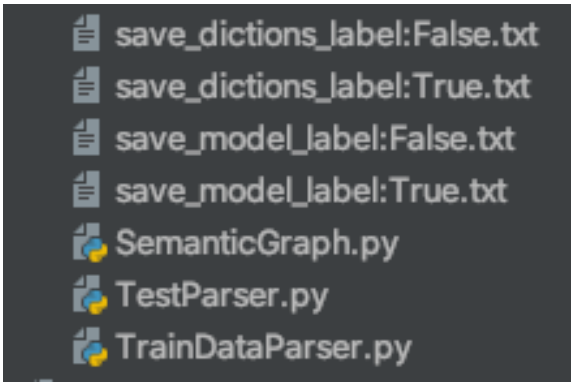
Report for Project II: Bi-lexical Semantic Graph

张瑞峰 1700012823

General View of Project

As shown on the right, the project is divided into three parts, TrainDataParser, SemanticGraph, TestParser, which are for parsing training data into features, main process as well as training part and generating output file.



They can be run through running SemanticGraph.py alone, or run as individuals or in any combinations, for they all have storing files. The model's output file is result.sdp, and the other two pairs of .txt files are used to run TestParser.py to create a result file, marking whether arcs are labeled. It's worth noting that result.sdp is not labeled, to get a labeled result, run TestParser.py with use_label = True.

Implementation

Parsing Data

The data parser includes a few basic parts.

First of course, the parser take in the training set and decompose it into paragraphs, and extract links.

And then, the parser judges the gold operation at each status. The approach is to do this with a transition-based model, which includes six basic operations: shift, delete, reduce, swap, left-arc and right-arc. The details are as follows[1]:

1. $LeftArc_r$: Adds a dependency arc from the word at the front of the queue to the word on top of the stack and selects the label r for the relation between the words.

2.$RightArc_r$: Adds an arc from the word on top of the stack to the word at the front of the queue and selects the label r for the relation between them.

3.Reduce: Pops a word from the stack.

4.Shift: Shifts a word from the queue to the stack.

5. Swap: Swaps the two words at the top of the stack.

So there're two ways to judge a transition: with and without arc's label. Each transition starts with root in stack and other words in queue, and ends with the queue emptied.

To generate the gold operations, the model simulates the process in each paragraph using the following strategies:

1. Judge whether the stack is empty, if it is then use Shift.

2. Judge whether the word on top of stack has no arcs toward or from the words behind it, if it doesn't then use Reduce. Because once a word enters the stack, it can't be linked with other words in the stack, and

[1] Modified from paper "Online Graph Planarisation for Synchronous Parsing of Semantic and Syntactic Dependencies"

all words in the queue have bigger id number, if it no longer has relations with the words in the queue, it should be be deleted.

3. Judge whether there is a relation between the stack top and the queue front, if there is then use an Arc transition. However, if last transition was an Arc one, obviously it mustn't be done again.

4. Judge whether there is a crossing-arc related to the two words on the top of the stack, if there is then Swap. To do this, during the process a list is maintained to see each word's relations, and if the word with smaller id number has a "smaller" list, in the lexicographical order (like [5] < [5, 6]), then the crossing arc is spotted.

5. If 1-4 are all rejected, then use Shift.

During the process, every status (stack and queue) and transition corresponding to it are documented.

As shown in the screenshot below, The strategies work very well, representing more than 99% of all the arcs. Also it sums all the transitions.

```
{'swap': 23443, 'shift': 745543, 'reduce': 745543, 'larc': 168854, 'rarc': 418699}
There are in total 587660 arcs, and the model generates 587553 arcs, 99.98179219276453%
```

Feature and Training

1. First Approach and Problems

The first approach is with vectorization and perceptron. To vectorize the word, the model uses one-hot encoding to encode both words and part-of-speech tags, and then concatenate them as a word vector. And the transition operations are also vectorized in one-hot encoding.

Then, the status is vectorized as follows:

$$V_s = V_{ws1} + V_{wq1} + V_{ws2} + V_{wq2}$$

where ws1 is the word on top of stack, wq1 is the word on front of the queue, ws2 is the next word to ws1 and wq2 is the same way, V stands for the word vector mentioned above and '+' is concatenate operation.

Using vectorized status and corresponding vectorized transition, the model builds the training set and the test set. And with a matrix of coefficients, the perceptron works like:

$$y^p = xW^T$$

$$update \quad W \leftarrow W + LR * (y^{gold} - y^p)^T * x$$

where x and y stand for vector representations of status and transition.

However, under the unlabeled training set, this method only has a transition prediction accuracy of 55.85% after enough epochs.

The reason is clear: there're around 2 million transitions in total, which greatly limits the length of the vectorization. In fact, previously the length of one-hot word vector is set to 1000, and then the status vector's length is over 4000, which is unrealistic under such amount of data. If more feature words are selected, the situation can be worse, which is beyond my PC's capabilities.

Also, the weight matrix can't be trained efficiently because the vectors are too sparse. To train W_{ij},

it requires the word to appear with a certain transition, and many groups of which never showed up in the training set.

2.Second Approach: Key Feature and Log-Linear Model

The general idea is to extract key features. To do this a dictionary is selected from the training data. More specifically, the key of the dictionary is like $(CheckID, KindID, ItemID, TransitionID)$ or $((CheckID, KindID, ItemID), (CheckID, KindID, ItemID), TransitionID)$,

where CheckID denotes where to look for features, including

0: top of stack, 1: front of queue, 2: second of stack, 3: second of queue, 4: the stack-top's previous word in sentence, 5: the stack-tops's next word in sentence, 6: the same as 4 for queue-front, 7: the same as 5 for queue-front, 8: the last transition, 9: the second last transition,

KindID denotes what kind the feature belongs to,

including 0: word, 1: part-of-speech tag, 2: transition,

ItemID is the feature's Id in that kind, and TransitionID is the transition the feature applies.

The keys are basic features and the combinations of two basic features representing the dependency.

After setting a threshold, the features that appear often are selected as keys, and each is assigned with a weight w. Then, use a simple log-linear model and gradient-decent optimizer to maximize $logp(y^{gold}|x, \theta)$. The entire implement is same as the previous lecture had mentioned and the one in course project I, so the details aren't displayed here.

3.Other Tricks

Because transitions have very different number of data items, when building the batch for training, the model selects equal number of data concerning each transition.

Also, the appearance thresholds are different for different transitions. For example if a certain transition, especially when considering labels, appears very few times, then the feature concerned with it should be taken at a lower standard. The dynamic threshold is set to $ln(number_{tran}) * threshold$, where threshold is a const number set to around 1.5.

To avoid consuming too much space, after every 1000 sentences, the program would examine the features to get rid of those that appears less times than this number.

Building Output File

The method uses the simple way: with stack and queue status given, it uses the trained weights to obtain a best next transition, and try to apply it. If it's not legal, like swap after swap, arc after arc or reducing, swapping without enough elements, it just applies a shift. This kind of approach may not be able to build as a semantic graph as best as the model can do, compared with beam search or other methods.

Results

Unlabeled

Excluding labels, there're in total 429715 features taken into account. As shown in the screenshot on the right, the final approach achieved transition accuracy at 80% on testing set (derived from training data).

However, this result is achieved at a time consumption of over one hour. Generally, a whole round is about 1000 steps with batch size 500. Because it's performance is a lot better than the labeled method, the output file is an unlabeled one.

```
Making result
Training:
Accuracy: 0.8253064310729067
Testing:
Accuracy: 0.815674413668683
Time spent:
Data parsing: 953.830203823 seconds
Iterations: 2987.36443171 seconds
Every iteration on average: 2.98736443171 seconds
```

Labeled

Including labels, there're in total 31 transitions and 55867 features taken into account. The transitions appear for 192 times at least and 596434 times at most. And training results are shown below.

The labeled transition accuracy is around 60%. However, it can't be said that the method is bad because the labeled features are just too many and takes too much time and space, and many useful features are abandoned for the model to be able to run. Even at this condition, running an entire parsing and training must take around one hour.

```
Testing:
Accuracy: 0.5962162676693161
Time spent:
Data parsing: 1474.4882420210001 seconds
Iterations: 2339.69580568 seconds
Every iteration on average: 7.798986018933333 seconds
```

It should be noted that these are the accuracy on making decisions, and a wrong move may cause sequential disasters, so I'm supposing that the result on the final output is worse. Meanwhile, the recall rate, for some reason, is always lower than precision, causing F1-score to be lower. I think that the unbalanced dataset had caused this.

Also, the model deals with top node very very bad. It almost judge every first word as top. I think it's because in the training set, when word 'root' is on the top of stack, it's often linked toward the queue, making the model believe that whenever 'root' is on stack top, a right-arc should be chosen. May the position of the word in the sentence should also be taken into the feature.

Conclusions

I had learnt very much during the work. The data structure, transition selection, and feature engineering, etc are all very challenging.

The model shows that the transition method truly works well. Although the learning structure is very simple, the method gained an accuracy of 80% on the unlabeled transition-selection. It should be noted that this result is under the circumstance that many features are already abandoned, and the model can't run for enough rounds, otherwise the time and space consumption are both beyond PC's capability.

Beyond the computational limitations, the project has some major drawbacks now:

1. Bad try except sentences

Because I don't know it very well and used it very much without really catching any errors, many things may go wrong, most importantly, sounding no alarm at all. I came across a few bugs concerning this and they really are time consuming. Also, the project can have more bugs undercover.

2. Casual feature selection

The features are selected not because they are special to certain transitions, but are merely because they appear more often. The effect is unknown but I think it exists. This may have caused the bad result on top selection, too.

3.Over simple optimizer and output parser

As mentioned above, the optimizer, especially gradient decent is too simple for this task, and the process of making output is too naive.

Of course, many other parts of the model can also be improved. For example, there’re certainly other ways to reduce computational and spacial complexity.

All in all, this project is a huge one for me, and it really takes time to meditate small things during such a giant work. It’s challenging, as well as exciting.