1 Active Learning

A common scenario in natural language processing is having far more data than annotations, since data collection is easy and annotation is hard. Two common techniques for making the best of this imbalance are semi-supervised learning and active learning. This report focuses on active learning; in this particular scenario, 'active learning' simply means using prior annotations to select which unlabeled sentences would be most useful to annotate.

We find that active learning, particularly using the 'tree entropy' metric, significantly boosts the usefulness of additional annotations.

Active learning is most useful at the very beginning of the learning process. At this point, the difference between a random sample of the unlabeled data (potential annotations) and a heuristically selected sample is the greatest. However, we are using a limited pool of unlabeled data (which is somewhat realistic; we will rarely annotate all of the data we have on hand). As we use more and more of the unlabeled data, the difference decreases until we reach the final batch of ~ 60 sentences, at which point the heuristic selection function must select the same sentences as the random function, since that's all there is left.

2 Comparison between heuristics

We used three heuristics to select unlabeled sentences to 'annotate' (of course, we did not annotate any sentences, we merely allowed ourselves to see the pre-existing annotations for certain sentences). The goal is to minimize annotation costs; we approximate that by trying to minimize the number of sentences we train on. It's debatable whether annotation costs are most directly related to token counts. Particularly if the sentences are preprocessed in some naive way (as the Penn treebank was), there might be other factors more important than token counts—aspects of verifying and completing a parse that make some sentence more costly than another.

1. **Tree entropy.** By measuring the distribution of trees and the confidence of the parser, we can select sentences that the parser is clueless about. If the parser thinks each of 20 trees are nearly equally probable, tree entropy will very high; if it vastly prefers a single parse to all the others, tree entropy will be low. We select those sentences where the parser produces high entropy, because those will presumably be most instructive.

2. **Top parse.** The parser assigns a probability to the highest parse in a sentence, relative to other potential parses. This is not directly related to such probabilities measured for other sentences, but by normalizing for sentence length, we can facilitate a crude comparison.

3. **Length.** Longer sentences will have more complex parses, which are presumably more instructive.

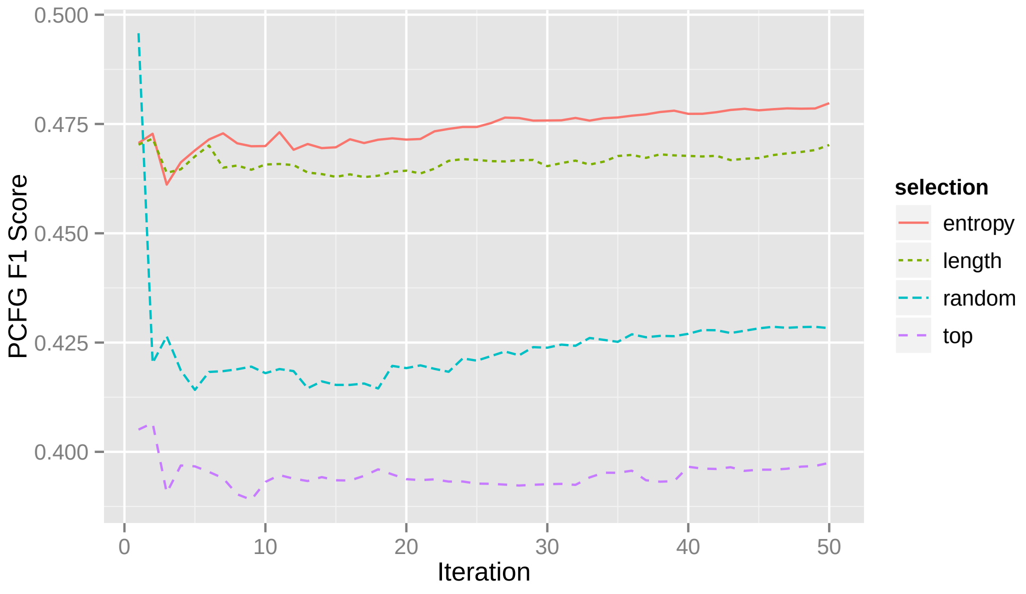
We compare these to an incremental learner that randomly selects a number of new sentences to annotate from the unlabeled pool.

As expected, the random selection function is the weakest at the beginning. The tree entropy measure is the best, followed by the length-normalized best parse, and then the sentence length.

3 Semisupervised

I also tried training the parser on it's own parses; I had the parser re-annotate the unlabeled sentences and then learn from those. This was not at all what the parser was built to do, but surprisingly, this was marginally helpful. I used the same selection functions, but sometimes in different order;

for example, I had it learn from the sentences with the lowest tree entropy, i.e., those sentences for which it was most confident in the best parse.



4 Conclusion

Ideas for improvements

In a truly active learning situation, we could construct the parser to gauge the most useful-to-annotate subtrees; our parser could determine what part of a tree it is highly confident about, and then present those parts of the tree to the annotator cemented together, leaving only the unconfident structure for the annotate to fill in.

4.1 Instructions

- All code and full SBT project, is available at `https://github.com/chbrown/nlp.git`.
- Commands used to run the code can be found in `hw03.README`.
- I made considerable changes to the Stanford Parser source code because it was emitting a huge number of logging messages and does not allow setting the log level. I was able to silence some of the messages via a custom TreebankLangParserParams class with pw() method overrides, but there were still far too many System.err's hard-coded into the Stanford Parser source. With all the jars that the library requires, I'm left with about 80 MB of free disk space on the CS cluster (which isn't too surprising considering I only have 256 MB to begin with). The Stanford parser quickly burned through that legroom with all the forcibly logged parse trees and parameters displays. So I replaced many of the forced standard error calls with log4j rootLogger.trace calls, which brought the library's output down to a much more reasonable size.