Training part-of-speech tagger using Log-Linear model

November 2018

1 Problem formulation

Given input in from of $\{\{(w_j, t_j)\}_{j=1}^{n_i}\}_{i=1}^{N}$ where w_j, t_j is jth word-tag pair in sentence i, we want to find $\theta \in \mathbb{R}^d$ that models empirically the marginal probability distribution $p(t|h)$ where h is a history (a window within a sentence). The marginal probability distribution is defined as follows

$$p(t|h;\theta) = \frac{exp(\theta^T f(h,t))}{\sum\limits_{t' \in S} exp(\theta^T f(h,t'))} \tag{1}$$

where $f(h,t) \in \{0,1\}^d$ is a vector of indicator functions indicating whether a feature presents within a given history or not and S is a set of all possible tags. By taking a logarithm we define a log linear function

$$log\big(p(t|h;\theta)\big) = \theta^T f(h,t) - log(\sum_{t' \in S} exp(\theta^T f(h,t'))) \tag{2}$$

By multiplying by -1 and introducing regularization parameter $\lambda > 0$ we have the following loss function to minimize

$$\mathcal{L} = log\big(\sum_{t' \in S} exp(\theta^T f(h,t'))\big) - \theta^T f(h,t) + \frac{\lambda}{2}\theta^T\theta \tag{3}$$

with gradients having following form

$$\frac{\partial \mathcal{L}}{\partial \theta_k} = \sum_{i=1}^{N}\sum_{t' \in S} f_k(h_i,t')p(t'|h_i;\theta) - \sum_{i=1}^{N} f_k(x_i,t_i) + \lambda\theta_k \tag{4}$$

2 Feature Extraction

2.1 Model 1

As a template for features for the first model we used more-or-less the same features as defined in [2] except that we haven't considered rare/not rare words as separate cases. Rather we have filtered out all features that appeared less than two times[1].

[1]This is justified since we assuming that the dataset consist of i.i.d samples and it is large, meaning that occurences of features converge to the real distribution.

Additionally, to reduce the number of features we haven't considered prefixes and suffixes of size 4.

w_{i+2}	A word that follows word w_{i+1}
w_{i+1}	Next word
w_i	Current word at position i
w_{i-1}	Previous word
w_{i-2}	The word before word w_{i-1}
t_{i-1}	Previous tag
t_{i-2}, t_{i-1}	Two previous tags
prefix3	Prefix of w_i of size 3 letters
prefix2	Prefix of w_i of size 2 letters
prefix1	Prefix of w_i of size 1 letters
suffix3	Suffix of w_i of size 3 letters
suffix2	Suffix of w_i of size 2 letters
suffix1	Suffix of w_i of size 1 letters
has_hyphen	1 if w_i contains hyphen, 0 otherwise
has_upper	1 if w_i contains uppercase letter, 0 otherwise
has_number	1 if w_i contains number, 0 otherwise

Table 1: Features for model 1.

Such vectorization resulted in 47401 features.

2.2 Model 2

Since train data for model 2 is much smaller, we haven't filtered out rare features and in addition to the first model we have added prefixes and suffixes of size 4. The table 2 shows the whole list of features for model 2. Applying the table resulted in 30111 features.

w_{i+2}	A word that follows word w_{i+1}
w_{i+1}	Next word
w_i	Current word at position i
w_{i-1}	Previous word
w_{i-2}	The word before word w_{i-1}
t_{i-1}	Previous tag
t_{i-2}, t_{i-1}	Two previous tags
prefix4	Prefix of w_i of size 4
prefix3	Prefix of w_i of size 3 letters
prefix2	Prefix of w_i of size 2 letters
prefix1	Prefix of w_i of size 1 letters
suffix4	Suffix of w_i of size 4
suffix3	Suffix of w_i of size 3 letters
suffix2	Suffix of w_i of size 2 letters
suffix1	Suffix of w_i of size 1 letters
has_hyphen	1 if w_i contains hyphen, 0 otherwise
has_upper	1 if w_i contains uppercase letter, 0 otherwise
has_number	1 if w_i contains number, 0 otherwise

Table 2: Features for model 2.

3 Training

We used Limited-memory BFGS[2] algorithm defined in scipy-package[3]. The first model took around 20 hours to train and 40 minutes to train the second one.

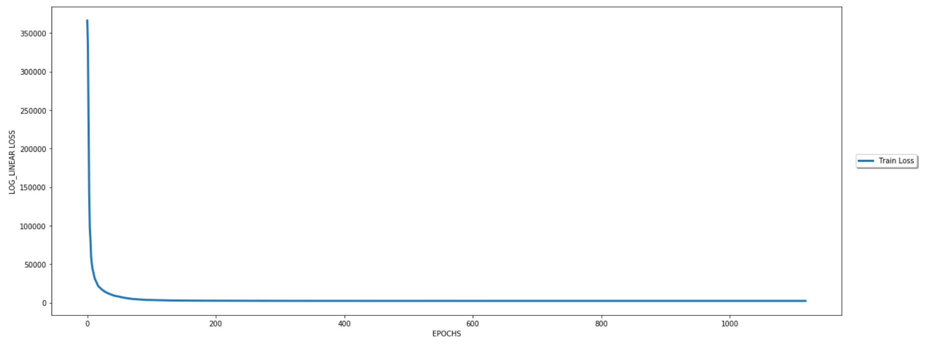


Figure 1: Training loss (3) using BFGS optimization algorithm with $\lambda = 0.005$ for model 1.

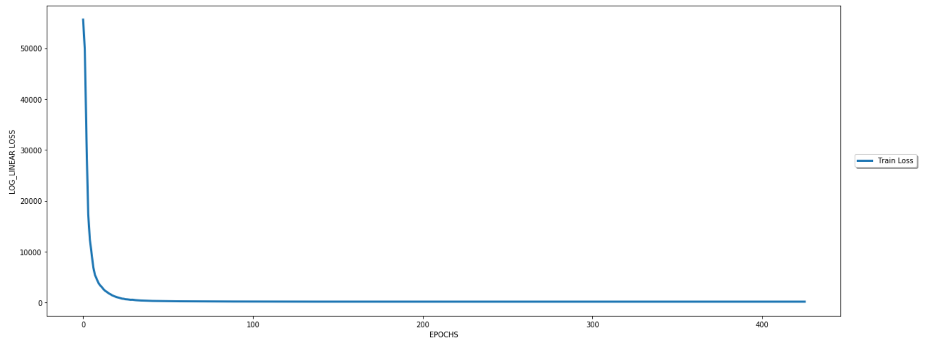


Figure 2: Training loss (3) using BFGS optimization algorithm with $\lambda = 0.005$ for model 2.

4 Inference

Given tokenized sentences as input $\left\{\{w_j\}_{j=1}^{n_i}\right\}_{i=1}^{M}$, for each sentence i, we want to find a sequence of tags $\{t_j\}_{j=1}^{n_i}$, by assuming the probability of form

$$p(\{t_j\}_{j=1}^{n_i}|\{w_j\}_{j=1}^{n_i}) = \prod_{k=1}^{n_i} q(t_k|h_k), \tag{5}$$

[2]`https://en.wikipedia.org/wiki/Limited-memory_BFGS`

[3]`https://docs.scipy.org/doc/scipy-1.1.0/reference/optimize.minimize-lbfgsb.html`

Using Viterbi algorithm, for each $(u, v) \in S_{m-1} \times S_m$ we define a reccurence relation

$$\pi(m, u, v) = max_{t \in S_{m-2}} \Big\{ \pi(m-1, t, u) q(v | h_m(t, u, \{w_j\}_{j=1}^{n_i}, m)) \Big\} \tag{6}$$

where S_m is a set of all possible tags at the position m within a sentence.

It is easy to see that for each element $\pi(m, u, v)$ we calculate probability $q(v|h_m(t, u, \{w_j\}_{j=1}^{n_i}, m)$ which results in $|S_{m-2} \times S_{m-1} \times S_m|$ computations for each word and overall complexity for a sentence with n words results in $\mathcal{O}(|S|^3 n)$ calculations. Although it does significantly improves the brute-force $\mathcal{O}(|S|^n)$ complexity, having a large set of tags (in our case 44) is still computationally expensive.

5 Results

5.1 Model 1

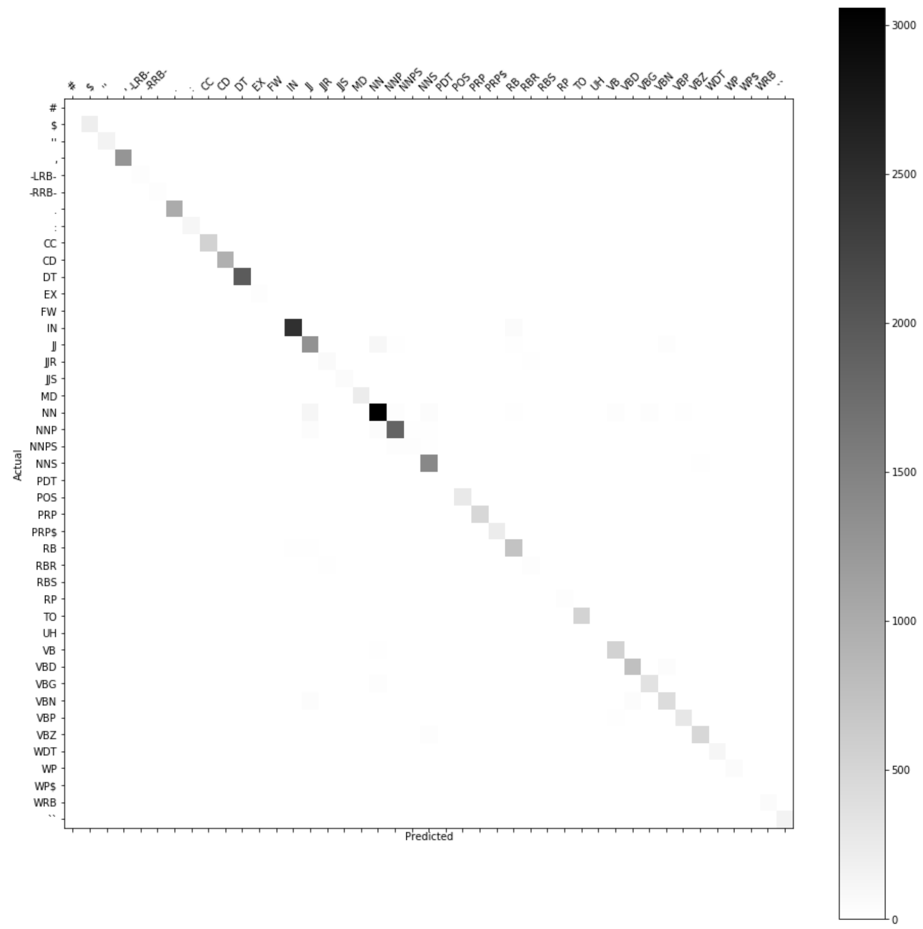
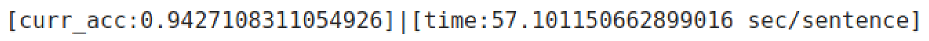


Figure 3: Confusion matrix for model 1.

Figure (3) shows the confusion matrix for model 1. The tags that have the worst performance are: IN, NN, NNP, DT, NNS, CD, ‘,‘, ‘.‘, RB, VBD. The ‘,‘ and ‘.‘ can be improved by adding additional feature indicating whether a word contains punctuation symbols. For other tags, there are multiple ways to improve their predictions. One way would be to understand what causes such performance: Analyze histories for such cases and look for features that cause the model to fail. For instance, it may be due to fact that some prefixes do improve the performance and some prefixes worsens it so one should create a list of blacklisted prefixes that won’t be included as features (or maybe it is better to remove prefixes of some length completely). Another way, similar to [1], is to use some form of ranking to improve the model: Take the output candidates from the first model and use them as input to the second model in an effort to improve the performance. The accuracy for test dataset that we have achieved is 0.943. Figure (4) shows its the screenshot. It also shows that the average time for inference for a single sentence is 57 seconds.



```
[curr_acc:0.9427108311054926]|[time:57.101150662899016 sec/sentence]
```

Figure 4: Screenshot of the accuracy for model 1.

5.2 Model 2

Since the size of the data for model 2 is small, to predict its performance we’ve trained the model 2 ten times each time randomly splitting the dataset by taking out 20% for testing and training with the rest 80%.

simulation	1	2	3	4	5	6	7	8	9	10	mean	err
accuracy	0.979	0.933	0.929	0.945	0.936	0.919	0.939	0.951	0.969	0.921	0.942	0.019

Table 3: Validation simulations for model 2.

Table 3 shows results for 10 simulations, their mean and standard deviation. The generalization error should be somewhere around 0.942. Taking the worst case, we project the accuracy to be $0.942 - 0.019 = 0.923$.

References

[1] Michael Collins. Ranking algorithms for named-entity extraction: Boosting and the voted perceptron. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 489–496. Association for Computational Linguistics, 2002.

[2] Adwait Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In *Conference on Empirical Methods in Natural Language Processing*, 1996.