1

Number of distinct words: 5069

Number of distinct tags: 45

Words with most distinct tags (3 or more):

offered	VBN,VBD,NNP,JJ	closed	VBD,VBN,JJ	west	NNP,JJ,NNPS	back	RB,VB,RP
up	RP,RB,IN,JJ	systems	NNP,NNPS,NNS	increased	VBD,JJ,VBN	recorded	JJ,VBN,VBD
that	IN,WDT,DT,RB	center	JJ,NN,NNP	work	VB,NN,VBP	balloon	NN,NNP,VB
close	RB,JJ,VB,NN	own	JJ,VB,VBP	out	RB,RP,IN	first	JJ,RB,LS
used	VBN,VBD,JJ	all	PDT,RB,DT	half	JJ,NN,PDT	put	VB,VBN,VBD
down	RB,RP,IN	hit	NN,VBN,VB	long	JJ,RB,NNP	file	VBP,VB,NN
feel	VBP,VB,NN	announced	VBD,VBN,JJ	second	JJ,LS,NN	control	NN,VB,NNP
lead	NN,VB,JJ	run	VBP,NN,VB	added	VBD,JJ,VBN	offer	VB,NN,VBP
in	IN,RP,RB	cut	VB,VBD,NN	over	RB,RP,IN	longer	JJR,RBR,RB
move	NN,VB,VBP	held	VBN,JJ,VBD	on	IN,RB,RP	playing	NN,JJ,VBG
off	RP,RB,IN	set	VBN,NN,VBD	general	JJ,NNP,NN	average	NN,JJ,VBP
reported	VBD,VBN,JJ	applied	NNP,VBN,VBD	finance	NN,NNP,VB	required	JJ,VBN,VBD
force	VBP,NN,VB	as	IN,JJ,RB	near	JJ,IN,RB	light	JJ,NNP,NN
french	JJ,NNP,NNPS	trade	NNP,NN,VB	third	NN,JJ,LS	like	IN,VB,JJ
real	JJ,RB,NNP	lower	VB,JJR,NNP	either	DT,CC,RB	one	CD,NN,PRP
industries	NNP,NNS,NNPS	public	NN,JJ,NNP	wonder	VBP,VB,NN	holding	VBG,NN,NNP

2

I implemented the maximum-likelihood estimation model.

The training set is shuffled after each time it is walked through.

For the convenience of later feature definition, I slightly changed the structure of the lattice. Previously, each word in an example corresponds to two columns in the lattice. There were two kinds of edges: (tag-)word-to-tag and tag-to-word. Now each word in an example corresponds to only one column in the lattice. There is only one kind of edge, where each edge corresponds to a triplet (previous tag, tag, word). By doing this, each local feature can be defined in terms of this triplet.

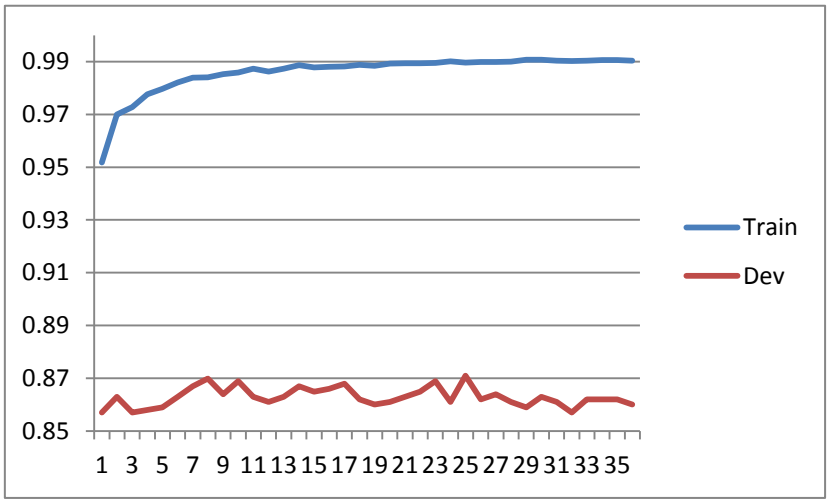
3

Without smoothing or using some prior, generative models compute word probability only by count-and-divide, so unseen words get a probability of 0, and consequently the entire sequence get a 0 probability.

A discriminative model doesn't care about the probability of the word; it only worries about how to tag the word. The model can make this tagging decision by looking at the set of features each word activates. In the worst case where the model has no idea at all, it believes all tags are equally likely.

4

With a learning rate of 0.3 and 30 iterations, the accuracy on `train` and `dev` are as follows:



The accuracy on `train` almost always increases, while after the 25th pass the accuracy on `dev` generally decreases. This means the model is overfitting after the 25th pass and we'd better stop there, where accuracy on `dev` is 0.8639523336643495 and on `train` is 0.9899006486575253.

However, in any sense, there's always a huge gap between the accuracy on `train` and on `dev` (0.99 vs 0.85). This implies that `train` and `dev` are very different from the model's perspective because the model doesn't generalize very well.

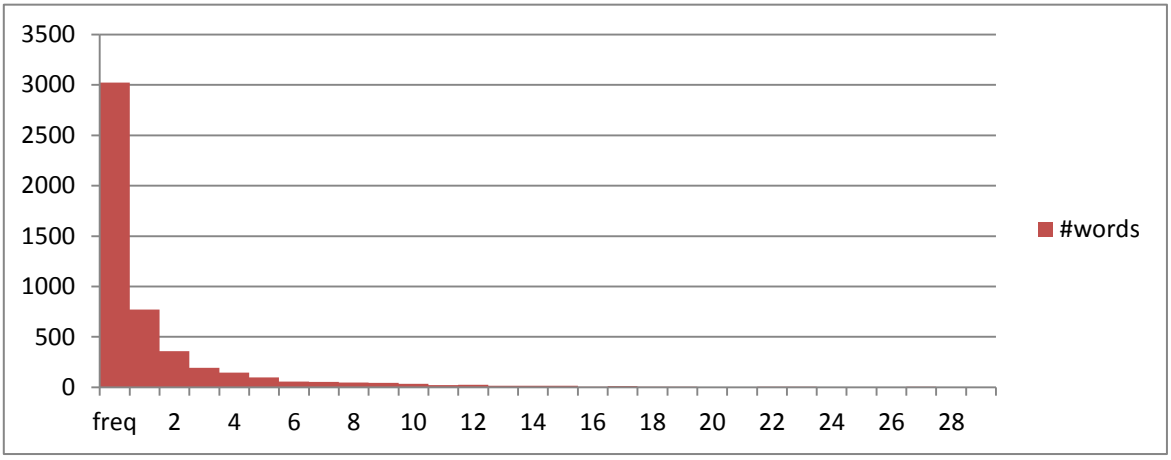
5

We tried Gaussian prior regularization. With a regularization parameter of 0.05, 0.01 or 0.005, the algorithm converges more quickly, but the accuracies on both `dev` and `train` are generally a few percent lower than before. When the parameter set to 0.001, we get lower accuracy on train (0.9768453896050578) but slightly higher accuracy on dev (0.8679245283018868).

Gaussian prior doesn't seem to be more helpful than early stop, and it's harder to tune. We don't use it in 6 and 7.

6-7

The following graph shows the distribution of word frequencies in `train`:



The main reason why the accuracy on `train` is so high is that many words occur only very few times, and they only get one tag. The model is originally implemented as the note suggests: "If a word w has been seen before, you only need to consider those tags for w that have been observed with w in `train.tags`." This means these rare words are always tagged correctly even if we don't train our model at all. But this doesn't help our prediction on `dev`, because the vast majority of these words won't ever appear again.

Even worse, these words won't be helpful for training feature parameters, either. For example, if we have a feature "all characters in word are digits & tag is CD", the corresponding parameter $\lambda_{\{all-digit,CD\}}$ never gets trained, because in training data such words are always labeled "CD" no matter what $\lambda_{\{all-digit,CD\}}$ is.

To solve this problem, for those rare words we could instead consider all the tags, but this would significantly slow down the program and incurs a lot of unnecessary computation. So what we do here is that for rare words that appears less than 5 times, in addition to their own tags, we also consider the most frequent 3 tags (i.e. NN, IN, DT).

Having done this, we define four kinds of features as follows:

$f_T^{(all-digit)} = 1$	word: contains at least one digit and contains no letters tag = T
$f_T^{(1st-cap)} = 1$	word: starts with a capital letter prevTag $\neq$ <s> tag = T
$f_{P,T}^{(prefix)} = 1$	word: P is its longest prefix and length(word) – length(P) >=2 tag = T
$f_{S,T}^{(suffix)} = 1$	word: S is its longest suffix and length(word) – length(S) >=2 tag = T

The list of English prefixes and suffixes are found on Wiktionary[1].

With learning rate 0.3, we stop the training after 11 iterations. We get accuracy 0.9902701371212743 on `train` and 0.9066534260178749 on `dev`.

The accuracy on `test` is 0.8837890625.

[1] Appendix:English prefixes: http://en.wiktionary.org/wiki/Appendix:English_prefixes
Appendix:English suffixes: http://en.wiktionary.org/wiki/Appendix:English_suffixes