CS 287
Assignment 5: Sequences and Entities

Due: Friday, April 15th, 11:59 pm

Finding and labeling named-entities in text is often the first step for full-scale information extraction tasks. Furthermore many interesting and practical NLP tasks can often be framed in this paradigm of identifying and labeling contiguous segments of text.

While there are many ways to frame this entity recognition problem, in this problem set we will focus on using sequence (Markov) models for this task. These models differ from the previous models that we have seen in the homework because instead of just making single predictions they require a search step to find the best sequence, i.e. maximizing the sequential scoring function $f(\boldsymbol{x}, c_{1:n})$. In order to solve this exactly, you will have to run dynamic programming over the sequence.

As you complete this assignment, we ask that you submit your results on the test data to the Kaggle competition website at `https://inclass.kaggle.com/c/cs287-hw5` and that you compile your experiences in a write-up based on the template at `https://github.com/cs287/hw_template`.

1 Data and Preprocessing

1.1 Data

The data for this problem set is in the `data/` directory. It consists of a subset of the CoNLL 2003 shared task for named-entity recognition. (The full specification is available here `http://www.cnts.ua.ac.be/conll2003/ner/`).

```
> head  data/train.num.txt
1       1       EU      I-ORG
2       2       rejects O
3       3       German  I-MISC
4       4       call    O
5       5       to      O
6       6       boycott O
7       7       British I-MISC
8       8       lamb    O
9       9       .       O
```

```
10      1       Peter   I-PER
11      2       Blackburn       I-PER

12      1       BRUSSELS        I-LOC
13      2       1996-08-22      O

14      1       The     O
15      2       European        I-ORG
16      3       Commission      I-ORG
17      4       said    O
```

For continuity, the data is in the same format as in HW 2. Each line of the file contains one tokenized word. The columns represent:

1. the global id of the word
2. the sentence id of the word
3. the tokenized word form
4. the BIO tag of the word

Sentences are separated by blank rows. Sentence boundaries are a classic source of errors, so you will have to handle these carefully. In addition we also include a tag dictionary file.

```
> cat data/tags.dict
O 1
I-PER 2
I-LOC 3
I-ORG 4
I-MISC 5
B-MISC 6
B-LOC 7
```

This is a mapping from each BIO tag in the CoNLL data set to a unique ID. It is important that you use this mapping since it is used for the Kaggle competition data. Note that B-ORG and B-PER are not seen in training. You might actually find that you code is faster if you ignore the B-tags altogether, as they are quite rare in NER.

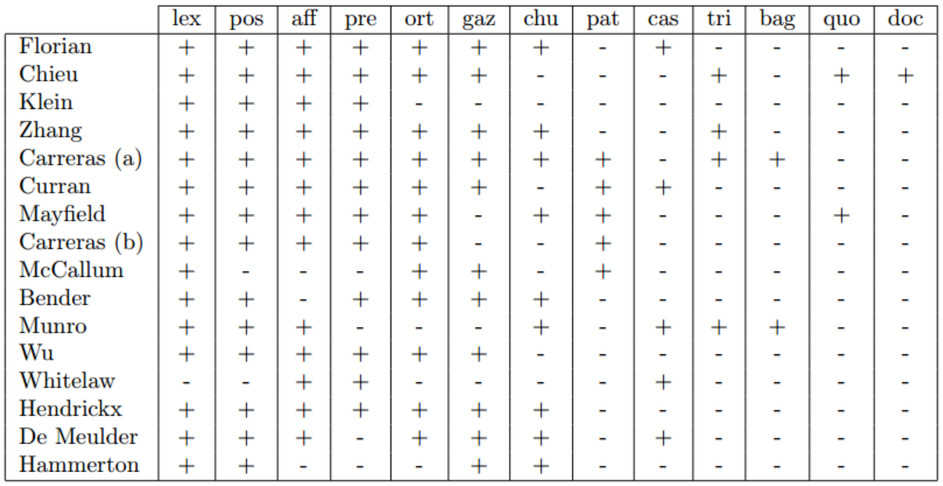
1.2 Preprocessing

For this assignment you will write your own preprocessing code in `preprocessing.py`. This code should transform the data representations given above into a format for multi-class classification. In particular you will,

- Clean the input data

- Pre-construct the features that are fed into the network
- Write out a vector of the pretrained embeddings for each word (for neural MEMM model)

For the sake of fairness, we recommend looking at the CONLL2003 shared task report to see which features might be useful Tjong Kim Sang and De Meulder (2003). We have included the key figure describing the features used by the teams.



	lex	pos	aff	pre	ort	gaz	chu	pat	cas	tri	bag	quo	doc
Florian	+	+	+	+	+	+	+	-	+	-	-	-	-
Chieu	+	+	+	+	+	+	-	-	-	+	-	+	+
Klein	+	+	+	+	-	-	-	-	-	-	-	-	-
Zhang	+	+	+	+	+	+	+	-	-	+	-	-	-
Carreras (a)	+	+	+	+	+	+	+	+	-	+	+	-	-
Curran	+	+	+	+	+	+	-	+	+	-	-	-	-
Mayfield	+	+	+	+	+	-	+	+	-	-	-	+	-
Carreras (b)	+	+	+	+	+	-	-	+	-	-	-	-	-
McCallum	+	-	-	-	+	+	-	+	-	-	-	-	-
Bender	+	+	-	+	+	+	+	-	-	-	-	-	-
Munro	+	+	+	-	-	-	+	-	+	+	+	-	-
Wu	+	+	+	+	+	+	-	-	-	-	-	-	-
Whitelaw	-	-	+	+	-	-	-	-	+	-	-	-	-
Hendrickx	+	+	+	+	+	+	+	-	-	-	-	-	-
De Meulder	+	+	+	-	+	+	+	-	+	-	-	-	-
Hammerton	+	+	-	-	-	+	+	-	-	-	-	-	-

Table 3: Main features used by the the sixteen systems that participated in the CoNLL-2003 shared task sorted by performance on the English test data. Aff: affix information (n-grams); bag: bag of words; cas: global case information; chu: chunk tags; doc: global document information; gaz: gazetteers; lex: lexical features; ort: orthographic information; pat: orthographic patterns (like Aa0); pos: part-of-speech tags; pre: previously predicted NE tags; quo: flag signing that the word is between quotes; tri: trigger words.

We would recommend starting with the most-common features and then possibly adding more if you have time. Some of them are more difficult to implement and/or less interesting. For instance "gaz" means a white-list of entities of that type (for instance countries of the world).

2 Code Setup

Write your main code in `HW5.lua`. For this assignment part, you can (and should) use the `nn` library in addition to the standard Torch library. You should not need to use any extra libraries.

2.1 Prediction and Evaluation

Despite the similarity of the input, the Kaggle competition setup is actually different than HW2. Instead of simply computing the accuracy of each of the predictions, we will instead follow the common definition of the task and use F-score as the metric. Formally define the F_β as

$$\frac{(\beta^2 + 1) \times prec \times recall}{\beta^2 \times prec + recall}$$

Where precision and recall are defined in the standard way with precision being the number of entities produced by the system being correct and recall is the number of true entities that are found by the system. A prediction is only considered correct if it gets the type and the span exactly right.[1] Here is an example of the correct sample dev output:

```
ID,Labels
ID,Labels
1,ORG-3
2,LOC-1
3,MISC-1-2 PER-4-5 ORG-13 ORG-15
4,ORG-14 ORG-16 ORG-18 ORG-25 ORG-36
5,ORG-3 LOC-12-13 ORG-15 LOC-30 PER-32-33
6,ORG-5 PER-15
7,ORG-1 PER-13-14 PER-16-17 ORG-27 LOC-29
```

Note that you will have to parse out the "I" and "B" tags to get the correct spans.

2.2 Hyperparameters

Several of the models described have explicit hyperparameters that you will need to tune. It is your responsibility to cleanly separate these out from the models themselves and expose as command-line options. This makes it much easier to run experiments and to utilize experimental scripts.

2.3 Logging and Reporting

As part of the write-up, you will need to report on the training and predictive accuracy of your models. To make this possible, your code should report on various metrics of the model both at training and test time. We will leave it up to you on which metrics to log, but we recommend reporting training speed, training set NLL, training set predictive accuracy, and validation predictive accuracy. It is your responsibility to convince us that the model is correctly training.

3 Models

For this assignment you will implement at least three models. We will warm up with a hidden Markov model and then implement an maximum-entropy Markov model (MEMM). Finally you will implement the structured perceptron. Finally you can extend the MEMM and CRF with non-linear elements. Note that this is in keeping with the competition itself. Tjong Kim Sang and De Meulder (2003) note that:

> The most frequently applied technique in the CoNLL-2003 shared task is the Maximum Entropy Model. Five systems used this statistical learning method. Three systems used Maximum Entropy Models in isolation (Bender et al., 2003; Chieu and Ng, 2003; Curran and Clark, 2003). Two more systems used them in combination with other

[1]Unfortunately Kaggle cannot yet reproduce the exact global metric used by the competition, so we utilize the *mean* F-Score `https://www.kaggle.com/wiki/MeanFScore` with $F_{\beta=1}$ metric

techniques (Florian et al., 2003; Klein et al., 2003). Maximum Entropy Models seem to be a good choice for this kind of task: the top three results for English and the top two results for German were obtained by participants who employed them in one way or another.

Hidden Markov Models were employed by four of the systems that took part in the shared task (Florian et al., 2003; Klein et al., 2003; Mayfield et al., 2003; Whitelaw and Patrick, 2003). However, they were always used in combination with other learning techniques. Klein et al. (2003) also applied the related Conditional Markov Models for combining classifiers.

3.1 Hidden Markov Model

We begin by implementing a standard hidden Markov model for this problem, which will play the role of our efficient count-based model (analogous to the role naive Bayes and ngram models have played in previous assignments).

For this model, you should estimate the transition and emission distributions directly from the tags and words passed in. The transition distribution is simply $p(\boldsymbol{y}_i|\boldsymbol{y}_{i-1})$ and the emission distribution is $p(\boldsymbol{x}_i|\boldsymbol{y}_i)$. Both of these are simply multinomial distributions and all the same ideas about smoothing can be applied here, e.g. Laplace smoothing for word generation.

Note: we have been purposefully informal in class about the initial and final distribution of the model. There are several different ways to handle this in practice. One simple idea we would recommend is to have pseudo-words <s> and < /s> to start and end the sentence with corresponding labels <t> and < /t> . You can then learn the transition to and from these tags. Remark in your report how you handle boundaries.

Once you have code to train your HMM, implement the Viterbi algorithm with backpointers for finding the highest-scoring sequence. Use the algorithm to find the best named-entities in the dev and test data set. We recommend implementing the Viterbi algorithm for a general Markov model (as described in the lecture notes), so that you can apply it to the other models in this problem set.

Extension: Implement a feature HMM as described in the lecture slides. Instead of having the tag emit generate the word itself, have it emit features associated with that word. In practice this looks like the naive Bayes model from HW2 combined with a transition distribution.

3.2 Maximum-Entropy Markov Model

Next we will implement Maximum-Entropy Markov Models. That is instead of multinomial transition and emission distributions, you will directly estimate $\hat{\boldsymbol{y}}(c_{i-1})$ as a function of the previous tags c_{i-1} and any aspects of the input (typically mainly word w_i but any features are allowed).

Recall that the term maximum-entropy here is synonymous with "multi-class logistic regression". Therefore really what you are doing here is preprocessing the input in such a form that you can train a multi-class logistic regression classifier with additional features of the previous tag. You can implement this classifier just using the `nn` library with mini-batch SGD. And in fact you may find it useful to start with your classifier from HW 2.

Once you have constructed the classifier, you can use it within your Viterbi algorithm from the above. You can use your model to compute each $\hat{\boldsymbol{y}}(c_{i-1})$ that you need for the function f, or you mind find it easier to just precompute all $\hat{\boldsymbol{y}}$ for the sentence. Either way, the core algorithm should remain unchanged.

Extension: Implement a neural network MEMM as described in the lecture slides. Instead of using a linear model, replace the network with a neural network model, using the GLOVE word embeddings that we have included in the assignment. This just changes the score function for constructing $\hat{\boldsymbol{y}}$, the rest of the model remains the same.

3.3 Structured Perceptron

For the final model implement the structured perceptron training algorithm as described in class. Note that this algorithm is quite different than the others described so far. Instead of training the model as a multi-class classifier, you will need to run search during training. You can then compute the gradients of the model using the argmax output sequence and the gold output sequence.

We recommend implementing this algorithm on top of a similar architecture as the MEMM. First start with all weights zeroed. Then for each example.

1. Use `:forward` on your position specific features to compute the scores $\hat{\boldsymbol{y}}$ (without the softmax)
2. Use Viterbi and these values to find the highest-scoring sequence.
3. Compare this sequence to the gold sequence and find the timesteps where they differ.
4. For each of these timesteps:
   - Call `:forward` to recompute each of the $\hat{\boldsymbol{y}}$
   - Manually construct a gradient with -1 on the true output c_i for $\hat{\boldsymbol{y}}(c_{i-1})$ and 1 on the predicted output c_i^* for $\hat{\boldsymbol{y}}(c_{i-1}^*)$
   - Call `:backward`
5. Update the weights with learning rate 1.

Structured perceptron will likely be slower to train then the previous models. However if you implement your Viterbi algorithm efficiently, the overhead can be quite small. In practice, computing scores with sparse features usually dominates the computational overhead.

Extension: Instead of using the algorithm as described you can improve performance by instead using the average of the weights of the structured perceptron through training (this has an effect similar to L2 regularization). We pose it as an exercise of how to efficiently keep this running average. In particular how to do this without touching all the parameters each update.

3.4 Additional Experiments

Once these models are constructed, you should also report on additional experiments on these data sets. One natural extension would be to implement the forward-backward algorithm. Various ideas you could try:

- Describe and compare the marginals produced by each of the models.

- Implement posterior decoding. Does this produce better outputs? Does it allow you to better calibrate f-score?
- Implement the marginal-based pruning method described in class. Quantify the difference
- Run CRF++ on this data.
- Implement beam search. Is it faster? How does accuracy compare?

4 Report and Submission

For your write-up, follow the report template at `https://github.com/cs287/hw_template`. Be sure to include a link to your code, Kaggle ID, and reports on your results.

In addition to submitting your Kaggle results, we also expect you to report on your experimental process. This should include data tables, graphs and discussion of any issues that you may run into.

References

Tjong Kim Sang, E. F. and De Meulder, F. (2003). Introduction to the conll-2003 shared task: Language-independent named entity recognition. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 142–147. Association for Computational Linguistics.