim(context\_embed, lexeme\_embed) + sim(context\_embed, gloss\_embed)$
                        Save the synset keys which have highest score so far

                        Replace the word embed of disambiguated lemma with its synset embedding

```

The implementation of distributional lesk is described step-by-step here:

1. Load 5000 semcor instances and also read *embeddings* from mapping, lexeme and synset files.
2. Sort the lemmas by the number of their synsets. (This way lemmas which are easier to disambiguate will be disambiguated first.)
3. Loop over each lemma (single instance of semcor).
4. For this lemma's context, get the *word2vec* word embeddings. (**Note:** Embeddings of context and glosses are computed by taking the mean of each word in respective list of strings, as mentioned in the paper.)
5. For this lemma, loop over its each synset.



```
THE FINAL PREDICTED SYNSET IS:  Synset('art.n.02')
 WORD EMBEDDINGS SUCCCCESSSSSSFULLY UPDATED WITH SYSNETS EMBEDDING
 35%|████████████████████████████████████████████████████████████
The maximum score is: 0.781497565007433
THE FINAL PREDICTED SYNSET IS:  Synset('continental.a.04')
 WORD EMBEDDINGS SUCCCCESSSSSSFULLY UPDATED WITH SYSNETS EMBEDDING
 35%|████████████████████████████████████████████████████████████
The maximum score is: 0.519114182858801
THE FINAL PREDICTED SYNSET IS:  Synset('year.n.03')
 WORD EMBEDDINGS SUCCCCESSSSSSFULLY UPDATED WITH SYSNETS EMBEDDING
 35%|████████████████████████████████████████████████████████████
The maximum score is: 0.8117101144547303
THE FINAL PREDICTED SYNSET IS:  Synset('want.v.02')
 WORD EMBEDDINGS SUCCCCESSSSSSFULLY UPDATED WITH SYSNETS EMBEDDING
 35%|████████████████████████████████████████████████████████████
The maximum score is: 0.8267541455224431
THE FINAL PREDICTED SYNSET IS:  Synset('descend.v.01')
 WORD EMBEDDINGS SUCCCCESSSSSSFULLY UPDATED WITH SYSNETS EMBEDDING
 35%|████████████████████████████████████████████████████████████
The maximum score is: 0.8746845279874684
THE FINAL PREDICTED SYNSET IS:  Synset('toller.n.01')
█
```

Figure 4: Distributional lesk: execution snap

6. Get the gloss embedding for this synset's definition.
7. This particular synset could be represented by multiple synset keys. Check if any of these keys' is present in *mapping_dict* and get the *wn_synset_id*. (This way it will help to hash the lemma id's *lexeme.txt* file).
8. Now, we have all the three embeddings required to compute the score as mentioned in the paper:

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

9. Outside of the loop in step-4.
10. Find the wn_synset_id corresponding to which the score is maximum.
11. Then update the word embedding of this lemma with the synset embedding of this wn_synset_id.
12. Outside of the outer loop.
13. You can evaluate the distributional lesk comparing the predicted synset keys with the correct label present in *.gold.key.txt* file.

The source code of distributional lesk can be found in **main.py** file with helper functions in **helper.py, loader.py and dict_utilities.py**. The accuracies observed for the distributional lesk on all three datasets are:

Algorithms/Datasets	Semcor	Senseval2	Senseval3
Vanilla Distributional lesk	38.43%	39.82%	38.70%
Distributional lesk with REMOVING STOPWORDS	34.02%	37.31%	34.08%

6 Sentence transformer and evaluation results

I extended distributional lesk by using sentence transformers. I referred this resource for using sbert in my algorithm : `https://www.sbert.net/docs/training/overview.html`. There are several pre-trained sbert models available. I tried specifically these two: *bert-base-uncased* and *multi-qa-MiniLM-L6-cos-v1*. The latter performs better than the former. Since, embeddings provided by sbert are 768-dimensional and we need 300-dimensionsal embeddings (as lexeme embeddings are 300-dim), I have employed PCA for dimensionality reduction. The **sbert** model can be customized by adding a layer of dimensionality reduction. The following piece of code fulfills this task:

```
...
pca = PCA(n_components=new_dimension)
pca.fit(train_embeddings)
pca_comp = np.asarray(pca.components_)

# We add a dense layer to the model,
# so that it will produce directly embeddings with the new size
dense = models.Dense(in_features=model.get_sentence_embedding_dimension(), out_f
                     activation_function=torch.nn.Identity())
dense.linear.weight = torch.nn.Parameter(torch.tensor(pca_comp))

model.add_module('dense', dense)
...
```

The source code for distributional lesk with sbert embeddings can be found in the file **sbert_finetune.py**. The step-by-step fine-tuning of the sbert model is explained here:

1. Initialize the pre-trained *sentence-transformer* model which you want to fine-tune.
2. Add a layer to incorporate **PCA** in the model, since we want to reduce dimensionality of the embeddings.
3. Now we need a dataset to train PCA. I used this particular dataset to train PCA: `https://sbert.net/datasets/AllNLI.tsv.gz`. There was no particular reason to choose this dataset.
4. Now, we have defined the sbert. Next we want to fine-tune it and hence need some training examples.
5. The Lesk Distance between the pairs of context and glosses help us define the sentence pairs with high similarity indicator and low similarity indicator. (For example similarity between context and its correct synset/gloss is high vs incorrect glosses.)
6. After fine-tuning sbert, it can be used in distributional lesk to get the embeddings of context and glosses.

7. Evaluation of sbert embeddings was performed on all three datasets :

Algorithms/Datasets	Semcor	Senseval2	Senseval3
Vanilla Distributional lesk	38.43%	39.82%	38.70%
Distributional lesk with sbert (*multi-qa-MiniLM-L6-cos-v1*)	37.26%	**50.96%**	34.10%
Distributional lesk with sbert (*bert-base-uncased*)	25.14%	26.07%	30.70%

7 Flair embeddings and evaluation results

The stack of Flair embeddings of *news-forward* and *news-backward* is used to get pre-trained embeddings. Since, these embeddings are **4096-dimensional**, we need **PCA** for dimesionality reduction. The source code for distributional lesk with flair embeddings can be found in the file **dist_flair.py**. The step-by-step explanation of lesk with flair embeddings :

1. Train a *PCA* model with semcor/senseval instances. I used 400 senseval3 instances to train PCA for 300-dimensional embeddings.

2. Initialize the stacked Flair embeddings and use PCA to transform them from *4096-dim* to *300-dim*.

3. Use these embeddings for gloss and context sentences and find lesk's similarity scores.

The evaluation of lesk with flair embeddings are summarized below:

Algorithms/Datasets	Semcor	Senseval2	Senseval3
Vanilla Distributional lesk	38.43%	39.82%	38.70%
Distributional lesk with flair embeds	34.63%	34.20%	34.59%

8 Comparison and Conclusion

The results show that the vanilla distributional lesk doesn't perform better than the two baselines. Although distributional lesk with sbert embeddings shows considerable improvement over vanilla distributional lesk on one of the datasets. We can improve further accuracy with sbert by fine-tuning sbert on a very large dataset. Moreover, we can try to fine-tune word embeddings with relevant dataset/instances to improve performanc of the dist lesk. The performance with flair embeddings is also not very good. The performance can be improved by training flair embeddings on our dataset. The following resource can be used to do train our own flair : `https://github.com/flairNLP/flair/blob/master/resources/docs/TUTORIAL_9_TRAINING_LM_EMBEDDINGS.md`.

The problem of Word Sense Disambiguation is interesting and holds huge importance in the Natural Language Processing domain. It has a long way to go and stands several challenges. Firstly, we don't have enough labelled and relevant datasets to compare and fine-tune embeddings. Secondly, fine-tuning embeddings and evaluation tasks are very high in time complexity. Thirdly, the ambiguities in the sense definitions make word sense disambiguation more challenging.

Although the recent research in WSD looks quite promising. This research [2] uses ground-breaking technique – *translation* to solve the problem and shows good improvement in results.

References

[1] Michael Lesk. 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, 1986.

[2] Yixing Luan, Bradley Hauer, Lili Mou, and Grzegorz Kondrak. Improving word sense disambiguation with translations. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4055–4065, 2020.

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

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