Homework #4: CMPT-413

Due in class on Mar 18, 2003

Anoop Sarkar – anoop@cs.sfu.ca

(1) (30pts) Consider the following CFG G with start symbol S that can generate the question: *what does Calvin like* and also the statement: *Calvin likes ice-cream* (among the other sentences it can generate)

$$
\begin{aligned}
S &\rightarrow \textit{WHNP does S} \\
S &\rightarrow \textit{NP VP} \\
\textit{VP} &\rightarrow \textit{V} \\
\textit{VP} &\rightarrow \textit{V NP} \\
\textit{WHNP} &\rightarrow \textit{what} \\
\textit{NP} &\rightarrow \textit{Calvin} \mid \textit{ice-cream} \\
\textit{V} &\rightarrow \textit{like} \mid \textit{likes}
\end{aligned}
$$

a. (10pts) Replace the rule $VP \rightarrow V$ in the CFG G with the two new rules shown below.

$$
\begin{aligned}
\textit{VP/NP} &\rightarrow \textit{V NP/NP} \\
\textit{NP/NP} &\rightarrow \epsilon
\end{aligned}
$$

These rules introduce two new non-terminals *VP/NP* and *NP/NP*. Modify the rest of the CFG G so that it can continue to generate the two sentences: *what does Calvin like* and *Calvin likes ice-cream*. Provide the new CFG.

b. (20pts) Provide a CFG based on the CFG you provided in Question 1a that can *accept* sentences listed below that are *not* marked with $*$ and *does not* accept sentences listed below that *are* marked with $*$:

Calvin likes ice-cream and Hobbes

Calvin likes ice-cream and hates beans

what does Calvin like and Hobbes hate

$*$ what does Calvin like and Hobbes hate beans

$*$ what does Calvin like ice-cream and Hobbes hate

Could you have provided an answer to this question without the modification proposed in Question 1a. Why not? Notice that the notion of *context-free*-ness impinges on your answer.

Incidentally, this constraint on sentences with conjunctions (e.g. *and, or,* and *but*) was discovered by John Ross in his 1967 MIT Linguistics PhD thesis. He called it the Across-the-Board constraint. The particular solution you are providing was first proposed by Gerald Gazdar in his 1981 paper in the journal *Linguistic Inquiry*.

(2) (160pts) Consider the sentence *Calvin saw the man with the telescope* which has two meanings, one in which Calvin sees a man carrying a telescope and another in which Calvin using a telescope sees the man. These two meanings can be represented by the derivations possible in the following context-free grammar (CFG) rules. This ambiguity between the *noun-attachment* versus the *verb-attachment* is called *PP-attachment ambiguity*.

$$
\begin{aligned}
S &\rightarrow SBJ\ VP \\
VP &\rightarrow V\ NP \\
VP &\rightarrow VP\ PP \\
NP &\rightarrow NP\ PP \\
NP &\rightarrow Det\ N \\
PP &\rightarrow P\ NP \\
SBJ &\rightarrow Calvin \\
V &\rightarrow saw \\
Det &\rightarrow the \\
N &\rightarrow man \mid telescope \mid hill \\
P &\rightarrow with \mid on
\end{aligned}
$$

Let S be the start symbol.

a. (10pts) Provide the set of non-terminal symbols and the set of terminal symbols for the above CFG.

b. (10pts) Using the above CFG draw the derivations possible for the input: *Calvin saw the man with the telescope on the hill*.

c. (140pts) The above CFG can be represented as Perl data in the following manner:

```
our (@start, %cfgrule, %cfgpos);

@start = ('S');
$cfgrule{'S'}   = [ ['SBJ', 'VP', '#' ] ];
$cfgrule{'VP'}  = [ ['V', 'NP', '#' ], ['VP', 'PP', '#' ] ];
$cfgrule{'NP'}  = [ ['NP', 'PP', '#' ], ['Det', 'N', '#' ] ];
$cfgrule{'PP'}  = [ ['P', 'NP', '#' ] ];
$cfgpos{'SBJ'}  = [ ['Calvin', '#' ] ];
$cfgpos{'V'}    = [ ['saw', '#' ] ];
$cfgpos{'Det'}  = [ ['the', '#' ] ];
$cfgpos{'N'}    = [ ['man', '#' ], ['telescope', '#'], ['hill', '#'] ];
$cfgpos{'P'}    = [ ['with', '#'], ['on', '#'] ];
```

The `%cfgrule` hash table contains the non-lexical rules (i.e. rules that do not contain any terminal symbols) while the `%cfgpos` hash table contains rules of the form $A \rightarrow a$ where A is a non-terminal and a is a terminal symbol signifying the part of speech tag rules in the grammar.

Using the above Perl representation of a CFG, implement the Earley parsing algorithm `earleyParse` as a Perl program. The Earley algorithm is described in Figure 10.16 on page 381 in the Jurafsky and Martin textbook.

Hint: You can represent the *state* data structure described in the algorithm as follows: A state: $(A \rightarrow B \bullet C, [5, 7])$ can be represented in Perl as
`$state = [ 'A', [ 'B', 'C', '#'], 1, 5, 7 ];`
where `$state->[0]` is the left hand side of the CFG rule, `$state->[1]` is a reference to a list of right hand side symbols for the CFG rule, `$state->[2]` is the location of the dot in the right hand

side, in this example, since the dot position is 1 the dot is immediately to the left of `'C'` in the right hand side of the rule, and finally `$state->[3]` and `$state->[4]` are the i, j values for the span of the dotted rule in the input string: $5, 7$ in this case. If you represent the state in this manner the following functions might be useful to you:

```
sub incomplete {
    my ($state) = @_;
    return (($state->[1]->[$state->[2]] eq '#') ? 0 : 1);
}

sub nextCat {
    my ($state) = @_;
    return ($state->[1]->[$state->[2]]);
}
```

How you implement the chart data structure in the implementation of `earleyParse` is up to you. The implementation `earlyParse` returns the chart after parsing of the input has finished. We can use the chart to decide whether the input string was accepted by the CFG or not. The following code is an example of how we can check the chart to see if a start symbol successfully spans the entire input string. Note that this code assumes that states are represented as described above. It also assumes that the chart is implemented as a reflist of reflists containing the states.

```
my @input = qw(Calvin saw the man with the telescope);
my $chart = earleyParse(@input);
my $length = $#input+1;
my $yes = 0;
for my $start (@start) {
    for my $finalState (@{$chart->[$length]}) {
        $yes = 1 if (parseSuccess($finalState, $start, 0, $length));
    }
}
print (($yes) ? "yes" : "no", "\n");

sub parseSuccess {
    my ($finalState, $start, $begin, $end) = @_;
    return (($finalState->[0] eq $start) and
            (! incomplete($finalState)) and
            ($finalState->[3] == $begin) and
            ($finalState->[4] == $end));
}
```

A sample final state for the input *Calvin saw the man with the telescope* which would succeed in this test (given the CFG above) is the state ($S \rightarrow SBJ\ VP\bullet, [0, 7]$) or in the Perl equivalent:
`['S', ['SBJ', 'VP', '#'], 2, 0, 7]`

Provide the Perl code for your implementation of the Earley parsing algorithm. Also provide a trace of your recognition algorithm for 3 sentences accepted by your program, and 3 sentences that are not accepted. Your trace should look like the traces supplied in the files `trace-1.txt` and `trace-2.txt`.

The Earley algorithm was proposed by J. Earley in his 1968 CMU CS PhD thesis as a more elegant form of the Cocke-Younger-Kasami parsing algorithm which was the first polynomial time parsing algorithm for CFGs. The Cocke-Younger-Kasami algorithm was independently discovered by the three people mentioned in its name and is often referred to as the CKY or the CYK algorithm. Both the Earley algorithm and the CKY algorithm take worst case time: $O(n^3)$ where n is the length of the input string.

(3) (210pts) In order to decide which of the two derivations in Question 2 is more likely, a machine needs to know a lot about the world and the context in which the sentence is spoken. Either meaning is plausible for the sentence *Calvin saw a man with the telescope* given the right context. However, in many cases, some of the meanings are more plausible than others even without a particular context. Consider, *Calvin bought a car with anti-lock brakes* which has a more plausible noun-attach reading, while *Calvin bought the car with a low-interest loan* which has a more plausible verb-attach meaning.

In this question, we will write a program that can *learn* which attachment is more plausible. In order to keep things simple, we will only use the verb, the final word (as a *head* word) of the noun phrase complement of the verb, the preposition and final word of the noun phrase complement of the preposition. For the sentence *Calvin saw a man with the telescope* we will use the words: *saw, man, with, telescope* in order to predict which attachment is more plausible. We will also consider only the disambiguation between noun vs. verb attachment for sentences or phrases with a single PP (we do not consider the more complex case in Question 2b).

Our *training data* (stored in the file `training.gz`) will be a set of 5-tuples as shown below. For each $v, n1, p, n2$ which could either have a verb-attachment (V) or a noun-attachment (N, i.e. n1), we have the correct attachment in each case provided by a human expert. After solving each step in this question, you will have a program that can predict for new examples of $v, n1, p, n2$ what the most plausible attachment should be in that case. Note that there are only two labels to predict: N or V and so this is a learning problem commonly referred to as a two-class classification problem.

v	n1	p	n2	Attachment
join	board	as	director	V
is	chairman	of	N.V.	N
using	crocidolite	in	filters	V
bring	attention	to	problem	V
is	asbestos	in	products	N
making	paper	for	filters	N
including	three	with	cancer	N
⋮	⋮	⋮	⋮	⋮

In order to learn attachment decisions from this data, we will use a learning algorithm called *Error-Driven Transformation-Based Learning* (TBL). Download the file `ppTBL-inc.pm` which is a Perl module that contains most but not all the functions you will need to train the learner and apply it to new test cases. Copy this file to the file `ppTBL.pm` which you will then add to in the following steps.

a. (10pts) TBL works by starting with an initial guess at the answer. Since we know the answers to the cases in the training data, we will start by learning from this data. First we need to provide an initial guess for each case in the training data. Write a Perl program `hw4_3a.pl` which will use the module `ppTBL.pm`. *Do not modify the file* `ppTBL.pm` *yet*. In `hw4_3a.pl` use the function `readCorpus` to read in the training data `training.gz`. `readCorpus` returns a reference to a list containing the training data examples.

   Use the function `applyDefaultRule` to make the initial guess over the training data. `applyDefaultRule` takes two arguments, a reference to the training data and the guess (`"N"` or `"V"` in this case).

   Your task is to provide the accuracy of this default rule on the training data. Use the function `evaluate` which takes two arguments: the training data and the guess. Which guess gives the higher accuracy and why?

b. (10pts) Let's assume we start with the guess (from Question 3a) that gives the higher initial accuracy on the training data. Once it has a guess, TBL then tries to improve this guess by learning *transformation rules* of the following type:

```
IF rule-condition is applicable
THEN change the current guess to one supplied in the training data
```

For this problem, we will always be changing our guess from N to V or vice versa. The *rule conditions* we will use are the words in the fields $v, n1, p, n2$. Just as in earlier n-gram models, we are unlikely to see all four words in unseen data and so we generate rule conditions which match supersets of the set containing all four words. The rule conditions we use are:

- $(v, n1, p, n2)$ (4 words)
- $(v, n1, p)$ and $(v, p, n2)$ and $(n1, p, n2)$ (3 words)
- (v, p) and $(n1, p)$ and $(p, n2)$ (2 words)
- (p) (1 word)

Assume the current guess is V, give the two rules that would disambiguate the attachment for the examples mentioned above: *(bought, car, with, brakes)* and *(bought, car, with, loan)*.

The function `findAllRuleConditions` from `ppTBL.pm` extracts all rule conditions of the above types from the training data. It takes the reference to the training data as input, and returns two scalar variables, `$ruleConditions` containing a reference to the rules (each rule has a ruleIndex) and `$rulesApplicable`, a reference to an array where for each ruleIndex we store a list of indexes of each training data example to which the rule condition applies. We will use this function in the following question.

c. (160pts) Now we are in a position to implement the TBL algorithm. The TBL algorithm tries to find transformation rules that *transforms* the current guess to a new one by matching the rule condition to the example in question. It iteratively finds the rule with the least error on the training data (where we know the right attachment). The algorithm works as follows:

algorithm Transformation-Based Learning

1. Apply the default rule to find the first guess for each example in the training data (from Question 3a)
2. Compute the accuracy over the training data for the current guess
3. If the accuracy did not change from the previous iteration (see Step 6) or if the maximum number of rules to be extracted from the training data has been reached then stop and return the list of transformation rules found (see Step 4b).
4. a. Find the transformation rule that has the least error over the training data (see function `getBestInstance` below)
   b. Add the transformation rule that was found to the list of transformation rules found so far.
5. Apply the transformation rule obtained in Step 4 to each example in the training data changing the current guess to obtain the new current guess whenever the rule condition is satisified (see function `applyRule` below)
6. Save the current accuracy and go back to Step 2

The above algorithm uses a couple of auxiliary functions: `getBestInstance` which reports the rule with the lowest error on the training data and `applyRule` which takes the best rule in each iteration and applies it to the previous guess to get the new guess over the training data.

These two functions are described in detail below:

function `getBestInstance` {
  Input: the current guess, the training data, and output from `findAllRuleConditions`
  let the current best rule `bestRule` = undefined
  let the current best accuracy `best` = -99999
  for each rule condition `c` obtained from `findAllRuleConditions` {
    for each example `i` in the training data {
      let `guess` be the current guess for example `i`
      let `newTag` = $\begin{cases} \text{V} & \text{if guess for example i is N} \\ \text{N} & \text{otherwise} \end{cases}$
      create rule `r` =
        "`If c applies to training example (v,n1,p,n2)`
          `then change the attachment from guess to newTag`"
      If the current guess is not equal to the attachment in the training data
        then increment the number of times rule `r` was applied correctly
        else increment the number of times rule `r` was applied incorrectly
    }
    for each rule `r` observed above {
      let accuracy(`r`) = correct applications of `r` − incorrect applications of `r`
      if accuracy(`r`) is greater than the current best accuracy `best`
        then let `bestRule` = `r` and let `best` = accuracy(`r`)
    }
  }
  returns: `bestRule`
}

function `applyRule` {
  Input: the training data, the current guess, the transformation rule to apply and
    the output from `findAllRuleConditions`
  `newGuess` = the current guess
  for each example `i` in the training data
    if the transformation rule applies
    then change the current guess for example `i` to the output of the rule
  returns: `newGuess`
}

First implement the two functions `getBestInstance` and `applyRule` and add them to the file `ppTBL.pm`. *Use* `$rulesApplicable` *from Question 3b to improve the speed of these two functions*. Looping over the entire training data for each rule is very time consuming. You will need to replace the lines which iterate over each example in the training data in the above pseudo-code with something that uses `$rulesApplicable` (read the comments in the source for `ppTBL.pm`).

Then write a Perl program in a file `hw4_3c.pl` which implements the TBL algorithm explained above. Run it on the training data `training.gz` and extract at least 175 transformation rules. Save these rules to a file, one rule per line. The rules file should look like this:

```
45 [^\s]+ [^\s]+ to [^\s]+ N V
333 [^\s]+ [^\s]+ at [^\s]+ N V
37 [^\s]+ [^\s]+ in [^\s]+ N V
650 [^\s]+ [^\s]+ from [^\s]+ N V
```

⋮

The first column is the rule index, the second through the fifth column is the rule condition, and the rule changes the attachment from the value in the sixth column to the value in the seventh column.

Provide the Perl code for the functions `getBestInstance` and `applyRule` and your Perl program `hw4_3c.pl`. Also provide the first 20 rules extracted by your program and each rule's accuracy on the training data (computed in Step 2 of the TBL algorithm).

d. (30pts) The output from your program `hw4_3c.pl` is an ordered list of rules that can now be applied to unseen data to predict the most plausible attachment decision given the four words *(v,n1,p,n2)*. Use the function `readCorpus` to read in our test data `devset.gz`. Then read in the list of rules stored in a file in the format specified in Question 3c using the function `readListOfRules`.

The guess from the trained TBL program is obtained by first applying the default guess and then applying each rule in turn from the ordered list of transformation rules. The final guess is obtained when all the rules have been applied. You can use the function `applyListOfRules` to apply your list of rules to the unseen test data which will provide the final guess of the trained TBL program. This function takes three arguments: the corpus, the rules read by `readListOfRules` and the maximum number of rules to apply. Then compute the accuracy of this method using the `evaluate` function as you did before except this time on the test data.

Starting with the application of one rule and going up to all the rules in your rules file, provide a graph of the accuracy on the test data for each value of the number of rules applied. It should look like the graph shown in Figure 1 (plotted for 400 rules here, your max number of rules can be 175).

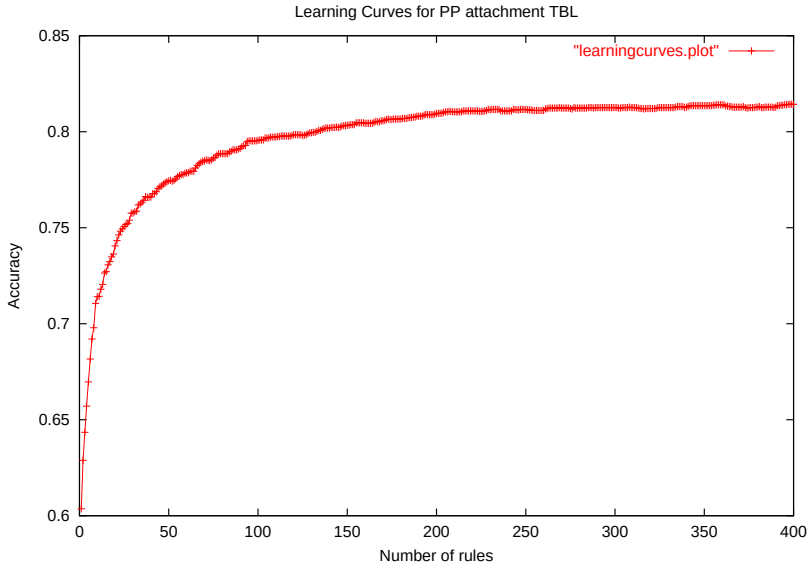

Figure 1: Learning curves for the TBL algorithm running on test data.

e. *(Optional)* Get an accuracy of greater than or equal to 88% on the test data `test.gz` using only the training data and get an automatic A+ for this course. No cheating of any kind is allowed including training on the test data in any way. First test your accuracy on the file `devset.gz` before attempting `test.gz`.

The error-driven transformation-based learning algorithm described above was proposed within the Computational Linguistics community by Eric Brill in a 1995 paper. It was initially proposed for part of speech tagging, although the learning algorithm has been adapted for various tasks since then. It was the first algorithm to successfully beat the trigram model combined with interpolation smoothing in the part of speech tagging task on the Penn Treebank. Some have noticed similarities between TBL and the General Problem Solver (GPS) algorithm proposed by Newell and Simon which uses means-ends analysis as a general strategy for learning in AI.