

Natural Language Processing

Tel Aviv University

Assignment 3: Tagging

Due Date: *December 31, 2020*

Lecturer: Jonathan Berant

In this home assignment we will implement different NER (Named Entity Recognition) taggers. You are provided with the data and supporting code.

For your convenience, coding sections are marked with ♣ and deliverables (**in addition to the code and the written solution**) are marked with ■.

0 Preliminaries

Submission Instructions The environment setup and submission tutorial can be found in the following notebook: https://colab.research.google.com/drive/1fnqF0n056CT_-AL6wm0MnCU1L5luCWpo?usp=sharing

1 ta preprocessing


- (a) ♣ As we saw in class, a common solution to the rare words problem is to pre-process the data and replace rare words with a category, or signature (e.g., numbers, dates, capitalization, prefixes, suffixes, etc. ...). Come up with good word categories and implement `replace_word` in `data.py`. The following paper can be helpful: <http://people.csail.mit.edu/mcollins/6864/slides/bikel.pdf>. You can test the efficacy of your implementation by evaluating your “most frequent tag” baseline (next problem).

2 Most frequent tag baseline


- (a) ♣ The most frequent tag baseline tags each word with its most frequent tag, as seen in the training set. Implement the training for the most frequent tag baseline in `most_frequent.py`.
- (b) ♣ Implement the prediction procedure for the most frequent tag baseline in `most_frequent.py`. Evaluate your tagger against the development set. The evaluation includes a confusion matrix and scores of precision, recall and F_1 (which is the harmonic mean of precision and recall - see https://en.wikipedia.org/wiki/F1_score). What is your F_1 score on the development set?


3 HMM tagger


- (a) ♣ **MLE estimators:** Use the training data to estimate the transition probabilities q and emission probabilities e . Fill the implementation of the training algorithm in the function `hmm_train` in `hmm.py`.

- (b)  **Viterbi:** Implement the Viterbi algorithm in `hmm_viterbi` function in `hmm.py`. The algorithm receives a sentence to tag as input, the counts computed by the training procedure and the hyper-parameters λ_i . The algorithm returns the highest probability sequence of tags according to q and e . Recall that the estimates for q should be based on a weighted linear interpolation of $p(t_i|t_{i-1}, t_{i-2})$, $p(t_i|t_{i-1})$ and $p(t_i)$. Tune the hyper-parameters on the development set, and document the optimal λ_i values in your written solution.

Note: a straight-forward implementation of the Viterbi algorithm can be inefficient, so you should add **beam tag pruning** (eliminating some tags for specific words). Training and evaluation (on dev set) should take less than 20 seconds. Document your pruning policy in your written solution.




- (c)  Implement the prediction procedure of the HMM tagger with Viterbi algorithm in `hmm_eval` in `hmm.py`. **What is your F₁ score on the development set?**

- (d)  **Forward-backward:** In `forward.py`, we define a scoring function $\psi(\hat{y}, y, j) = P(\hat{y} | y, j)$ where y, \hat{y} are NER tag. Using a variant of the forward-backward algorithm shown in class (slide 31 in class 8 slides) calculate the marginal probability for $P(y = \text{LOC} | j = 8, |x| = 14)$.

- (e)  **Theoretical question 1:** Give an example for transition parameters Q , emission parameters E and a sentence s , such that the greedy inference algorithm does not result in the highest probability sequence. Your solution should include the probabilities calculations, as well as the samples on which the parameters were estimated.


4 Maximum Entropy Markov Model (MEMM) tagger

In this part you will implement the MEMM tagger (a locally-normalized log-linear model). The learning part is already given in the skeleton code using `scikit-learn`.

- (a)  **Feature engineering:** Implement features for your model. You should implement the features from **Chaparkhi (1996) mentioned in class**, but you can add more feature templates if you want.
- (b)  **Greedy inference:** Implement a greedy inference algorithm, where you tag a sentence from left to right with your trained model. Fill your implementation in the function `memm_greedy` in `memm.py`.
- (c)  **Viterbi:** Implement the Viterbi algorithm for MEMMs. The tag distribution should be inferred from the trained model. Fill your implementation in the function `memm_viterbi` in `memm.py`.

Note 1: prediction in this model is likely to be much slower than in the HMM model. You should consider optimizing your implementation by caching predictions, avoiding unnecessary feature extraction, etc. **Training and evaluation (on dev set) should take less than 4 minutes.** Document any optimization you have performed in the written solution. Naive implementation will not receive the full score.

Note 2: As mentioned before, you should not change the code, besides in the marked sections. You can use the method `build_extra_decoding_arguments` to pass any additional argument to your decoding procedure.

- (d)  Implement the prediction procedure of the MEMM tagger with Viterbi and greedy inference in `memm_eval` in `memm.py`. What are your F₁ scores on the development set?

- (e) Sample errors from your best model and analyze them. What are common failure cases for your model. Where does it struggle? Summarize the results of your analysis in the written solution.

5 BiLSTM tagger

We are going to implement an NER tagging model that uses BiLSTM¹ (see Figure 1) and predicts a label for each token independently using contextualized representations. The model uses cross-entropy loss per token, and has the following form:

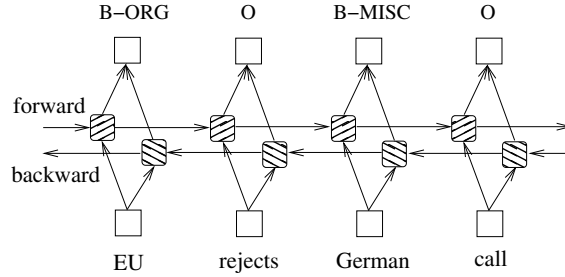


Figure 1: A bidirectional LSTM network.

$$\begin{aligned}
 \mathbf{e}^{(t)} &= [\mathbf{x}^{(t-1)}E, \mathbf{x}^{(t)}E, \mathbf{x}^{(t+1)}E] \\
 \mathbf{h}^{(t)} &= \text{BiLSTM}(\mathbf{e}^{(t)}) \\
 \hat{\mathbf{y}}^{(t)} &= \text{softmax}(\mathbf{h}^{(t)}W + \mathbf{b}) \\
 J &= \text{CE}(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) \\
 \text{CE}(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) &= - \sum_i y_i^{(t)} \log \hat{y}_i^{(t)}
 \end{aligned}$$

where $E \in \mathbb{R}^{V \times D}$ is the embeddings matrix, $\mathbf{e}^{(t)}$ is the word representation (notice that we also concatenate embeddings of the neighboring words, as it improves the model performance), $\mathbf{h}^{(t)}$ is dimension H (the LSTM output dimension of each direction is of size $\frac{H}{2}$) and $\hat{\mathbf{y}}^{(t)}$ is of dimension C , where V is the size of the vocabulary, D is the size of a word embedding, H is the size of the hidden layer and C are the number of classes being predicted.

The predicted label is outputted with $\arg \max$ independently per token (no fancy decoding):

$$\text{label}^{(t)} = \arg \max \hat{\mathbf{y}}^{(t)}$$

- (a) Implementing an RNN (in our case, LSTM) requires us to unroll the computation over the whole sentence. Unfortunately, each sentence can be of arbitrary length and this would cause the RNN to be unrolled a different number of times for different sentences, making it impossible to batch process the data.

The most common way to address this problem is to pad our input with zeros. Suppose the longest sentence in our input is M tokens long, then, for an input of length T we will need to:

- Add “0-vectors” to \mathbf{x} and \mathbf{y} to make them M tokens long. These “0-vectors” are still one-hot vectors, representing a new NULL token.

¹BiLSTM (bidirectional LSTM) is simply the concatenation of the outputs of an LSTM reading the input forward and an LSTM reading the input backward.

- Create a masking vector, $(m^{(t)})_{t=1}^M$ which is 1 for all $t \leq T$ and 0 for all $t > T$. This masking vector will allow us to ignore the predictions that the network makes on the padded input.
- Of course, by extending the input and output by $M - T$ tokens, we might change our loss and hence gradient updates. In order to tackle this problem, we modify our loss using the masking vector:

$$J = \sum_{t=1}^M m^{(t)} \text{CE}(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)})$$

- i How would the loss and gradient updates change if we did not use masking? How does masking solve this problem?
 - ii ♣ Implement pad sequences in your code. You can test your implementation by running `python q5/ner_bilstm_model.py test_padding`.
- (b) ♣ Implement the rest of the code for data preprocessing, the model and its training, by appropriately completing functions in the `DataPreprocessor`, `NerBiLstmModel` and `Trainer` classes (search for TODOs in `q5/ner_bilstm_model.py`, use the hints!). Assume only fixed length input. Before continuing to the full training, you can check that your code is not crashing by running `python q5/ner_bilstm_model.py test_training`.
- (c) Train your model using the command `python q5/ner_bilstm_model.py train`. You should get a development F_1 score of at least 88%. Training should take a few minutes with the Google Colab GPU. The model and its output will be saved to `results/<timestamp>/`, where `<timestamp>` is the date and time at which the program was run. The file `results.txt` contains the formatted output of the model's predictions on the development set, and the file `log` contains the printed output, i.e. confusion matrices and F_1 scores computed during the training.



variables: `log` and `predictions.conll` in `q5` folder.

Acknowledgements This question was adapted from Stanford's CS224n course. Their contributions are greatly appreciated.