Computational Linguistics: Practical

Mr, Omer Gunes

Candidate number: 291873

Contents

Overview

This report consists of four parts. First part focuses on the implementation detail of Part-of-speech(POS) tagger based on Hidden Markov Model (HMM) and the performance of the model while the model knows all or some words in the corpus. Second part will give the results of the HMM after several different smoothing techniques[4] deployed to the model. This project tested the result for both trigram and bigram smoothing. The last part of the project will give the result while we deploy another powerful model: Neural Network (NN) to model the tagging. Notice that all the code associated with this work could be found on the github repository[1].

Bigram HMM

Implementation Detail

The implementation of learning part is trivial and you could view the code on the github. This is just counting n-grams and do a Maximum Likelihood Estimation(MLE). For the predicting part, it is easy to see that this is the computational bottleneck of the model. Hence, it is implemented in a manner to exploit all computational power of the CPU. The cross validation uses multi-threaded implementation, while the Viberti algorithm used in prediction is implemented in a Matrix-Matrix multiplication manner. That could exploit the extra power provided by SIMD instruction set on modern CPU architecture[3].

Performance

The results is obtained by doing a 12-folded validation. '12-folded' is because there are 4 cores available in the workstation. The model is trained with two different ways. The first model knows all words in the corpus. The second model only knows some words in the corpus. For second model, we just need to input the same training set as the one for training transmission probability to train the emission probability. The performance is measured by per-word accuracy. That gives us interesting result.

Model knows all words	Model only knows some words
0.9263	0.9441

That means while the model sees unknown word, just guessing the tagging by transmission probability deteriorates the result.

Smoothing of HMM

Overview

One could easily extend the previous bigram model to traigram one. Yet, due to the scarcity of the training data, doing a MLE on trigram model will generate lots of zero probability. Once Viberti sees zero, it will immediately refuse to predict on this branch and deteriorate the performance.
Suppose we want to solve this problem on a trigram model, it is intuitive to think about using a bigram model to predict the transition probability while trigram one meets a zero entry. That is the basic idea of smoothing. In this project, five smoothing techniques are implemented, and the effect of smoothing on the trigram and bigram model are examined.

[1] https://github.com/ET-Chan/POSTagger

Performance

All the results below are obtained by doing a 12-folded validation, indicating there are some words are unfamiliar with the models.
For trigram model:

No smoothing	Adaptive	Katz	Witten-Bell	Kneser -Ney	Knesey-Ney modified
0.0667	0.9285	0.8795	0.9249	0.9245	0.9248

For bigram model:

No smoothing	Adaptive	Katz	Witten-Bell	Kneser -Ney	Knesey-Ney modified
0.9186	0.9220	0.9203	0.9221	0.9220	0.9220

Comparing two models, you could see trigram models generally perform slightly better than the bigram one except for the Katz and 'No-smoothing'. It shows that without deploying smoothing technique on trigram model, the result is not acceptable, as it is even worse than unigram model.

Comparing the effects after deploying different smoothings on the model, one could find out that the adaptive smoothing works best. Knesey-Ney modified smoothing works slightly better than the original one. Noticeably, all of them actually perform very similarly in terms of per-word accuracy. It is hard to say which smoothing is the best as it could be only the noise causing the difference.

Neural Network

One way to address the problem of unknown word is to introduce extra words to the model. On other words, to make the model contain certain amount knowledge of the language. We could achieve that by firstly train a Language model. Suppose, given a sequence of sentence, the model outputs accessibility of the sentence. Specifically, the score of 'This is a cat' should be higher than the score of 'This cry a torch'. Then, we transfer the knowledge contained in the Language model to the POS model. The language model we trained here is a Neural Network model, the structure could be refered to[6]. Or you could visit the github repository. It is self-explanatory.

In terms of training, firstly, the language model is trained with the whole English Wikipedia in a unsupervised manner. The model is trained with ADAM optimizer[7]. The dataset is obtained from [2] and pre-process by Wikipedia Extractor [1] and Stanford NLTK [8].
The neural networks implemented on this practical are all based on Torch 7[5]. In order to accelerate the training process, the Lookup table used in the models is LookuptableGPU in fbcunn. The language model was trained on GTX 860M with i7-4710MQ for 16 hours. Afterwards, we could extract the parameters from the first layer of the language model, namely the lookup table, to see the similarities of the words found by the model. The following table gives the words model think are similar.

MUSIC	CHINA	MOUSE	IMMIGRANT	HITLER
FILM	INDIA	EGG	REFUGEE	STALIN
PRODUCTION	JAPAN	KEYBOARD	FEMINIST	WALSH
BUSINESS	EUROPE	FUNK	MILITANT	RHODES

With the power of GPU, only 16 hours training we could obtain an acceptable embedding of words. Now, we transfer these parameters to the first layer of our neural POS tagger, and train the parameters jointly to get a fine-tuned model. The per-word accuracy of the model is as follows.

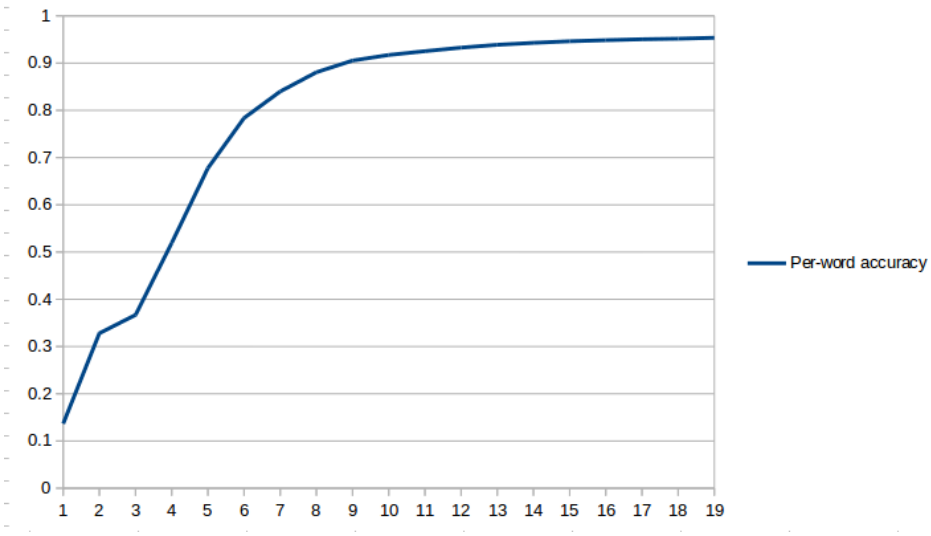

Figure 1: The convergence of the model while epoch is low

.

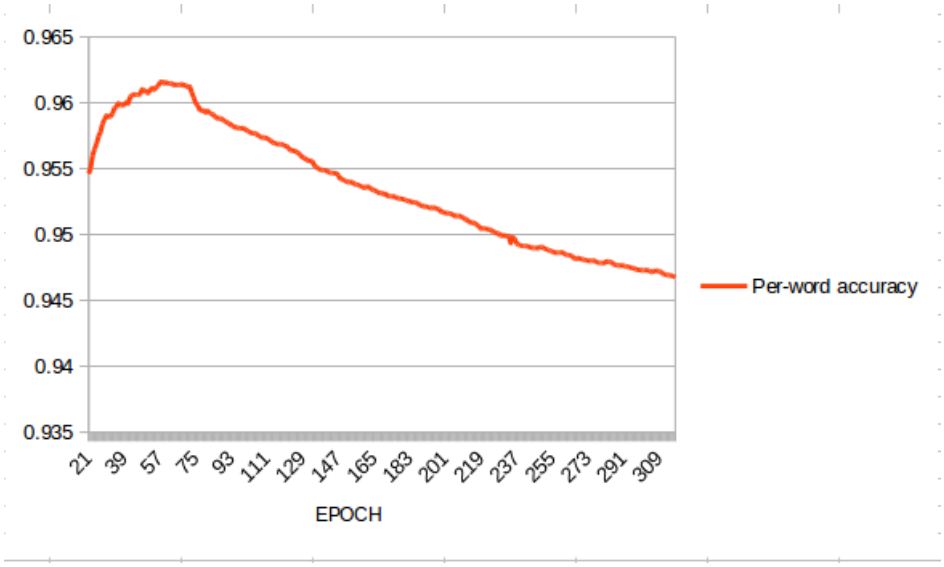

Figure 2: The convergence of the model while EPOCH is high

From fig. 2, you could see a clear sign that the model starts to overfit. By deploying early-stopping, we could prevent the model from overfitting. The best accuracy achieved by this model is 0.9614, which is 4 % higher than the HMM model. Although this figure is slightly worse than the one reported in [6], it could be argued that the supervised data provided in this practical is way less than the one they used to train their model.

Acknowledge

I hereby thank for the help of Mr. J Buys and Mr. Omer Gunes, two practitioners in the practical session. Their advices on smoothing techniques and the unsupervised learning really help me a lot in this work.

Reference

[1] Wikipedia extractor,https://github.com/bwbaugh/wikipedia-extractor.

[2] Wikipedia latest english dump, http://dumps.wikimedia.org/enwiki/20150205/.

[3] John Canny and Huasha Zhao. Bidmach: Large-scale learning with zero memory allocation. In *BigLearn Workshop, NIPS*, 2013.

[4] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics, 1996.

[5] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[6] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537, 2011.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[8] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014.