Assignment 3
NLP 202: Natural Language Processing II

University of California Santa Cruz

This assignment is to be done in Python 3.

Your final writeup for this assignment, including the problem and the report of the experimental findings of the programming part, should be no more than six pages long. Most of your grade will depend on the quality of the report. You should submit it as a PDF. We *strongly* recommend typesetting your scientific writing using LaTeX. Some free tools that might help: Overleaf (online), TexLive (cross-platform), MacTex (Mac), and TexStudio (Windows).

1 Overview

The purpose of this homework assignment is to gain familiarity with linear and neural models for a structured prediction task. You will implement a named entity recognition (NER) system and evaluate it on the CoNLL 2003 shared task [7]. Your task is to implement a BIO tagger for four kinds of entities: people, locations, organizations, and names of other miscellaneous entities.

1.1 Data format

The data format is four columns separated by a space. Each token in the sentence has been put on a separate line, and the sentences are separated by a blank line. The first column is the token, and second is the part-of-speech tag, the third is a syntactic chuck tag (which we will not use), and the fourth is the named entity (NE) tag. The NE tags use a modified BIO tagging scheme, where 'O' denotes outside, 'I' denotes inside or begin, and 'B' is only used for the start of a new NE in the case of two identically tagged NEs in a row. Here is an example sentence:

```
France NNP I-NP I-LOC
and CC I-NP O
Britain NNP I-NP I-LOC
backed VBD I-VP O
Franz NNP I-NP I-PER
Fischler NNP I-NP I-PER
's POS B-NP O
proposal NN I-NP O
.  .  O O
```

1.2 Tagging

Your tagger should take as input a file in this format, ignore the last column (don't cheat!), and produce a new file with a fifth column containing the tags produced by your tagger.

2 Part 1: Linear Model with Perceptron Algorithm

In this part, you will implement a tagger with a linear model trained with the Perceptron algorithm.

2.1 Features

We will use the following real-valued features in the model. All of these are conjoined with the current tag. (In our feature names + represents conjunction.) An example feature name and value is given for the word `France` in the above sentence. Remember that although we show features for the tag sequence in the example, your Viterbi decoder will need to consider all tags for each word. Be sure to include start and stop symbols in $\boldsymbol{x}$ and $\boldsymbol{y}$, so $x_0 = \langle \text{START} \rangle$ and $x_{n+1} = \langle \text{STOP} \rangle$, and similarly for $\boldsymbol{y}$.

1. Current word w_i. Example:
   `Wi=France+Ti=I-LOC 1.0`
   We always put the ouput label (`Ti=LABEL`) at the end of feature names.
2. Previous tag t_{i-1}. Example:
   `Ti-1=<START>+Ti=I-LOC 1.0`
3. Current POS tag p_i. Example:
   `Pi=NNP+Ti=I-LOC 1.0`
4. Shape of current word s_i. Just replace all letters with `a` or `A` depending on capitalization, and replace digits with `d`. Example:
   `Si=Aaaaaa+Ti=I-LOC 1.0`
5. Length k prefix for the current word, for $k = 1, 2, 3, 4$. Examples:
   `PREi=Fr+Ti=I-LOC 1.0`
   `PREi=Fra+Ti=I-LOC 1.0`
   Our convention is that if the current word is shorter than k for a prefix of length k, it is not duplicated for that prefix. So the word `to` fires the features `PREi=t+Ti=LABEL` and `PREi=to+Ti=LABEL` just once.
6. Is the current word in the gazetteer for the current tag? Example:
   `GAZi=True+Ti=I-LOC 1.0`
   Our convention is that if `Galerie Intermezzo` is in the gazetteer for the tag `LOC`, then this feature fires for both the unigrams `Galerie` and `Intermezzo`.
7. Does the current word start with a capital letter? Example:
   `CAPi=True+Ti=I-LOC 1.0`

2.2 Implementation

Implement the Viterbi algorithm and train with the structured perceptron and structured SVM for a named entity recognition system. We are providing starter code that implements a simple NER system using a linear model with features, but with key pieces missing.

- **Feature Functions**: We have provided some basic features, but you need to implement all the features listed above.
- **Decoder**: You need to implement the Viterbi algorithm, which is given in the lecture slides from last quarter. You can copy and modify the algorithm you implemented last quarter.

2.3 Structured Perceptron Training: SSGD

Now that you have implemented the decoder, perform training with the structured perceptron loss function using stochastic subgradient descent (SSGD) and early stopping for this model. Because the perceptron loss function scales linearly with the weight vector, neither the regularizer nor the step size have a meaningful effect on the perceptron loss for a linear model. For this reason, use stepsize 1 and do not include a regularizer.

Deliverables In the writeup, be sure to fully describe your models and experimental procedure. Provide graphs, tables, charts or other summary evidence to support any claims you make.

1. With features 1-4, report the precision, recall, and F1 of your model on the dev and test sets after training with early stopping on the dev set.
2. Submit the output of the model for both the dev and test sets.
3. With this limited feature set, look at the output on the dev set and compare to the gold labels. What kinds of errors do you see, and why do you think this is occurring? The model is trying to fit the training data as best it can, and sometimes has to make compromises. What patterns in the output tags do you seen that work well for some sentences but poorly for others?
4. With the full feature set, report the precision, recall, and F1 of your model on the dev and test sets after training with early stopping on the dev set.
5. Submit the output of this model for both the dev and test sets.
6. With the full feature set, look at the output on the dev set and compare to the gold labels. What errors have been fixed when compared to the limited feature set? What kinds of errors do you still see, and why do you think this is occurring?
7. With the full feature set, look at the model file. It is in the format of `feature_string feature_weight`. What are the 5 features with the highest weight in the model, and what are the 5 features with the lowest weight? What does this tell you? Give two features with weights that you find interesting or surprising.

Debugging To help debug your code, we suggest training on one sentence at first. The perceptron algorithm should converge to correctly label this sentence. If it doesn't, there may be a bug in the decoder or the training algorithm.

It is also helpful to print out the average value of the loss function at the end of each epoch, averaged over the epoch. It should roughly go down as training proceeds.

2.4 Structured Perceptron Training: Adagrad

Perform training with the structured perceptron loss function using the Adagrad optimizer. For the same reasons as before, use stepsize 1 and do not include a regularizer.

Adagrad is implemented the same as SSGD, but with a modified equation for updating the parameters. The formula for the Adagrad update is as follows. Like SSGD, each gradient update is called a time step. Let t count the number of gradient updates that have been performed (and does not reset after each epoch.) Let $g_{t,i}$ be the ith component of the gradient at time step t. We keep a running total of the sum of squares of all the components of all the previous gradients: $s_{t,i} = \sum_{\tau=1}^{t} g_{\tau,i}^2$. This includes gradients from all previous epochs. At time step t, the ith parameter $\theta_{t,i}$ is updated like so:

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{s_{t,i}}} g_{t,i}$$

Deliverables In the writeup, be sure to fully describe your models and experimental procedure. Provide graphs, tables, charts or other summary evidence to support any claims you make.

1. Report the precision, recall, and F1 of your full feature set model on the dev and test sets after training with early stopping on the dev set.
2. Submit the output of your model for both the dev and test sets.

3 Part 2: Neural Model with CRF

3.1 Task

The purpose of this part of the homework is to do sequence labeling using a neural model. You will need to implement a Bi-LSTM CRF (and its variants) in PyTorch (section 3.2) and train it to predict the best label sequence using Viterbi decoding. Report your results in each case.

3.2 Implementation

1. **BiLSTM-CRF**: Implement a BiLSTM CRF with minibatch training and train it on the training data. See Appendix A for help with implementing minibatching. You may train on a subset of the data if it is too slow to train on the whole data. The loss function to be optimized is log loss:

$$-\text{score}_\theta(x_i, y_i) + log \sum_{y' \in Y} e^{\text{score}_\theta(x_i, y')} \tag{1}$$

   Use the forward algorithm to compute the partition function and the derivative of the log likelihood with automatic differentiation.
   You may refer to and use code from the tutorial here as starter code: https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html
2. Write a Viterbi decoder and run your model on the dev and test data. You can reuse the Viterbi decoder from the previous assignments(s) that you have implemented or code from the tutorial.
3. **Character CNN with max pooling layer**: In your BiLSTM-CRF, add a character-level CNN embedding layer with maxpooling for each token. Include these character-level embeddings in addition to traditional the word embeddings. [6] (Refer to the slides from the lecture on CRFs from NLP 201 about character-level CNNs.) These references are just suggestions: you are free to include these embeddings in any manner you see fit.
4. (Extra credit) Modify your BiLSTM-CRF model to use a different loss function taking a cost function into account. You can optimize the loss function based on ***one*** of the following criteria (your choice – note the minus signs in front of first term in these equations):

   - Max margin or SVM loss

$$-\text{score}_\theta(x_i, y_i) + \max_{y' \in Y} \left(\text{score}_\theta(x_i, y') + \text{cost}(y_i, y')\right) \tag{2}$$

- Softmax margin [3]

$$-\operatorname{score}_\theta(x_i, y_i) + log \sum_{y' \in Y} e^{(\operatorname{score}_\theta(x_i, y') + \operatorname{cost}(y_i, y'))} \tag{3}$$

- Ramp loss [2]

$$-\max_{y \in Y} \operatorname{score}_\theta(x_i, y) + \max_{y' \in Y} \big(\operatorname{score}_\theta(x_i, y') + \operatorname{cost}(y_i, y')\big) \tag{4}$$

- Soft ramp loss [4]

$$-\log \sum_{y' \in Y} e^{\operatorname{score}_\theta(x_i, y)} + \log \sum_{y' \in Y} e^{(\operatorname{score}_\theta(x_i, y') + \operatorname{cost}(y_i, y'))} \tag{5}$$

Use the hamming loss as the cost function.

$$cost_{hamming}(\hat{Y}, Y) = \sum_{j=1}^{|Y|} \delta(\hat{y}_j \neq y_j) \tag{6}$$

Multiply the cost function by a constant factor as a hyperparameter which you are free to choose.

Deliverables In the writeup, be sure to fully describe your models and experimental procedure. Provide graphs, tables, charts or other summary evidence to support any claims you make.

1. Report the precision, recall, and F1 of the best model of each kind (from section 3.2) on the dev and test sets after training. You may report results from a model trained on a subset of the training data. Try to use at least a 1000 sentences. Make sure to report all of your assumptions in the report, including hyperparameter values.
2. To demonstrate that you have implemented minibatching correctly, report the training time for one epoch for at least two different batch sizes. You may use a subset of the data.
3. Submit the outputs of the best model for both the dev and test sets, for each kind of model (1, 3, and 4 if you did it).
4. Look at the output on the dev set and compare to the gold labels. What are your observations about the errors your model makes?

4 Submission Instructions

Submit a zip file (`A3.zip`) on Canvas, containing the following:

- **Output**: Submit the output of your code on the dev and test sets if it is asked for in each section.
- **Code**: You code should be implemented in Python 3, and needs to be runnable. Submit your code together with a neatly written README file to instruct how to run your code with different settings. We assume that you always follow good practice of coding (commenting, structuring), and these factors are not central to your grade. However, please provide well commented code if you want partial credit. If you have multiple files, provide a short description in the preamble of each file.
- **Report**: As noted above, your writeup should be six pages long, or less, in PDF (one-inch margins, reasonable font sizes). Include your name at the top of the report. Part of the training we aim to give you in this class includes practice with technical writing. Organize your report as neatly as possible, and articulate your thoughts as clearly as possible. We prefer quality over quantity. Do not flood the report with tangential information such as low-level documentation of your code that belongs in code comments or the README. Similarly, when discussing the experimental results, do not copy and paste the entire system output directly to the report. Instead, create tables and figures to organize the experimental results.

A Appendix: Implementation of Minibatching for CRF

- **Masking the inputs**
  Generally, LSTMs can accept variable length inputs. However, to perform minibatching using DataLoader, all the inputs in a batch need to be of the same length. This is done by a process called **masking**, where an input sequence is *padded* with a padding token to make it of the same length as the maximum length input in the batch.
  When preprocessing the data, we convert the input sequence into a sequence of indices. A sentence after preprocessing may look like this: $[12, 1, 5, 6]$
  If the max sentence length in a batch is 10, and the padding_id is 0, then the padded version of the sentence would be: $[12, 1, 5, 6, 0, 0, 0, 0, 0, 0]$
  Since this is a sequence labeling task, each word in the input sentence is associated with a label. Therefore, we will need to pad both the input sentence and the label sequence during training. Before the input goes into the LSTM, we will need to unpad the sequence. In order to be able to do this, we store the lengths of the sentences before we pad them. The *collate_fn* parameter in DataLoader lets us write a customized method to process the batches. To do the padding, and subsequent unpadding, you may use the *pad_sequence()*, *pack_padded_sequence()* and *pad_packed_sequence()* methods defined in torch.nn.utils.rnn

- **Calculating batch loss**
  In the reference code, the loss is computed for each gold and predicted sequence inside the main training loop. In the mini-batched version, we compute the loss over each batch inside the main training loop. Here are some required changes to the reference code to bear in mind:
  - _get_lstm_features(): _get_lstm_features() computes the lstm features. However, now, we have padded the inputs, therefore, we unpack the sequence before feeding it into the lstm, and then pack the output of the lstm. Use *pack_padded_sequence()* and *pad_packed_sequence()*. Note that the lstm_features tensor is now batch_size x maxSeqLen x tagset_size, different from sentence_length x tagset_size, like in the reference code.
  - forward_alg(), _viterbi_decode(), _score_sentence(): forward_alg() now compute the scores over each batch, instead of each sentence. The scores for all sentences in a batch are added to get the total score for a batch. The padded values do not contribute to the scores, therefore a masked version of the input is required to do these computations. Similarly, _score_sentence() gives the score of a provided tag sequence for a batch of sentences.

To handle and manipulate your tensors, you may need to use some of the following tensor operations/methods. Further documentation at `https://pytorch.org/docs/stable/tensors.html`

- **view() and reshape()**
  tensor.view() or tensor.reshape() returns a new tensor with the same data and number of elements as the old one, but a different shape specified in the parameters. Refer to the examples here. -1 is used as a parameter to derive the length of one of the dimensions based on the other dimensions and the number of elements, and only one dimension may be specified as -1.
- **squeeze(), unsqueeze()**
  squeeze() and unsqueeze() are reverse operations of each other. tensor.squeeze(input) returns a tensor with all the dimensions of input of size 1 removed. tensor.unsqueeze(input, dim) returns a new tensor with a dimension of size 1 inserted at the specified position (dim). Documentation for squeeze and unsqueeze

- **contiguous()**
  tensor.contiguous() makes a copy of tensor so the order of the elements would be the same as if a tensor of the same shape was created from scratch.
- **transpose()**
  transpose(input, dim0, dim1) returns a tensor that is a transposed version of input. The dimensions dim0 and dim1 are swapped.
- **permute()**
  tensor.permute(*dims) rearranges the original tensor according to the desired ordering (*dims) and returns a new multidimensional rotated tensor
- **flatten()**
  flatten(input, start_dim=0, end_dim=-1) flattens a contiguous range of dims in input. Examples here
- **expand()**
  tensor.expand() returns a new view of the self tensor with singleton dimensions expanded to a larger size. Passing -1 as the size for a dimension does not change the size of that dimension.
- **scatter_() and gather()**
  scatter_() and gather() are reverse operations of each other. gather(input, dim, index) gathers values of indices along an axis specified by dim. scatter_(dim, index, src) writes all values from the tensor src into self at the indices specified in the index tensor.
- **masked_select()**
  masked_select(input, mask) returns a new 1-D tensor which indexes the input tensor according to the boolean mask (which is a BoolTensor).
- **masked_scatter()**
  tensor.masked_scatter_(mask, source) copies elements from source into tensor at positions where the mask is one.
- **eq(), ge()**
  tensor.eq(other) or tensor.ge(other) computes element-wise equality or $\geq$ for tensor and other, where other can be a number or a tensor whose shape is broadcastable with the first argument..

References

[1] *Proceedings of the International Joint Workshop on Natural Language Processing in Biomedicine and its Applications (NLPBA/BioNLP)*, Geneva, Switzerland, August 28th and 29th 2004. COLING. URL https://www.aclweb.org/anthology/W04-1200.

[2] Olivier Chapelle, Chuong B Do, Quoc V Le, Alexander J Smola, Choon Hui Teo, et al. Tighter bounds for structured estimation. In *Proc. of NIPS*, 2008.

[3] Kevin Gimpel and Noah A Smith. Softmax-margin crfs: Training log-linear models with cost functions. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 733–736, 2010.

[4] Kevin Gimpel and Noah A Smith. Structured ramp loss minimization for machine translation. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 221–231, 2012.

[5] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[6] Xuezhe Ma and Eduard Hovy. End-to-end sequence labeling via bi-directional lstm-cnns-crf. *arXiv preprint arXiv:1603.01354*, 2016.

[7] Erik F. Tjong Kim Sang and Fien De Meulder. 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*, pages 142–147, 2003. URL https://www.aclweb.org/anthology/W03-0419.