Project 3: Parsing

CS 4650/7650: Natural Language Processing

February 21, 2013

In this project, you will build rule-based and statistical parsers. You will build on 3rd-party code. The project has two separate parts: rule-based CFG parsing, and statistical dependency parsing.

1 CFG Parsing for Twitter

In this section, you will build a context-free grammar for Twitter, using the dataset `oct27.dev` from Project 2.

My code `parseTwitter.py` has a function `evalParser()`, which by default will use your grammar to parse all sentences of length less than 10. It will also parse scrambled versions of those sentences, which are assumed to be ungrammatical (of course this is not always true, so the accuracy will be a lower bound). My code calls the `nltk` library, which you may like to learn more about here (`http://nltk.org/`) and especially here (`http://nltk.org/book/ch08.html`).

The ideal parser will produce exactly one parse for every valid English sentence, and zero parses for the scrambled sentences. The scoring works like this:

- **true positive**: your parser produces at least one analysis for the original sentence
- **false negative**: your parser fails to produce any analysis for the original sentence
- **false positive**: your parser produces at least one analysis for a scrambled sentence
- **recall**, **precision**, and **f-measure** are as defined in class
- **parses-per-sent**: the number of parses per analyzed sentence (not counting sentences which could not be parsed). This is a measure of the ambiguity in your parser, and should be as low as possible

Your parser should be specified as a text file, e.g. `lastname-task1.cfg`. (All grammars should be submitted and should follow this naming convention.) Each line in a grammar specifies a set of productions, e.g.

```
S -> NP VP | S "&" S
NP -> "D" "N" | "^"
VP -> VP NP | "V"
```

Quotes are used for terminal symbols. For simplicity, your terminal symbols will be at the part-of-speech level, not the original words. You can assume "gold" part-of-speech tags as provided in the dataset.

Once you have created a CFG, you can evaluate it using `evalParser`

```
import parseTwitter
output = parseTwitter.evalParser("file:lastname-task1.cfg")
```

This function takes an optional argument `debug=True`, which prints some information that can help you improve your parser.

1.1 Getting started

Deliverable 1 Create a CFG that gets perfect recall and only one parse per sentence.

Sanity check It should have very few non-terminals.

1.2 Grammar design

Now you will create the best CFG that you can. The grammar in Deliverable 1 gets (at least) 28.6% f-measure, so you'll have to do better than that. Take a close look at the tag definitions in the POS paper [GSO+11], and also think about the English grammars discussed in class and in the text.

Deliverable 2 Create the best CFG that you can. Try to maximize f-measure without producing too many analyses per sentence. Submit the grammar and indicate the results in your writeup.

Sanity check I designed a grammar that gets 40% f-measure with 21.8 parses per true-positive sentence, and another grammar that gets 37% f-measure with only 4.7 parses per sentence.

Deliverable 3 `evalParser` has an the optional argument `max_len`. Set `max_len=20` to try to parse sentences of length < 20. Report the f-measure and average number of parses.

Sanity check My grammar that produced 4.7 parses/sentence now produces 372 parses/sentence!

Deliverable 4 One especially tricky aspect of Twitter parsing are the "verbal" tags "L", "M", and "Y." These tags are used for tokens that include both a noun and a verb, like *she'll*. How do you handle them in your parser?

1.3 Terminal refinement

Try to refine the tagset. The function `evalParser` takes an additional optional argument, `preprocessor`. This argument is the pointer to a function that takes a set of words and tags and performs some additional processing. I've provided a simple example function, `preprocess()`, which searches for the word "to" and gives it a new tag "2". You can then use this new "2" tag in your grammar, giving it different treatment from other prepositions — for example, it can produce infinitive verb phrases. Other ideas would be to introduce special handling for present and past participle verbs, case marking for pronouns, etc. Section 4 of Klein and Manning's "Accurate Unlexicalized Parsing" might give you some other ideas [KM03].

The syntax for using `preprocess` is:

```
out = parseTwitter.evalParser("file:name-task3.cfg",
                              debug=True,
                              preprocessor=parseTwitter.preprocess)
```

Deliverable 5 Modify the `preprocess()` function to include at least three new tags, and develop a new grammar that uses these tags. Try to improve your f-measure from the previous deliverable. Explain the idea behind what you did.

Deliverable 6 **Bakeoff!** Submit your best grammar as `lastname.cfg`. I'll evaluate them all on the test data and present the results in class.

2 Unlabeled dependency parsing

In this section you will work with a structured perceptron dependency parser, which is very similar in design to the structured perceptron sequence labeler that you built in Project 2. In this case you don't need to implement the inference or the learning, just the features.

You can load in the parser with the following commands:

```
import sys
sys.path.append('parsing/')
sys.path.append('util/')
import dependency_parser as depp
dp = depp.DependencyParser()
dp.read_data("english")
```

Deliverable 7 Report the number of features.

2.1 Training

Now train the parser:

```
dp.train_perceptron(10)
```

Sanity check After 10 iterations I get 51.0% accuracy on the development data

The existing parser has only a single arc feature template: the pair of tags involved in each dependency arc. This feature template is instantiated into many different features, as you saw in the deliverable above.

You can see how the POS feature template is implemented in the function `create_arc_features()` in the file dependency_features.py. Each feature template is represented as a list of arguments, including a feature template counter. The POS tag feature is implemented as

```
f = self.getF((k,instance.pos[h],instance.pos[m]),add)
```

where the first element inside the list is the feature counter (which prevents collisions), and the second two elements are the POS tags of the head and modifier nodes. The `Instance` class is defined in dependency_reader.py.

2.2 More features

Now you will add more features to try to improve performance. Every time you add features you will want to reinstantiate your parser and call `read_data()` again.

Distance First, add features to quantify the distance between the head and modifier in each dependency arc. You can add features for every possible distance, but it may be better to threshold the features at some upper and lower limit.

Deliverable 8 How does this impact dev set accuracy (after 10 iterations) and the total number of features?

Sanity check In my case, it makes a very substantial difference, more than 10% absolute improvement.

Lexical features Add lexical features between the word of the head and the tag of the modifier, and vice versa.

Deliverable 9 How does this impact the total number of features? How does it impact the development accuracy? How does it impact the **training** set accuracy?

Sanity check Again I find a very substantial improvement in development set accuracy.

Bilexical features Add bilexical features between the word of the head and the word of the modifier.

Deliverable 10 How does this impact the total number of features? How does it impact the development and training accuracy?

Context features Add context features that consider the tags adjacent to the head and modifier. You may wish to consider various tag combinations, such as

- $\langle t[h], t[h-1], t[m] \rangle$: head, head-left, modifier
- $\langle t[h], t[m], t[m+1] \rangle$: head, modifier, modifier-right
- $\langle t[h], t[h-1], t[m], t[m+1] \rangle$: head, head-left, modifier, modifier-right
- etc

Deliverable 11 Describe what context feature templates you have added. How do they impact the total number of features? How does it impact the development and training accuracy?

2.3 Bakeoff

You can try to develop any other features that you like. Please explain all the features that you have. After identifying your best feature set, run `dp.test()`. This will produce a file data/deppars/english_test.conll.pred.

Deliverable 12 Rename this to lastname.conll.pred and include it in your t-square submission.

3 7650: Applications of Dependency Parsing

Find a paper that applies dependency parsing to help solve some other problem: information retrieval, question answering, information extraction, paraphrase, semantic analysis, translation, etc. Your paper should be selected from the proceedings of ACL, NAACL, EMNLP, WWW, SIGIR, or EACL.

Deliverable 13 Report on:

- What is the problem that is being solved?
- Why is dependency parsing expected to help?
- Do they use labeled or unlabeled dependency parses?
- What dependency parser do they use?
- How is dependency parsing included in the system?
- What improvement, if any, does it make?

And add any other details of interest.

References

[GSO+11] Kevin Gimpel, Nathan Schneider, Brendan O'Connor, Dipanjan Das, Daniel Mills, Jacob Eisenstein, Michael Heilman, Dani Yogatama, Jeffrey Flanigan, and Noah A Smith. Part-of-speech tagging for twitter: annotation, features, and experiments. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 42–47. Association for Computational Linguistics, 2011.

[KM03] Dan Klein and Christopher D Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics, 2003.