Semantic Role Labeling with Structural Perceptrons: An Experiment

Li, Ziyao
1500017776 / Yuanpei College
leeeezy@pku.edu.cn

Abstract

This article is the Project 1 of *Course EMNLP*. The main task is to realize a model to perform Semantic Role Labeling (SRL) on Chinese PropBanks. The experiment regard this SRL problem a Word-level Sequence Labeling Task and implemented Structural Perceptrons Algorithms (Collins, 2002) along with a Viterbi decoding process over the problem. Due to the limitation of training time, only unigram and bi-gram features are explored. Besides, a BIOES labeling standard is adopted instead of the old-fashioned BIO system.

1 Introduction

1.1 Semantic Role Labeling

Semantic Role Labeling (Carreras and Marquez, 2005) is a task to identify the $(Predicate, Argument)$ relationships in given corpora and to briefly categorize the arguments into a coarse classification noted as *subjects*, *objects*, *complements* etc. The problem is important as a preposed task of knowledge mining applications such as Entity Linking and Relation Extraction. Various methods to solving the problem were proposed in different NLP conferences in the past several years after the Conll 2004 task (Sun et al., 2009) (Punyakanok et al., 2008).

1.2 Data Description

The data used in this experiment are mainly derived from Chinese Proposition Bank (CPB), while the test dataset is generated without gold labels (provided by the course). Below in Table 1 are some details of the data. Also, data from Chinese TreeBank (CTB) are also explored to train a Chinese parser to derive more features. Part-of-Speech (POS) tags are provided in the data as ground truth.

Data	Sentences	Predicates
Train	8828	31346
Develop	561	2064
Test	803	1874

Table 1: Numbers of sentences and predicates in the data.

1.3 Structural Perceptron

Structural Perceptron (Collins, 2002) is a discriminant model designed for various tasks. The concept of the model is very simple, encouraging features of correct labels while discouraging those of misclassified labels. Although seems naive, the model has very good performance and efficiency.

2 Code Pipeline

This section is to briefly illustrate the codes provided in the attachment.

Firstly, we combine the .text file and the .props file together and transform the original labeling system into the BIOES system. We treat each predicate in a sentence as an individual sample. Then, a script is implement to combine the features from the parsing trees with the BIOES labeled data, forming a .samples file.

Secondly, we use a *CRF++* style template file to announce the uni-gram features and bi-gram features to explore according to the *.samples* file. More than 50 uni-gram templates are defined over 11 explicit features, while only a constant bi-gram feature is defined to save on complexity. The script then ignores features which has a frequency less than 5 to avoid overfitting and outputs the derived features into a *.features* file. Note that features are only derived and encoded on the training set.

Thirdly, we read this *.feature* set and train a Structural Perceptron defined in *perceptron.py*. Basic methods, such as *train()*, *eval()*, *predict()*, *save()* and *load()* are implemented in the class *Perceptron*.

Finally, the model is trained and saved, and predictions are made and then translated into a *.props* style output *.pred*.

3 Features

Table 4 shows a set of designed explicit features in *.samples* file, upon which 60 uni-gram feature template generates $434,878 \times 26$ uni-gram features (26 is the total number of the BIOES labels). 27×27 bi-gram features (adding a START and an END feature) are defined as well to make the BIOES labels commensurate with each other. The explicit features comes from POS tags, parsing trees and dependency trees. The dependency trees are generated from parsing trees with the code *tree.py* accomplished in Homework 2 of the *Course EMNLP*. Besides, to avoid possible heterogeneity between the gold parsing trees and the predicted ones, parsing tree features are derived from the predicted ones even when the gold parsing trees are available for the training set.

The feature templates are announced in *features.tpl* in a CRF++ style. Mainly a combination of up to three independent features are explored, horizontally or vertically. The interested can browse the file for a more detailed explanation.

One small trick is also implemented since predicates are given in SRL tasks. When conducting Viterbi Algorithm, if the predicates are not specially handled, they can be predicted as strange labels rather than *S-V*. However, the code would be redundant and potentially buggy if one tries to alter the Viterbi Algorithm at the positions of predicates. We achieve identical effect to lock the predicates using a simple trick, that is when detecting that current word is the very predicate in the sample, we mask all entries but *S-V* as negative infinity. This trick ensures that only *S-V* can be entered when calculating maximums and retrieving the sequences. If predicates can be rather long, this trick can be easily extended.

4 Results

Table 2 shows the results over the development set (50 epochs training on 31,346 samples in 8,828 sentences, evaluating on 2,064 samples in 561 sentences).

Perceptron	Precision	Recall	$F_{\beta=1}$
Overall	78.58%	76.47%	77.51
A0	71.64%	70.59%	71.11
A1	81.02%	81.86%	81.44
A2	76.09%	64.22%	69.65
A3	76.92%	66.67%	71.43
A4	57.14%	44.44%	50.00
AM	81.94%	77.30%	79.55
V	100.00%	100.00%	100.00

Table 2: Structural Perceptron performance over the development data.

CRF++	Precision	Recall	$F_{\beta=1}$
Overall	75.04%	71.29%	73.12
A0	64.92%	56.42%	60.37
A1	80.07%	79.61%	79.84
A2	75.69%	62.84%	68.67
A3	90.00%	60.00%	72.00
A4	100.00%	66.67%	80.00
AM	76.45%	75.47%	75.96
V	100.00%	100.00%	100.00

Table 3: CRF++(v0.54) performance over the development data.

Besides, an attempt using directly the CRF++ (v0.54) (Lafferty et al., 2002) toolkits to accomplish the sequence labeling task was made as well (47 epochs over the same data). Features are designed identical as the Perceptron Algorithm. Table 3 shows the detailed performance.

A better performance can be observed in the Structural Perceptron Algorithm. Besides, the time each epoch takes in Perceptron Algorithm is a lot shorter than that of a CRF algorithm.

References

Xiavier Carreras and Lluis Marquez. 2005. Introduction to the conll-2005 shared task: semantic role labeling. In *Proceedings of the Ninth Conference on Computational Natural Language Learning*, pages 152–164.

Michael Collins. 2002. Discriminative training methods for hidden markov models: theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 1–8.

John Lafferty, Andrew Mccallum, and Fernando Pereira. 2002. Conditional random fields: Prob-

Instance	Meanings
(China)	Word.
NN	POS.
(established)	Predicate.
VV	Predicate POS.
VP=(VV)	The macro-structure of the Predicate in the parsing tree.
-7	The distance from the predicate to the word.
bf	Whether the word is before (*bf*), at (*at*) or after (*af*) the predicate.
NR>NP>IP<VP<VV	The reduced path from the word to the predicate in the parsing tree. ”>”, ”<” indicates the direction traveling up in the tree. The path is reduced means identical adjacent nodes are merged into one.
(Spandex)	The head-word of the current word in a dependency tree. The dependency tree is generated from the parsing tree with the code *tree.py*.
NN	Head word POS.
notPred	Whether the word is the predicate (*isPred*) or not (*notPred*).

Table 4: Explicit features generated in the *.samples* file. The instance here is the first word of the first sample in *dev.samples*.

abilistic models for segmenting and labeling sequence data.

Vasin Punyakanok, Dan Roth, and Wen-tau Yih. 2008. *Computational Linguistics*, 34(2):257–287.

Weiwei Sun, Zhifang Sui, Meng Wang, and Xin Wang. 2009. Chinese semantic role labeling with shallow parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 1475–1483.

Appendix

The codes are written with Jupyter Notebook. Codes in sequences are the very pipeline described above in the corresponding section. Only original and predicted data are saved and submitted due to the size of the middle results (over 1 GB). The interested can run the whole process of the pipeline to verify the test results by first implementing the data and then run the experiment code.

It is a future task to merge all the separate data handling scripts into an end-to-end system due to the limitation of time.