IDS 566 Mini Project 2

1. Preprocessing

1.1. Dataset Cleaning

To clean up the training and validation data, we created a function clean_data which took an input file and wrote the cleaned version to a specified destination. For each line of the input file, the function cleans the words before and after the target word. For those before and after contexts, we have a clean_word function that checks if the word is punctuation or a stopword for each word. Otherwise, we lemmatized and retained the word.

1.2. Dictionary Cleaning

For the dictionary-based system, we created different functions to remove stopwords, lowered the case, renamed the column, and lemmatized the words.

2. Supervised WSD Model

2.1. Feature Vector Extraction

The function feature_extract takes input data and parameters, including window size, whether to use co-locational, co-occurrence features, and simple count function for co-occurrence features. First, we created an entry in a list for each input data line. Each entry is a tuple consisting of (a target word, sense, and feature dictionary). The dictionary's keys are the feature, and the value is the value of that feature in the example. We extracted the target word, sense, and examples using a simple regex to get a list of the parts and parts of speech of the target word from the first part of the example. Then, for each word within the window, we created a co-occurrence feature in the example's feature vector where the key is the word, and the value is its count. For each word in the window, we also extracted co-locational features. For instance, if the 5th word following the target word were "jaguar," a feature would be added to this example's feature vector. The key would be "(after,5)", and the value would be the word. We also pulled the POS of that word creating a feature with key (after-pos,5) and value being the pos of the word, as tagged by our pos-tagger from NLTK.

2.2. Classifier Construction

$$P(s_i) = \frac{count(s_i, w)}{count(w)} \qquad P(f_j|s_i) = \frac{count(f_j, s_i)}{count(s_i)}$$

To classify a set of examples, we wrote a function classifier that takes test data. It also takes a k, the value used in smoothing, softcore, and whether to use soft scoring. In the classifier, we reviewed each example and generated a probability for each sense the target word has in our training set. We developed the possibilities by going over each feature. Then, we pulled the value for that feature from our test data. We learned the prior probability of each sense and the individual feature probabilities. We use the log of probability to avoid underflow. Further, we add λ smoothing to avoid counts of zero, which are negative infinity in logs. We did this for each sense of each target word example and took the argmax to produce a prediction.

2.3. Extension: Soft-scoring

When calculating soft-scoring for accuracy, we process the classification probability scores of each word after they've been calculated using the supervised model. To prevent underflow, we modified all probability scores of each word to the exponential of probability minus max probability, where max probability is the maximum probability for that word. In addition, we normalize the scores to calculate the actual probability of each sense prediction. During the accuracy calculation, instead of adding 1 for each correct prediction, we add the predicted probability for benchmark sense.

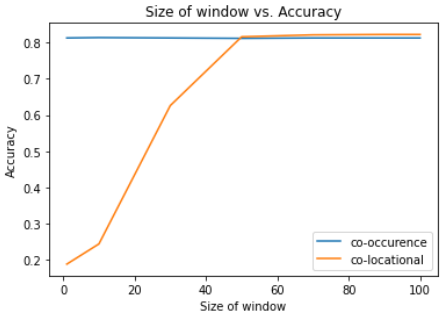
2.4. Experimentation

We tested our implementation of the supervised model in various ways using the validation data. The first test we did was to determine our baseline: simply predicting the most common sense for each word; this gave an accuracy of 81.14%. Then we tested co-occurrence and co-locational features. We tested co-occurrence features for window sizes from 1 to 100 with step size ten. Still, We stopped going beyond that because, for all window sizes, we got a similar accuracy, a slight improvement above the baseline. Our testing of co-location features showed poor performance at the beginning but improved quickly as window size increased.

2.4.1. Hard-comparing

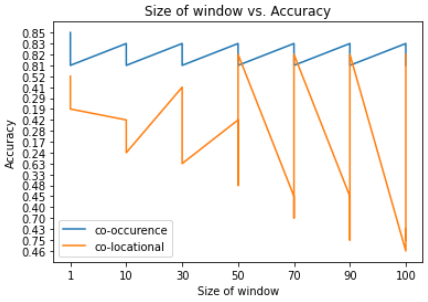
- Add 1 Smoothing

Windowsize	Co_occurrence	Colocation
1	0.8124330118	0.1886387996
10	0.8135048232	0.2443729904
30	0.8124330118	0.6259378349
50	0.8113612004	0.8156484459
70	0.8124330118	0.8210075027
90	0.8124330118	0.822079314
100	0.8124330118	0.822079314



- Add λ Smoothing

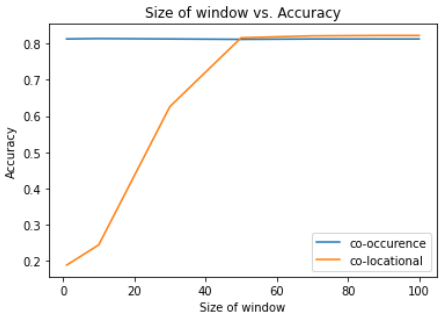
Windowsize	Add-k-smoothing	Co_occurrence	Colocation
1	0.001	0.8488745981	0.5219721329
1	0.01	0.8295819936	0.4083601286
1	0.1	0.8156484459	0.2893890675
1	1	0.8124330118	0.1886387996
10	0.001	0.8295819936	0.4222936763
10	0.01	0.822079314	0.2840300107
10	0.1	0.8167202572	0.1747052519
10	1	0.8135048232	0.2443729904
30	0.001	0.8274383708	0.40943194
30	0.01	0.81886388	0.281886388
30	0.1	0.8177920686	0.2775991426
30	1	0.8124330118	0.6259378349
50	0.001	0.8263665595	0.4169346195
50	0.01	0.8199356913	0.3290460879
50	0.1	0.81886388	0.4812433012
50	1	0.8113612004	0.8156484459
70	0.001	0.8274383708	0.4458735263
70	0.01	0.8199356913	0.4019292605
70	0.1	0.8199356913	0.6956055734
70	1	0.8124330118	0.8210075027
90	0.001	0.8274383708	0.4544480171
90	0.01	0.8199356913	0.4287245445
90	0.1	0.8199356913	0.7481243301
90	1	0.8124330118	0.822079314
100	0.001	0.8274383708	0.4555198285
100	0.01	0.8199356913	0.4340836013
100	0.1	0.8199356913	0.7491961415
100	1	0.8124330118	0.822079314



2.4.2. Soft-scoring

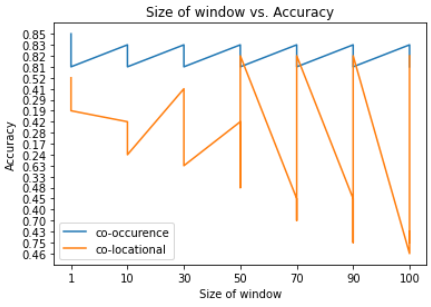
- Add 1 Smoothing

Windowsize	Co_occurrence	Colocation
1	0.8124330118	0.1886387996
10	0.8135048232	0.2443729904
30	0.8124330118	0.6259378349
50	0.8113612004	0.8156484459
70	0.8124330118	0.8210075027
90	0.8124330118	0.822079314
100	0.8124330118	0.822079314



- Add λ Smoothing

Windowsize	Add_k_smoothing	Co_occurrence	Colocation
1	0.001	0.8488745981	0.5219721329
1	0.01	0.8295819936	0.4083601286
1	0.1	0.8156484459	0.2893890675
1	1	0.8124330118	0.1886387996
10	0.001	0.8295819936	0.4222936763
10	0.01	0.822079314	0.2840300107
10	0.1	0.8167202572	0.1747052519
10	1	0.8135048232	0.2443729904
30	0.001	0.8274383708	0.40943194
30	0.01	0.81886388	0.281886388
30	0.1	0.8177920686	0.2775991426
30	1	0.8124330118	0.6259378349
50	0.001	0.8263665595	0.4169346195
50	0.01	0.8199356913	0.3290460879
50	0.1	0.81886388	0.4812433012
50	1	0.8113612004	0.8156484459
70	0.001	0.8274383708	0.4458735263
70	0.01	0.8199356913	0.4019292605
70	0.1	0.8199356913	0.6956055734
70	1	0.8124330118	0.8210075027
90	0.001	0.8274383708	0.4544480171
90	0.01	0.8199356913	0.4287245445
90	0.1	0.8199356913	0.7481243301
90	1	0.8124330118	0.822079314
100	0.001	0.8274383708	0.4555198285
100	0.01	0.8199356913	0.4340836013
100	0.1	0.8199356913	0.7491961415
100	1	0.8124330118	0.822079314



3. Dictionary-Based WSD Model

3.1. Dictionary Construction and Lesk Algorithm implementation

In our Dictionary-based model, we used the given dictionary to derive the meanings of words. The overlap score computes the number of individual words that overlap between the definitions of two words. The consecutive overlap score calculates the number of consecutive overlaps between the definitions. First, we predict the best sense = 1 for all the words in both test and validation sets for the baseline. We implemented three lesk algorithms: simple, original, and corpus lesk algorithms.

3.2. Experimentation

```
Accuracy of validation data for simple_lesk: 49.08896034297964
Accuracy of validation data for original_lesk: 41.37191854233655
Accuracy of validation data for corpus_lesk: 83.38692390139335

Accuracy of validation data for adv_original_lesk: 40.08574490889603
Accuracy of validation data for adv_corpus_lesk: 79.52840300107181
```

We first constructed the model without rewarding the consecutive overlap. After that we optimized the model by considering the reward of consecutive overlap. Consecutive overlap score is bigrams for target and context senses that are generated and the overlapping words are counted. The count was added twice to the single word score to give the higher weightage. The results showed that the corpus lesk had the highest accuracy on the validation data.

4. Analysis (Observations)

The given data, particularly the training and validation data, made it difficult for the supervised system to perform better than the baseline model. The training data had an average of 222.81 examples per word. The max was 2,536 examples for one word, and the minimum was shallow at 19. This means that there were very few examples for each word. To make matters more complicated, most of the data was biased. For example, the term ‘capital’ had 278 total examples. Among them, 258 are for sense 1, 18 for sense 2, 1 for sense 3, and 1 for sense 5. The bias in the given data made it difficult for the supervised model to create an accurate model. We believe this is one of the possible reasons the performance didn’t score much above the baseline using our supervised model. For the dictionary-based model, we found that our implementation of simplified Lesk underperformed concerning the baseline. However, our reason is that we still were not using WordNet to the best of our abilities. Moreover, running tests for the dictionary was time-consuming, considering that many tests took more than one hour to complete. Lastly, we tried to see the difference between the supervised and dictionary-based models. The supervised approach has the advantage of large amounts of training data compared to a dictionary-based system, given that there is adequate training data for the supervised model. However, having training data that is very biased towards one or another or not representative of actual uses of a specific sense can make the model wildly inaccurate. The dictionary-based model solves this by using already existing dictionaries to determine likelihood, thus not relying on adequate data.