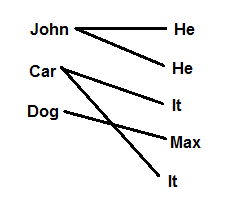
Jesse Waite

10752500

Structured Prediction, homework 1

1. The LaSO framework uses the concept of priority queue-based tree search to learn a parameter vector **w** capable of guiding the search toward y-good terminal nodes. The problem of cluster-based coreference resolution corresponds with finding a suitable assignment of mentions (“John”, “he”, “his”) to clusters C1, C2, thru Ci. In the context of cluster-based coreference resolution, structured prediction corresponds to finding a weight vector **w** such that for any input sequence X, we find a correct bipartite matching of real entities **E**={e1,e2, … en} to subject references **R**={r1,r2, … rn}. Here is one such example assignment for the sentence “**John** drove the **car** so **he** could walk the **dog**. But **it** broke down, so **he** fixed **it** while **Max** waited.”:



Clearly, the assignment of edges in this bipartite graph fits within our overall framework of structured prediction as a graph coloring problem. Here, edge colors would be binary, “1” if the entity and reference correspond, and “0” otherwise, subject to the constraint that each reference corresponds to only a single entity.

Although this problem clearly fits our structured prediction framework, I’m uncertain how to map cluster-based coreference into the tree-based framework provided by LaSO, since it isn’t immediately clear how bipartite matching reduces to a recursive-style tree search problem.

Nonetheless, one way to do so would be to read the input sequence backward, assigning references in **r**  to unique entities in **E**. The assumption for this approach is that entities always occur upstream of their references. To map this approach into LaSO, each node in the search tree corresponds to the occurrence of a reference when reading the input word sequence in reverse. The queue is composed of potential candidate entities with which to bind the current reference, with the learning goal requiring that the correct entity is highest-ranked. Thus, for a beam of size 1, we would require that the correct entity was predicted for every reference encountered. For a beam size of infinity, the beam would include all possible entities, but we would still define an error preferring higher rankings for the correct entity. The successor function in this model would simply provide the next reference to be assigned to some upstream entity. This gives the search space definition:

I: The first reference occurring in the reversed sequence.

S: The next reference occurring in the reversed sequence.

H(): A ranking function over entities, constrained to rank the correct entity highest on the queue

Search error: Occurs whenever the ranking of entities does not contain the correct entity, within a beam of size *b.*

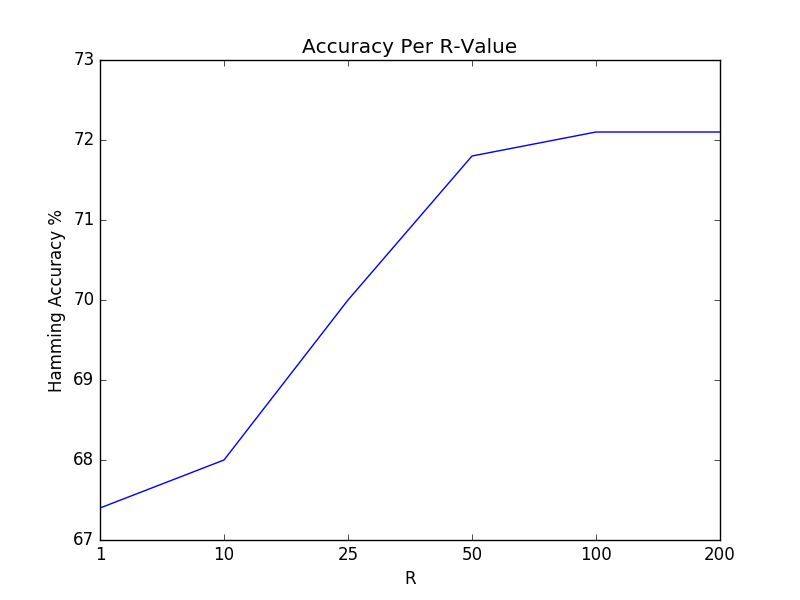
Notably, this is a degenerate use of LaSO, since there is no beam update. The reason is that the successor doesn’t generate nodes from the queue, but simply grabs the next reference to resolve to some entity. Hence, this is really just an iterative method of reducing the error in search ranking, so my method doesn’t leverage search beyond a single layer.

2. b. The plots of accuracy for R=20, eta=0.01 are shown for the net-talk and ocr datasets, for maxIterations of 1, 10, 25, 50, and 100. For both datasets, the accuracy did not improve much after the first 15 or so iterations, as shown, for all features representations. The left-hand figures are for the ocr data, and the right are for the net-talk data. The first row are for the first-order features, the second row for second-order features, and the last for third-order features.

|  |  |
| --- | --- |
| C:\Users\jesse\AppData\Local\Microsoft\Windows\INetCacheContent.Word\phi1OcrAccuracy.png | C:\Users\jesse\AppData\Local\Microsoft\Windows\INetCacheContent.Word\phi1NetAccuracy.png |
| C:\Users\jesse\AppData\Local\Microsoft\Windows\INetCacheContent.Word\phi2OcrAccuracy.png | C:\Users\jesse\AppData\Local\Microsoft\Windows\INetCacheContent.Word\phi2NetAccuracy.png |
| C:\Users\jesse\AppData\Local\Microsoft\Windows\INetCacheContent.Word\phi3OcrAccuracy.png | C:\Users\jesse\AppData\Local\Microsoft\Windows\INetCacheContent.Word\phi3NetAccuracy.png |

As shown, the accuracy did not improve for feature representations, nor for the different datasets, after the first 15 or so iterations. The net-talk data was apparently easier to learn, while the ocr data had lower test accuracy. The results show that iterations were not the biggest bottleneck of the learning algorithm, nor did larger feature representations substantially improve test accuracy, even with many training iterations.

2. d. Next, we tested the effect of varied the R parameter to observe its impact on test performance. The hamming accuracy is shown below for R = 1, 10, 25, 50, 100, and 200, using only first-order features. Increasing R improved accuracy, however not considerably, and the improvement tapered beyond R = 50.



2. e. As we observed, the bottleneck in the algorithm was not the number of iterations, nor the complexity of the features. This eliminates the learning algorithm and the complexity of the features as possible causes for the upper performance bound. Most importantly, the R parameter yielded the greatest improvement in algorithmic performance, but only up to a limit. This implies that the random search algorithm was the core bottleneck of the algorithm, since the randomly generated base strings in the inference procedure provided an unprincipled basis for searching for better inferences. In short, to improve the algorithm, we would need a better and more principled search algorithm, one that based its inferences on a better search method, or that generated initial inferences based on learned information.

2. f. One observation of randomized greedy search is that the uniform randomness of the start point for each search creates instability in the learning algorithm. Generating random label sequences will often yield sequences with labels that closely map to the input x-sequence, and thereby the sequence is already biased to climb toward knowledge it has already learned. In short, randomly-generated sequences are not the worst-case starting point for beginning a search procedure, since even randomness lends some bias by generating label positions that closely match their corresponding positions in the input x-sequence.

One way to get around this is to generate worst-case strings at every iteration. That way, the search procedure is forced to start learning from the label sequence furthest from the target sequence. This is a method for maximizing the amount of search performed by the search procedure, such that we maximize the number of mistakes made by the learner, and minimize the likelihood of its learning inferences being partially and accidentally correct.

To test this, instead of generating random sequences within *InferRGS()*, I simply return a list of the same random character repeated *n* times, where *n* is the length of the input sequence. For instance, “aaaaaa” or “bbbbbbb”. The effect is that the search procedure is forced to hill-climb according to only its current weights, not from positions with some small chance of being partially correct.

As shown, the search method works surprisingly well, for 15 iterations, eta=0.01, and first-order features. The search method appeared to work better than using completely random sequences, but not significantly. At most, it can be claimed the method didn’t harm performance.

