Project: Maximum Entropy Tagger
Project report INF5830

Lars Bungum

November 27, 2007

1 Introduction

The framwork for this project is a term project for the course INF5830 at UIO, in which one of the taught NLP technologies were to be tried out. This needs a mention, because in particular the final of my test runs gave a very fine result. Being a student, though, I boldly risk the submission of these results, prior to thorough scrutiny of my implementation, particularly evaluation.

Work with the project was largely divided in two parts, namely creation of proper features for a Maximum Entropy Tagger (henceforth: MaxEnt) and then implementing an algorithm to decode the optimal sequence of tags given an input text. For each literal word there is a part-of-speech (POS) tag, and I will use the first 21 sections of the PBT to extract information about under which conditions each POS-tag is handed out to a previously not seen word (from a section not used in training), facilitated through a MaxEnt approach. Inspiration was drawn from the curriculum article of Ratnaparkhi [1] although my actual model ended up looking quite differently in terms of both search and feature selection.

2 The data

The data being used was the Penn Treebank (PBT) consisting of 24 years of the Wall Street Journal (WSJ). The treebank is annotated in a convenient format, with a tree structrure like this:

```
1 ( (S
2     (NP-SBJ
3       (NP (NNP Donald) (NNP Trump) )
4       (\, \,)
5       (SBAR
6         (WHNP-1 (WP who) )
7         (S
```

```
8             (NP-SBJ (-NONE- *T*-1) )
9             (VP (VBD faced)
10              (NP
11                (NP (VBG rising) (NN doubt) )
12                (PP (IN about)
13                  (NP
14                    (NP (PRP$ his) (NN bid) )
15          (...)
```

There is more annotated information here than I need in my project, and I consequently extract the bottom layer of the structures, which is a Lisp SEXP with a list of two elements, both of which are atomic. My code checks for this, which means I extract from the first two lines nothing (no such lists) and from line 3 (NNP Donald) and (NNP Trump), and from line 4 $(\backslash, \backslash,)$. For each literal word in the PBT there is such a pair, which I use for training, and later testing of my tagger.

I use sections 0-21 for training and section 23 for tagging. I see that this deviates slightly from the results posted at `http://aclweb.org/aclwiki/index.php?title=POS_Tagging_(State_of_the_art)`, but mostly the same.

3 MaxEnt Approaches

The MaxEnt approaches has their name from Information Entropy, seeking to extract the weight distribution that has the highest entropy, but still fits the data, in order not to assume more than you can about the data material. For the task of tagging, the problem is to select the most likely category for a given word, in its context. An aribtrarily large set of binary features can be created, that are either 1 or 0 in the given context. If the suffix is "ing" and the previous word is *is* then the word before that is *am*, this word is likely to be a verb, but if it is *favorite* its more likely to be a noun. One can here select features for each word in question during training, and see if you find them again during testing. So even though you might not find the exact same word sequences, you're likely to find at least some of them. And the bounty of the MaxEnt probability distribution is that you get the weights for each feature, that in turn can be used to select the appropriate category. Timely to prove, this results in an equation that gives you the probability of a category given a context like this:

$$P(c|x) = \frac{1}{Z} exp \sum_a \lambda_a f_a(c, x)$$

where only it can be shown that it is sufficient to maximize only the sum expression above, to get the most likely category. That means you don't need to calculate the normalization factor (the exping of all possible

contexts) to get the maximum category, but you need it to get an actual probability.

That, however, gives you the optimal category (POS) of a word in its context, but computing these standalone does not necessarily give the best sequence, which is what you are interested in POS-tagging, because previous tags can be included in the feature set.

4 The experiment

4.1 TADM

As I mentioned briefly in the introduciton I roughly split this project in half, where half it was about extracting the features from the training corpus and feeding them to The Toolkit for Advanced Discriminative Modeling, tadm, `http://tadm.sf.net`. That means I developed code for extracting the above-mentioned data pairs and then extracting features from them and writing them to a file that could be used as input to tadm. I chose the file format of the python scripts, mainly beause its easier to read than digital feature files, but also because they contained some nice features like a model storage. I added a function to them that exported the model at last to an easy readable SEXP format for later use in other scripts. The file format looks like this:

> IN PREVWORDĪN PREPREVWORD=INSTALLED PREPREPRE-
> VWORD=* CURRENTWORD=AFTER NEXTWORD=THE NEXTNEXTWORD=OCTOBER
> NEXTNEXTWORD=1987 PREVTAG=INPREPREVTAG=VBN PREPREPREVTAG=-
> NONE-

A feature is created for each of these equations, coupled with the possible POS-tags, for example ((PREPREVWORD=INSTALLED) IN) , a feature that receives the value one for contexts where the pre-previous word is INSTALLED with the POS IN. After a bit of hard work on the part of campus hardware, weights are returned for all of these features. I did not go into the inner workings of these algorithms at all, although I did “black box” compare the output of tadm on a couple of different machines.

4.2 The context cocept/Initial features

For each word I generated a context, implemented through a (7 2)-dimensional array in LISP, looking like this:

1	IN	PREPREPREVWORD	PREPREPREVTAG
2	INSTALLED	PREPREVWORD	PREPREVTAG
3	*	PREVWORD	PREVAG
4	AFTER	CURRENTWORD	
5	THE	NEXTWORD	
6	OCTOBER	NEXTNEXTWORD	
7	1987	NEXTNEXTNEXTWORD	

this is also acting like a FILO buffer where the context is "bumped" for each new word in the observation sequence (words to be tagged). I think it is fairly easy to see how the features are extracted, and after that is done for this word, the context is bumped into a new one looking like this:

1	INSTALLED	PREPREPREVTAG	PREPREPREVTAG
2	*	PREPREVWORD	PREPREVAG
3	AFTER	PREVWORD	PRETAG
4	THE	CURRENTWORD	
5	OCTOBER	NEXTWORD	
6	1987	NEXTNEXTWORD	
7	NNP	CRASH	NEXTNEXTNEXTWORD

from which features are again extracted. This also shows my first feature choice. These are very basic features, but once I had the feature extraction scheme working, I focused on the implementation of the Viterbi algorithm for a MaxEnt Markov Model (henceforth: MEMM) which for me was by far the most challenging task. I estimated that after that was accomplished it would be easy (although time-consuming) to refine the selection of features, which also proved to be the case. I used the weights from this run to develop the MEMM though, and I refer to it as the "Simple First Run".

4.3 The "Gold Standard Run"

When I extracted feature files for training, I also processed section 23 for testing in the same way. This left me with a feature file where all previous tags were entirely correct, as if I wanted to train on section 23. But what I instead did was to use a nice feature of the tadm Python script fauna, that evaluates a feature file like this and counts how many time your model predicts the tag that should be there according to the feature file. I refer to this as the "gold standard run", because it is a measure of how good the MaxEnt model is, because it gives you the accuracy of predictions where the history of the sequence is always right.

Obviosuly when you do decodeing you are working out the sequence yourself, but I found it useful as a ceiling for how good the tagger can get, given the model in question.

4.4 The "No Standard" run

Similarly I created a feature file erasing all tag history, which shows how you would do if you only knew the words in the context, and never even guessed about the last tag. I interpret this as the floor of bad the search algorithm can be given the model, you are in trouble if you get a worse sequence than the easy-computable one without tag history.

4.5 Feature refinement

After I was able to decode a sequence close to the "Gold Standard" run in quality I returned to refining the features. Due to resource issues I first ran a trimmed model with the only difference being that the model was now case sensitive, and adding bigram and trigram features, that is the combination of PREVTAG and PREPREVTAG in one feature, and the combination of PRE-PREPREV-PREVTAG in one feature ((PRE-PREPREV-PREVTAG NNP-VBD-NN) IN) as an example. In the refined versions, I also set the readtable up, so that I could preserve case-sensitivity of words.

Finally I was able to do a full run with the features in the above paragraph as well as 3-letter suffix and prefix information for all 7 words of context. This resulted in a very heavy model, with 1 498 134 active out of a total of 21 519 966 features.

For the feature refined models I only did a "Gold Standard Run" establishing their ceilings. I interpret the relative results of the simple model to the Gold Standard Run as a measure of what I could expect of viterbi decoding of them.

4.6 The Viterbi decoding of a MEMM

A MEMM is similar in its appearance to a HMM, in the sense that you compute a trellis containing the probablility, delta, of being in a certain state at a certain time (in POS-tagging the state represents a POS like in a HMM), but is very different in the computation. Although the computation of the delta(state,time) depends on both the previous tag and the current observed word, the computation is done in one. So you as I iterate over the previous states to maximize the product of delta(s',t-1) (s' representing a choice of s' from $\mid S \mid$ and the probability of attributing the *current* tag given the observed word and other context, the information about which tag is currently s' has to be baked in to the context.

Given a context as shown above, the (2 0) entry, the tag of the second slot has to be entered for all the states you iterate over, and the probability of being in the state you are actually in, in the trellis has to be computed. This is not a maximization problem, but it is the problem of computing being in a given state at a given time. Of the 46 states, 46 are always suboptimal

in the trellis. So you are interested in the probility of being in each state, representing a POS given all the different states as PREVTAG in turn, times the probability of being in just that state in the previous time step. The maximum of this product (combination of PREVTAG inside and outside context) is then slotted into the new delta cell. When you're at the end of your trellis, the highest delta is chosen, and you unwind the sequence.

I refer to Roman Finkelsteins (in German) `http://www-ai.cs.uni-dortmund.de/LEHRE/PG/PG520/MATERIAL/MEMM.pdf` presentation for a very good step-through show of the Viterbi decoding of a MEMM, that shows what I just stated in a diagram. Even after seeing int visualized like this it was still challenging to actually implement it, as it means the generation of so many different contexts from which the feautres are extracted and then categorized. Each time you set the PREVTAG of your current context to the s' from iteration, the PREPREVTAG and PREPREVTAG that are optimal choices in delta(s',t-1) must also be unwound before computation.

I have based my implementation of the Viterbi MEMM on Robert Wilenskys lisp code implementations of a HMM, written at a much higher level of sophistication than what I do from scratch, possibly standing out a little. This was though quite severly adapted to faciliate the decoding of a MEMM instead of a HMM. `http://www.cs.berkeley.edu/~wilensky/lispcraft/`

4.6.1 Assumptions

Preprocessing. The files for training are not used directly from the PBT, but run through a sed filter, that escapes special characters, such as punctiation marks, that have special meanings in Lisp. For me it was easier to filter the files first, instead of making further changes to the readtable as the files were loaded. I don't think this step is significant to the performance, though.

I reset the context buffer after each file, representing a hundreth part of a WSJ year, making the assumption that the beginning of the next section part (one of the hundred files) is independent of the last word in the previous section. It seems reasonable, but it is a choice on my part nonetheless, and should be mentioned. Viterbi decoding is run for each sentence (SEXP), which is short enough to avoid any problems with the probabilities getting too small for the computer to handle, which will happen at some point in an infinate Markov Model, as the probability of being in a given state in a given time is either equal or reduced compared to the top choice of the previous time step.

5 Results

Model	Gold Standard	No Standard	Viterbi MEMM
First run	0.9564	0.91763043	0.938118 (1-74) / 0.94893193 (75-99)
First refinement	0.96676964		
Second (full) refinement	0.98.1486680649		

The two values for the Viterbi MEMM decoding are because of an improvement in the algorithm. I think evaluating sections 74-99 is enough to say something about the accuracy, but I want to be specific about what was actually done. I would think this is an indication about how close we should be able to get to the other Gold Standard Runs as well, but this computation is very heavy (12 hours for the entire section 23), and it was only at the finishing stretch of the project I had both a Viterbi decoder and refined feature weights, so I didn't get to test that out.

As I noted in the introduction these results are very good. I have not tweaked any other settings, as for instance a Gaussian Prior variance, or a trehshold for which features to omit (because of low frequency). When this is done in the literature there has always been improvement, so until disproven I assume this would be possible also here. But I choose the humble approach, there can be something here I didn't think of.

One thing I can think of noted is that I do not delete -NONE- tags for traces. And since the feature for attributing the NONE tag for a NONE pseudo-word (all traces are conveniently equipped with these) has a very high weight, since the NONE word is always NONE, and same for traces, this obviously boosts accuracy, with having such easy features that you always get right.

I also think it should be noted that the scores for the simple run are quite high, at least to me, given just its simplicity. In this run, everything was also upper case, using a standard lisp readtable. This was changed in the refined models.

For the second refinement, I made sure to rerun that, and from what I can see the model is capable of coming up with these tags, without knowing them from beforehand. And the way of measuring it is very easy, you have the right tag, or you don't.

6 Evaluation

Obviously the viterbi decoding is the most interesting part, because that's what the WSJ POS-tagging task is really about, and I should have (will) try that for the bigger models. Part of the reason I couldn't do it was because the actual export of the huge text files used to transfer the model into lisp

takes so many hours that I chose to drop it, to at least collect Gold Standard values. But I had the one for the Simple Run, that I have used for development.

I mentioned already the inclusion of NONE-tags. They could also have been taken out in a test run.

I should also have conformed to the WSJ POS-tagging task from the ACL Wiki, making my results directly comparable to the posted results there. I can not see how using the section choices from there would have made my task computationally more difficult, although it is likely to have diminished results slightly, because of less training and more testing data.

And obviously the point about review, since the results are so encouraging even before any parameter tweaking is done. Possibly this is because of my "greed" just adding a very large number of features whose computation were made tractable only by the availability of a fast machine.

7 Discussion

7.1 Complexity of Viterbi decoding

In the way I have implemented the decoder, I need to extract probabilities, and not only maxima as explained above in section 4.6. I have given it some thought, but I still maintain the position that you need to compute the entire probability distribution of each context in the MEMM trellis, because you are not looking for the most likely category at each point, but on the other hand the exact probability of an exact tag, that most likely is not no be found in the optimal sequence.

Obviously I'm not in a position to rule out that the problem has a simpler mathematical solution, but I think it will be difficult, because the context from wich the probability distribution is computed is different in each state. You don't maximize over states in the trellis before you are at the last time step, where you find the highest probability and start unwinding the optimal sequence (stored in the psi array in my implementation).

7.2 Relation of Viterbi decoding to Gold Standard runs

Now that I have the infrastructure, the decoder and binary models of both the refined run, the answer for my bigram model would be an easily solved empirical question in terms of complexity, but not time. The binary models can be loaded into the tadm Python and text models can be written from them, which can be loaded into lisp and the decoding can be run on section 23 with out any acutal programming to the model. My expectation though is that the difference between the gold standard and the viterbi decoding would be comparable. Most likely the floor performance of the model is higher as well, and this gives you a very good chance of making the right

desicion at one time point, that will make it even easier to get it right the next time, and so on. Because of this, I don't think that the improvement by going to a trigram MEMM creating (* 46 46) virtual states, with having to create 46 times as many contexts at each point in the run would help significantly. You would get a better sequence, but at a very small rate of improvement is my intution.

Continuing to shoot from the hip, I would expect that you need a n-gram MEMM corresponding to the depth of TAG history you store in your model to be *sure* of decoding the optimal sequence, in my case a quadgram model with 46^4 or 4 477 456 virtual states. I don't know how to prove this either theoretically or empirically though.

7.3 Further work

It will be interesting to see how the MEMM copes with the much heavier models in terms of computational time and performance. This would be the first thing I would like to try out, since I can't see how this would be a problem just doing at this point, although it obviously will require many a CPU hour.

7.4 Error analysis

I compared my errors (which tokens were most often mistaken with which) to Ratnaparkhis error analysis from 1996, but I don't think that was so interesting. First of all my errors were from the Gold Standard Run of the flawed Simple Run Model, and then because different features were used, different unsurprisingly came about. It is perhaps possible to not that his features, focussing more on suffix of the current word were more "lexically" strong avoiding as many NN for NNP errors as my Simple Run had.

It would be more interesting to do an error assesment again of a full viterbi decoder based on the heaviest model.

References

[1] Adwait Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In Eric Brill and Kenneth Church, editors, *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 133–142. Association for Computational Linguistics, Somerset, New Jersey, 1996.