Homework 1

Luke Weber, S.I.D. 11398889
CptS 580 — Structured Prediction
Prof. Jana Doppa, Ph.D.

February 21, 2017

1 LaSO Framework Advice on NLP Domain

1.1 Search Space

Tom might want to consider a somewhat standard search space (from what I've seen) for structured prediction problems: Let there be a search tree T of depth D, where D represents the quantity of mentions in each given document. Each node $n_i \in T$ contains the structured input vector of mentions $x =< m_1, m_2, ..., m_D >$ with a corresponding partially completed $\hat{y} =< y_1, y_2, ..., y_D >$, where $y_j \in \{C_1, C_2, ..., C_k\}$, for k clusters and for $1 \leq j \leq D$.

When searching, the start state $S \in T$ will contain both x and $\hat{y} =< -, -, ..., - >$, where $\hat{y}$ begins as the vector of completely unassigned clusters. The children of S will be nodes each containing one unique possible assignment of the first cluster y_1; and since there are k possible clusters with which to assign y_1, there are correspondingly k child nodes of S. Then, for each of these children of S, a similar process takes place: there exists k child nodes each with a unique assignment of the y_2 cluster in $\hat{y}$. And so the process continues until all of the D clusters in $\hat{y}$ have been assigned, and we arrive at the depth of T containing only terminal nodes. Therefore, the completeness of the labelling (i.e. assignment of clusters) on each $\hat{y}$ depends only on the depth of n_i.

1.2 Heuristic Analysis

If Tom decided to learn an accurate heuristic to guide the search process, the only notable drawbacks would its memory and time constraints (Yuehua Xu and Alan Fern, *On Learning Linear Ranking Functions for Beam Search*). However, this is for an unrestricted search, such as standard Breadth-first Search (BFS), where the branching factor makes our search space grow exponentially as we increase tree depth — we lose tractability here.

One way to maintain tractability is to use Breadth-first Beam Search, as suggested by Xu and Fern. If Tom does this, a learned heuristic (i.e. a ranking function) could outperform many modern heuristics.

2 Structured Perceptron with RGS

(a) See **Attachment 1** for the complete Python code (also available at `https://github.com/lukedottec/StructuredPerceptron`).

(b) Visit **Attachment 2** for the plot of the Hamming accuracy as a function of the number of iterations, for the number of maximum learning iterations $MAX = 100$, learning rate $\eta = 0.01$, and number of restarts (in RGS) $R = 20$. Note that it was not explictly stated, but we used a unary feature representation for ϕ here!

(c) Repeating (b) with varying dimensions on ϕ, we have:

- First-order: $\phi \in \mathbb{R}^d$ where $d = m * k + k^2$. See **Attachment 3** for the plot.
- Second-order: $\phi \in \mathbb{R}^d$ where $d = m * k + k^2 + k^3$. See **Attachment 4** for this graphing.
- Third-order: $\phi \in \mathbb{R}^d$ where $d = m * k + k^2 + k^3 + k^4$. See **Attachment 5** for the plot.

(d) For our last plot, see **Attachment 6**. Here we plot the Hamming accuracy as a function of the number of restarts R allowed within our Randomized Greedy Search inference process. Model parameters are similar to (b), but with a first-order ϕ representation.

(e) Many methods were employed to diagnose the (poor) performance of my model:

- Limit alphabet α of labels, thus reducing the amount of characters we have to learn. Varied ϕ's order (1, 2, 3, and 4) within each of these batch of labels using limited α's. Note that limiting the alphabet has non-trivial consequences, and pushes the model to determine patterns between characters in α that wouldn't ordinarily happen. For that reason, this was not the most intelligent test. However, it was found that with very limited α (e.g. $\alpha = \{a, b, c\}$), increasing the degree of ϕ improved the accuracy in a non-linear — possibly exponential — manner. For instance, we jumped from 40% up to 77% accuracy merely by increasing the features in ϕ from unary to first-order. This was just an interesting observation.
- Maintained constant scoring function (i.e. $\vec{w} = 0$) with no update, as compared to identical training procedure but with a mistake-driven weight update. On every occurrence, the model with the constant scoring function gave equal to or less than the accuracy of the model with the updating scoring function. This was good news.
- Compared performance of my model with that of others. Mine fell short every time. Not good news, but motivating because obtaining accuracies above 50% were seen as a possibility.
- Tested nearly every modular function in the code for correct inputs and outputs, went to the whiteboard to model how each function should work, cross-validated with my Python implementation, and could find nothing. Still, I'm convinced the error is small and subtle.

(f) Nothing here. The time was simply not there to experiment with other forms of inference. I would certainly like to see the improved performance with other methods, and possibly even explore the possibility of a deterministic process — in fact, a deterministic inference method would be amazing, and would make debugging the model a much more peaceful process.