NLP Project 4

Ryan Enderby (rae83)
Liane Longpre (lfl42)
Jordan Stout (jds459)

Basic observation and analysis

Brown pre-trained cluster analysis

We began this project by analyzing the pretrained Brown clusters. We looked at both the 1000 cluster and 3200 cluster datasets, each with frequency cutoff 50. Across both sets, one of the most interesting things to note was that many clusters contained words that were opposites of each other. Take the cluster that only contains “can” and “cannot.” Naively, when first presented with the concept of clusters, we assumed that they would contain words of close semantic meaning. As such, seeing polar opposites was initially curious, but made perfect sense when considering how clusters are derived.

The Brown clusters are based on a hierarchical clustering of words based on their occurrence contexts. Linguistically, “can” can appear in place of “cannot” in any sentence (take this one for example). Although as humans we can intuitively and clearly see that this would invert the meaning, a computer cannot. And as such, the contexts of the “can” and “cannot” could be very similar, and therefore end up in the same cluster, regardless of what they actually mean.

Another interesting observation was finding words of different POS and seemingly little semantic relation within clusters. For example, there is a cluster in the 1000 cluster set that contains words about water features: “inland,” “underwater,” “downstream,” “environmentalist,” but also two abbreviations: “MBA” and “LBO.” This is because of the very domain-specific use of words like downstream and underwater, namely in finance when they refer to players in a market and the possibility liquidity status of companies. As such, we saw that words that are only related in highly-specific domains could end up in cluster together, regardless of general use.

We ended up choosing the 1000 clusters dataset because of its inclusion of highly-occurring uncertain words like “perhaps,” which did not even occur in the 3200 cluster set. Further, the 1000 cluster set had “some” in a cluster with words like “many” and “most,” whereas the 3200 set had it in a cluster by itself and nowhere else. We reinforced this decision by spot checking some of the highest occurring words with uncertainty tags from our training documents to find which cluster set contained the most.

We identified a few problems likely to arise from using clusters to update project 2, the uncertain sequence and sentence tagging system. Words of opposite meaning and domain-specific relations in clusters could introduce a larger number of false positives into our system when clusters are used to identify words with a probability of being uncertain. Through our implementation and analysis, we found this to be true.

Identification of the sparsity problem

We first saw the issue of sparsity when testing our original uncertainty-tagging system. The system disproportionately guessed the “O” tag, meaning that it more often than not would tag both certain and uncertain phrases as being certain. This was because of the sparsity of uncertain tags in a relatively small training set. Relative to “O” tags, there was a small number of “B” and “I” on which to train our system.

We proved that this was an issue when we started testing different downsampling and upsampling methods. We achieved our best results when we would repeat uncertain phrases in training multiple times, and randomly omit documents that had no uncertain phrases. As such, we saw that the sparsity of uncertain tags was a problem

This observation motivates our cluster-based modifications. To overcome the sparsity of uncertain examples, we implement the following changes to how emission probabilities are calculated. We now use clusters of words. For a word that is tagged in training as uncertain, we increase the count of not only that word, but all words in its cluster as well. This method assumes words in the same cluster as the uncertain word will also indicate uncertainty. This will effectively help upsample uncertain examples, and broaden our ability to recognize a range of uncertain words, even when some words appear far fewer times than other uncertain words in our training set.

Improvement over previous system

From our original project 2 work, we saw that upsampling uncertain tags was necessary due to the sparsity of uncertain tags within training documents. A large portion of training documents even had no uncertain tags at all. As such, we wanted to use pre-trained clusters in a way that upsampled the number of words with probability of having an uncertain tag. We change our original system’s method of calculating emission probabilities to consider information extracted from word clusters.

Our original system implements a hidden markov model and uses the Viterbi algorithm to find the most likely BIO tag sequence. Transition probabilities are the bigram probabilities of the next tag sequence (e.g. P(I|B)). The emission probabilities are the probabilities of the word given the BIO tag (e.g. P(“perhaps”|B)). We also built in the ability to upsample by doubling any sentence that contains uncertain words.

$$P(word \mid tag) = \frac{count\,(word\,,\,tag)}{count\,(tag)}$$

For this assignment, our improved system alters the original system by changing the calculation of emission probabilities. In training, each time an uncertain tag is encountered, we alter the counts used in calculating emission probabilities. For every word in the cluster of the original uncertain word, we can add +1 to the count of the bigram (tag, word) as well as the unigram count of the tag (in order to maintain probabilities summing to 1). In doing so, we effectively add probability for every word in the cluster occurring given an uncertain tag.

We made this design choice based off of our analysis of clusters and the thesis that if a word is seen as uncertain during training, other words in its cluster are likely to be used as uncertain. As such, to upsample uncertain words in training, we decided to add count to words in the same cluster. This decision was further supported by the observation that in doing so we are handling seen uncertain words, words that might be seen in training but not seen in an uncertain context, and words that do not appear in training at all. As such, our system is now more robust for handling different scenarios.

We experiment with adding counts for all words in a cluster different percentages of the time. When originally adding counts for all word clusters that contain any uncertain word, analysis of clusters revealed that sometimes the clusters of uncertain words did not accurately contain words that were indicative of uncertainty. As such, we saw that we could be adding count to words that were unlikely to ever be seen as uncertain. We also kept our original upsampling method that gave us the ability to double the count of uncertain words and tags when seen.

Integration of word clusters in emission probability calculation improved our system. This method of upsampling performs better than our best upsampling method from project 2. Analysis is provided in the next section. Using word clusters improves handling of unseen words. If certain uncertain words do not appear in our training data but do appear in a word cluster, the probability of the word being tagged as uncertain is not zero (as it would have been in our previous system).

Experiments and analysis

Experiment approach

In analyzing our system, we compared adding counts to words in a cluster different percentages of the time, as well as experimented with combining our cluster approach with an upsampling method from our original project 2 submission. For example, in the table below, "% of clusters added" means the percentage of time that we added +1 to the count of all words in a cluster when one word in the cluster was found to have an uncertain tag during training.

The "Doubled B/I tags" and "No doubling of B/I" tags refers to our original upsampling system. Since the number of uncertain tags in the training docs was considerably smaller than the the number of 'O' (not uncertain) tags, this upsampling method involved adding multiple counts of the (uncertain_tag, uncertain_word) pairs to our bigram probability calculations instead of a single count. During our project 2 experiments, we found the greatest improvement in score when we doubled the count of uncertain words. We experimented with our cluster-based upsampling method in use both with and without this tag-doubling upsample method.

Evaluation

To calculate the precision, recall and F-score, we evaluate on a validation set. A proportion of documents (here set to 10%) is randomly chosen and excluded from training, and included instead in a validation set. After training the model on the training set, we then look at

the output of the model on the validation documents, comparing the model output to the provided answers.

Results are shown in the table below. At 100% (the top row), we added count to all words in a cluster whenever any word in that cluster was found to have an uncertain tag during training. In the last row, we never added count to words in the cluster. This last row therefore represents our original system, both with and without doubling uncertain tags when they are seen in training.

Results

Our best results in terms of recall and F-score happen when we add count for all words in a cluster 15% of the time. Our highest precision occurred when we added count for all cluster words 25% of the time and did not double B or I tags. However, highest recall at 15% of clusters was *with* doubling uncertain tags, and highest F-score was without. This illustrates the clear tradeoff between precision and recall that we faced in the design of our system.

Combining both upsampling methods yielded the best recall, and this makes sense, since the more uncertain tags we add count for during training, the more likely we are to identify the real uncertain tags during validation. However, this comes at the expense of precision, as we are clearly now returning a higher number of false positives. Since a cluster often contains some relevant words and a few that are less than intuitive, it makes sense that this method would increase the number of false positives. We are in some cases even adding count for words that exist in a cluster but not in our training set. As such, we correctly identify more uncertain words, but also incorrectly identify certain words as uncertain.

Highest precision occurred when adding 25% of clusters and *without* doubling uncertain tags. The precision for this was 0.613, whereas precision for 25% of clusters *and* doubling was only 0.282. Similarly, recall for the non-doubling system was half that of its doubling counterpart. Again, this intuitively makes sense as we are reducing the number of false positives.

It is exciting to note that both the lowest and the third-lowest F-score returned were with our original project 2 model. In both the doubling and non-doubling columns, in almost all cases, both precision *and* recall were higher when adding count for any given percentage of clusters. This is probably because our baseline system is only capable of identifying a word as uncertain if it has seen it as uncertain before. The old upsampling method therefore only improved our old system's ability to identify words it has already seen as uncertain. In our new system, even though it may introduce more false positives, the clusters allow us to handle both words that we have seen as uncertain, as well as some words that might be uncertain that were not listed as such in training, and words that appear in clusters but did not appear anywhere in training. As such, even with more false positives, precision increases as we are able to handle more scenarios involving uncertain words.

Although we are reporting these results as an average over 30 trials, and our code is currently set up to run over 10 trials (for speed, just in case the TAs plan to run the Python file), there is still a considerable amount of variability in results. I expect that this has to do with how we randomly select documents to be excluded from training and serve instead as a validation set. Since many of the training documents have few or no uncertain tags at all, there is a

reasonable chance that the validation set will contain very few uncertain tags. As such, the training set will present the documents that have higher concentration of uncertain tags, which are then upsampled via various means, thus giving us a higher number of false positives, lowering our F-score.

% of clusters added	Doubled B/I tags	No doubling of B/I tags
100	Precision: 0.263 Recall: 0.589 F-score: 0.363	Precision: 0.500 Recall: 0.281 F-score: 0.359
50	Precision: 0.282 Recall: 0.542 F-score: 0.371	Precision: 0.455 Recall: 0.357 F-score: 0.400
25	Precision: 0.282 Recall: 0.605 F-score: 0.385	**Precision: 0.613** Recall: 0.305 F-score: 0.407
15	Precision: 0.289 **Recall: 0.663** F-score: 0.403	Precision: 0.549 Recall: 0.373 **F-score: 0.444**
10	Precision: 0.232 Recall: 0.565 F-score: 0.329	Precision: 0.507 Recall: 0.296 F-score: 0.374
0	Precision: 0.242 Recall: 0.500 **F-score: 0.326**	Precision: 0.454 Recall: 0.260 F-score: 0.331

Work distribution

Liane implemented the changed emission probability algorithm by adding counts to cluster words. Ryan implemented creating the clusters as well as experimented with percentage of clusters added and doubling uncertain tags. Jordan helped both with implementation. The write-up was worked on together by the group.

Code

All code is written in Python 2.7. We prototyped in IPython notebooks and exported as HMM.py. Running from the command line will print out the average results for a number of training and validation trials (currently 10), as well as generate the output documents that were specified in project 2. HMM.py requires the follow packages: baseline, os, csv, collections, numpy and nltk.

HMM.py imports some functions from baseline.py for producing the output span and sentence prediction csv files. All six folders of training, test-private, test-public, and their edited counterparts need to be in the same directory structure as the Python files. As it is in the zip archive, HMM.py can be called from the command line to print out averge results and generate output documents.