Project 2: Report on Word Sense Disambiguation

1 Introduction

Our code is written in Python and uses several libraries, including NLTK. We used NLTK's WordNetLemmatizer for lemmatizing words in our data, NLTK's maximum entropy POS tagger based off the Penn TreeBank. Finally we also used NLTK's corpus of stopwords and NLTK's WordNet corpus for accessing more dictionary information. To run our code, please reference our README.

2 Preprocessing

2.1 Dataset Cleanup

To clean up the training, validation and test data we began by writing a function `cleanFile` which took a source file and wrote the cleaned version to a specified destination. For each line of the source file the function cleans the words before and words after the target word in the file. For those before and after words we have a `clean` function which for each word checks if the word is a punctuation, or a stopword, if so the word is thrown out. Otherwise the word is lemmatized and retained:

```
return [lemma.lemmatize(word) for word in words if
  (re.match('.*\w+.*',word) != None) and (word not in stopword.words('english'))]
```

2.2 Dictionary Cleanup

For the dictionary-based system, our input file `dictionary.xml` needed to be slightly reformatted for the Python library `lxml` to accept and parse correctly. In `utilities.py`, the function `fixDoubles` preprocesses the dictionary, fixing the instances of double senses.

```
  for i in range(20): # doing this multiple times for multiple '"'
      dictContent = re.sub('examples="(.*)"(.*)" />', \
          r'examples="\1\2" />', dictContent)
```

One of "drug"'s senses references its previous senses 1 and 2: "1&&2". We simply dropped this dictionary entry as it would not drastically affect the overall performance of our system.

`fixDoubles` generates `dictionary_processed.xml`, which we read in using Python's lxml library. Usage of this processed dictionary is described in the following Dictionary system-specific section.

3 Naive Bayes

3.1 Feature Vector Construction

In `naive_bayes.py`, the function `constructSet` takes a source as well as parameters specifying window size; whether to use co-locational, co-occurrence features or both; whether to use the part of speech of the target word in the example as a feature; and what count function to use for co-occurrence features. Co-occurrence features could be counted using either a boolean count (does the word appear or does it not), or a simple count (how many times the word appears). For each line of the source file we would create an entry in a list. Each entry is tuple of *(target word, sense, 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 by using a simple regex to get a list of the parts. Part-of-speech of the target word was extracted from the first part of the example which is structured as: `<word>.<pos>`. 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 the count of it (or simple boolean, did it appear or did it not). For each word in the window we also extracted co-locational features. For instance if the 5th word following the target word was "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.

3.2 Model Construction

In order to construct a naive bayes model (class `naive_bayes`) of the examples generated in the previous section, we generated the appropriate counts necessary. In our naive bayes' constructor we iterate over all passed in examples. We maintain a list of word counts in `self.wordcounts` which is a dictionary mapping target words to how many examples are for that word. We also maintain `self.featureLists` which is a dictionary which maps target words to sets of all features for each target word, each set is created as we iterate through examples and gather all features from each example for each word. Further we maintain a dictionary of dictionaries holding sense counts. For each word we have an entry in `self.senseCounts` which holds a dictionary where the keys are the sense and the value are the number of times that sense appears in the training set.

Finally we create `self.featureCounts`, this is a dictionary with keys of (word,sense). Then for each key there is a dictionary with keys corresponding to each feature. These features values are dictionaries which have a key for each value for that feature that appears in the training set with the number stored in the dictionary being the count of the number of times the feature with the specific value has shown up for that (word,sense) pair in the training set. To illustrate this, consider the case when we have two following feature vectors for the word "affect" with the sense "1":

```
{"f1" : "dog", "f2" : 3, "f3" : "hello"}
{"f1" : "dog", "f2" : 2, "f4" : "boat"}
```

In this case `self.featureCounts` would have one entry: ("affect","1"), this entry would look as follows:

```
{ "f1" : { "dog" : 2 }, "f2" : { 2 : 1, 3 : 1 }, "f3" : { "hello" : 1 }, "f4" : { "boat" : 1 }
}
```

It is important to note that we do not maintain count of features which are part of a specific word but do not appear in an example. For instance the second example in our previous case is missing the feature "f3" so in actuality it has a feature of "f3" with a value of null (0 if it was a boolean/number feature), but we do not count this as we can generate that count on the fly.

3.3 Classification

In order to classify a set of examples we wrote a function `classify` which takes a testSet (one processed by the function `constructSet` described previously). It also takes an alpha, the value used in smoothing; softscore, whether to use softscoring or not; and kaggle, whether to write a file compatible with kaggle storing results. In `classify` we go over each example and generate a probability for each sense the target word has in our training set. We generate the probabilities by going over each feature in our `self.featureLists` for the target word.

We pull the value for that feature from our test example if it has one, if it doesn't then the value for that feature is null/0. We then find the corresponding entry in `self.featureCounts` if the feature is not null/0 in this example, if the entry does not exist (i.e. the value does not occur in the training set for this sense) then we add

```
math.log(alpha/float(sc+alpha*len(self.featureLists[word])))
```

to our log probability–sc is the sense count. If the entry does exist, replace alpha with the count in `self.featureCounts` plus alpha in the above formula. If the feature in the test example has null/0 value then we find how many examples in the training set correspond to that value by counting how many do not have that value and subtracting that number from the sense count, the code for that addition to log probability is as follows:

```
math.log((sc-sum(self.featureCounts[(word,sense)][key].values())+alpha) /
    float(sc+alpha*len(self.featureLists[word]))}.
```

If there are no training examples with non-null/0 value then the numerator in the above code becomes `(sc+alpha)`.

The log probability for each training example also has the log of `(sc/wc)` added to it, where sc is sense count and wc is word count for the target word. Thus for each example we calculate an approximation (the log of an approximation) of: $P(s_i)\prod_{j=1}^{n}P(f_j|s_i)$ for our model. We approximate these probabilities using $P(s_i) \approx count(s_i|w)/count(w)$ and $P(f_j|s_i) \approx count(f_j|s_i, w)/count(s_i|w)$. We use the log of probability to avoid underflow. Further we use add-alpha smoothing to avoid having counts of zero which are negative infinity in logs (or 0 for probability, meaning the sense would never be selected). This is especially important because the feature vectors are often very sparse.

We do this for each sense of each target word example and take the argmax to produce a prediction. We do this for each example in the test set and then output a tuple of our accuracy and a list of tuples corresponding to `(actual_class,predicted_class)` for each example.

3.4 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 Naive-Bayes model. In order to prevent underflow, we scale all the probability scores pr of each word first by modifying them to become $e^{pr-maxProb}$ where $maxProb$ is the maximum probability for that word. We then normalize the scores to calculate the actual probability of each sense prediction. Then, during calculation of accuracy, instead of adding 1 for each correct prediction, we add the predicted probability for the golden standard sense.

4 Dictionary-based

4.1 Dictionary Construction and Lesk Algorithm implementation

In our Dictionary based WSD, we made use of both the given dictionary as well as WordNet in order to derive meanings of words. (We should note that we are using the default WordNet version 3.0, not 2.1, in our derivations. This turns out to be okay, since we are not using the provided wordnet sense integers.)

In `utilities.py`, `fixDoubles` generates `dictionary_processed.xml`, a preprocessed dictionary for which we can read in using Python's lxml library. From here, we transform the data into a dictionary, a lookup-table where the word is the key. The value, if the word is found, is the tuple of (part of speech, sub-dictionary). The sub-dictionary was further a map with key as the senseID and value as another tuple of (definition, example list, and wordNet Senses). Whenever a word was not found in the dictionary, we used WordNet to extract the word's senses and definitions, and then updated our dictionary to contain this word.

Once we know the list of senses, each sense is used in the `computeOverlap` function to calculate the *distant overlaps, consecutive overlaps, and context overlaps.* We made a few optimizations to this function in order to pick relevant words from the context and calculate overlap which you can find in the section below.

The optimizations we made to Lesk Algorithm mostly revolve around the `computeOverlap` function. Our function takes in the target word, sense definitions from both wordNet as well as the given dictionary, examples and context words within a certain window to calculate overlaps. In order to find relevant words, we kept our window to a default 10 words and further experimented by increasing and decreasing the window. We used three overlap measures to calculate the overlap rewarding each one appropriately.

The overlap score computes the number of individual words that overlap between the definitions of two words. This is calculated in the function `getOverlap`. Here we made two optimizations: adding in words given in examples to check overlaps and adding in WordNet definitions.

The first optimization we did was to add words that occur in examples of our target word to the definition itself such that it is of the following form:

['available', 'wealth', 'asset', 'take', 'lot', '', 'turn', 'idea', 'product', 'accumulation', '', 'led', **'rise'**, ...]

Now consider capital as a target word from the test set with the following preprocessed sentence to disambiguate from.

*first nine month 1988 net 85 million 3.76 share mr. simmons said third-quarter result reflect continued improvement productivity operating margin said **capital** spending next year **rise** 45 million 35 million year*

Notice that the word "rise" that occurs in context overlaps with the word 'rise" used in the definition words list that actually comes from an example. Therefore, it attaches the sense of "wealth asset" to capital which in this context of the test sentence is correct. From this we concluded that, words from the examples that are usually used around the target word if at all occur in our test sentence are highly likely to correspond to the same sense. This score is added as a part of def_overlap score and weighted accordingly.

The second optimization we did was to add WordNet definitions. We used the WordNet integers which were processed as a part of our original dictionary to determine the target word. We use this modified target word with appropriate WordNet sense to get a definition from WordNet.

```
wnstring = target + "." + pos + "."
    if wnint < 10: wnstring += "0" + str(wnint)
    else: wnstring += str(wnint)        # capital.n.02
    try:
        wndef = utilities.cleanString(wn.synset(wnstring).definition)
        def_words.extend(wndef.split(' '))
```

The consecutive overlap score computes the number of consecutive overlaps that occur in between the definitions. This is done in the function `consecutiveOverlaps` which given two sentences in the form of string lists counts the number of consecutive words that occur in both.

And finally, context overlap as we call it is a measure of the number of words from the context that overlap with the definitions.

```
if context_word in def_words:
    context_overlap += 1
```

We used the context overlap measure because we noticed that words used within context sometimes do overlap with the definition of the target word. For example from our validation set, consider the target word capital and our preprocessed test sentence:

```
capital.n | 0 | Its plans to be acquired dashed , Comprehensive Care Corp. said
it plans to sell most of its psychiatric and drug abuse facilities in California
and some other assets to pay its debt and provide working capital .
```

The definition of capital contains words like asset, wealth, business, resources etc. Notice here that the word asset that occurs in the definition also occurs within the context of the test sentence. In that case, the number of context overlaps increases by 1 and gets weighed in appropriately. Our algorithm predicts the "asset" for capital which is correct in this context. The justification for this is that since a definition can be a simple expansion of the word, the words used in the definition are likely to be used around the context as well to give it further meaning. Another reason for the potential benefits of using context overlaps is if there is not enough data to work with, or if the definition is short and concise.

With these three measures, our metric system to calculate the overlap score is as follows:

$$\text{overlap} = k \times \text{context overlaps} + m \times \text{definition overlap} + n \times \text{consecutive overlap} \quad (1)$$

We experimented with the values k, m, and n, and came to the conclusion that fairly-balanced values (like 3, 5, 8 respectively) generates higher accuracy.

4.2 Soft-scoring

To do soft-scoring for our dictionary implementation we had to find a way to generate confidence in each sense. To do this we simply normalized our overlaps scores we calculated for each sense. To account for senses that might have zero overlap we used a small alpha smoothing parameter so they did not have zero probability under our soft-scoring model. We then compute accuracy in the same way as we did for the Naive Bayes model.

5 Experimentation

We are kaggle team abms.

5.1 Naïve-Bayes

We tested our implementation of Naive Bayes in various ways using the validation data. The first testing we did was 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 and the part-of-speech feature. Because the training data for each word is very biased and very minimal (for instance 'capital' occurs 278 total times: 258 for sense 1, 18 for sense 2, 1 for sense 3, 1 for sense 5) we stopped multiplying the probability for each sense by the probability of the sense occurring in the training set for that word. Thus we are assuming that the training data is not biased at all for our testing of features. We quickly realized that the POS feature had the same value for all training examples for each word, making our Naive Bayes simply choose the first feature every time, resulting in an accuracy of only 55.84%. We tested co-occurrence features for window sizes from 1 to about 20, but stopped going beyond that because for all window sizes we got the same accuracy: 81.35%, a slight improvement above the baseline, but very minimal. Our testing of co-location features were more fruitful, showing poor performance at the beginning, but improving quick as window size increased, eventually peaking at 82.21% (see Figure 1).

We also tested how the accuracy of our Naive Bayes model would change on the validation set as we increase the maximum possible number of training examples for each sense of each word. For this test we used a Naive Bayes model using only co-occurrence features with a window size of 10. We tested both soft-scoring and hard-scoring. We found that accuracy started fairly low for both and as training size increased, accuracy increased quickly for both scoring systems, with soft-scoring always slightly lower than hard (see Figure 2).

5.2 Dictionary-based

The parameters we were able to manipulate for experimentation were the two optimizations: wordNet integers, adding in examples and changing weights for each of the overlaps. For the baseline, we predicted the best sense = 1 for all the words in both test and validation sets. Our reported accuracies:

Test: 0.54186

Validation: 0.54126

For the validation data, here is a report of our results. Note: for all the data reported below, the weights $k = 3$, $m = 5$, $n = 8$, determined through experimentation to be fairly accurate. (These weights are the relative scales for contextual overlap, definition overlap, and consecutive overlap for each target word and its sense and context.)

Window size	With examples	With WordNet defs	Accuracy
5	N	N	0.33112
5	Y	N	0.36763
5	N	Y	0.44653
5	Y	Y	0.43194
10	N	N	0.32904
10	Y	N	0.35798
10	N	Y	0.44587
10	Y	Y	0.43301

A brief guide: window size is the number of words left of and right of the target word with which we are taking definitions to compute overlap with the target word definitions. "With Examples" means, in our system, we additionally take the provided examples in our dictionary and treating the words as part of the definition. "With WordNet defs" means, we use the WordNet integers corresponding to WordNet 2.1 senses and looking up additional definitions to include in our computation of overlap.

Observations: It looks like including WordNet definitions helps improve our system tremendously, as much as a 10% increase in accuracy. In both window sizes, using only WordNet, and no examples, produced the most accurate sense predictions. It looks like the inclusion of examples can help performance (probably due to the shortness of many dictionary definitions), although a mix of examples and WordNet may not be fruitful. One of our additional experiments was with a window size of 8, with $k = m = n = 1$: **0.4523**. This result supports the possibility of balanced weights for each overlap type. We will discuss why we could not match the baseline later.

For our test data to upload to Kaggle, here is a comprehensive chart of our results. (For all, we did not use soft-scoring, as it did not perform up to par.)

Test	Window size	k, m, n	Examples?	WordNet?	Accuracy
Baseline	–	–	–	–	*0.54186*
–	5	1,5,15	N	N	0.35681
–	5	1,5,15	Y	N	0.41348
–	5	1,4,5	N	Y	0.43221
–	5	1,4,5	Y	Y	0.4296
–	5	1,6,3	N	Y	0.43287
–	5	1,3,6	N	Y	0.43083
–	5	3,5,8	N	Y	0.43134
–	6	1,5,15	Y	Y	0.43287
–	8	5,1,15	Y	Y	0.43492
–	10	1,5,15	Y	N	0.39051
–	12	5,1,15	Y	Y	0.4367
–	15	3,5,8	Y	Y	0.44666
–	16	5,1,15	Y	Y	0.44181
–	20	1,4,5	N	Y	0.43849
–	20	5,1,15	Y	Y	0.44283
–	20	3,5,8	Y	Y	**0.45202**
–	20	3,5,8	N	Y	0.4367
–	5	3,5,8	N	N	0.35911
–	5	3,5,8	Y	N	0.38617
–	5	3,5,8	N	Y	0.44564
–	5	3,5,8	Y	Y	**0.45559**

We should note that one of our preliminary tests was simply returning the first sense for each word as ordered in the dictionary, regardless of any context or definition. This produced the high accuracy of **0.54186**. Some observations: a higher window size does not immediately translate to higher accuracy; however, WordNet definitions help tremendously. There is almost a divide between accuracies when WordNet definitions are used and are not used. Our initial choices for k, m, n, like 1,5,15, was motivated by the fact that consecutive overlaps are very significant and are a likely indicator for sense. However, our highest results where when we used fairly balanced weights for contextual, definition, and consecutive overlaps, like $k = 3$, $m = 5$, $n = 8$.

5.2.1 Conclusion

- We could not reach the accuracy of the baseline because senses overlapped: we loaded in more senses than the test and validation data used. Often, the first and second senses were much more popular, with up to 70–80% of cases being in these senses. (Perhaps we could have loaded a partial dictionary, sacrificing senses for better precision with the most common senses.)
- We could not reach the accuracy of the baseline because the WordNet integers allowed us to get a much more substantial dictionary: more senses allows for more incorrect overlaps.
- Examples help more substantially in a small window size when k is relatively high; it weighs context overlaps more highly, which is very apparent, since examples are all context.

- Examples do not help much with a large window size; they are prone to dilute the data, especially with WordNet definitions.
- With a bigger window, consecutive overlaps are likely to become more useful and less sparse simply due to the inclusion of more words. (Supported by our data)

6 Discussion

The data provided to us, in particular the training set and validation set, made it a struggle for the supervised system to perform well. The task at hand essentially required a separate Naive Bayes model for each target word, meaning that data was sparse. The training set had a median of only 128.5 examples per word, and a mean of 222.81. The max wasn't that large, at 2,536 examples for one word, and the minimum was incredibly low at 19. This means that there were very few examples for each word. To make matters more difficult most of the data was very biased. For instance, 'capital' had 278 total examples: 258 for sense 1, 18 for sense 2, 1 for sense 3, and 1 for sense 5. Because of this bias in the data it made it very hard for Naive Bayes to create an accurate model of what words might be important in context for most senses of most target words. We believe this is the reason we struggled to score much above the baseline using our supervised model.

After analyzing our models we found that the most important class of features were the part-of-speech co-locational features. Using only these features we achieved an accuracy of 82.32% on the validation set (window size 90). Second best were word co-locational features which achieved an accuracy of 81.89% on the validation set (window size 90). Third best were co-occurrence features which had an accuracy of 81.35% (window size 90) on the validation set. We believe that POS co-locational features were best because they generalized more than word co-locational features, as having a noun at the 5th position following the target word is more general than having 'cat' at that same position as a feature. We believe that co-locational features were probably powerful than co-occurrence features because they are for specific positions rather than general words over the whole window.

Running tests for the dictionary was a bit painstaking, considering that many tests (with a window size of 20) took as much as 40 minutes to complete. We realize that our implementation of simplified Lesk underperformed with respect to the baseline (discussed above), but our reasoning is that we still were not using WordNet to the best of our abilities, even when looking up definitions in WordNet 2.1.

The supervised approach uses training data to create a model of the probability of features drawn from those training examples (such as co-occurrence and co-locational features) to determine how well a test example fits the model for a certain sense. The dictionary approach uses already existing dictionaries to compare how well a test example's in-context words' definitions overlap with the definitions for the target word's senses to pick which sense is most likely. The supervised approach has the advantage of doing very well with large amounts of training data in comparison to a dictionary based approach which will typically not do as well, given that there is adequate training data for the Naive Bayes model. However, this makes Naive Bayes very susceptible to problems with training data. For instance, having a training set which is very biased towards one sense or another, or is not representative of real uses of a specific sense can make the model very inaccurate. Further often there is not enough data to provide a supervised learning approach an adequate representation of the general use of a sense. This means the model will be very specific for each sense and thus not generalizable. The dictionary approach solves this by using already existing dictionaries to determine likelihood thus not relying on getting adequate data.

7 Figures

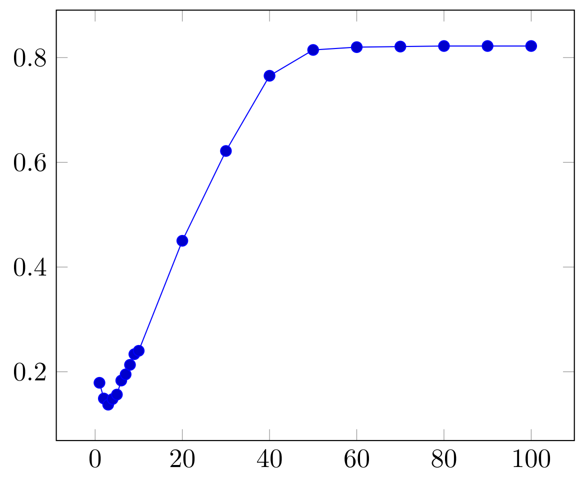

Figure 1: Size of window vs. accuracy for co-locational features

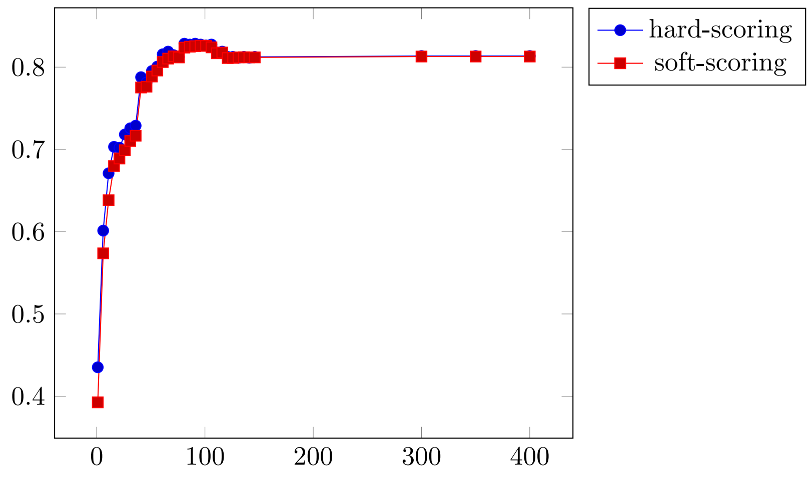

Figure 2: Maximum size of training set for a single sense vs. accuracy for a naive bayes model with window size 10 using only co-occurrence features.