CS 224N / Ling 280

Assignment 3: MaxEnt Sequence Classifiers & Treebank Parsing

Due Wed, 16 May 2007, 5pm

0 Overview

This assignment looks at named entity recognition and parsing. The aim is to examine whether pre-chunking of named entities can improve the performance of a statistical parser trained on financial newswire text when applied to the task of parsing biomedical research articles. You will build a maximum entropy classifier, which will be incorporated into a maximum entropy Markov model for doing named entity recognition on biomedical text. You will also implement the parsing algorithm for a broad coverage statistical treebank parser. We have included in the support code the ability to chunk entities into a single word, and then to pass this chunked sentence to the parser, so that you can then informally compare the performance of the parser on chunked and unchunked input.

1 Setup

We've put everything for both portions of this assigment (both Java starter code and data files) in the directory `/afs/ir/class/cs224n/pa3/`. The Java starter code is in `pa3/java/`, and the data is in `pa3/data/`. Copy the starter code to your local directory, and make sure you can compile it. Spend some time looking through the principal source files for this assignment. For the maximum entropy portion, these are:

```
src/cs224n/assignments/MaximumEntropyClassifierTester.java
```

You can do a quick test run for the first one using the command

```
java cs224n.assignments.MaximumEntropyClassifierTester -mini
```

For the parsing portion, the principal file is:

```
java/src/cs224n/assignments/PCFGParserTester.java
```

Make sure you can run the main method of the `PCFGParserTester` class. You can either run it from the data directory, or pass that directory in as the `-path` value. Use the `-data` value to specify which dataset to use. Running:

```
java -server -mx500m cs224n.assignments.PCFGParserTester
-path /afs/class/cs224n/pa3/data/parser/ -data miniTest
```

will train and test your parser on a few sentences from a toy grammar. Running:

```
java -server -mx500m cs224n.assignments.PCFGParserTester
-path /afs/class/cs2224n/pa3/data/parser/ptb/ -data treebank
```

will train and test your parser on the WSJ section of the Penn Treebank dataset.

2 A Maximum Entropy Markov Model (MEMM)

The backbone of a maximum entropy Markov model (MEMM) is the maximum entropy classifier, which you will be responsible for building. As you learned in class, an MEMM is trained in exactly the same way as a regular maximum entropy model, where each word corresponds to one datum, and the correct label for the previous word can be used in features for the current word. The primary difference comes at test time, when a Viterbi decoder (which we have provided) must be used to find the best possible sequence of labels, instead of greedily finding the best label at each point.

2.1 Building a simple maxent classifier

Look through the code in `MaximumEntropyClassifierTester.java`. This class contains several subclasses. The most important two are:

`MaximumEntropyClassifier` (implements `classify.ProbabilisticClassifier`)
`MaximumEntropyClassifierFactory` (creates the former)

Look at the `main` method to see the overall program flow. There are two modes you can run this `main` method in. If you supply `-mini` as the first command line argument, you'll get a miniature classification problem from the `miniTest()` method. I recommend working with this branch first, since it's easier to debug. `miniTest()` creates several training datums and a test datum. Each datum represents either a cat or a bear and has several features (which are just strings). The training data are passed to a `MaximumEntropyClassifierFactory` which uses them to learn a `MaximumEntropyClassifer`. This classifier is then applied to the test data, and an accuracy (and distribution over labels) is printed out.

While the starter code contains a fully-functioning pipeline for training and testing a classifier, the classifier it builds is as dumb as a rock, and does *not* use maximum entropy. Your job is to turn the placeholder code into a maximum entropy classifier by filling in the three chunks of code marked by "`// TODO`" lines . First, look at

```
MaximumEntropyClassifier.getLogProbabilities()
```

This method takes a datum, and produces the (log) distribution, according to the model, over the possible labels for the datum. There will be some interface shock here, because you're looking at a method buried deep inside the implementation of the rest of the classifier. You are given several arguments, whose classes are defined in this same Java file:

```
EncodedDatum datum
Encoding<F, L> encoding
IndexLinearizer indexLinearizer
double[] weights
```

The `EncodedDatum` represents the input datum. It is a sparse encoding, which tells you which features were present in that datum, and with what counts. When you ask an `EncodedDatum` what features are present, it will return feature *indexes* instead of feature objects — for example it might tell you that feature 121 is present with count 1.0 and feature 3317 is present with count 2.0. If you want to recover the original (`String`) representation of those features, you'll have to go through the `Encoding`, which maps between features and feature indexes. Encodings also manage maps between labels and label indexes. So while your `miniTest()` labels are "cat" and "bear", the `Encoding` will map these to indexes 0 and 1, and your returned log distribution should be a double array indexed by 0 and 1, rather than a hash on "cat" and "bear".

So, you first have a feature ("fuzzy") and a label ("cat") which map to some feature index (say 2) and some label index (say 0). As outlined above, these are managed by the `Encoding`. The job of the `getLogProbabilities()` method is to return an array of doubles, where the indexes are the label indexes, and the entries are the log probabilities of that label, given the current datum. To do this, you will need to properly combine the feature weights (the λs). The double vector `weights` contains these feature weights linearized into a one-dimensional array. To find the weight for the feature "fuzzy" and the label "cat," you'll need to take their indexes (2 and 0) and use the `IndexLinearizer` to find out what index in `weights` to use.

Your job here is to correctly fill in and return an array of log probabilities. Try to do this as efficiently as possible — this is the inner loop of the classifier training. Indeed, the reason for all this primitive-type array machinery is to minimize the amount of time it'll take to train large maxent classifiers.

(Here's a tip: you will likely find that your code spends much of its time taking logs and exps. You can often avoid a good amount of this work using the `logAdd(x, y)` and `logAdd(x[])` functions in `math.SloppyMath`.)

Run `miniTest()` again. Now that it's actually voting properly, you won't get a 0/1 distribution anymore — you'll get 50/50, because, while it is voting now, the weights are all zero. The next step is to fill in the weight estimation code. Look at

```
Pair<Double, double[]> calculate(double[] x)
```

buried all the way in

```
MaximumEntropyClassifierFactory.MaximumEntropyClassifier.ObjectiveFunction
```

This method takes a vector `x`, which is some proposed weight vector, and calculates the value of the objective function we're trying to optimize in training, along with its derivatives. Our objective function is the negative conditional log likelihood of the training data $\langle C, D \rangle$:

$$F(\vec{\lambda}) = -1 \cdot \left[\sum_{\langle c,d \rangle \in \langle C,D \rangle} \log p(c \mid d, \vec{\lambda}) \right]$$

$$p(c \mid d, \vec{\lambda}) = \frac{\exp\left[\sum_i \lambda_i f_i(c,d)\right]}{\sum_{c'} \exp\left[\sum_i \lambda_i f_i(c',d)\right]}$$

(Note that the is here correspond to the indexes generated by the `IndexLinearizer`.) The derivatives of the log likelihood are therefore:

$$\frac{\partial F(\vec{\lambda})}{\partial \lambda_i} = -1 \cdot \left[\left[\sum_{\langle c,d \rangle \in \langle C,D \rangle} f_i(c,d) \right] - \left[\sum_{\langle c,d \rangle \in \langle C,D \rangle} \sum_{c'} p(c' \mid d, \vec{\lambda}) f_i(c',d) \right] \right]$$

Recall that the left summation is the number of times the feature i actually occurs in examples with true class c in the training, while the right sum is the expectation of the same quantity using the label distributions the model predicts.

The current code just says that the objective is 42 and the derivatives are flat. Note that you don't have to guess at `x` — that's the job of the optimization code. All you have to do in the `calculate()` method is evaluate proposed `x` vectors. You have available as method arguments the data, the string-to-index encoding, and the linearizer we discussed before:

```
EncodedDatum[] data;
Encoding encoding;
IndexLinearizer indexLinearizer;
```

Write code to calculate the objective function and its derivatives, and return the `Pair` of those two quantities.

Now run `miniTest()` again. This time, the optimization should find a good solution, one that puts all of the mass onto the correct answer "cat."

Almost done! Remember that putting probability 1.0 on "cat" is probably the wrong behavior here. To smooth, or regularize, our model, we're going to modify the objective function to penalize large weights. In the `calculate()` method, you should now add code which modifies the objective function as follows:

$$G(\vec{\lambda}) = F(\vec{\lambda}) + \sum_i \frac{\lambda_i^2}{2\sigma^2}$$

The derivatives change correspondingly:

$$\frac{\partial G(\vec{\lambda})}{\partial \lambda_i} = \frac{\partial F(\vec{\lambda})}{\partial \lambda_i} + \frac{\lambda_i}{\sigma^2}$$

Run `miniTest()` one last time. You should now get less than 1.0 on "cat" (0.73 with the default sigma).

2.2 Feature engineering for a maxent classifier

You will be doing named entity recognition on biomedical data, and the task is to identify the following types of entities: `cell_line`, `cell_type`, `DNA`, `RNA`, and `protein`. Don't worry if you don't know much (or anything) about biology, you'll find that it really shouldn't matter. We have also included newswire data from the MUC shared task, which contains entities of the types `PERSON`, `LOCATION`, `ORGANIZATION`, `MONEY`, `PERCENT`, `DATE`, and `TIME`. You are absolutely *not* required to feature engineer for the newswire data, or even run your system on it, we have merely included it for those who are interested and feel like playing around with it.

Now that your classifier works, goodbye `miniTest()`! Run the `main` method with a single argument of

```
/afs/ir/class/cs224n/pa3/data/ner/genia
```

or wherever you may have copied the data. It now loads the named entity recognition data from the data directory and converts each data instance into a list of `String` features (for the newswire data, replace `genia` with `muc7`). Currently, it adds only two features: one for the current word and one for the pervious label. This should train relatively quickly (around a few minutes). It will print output to standard out, using a very similar format to the imput data. There will be one word per line, with three columns. The first column will contain the word, the second column the correct answer, and the third column will contain the answer guessed by the model. We have provided a script for computing the per-entity F-score. This script is located at `/afs/ir/class/cs224n/pa3/bin/nerEval`. The model you just built won't work well - that's where you come in!

Your job here is to flesh out the feature extraction code in

```
MaximumEntropyClassifierTester.extractFeatures(List<String> sentence, int
position, String prevLabel)
```

This `List<String> sentence` contains all words in the sentence, the `int position` tells you the position in the sentence over which you should build your features, and the `String prevLabel` tells you the label of the previous word in the sentence. There are lots of features you can add here - this is your chance to really get creative, and use the error analysis skills you've been building all quarter. You can add orthographic features, such as whether the word starts with a capital letter, or contains a number, or any hyphenation. You can use regular expressions to check if the word contains a spelled out greek letter. You can also use feature conjuctions, such as whether the word is capitalized AND the previous label. Go crazy, and try all sorts of stuff. One thing to be aware of though is sparseness - pairs of words, such as current word AND previous word, are all well and good, but they won't occur all that often and you will have a lot of them. Try to think of features which will get activated with some frequency.

3 Building A PCFG Parser

In this portion of the project, you will build a broad-coverage probabilistic parser. You're free to build a beam-decoded shift-reduce parser, or implement any other general (P)CFG parsing solution you find interesting and manageable. However, the bulk of the support code assumes that you will be building a parser where the grammar rules have been pre-transformed into exclusively unary and binary grammar rules. The easiest thing to build would be a probabilisitic generalized CKY parser. Another good thing to attempt to build is an agenda-driven PCFG parser.

At the beginning of the main method in `PCFGParserTester` some training and test trees are read in. Currently, the training trees are used to construct a `BaselineParser` that implements the `Parser` interface (which only has one method: `getBestParse()`). The parser is then used to predict trees for the sentences in the test set. For each test sentence the parse given by your parser is evaluated by comparing the constituents it generates with the constituents in the hand-parsed version. From this, precision, recall, and the F1 score are calculated.

This baseline parser is really quite poor—it takes a sentence, tags each word with its most likely tag (i.e., a unigram tagger), then looks for occurrences of the tag sequence in the training set. If it finds an exact match, it answers with the training parse of the matching training sentence. If no match is found, it constructs a right-branching tree, with nodes' labels chosen independently,

conditioned only on the length of the span of a node. If this sounds like a strange (and terrible) way to parse, it should, and you're going to provide a better solution.

You should familiarize yourself with these basic classes:

`ling.Tree`	CFG tree structures (pretty-print with `ling.Trees.PennTreeRenderer`)
`Lexicon`	Pre-terminal productions and probabilities
`Grammar, UnaryRule, BinaryRule`	CFG rules and accessors

`Tree` is a linguistic tree class that you will use no matter what kind of parser you implement. `Lexicon` in a minimal lexicon, but it handles rare and unknown words adequately for the present purposes. You can use it to determine the pre-terminal productions for your parser if you like. `Grammar` is a class you can use to learn a PCFG from the training trees. Since the training set is hand-parsed this learning is very easy. We simply set

$$\hat{P}(N^j \rightarrow \zeta) = \frac{C(N^j \rightarrow \zeta)}{\sum_\gamma C(N^j \rightarrow \gamma)}$$

where $C(N^j \rightarrow \gamma)$ is the count observed for that production in the data set. While you could consider smoothing rule rewrite probabilities, it is sufficient for this assignment to just work with unsmoothed MLE probabilities for rules. (Doing anything else makes things rather more complex and slow, since every rewrite will have a nonzero probability, so you should definitely get things working with an unsmoothed grammar before considering adding smoothing!) `UnaryRule, BinaryRule` are simply the classes the grammar uses to store these learned productions. They each bear the frequency estimated probabilities from the training set.

(If you build an agenda-driven PCFG parser, you will also find `util.PriorityQueue` useful. It is a fast priority queue implementation, but it does not support a promotion / increasePriority operation. If you want to increase the priority of an edge in the queue, you need to add that edge a second time, with the new higher priority, and be aware that whenever you pop an edge off of the queue, it might be a duplicate "dead" edge. We made this design decision because, in our experience, highly-tuned agenda-driven parsers spend a large fraction of their time dealing with the overhead that queues supporting promotion incur. But you are of course welcome to change the priority queue if you'd really like it to support promotion.)

Although it is not required, we strongly recommend that you get your parser working on the `miniTest` dataset before you attempt the `treebank` dataset. The `miniTest` data set consists of 3 training setences and 1 test sentence from a toy grammar. There are just enough productions in the training set for the test sentence to have an ambiguity due to PP-attachment. There are unary, binary, and ternary grammar rules in in training sentences.

As we discussed in class, most parsers require grammars in which the rules are at most binary branching. You can binarize and unbinarize trees using the `TreeAnnotations` class. The implementation we give you binarizes the trees in a way that doesn't generalize the grammar at all. You should run some trees through the binarization process to get the idea of what's going on. If you annotate/binarize the training trees, you should be able to construct a `Grammar` out of them using the constructor provided.

Your first job is to build a parser using this grammar. For the `miniTest` dataset your parser should match the given parse of the test sentence exactly. Once you've got this working you can move on to the `treebank` dataset.

Scan through a few of the training trees in the `treebank` dataset to get a sense of the range of inputs. Something you'll notice is that the grammar has relatively few non-terminal symbols (27

plus part-of-speech tags) but thousands of rules, many ternary-branching or longer. Currently, sections 2 through 21 of the Treebank are read in as training data, section 24 is used as validation data, and section 22 is the test data. (You can look in the data directory if you're curious about the native format of these files.) At the moment, only the training and test set are used, but you are welcome to use the validation set too if you see a use for it. The static integer `MAX_LENGTH` determines the maximum length of sentences to test on (it does not affect the training set). You can lower `MAX_LENGTH` for preliminary experiments, but your final parser should work on sentences of at least length 20 in a reasonable time (5 seconds per 20-word sentence is achievable with some optimization).

Once you have a parser working on the Treebank, your next task is improve upon the supplied grammar by adding 2nd-order vertical markovization. This means using parent annotation symbols like `NP^S` to indicate a subject noun phrase instead of just `NP`. You can test your new grammar on the `miniTest` data set if you want, though the results won't be very interesting. When you test it on the Treebank the results should be pretty impressive: a 3–4% improvement over the a parser using the original grammar.

At this point an F1 performance of 80% is probably achievable (but don't worry too much about this exact figure—it's just a ballpark).

3.1 Coding Tips

Whenever you run the java VM, you should invoke it with as much memory as you need, and in server mode:

```
java -server -mx500m package.ClassName
```

On many machines, you'll get much faster performance than just running with no options.

If your parser is running very slowly, run the VM with the `-Xprof` command line option. This will result in a flat profile being output after your process completes. If you see that your program is spending a lot of time in hash map, hash code, or equals methods, you might be able to speed up your computation substantially by backing your sets, maps, and counters with `IdentityHashMap` instead of `HashMap`. This requires the use of something like an `Interner` for canonicalization. Ask around if you're not sure what that would entail, some people in the class already know this trick.

4 Combining NER and Parsing

Luckily for you, this last section requires no coding because the glue for plugging the two systems together is provided. Your parser was trained on newswire, but your NER system was trained on biomedical data. If you try parsing biomedical data with a parser trained on newswire, you'll find it won't do so well, in part because of the large number of unknown words. We have provided an additional 500 sentences of parsed biomedical data. In this portion of the assignment, you are to test your parser on this biomedical data, and then examine the output. You will need to specify on the command line where the new test data is located:

```
java -server -mx500m cs224n.assignments.PCFGParserTester
-path /afs/class/cs2224n/pa3/data/parser/ptb/ -data treebank
-testData /afs/class/cs2224n/pa3/data/parser/gtb/
```

You should then run the system again, but where the entities in the biomedical data are first chunked together into single words. You will do this by running the parser with an additional command line arguments which tells it where the NER training data is:

```
java -server -mx500m cs224n.assignments.PCFGParserTester
-path /afs/class/cs2224n/pa3/data/parser/ptb/ -data treebank
-testData /afs/class/cs2224n/pa3/data/parser/gtb/
-nerTrainFile /afs/class/cs2224n/pa3/data/ner/genia
```

It will then train your NER system after it trains the parser, and use that NER model for chunking. **Do not pay attention to the scores that the parser gives you on this data**, as they will be meaningless on account of the fact that the actual words in the sentence are now different than those in the gold-standard tree. Your task here is to examine both sets of outputs, and see what types of errors the chunking fixes, what types it introduces, and generally how it affects the output.

5 The Requirements

For the MEMM portion, the following is expected:

- A correct implementation of a maxent classifier.
- A well-designed set of feature templates for biomedical entity classification.
- Describe the feature templates you used, and also how good and effective they are for the classification problem.

For the parsing portion, the following is expected:

- A PCFG parser, probably a CKY parser or an agenda-driven parser, that can parse sentences of at least length 20 in a reasonable amount of time (definitely under a minute!).
- An improved grammar with 2nd order vertical markovization.

And, finally, for the combination portion:

- Error analysis and an intelligent discussion of ways in which pre-chunking with NER improves or hurts the performance of the parser on biomedical text

Anything beyond this is optional.

6 Write-up

For the write-up, you should describe what you built, what choices you had to make, why you made the choices you did, how well they worked out, and what you might do to improve things further. In particular, for the maximum entropy model:

- In your implementation of the maxent model code, we'll be looking first for correctness, and secondarily for efficiency.

- For the named entity recogntion model, an important criterion will be coming up with good and effective features for this task. Describe the feature templates you used, why you choose them, and how well they worked.
- Make sure you describe any data analysis and testing you did as part of finding good features, and show any revealing data analysis on your final results which shows what it is good and bad at.

For the parser model:

- A brief description of your implementation including any important design decisions you made.
- An evaluation of successes and failures of your parser. You don't have to do a detailed data analysis, but you should give a few examples to illustrate any errors your parser makes often and describe its performance in general.
- A discussion of further improvements you might make to deal with some of the observed errors.
- An explanation of any extensions you implemented and an analysis of whether or not they improved performance.

And finally, for the NER/parser combination:

- A clear analysis with examples to illustrate how the performance changes.

7 Submitting the assignment

7.1 The program: electronic submission

You will submit your program code using a Unix script that we've prepared. To submit your program, first put all the files to be submitted in one directory on a Leland machine (or any machine from which you can access the Leland AFS filesystem). This should include all source code files, but should not include compiled class files or large data files. Normally, your submission directory will have a subdirectory named `src` which contains all your source code. When you're ready to submit, type:

```
/afs/ir/class/cs224n/bin/submit-pa3
```

This will (recursively) copy everything in *your* submission directory into the official submission directory for the class. If you need to resubmit it type

```
/afs/ir/class/cs224n/bin/submit-pa3 -replace
```

We will compile and run your program on the Leland systems, using `ant` and our standard `build.xml` to compile, and using `java` to run. So, please make sure your program compiles and runs without difficulty on the Leland machines. If there's anything special we need to know about compiling or running your program, please include a `README` file with your submission. Your code doesn't have to be beautiful but we should be able to scan it and figure out what you did without too much pain.

7.2 The report: turn in a hard copy

You should turn in a write-up of the work you've done, as well as the code. **Your write-up must be submitted as a hard copy.** There is no set length for write-ups, but a ballpark length might be 6 pages, including your evaluation results, a graph or two, and some interesting examples. They are to be turned in to a box outside of Professor Manning's office.