Homework #8: CMPT-413

Distributed on Mon, Mar 22; Due on Mon, Apr 5

Anoop Sarkar – anoop@cs.sfu.ca

(1) (Submit files: `ppTBL.pm`, `run-ppTBL.pl`, `test-ppTBL.pl`, `test-ppTBL.eps/pdf`)
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. This ambiguity between the *noun-attachment* versus the *verb-attachment* is called *PP-attachment ambiguity*.
In order to decide which of the two derivations 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 with more than one PP.
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) which is explained for part-of-speech tagging in Section 8.6 (p.307) of the Jurafsky & Martin textbook. You can refer to the textbook for a general description, but for your implementation use the detailed description in this homework.
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. 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 `run-ppTBL.pl` which will use the module `ppTBL.pm`. *Do not modify the file* `ppTBL.pm` *yet*. In `run-ppTBL.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 first task is to make `run-ppTBL.pl` print out the accuracy of this default rule on the training data. Use the function `evaluate` which takes two arguments: the training data and the guess. Pick the guess that gives the higher accuracy.

b. Let's assume we start with the guess (from Question 1a) 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 for *(bought, car, with, brakes)* the rule might be *if n2=brakes then change V to N* and if the current guess is N for *(bought, car, with, loan)* the rule might be *if p=with and n2=loan then change N to V*.

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.

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 1a)
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.

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* `$rulesApplicable` *from Question 1b to improve the speed of these two functions*. Looping over the entire training data for each rule is very time consuming. You will need to replace the lines which iterate over each example in the training data in the above pseudo-code with something that uses `$rulesApplicable` (read the comments in the source for `ppTBL.pm`).

Then implement the TBL algorithm explained above in the file `run-ppTBL.pl`. 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.

c. The output from your program `run-ppTBL.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)*. Write a new Perl program called `test-ppTBL.pl` to apply the learned rules to unseen data. 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 1b 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. Use the function `applyListOfRules` as a guide to write a new function `applyListOfRulesAndEvaluate` to apply your list of rules to the unseen test data which will provide the final guess of the trained TBL program and evaluate the accuracy on the test data after each rule application. An example of using `applyListOfRules` using 10 rules is as follows:

```
my $corpus = ppTBL::readCorpus("test.gz");
my $TBLrules = ppTBL::readListOfRules("rulesfile");
my $guess = ppTBL::applyListOfRules($corpus, $TBLrules, 10, 'N');
my $accuracy = ppTBL::evaluate($corpus, $guess);
print "accuracy on test=$accuracy\n";
```

Starting with the application of zero rules (using only the default) 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 in the file `test-ppTBL.eps/pdf`. It should look like the graph shown in Figure 1 (plotted for 400 rules here, your max number of rules can be 175).

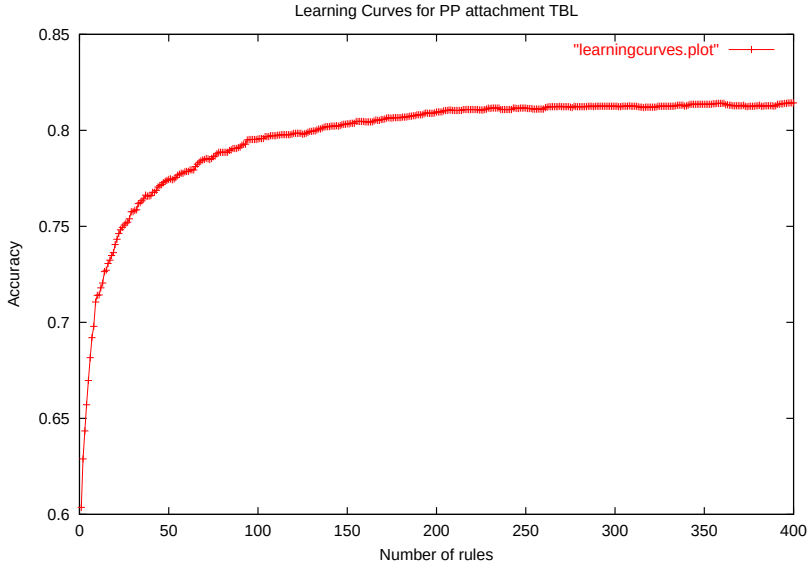

Figure 1: Learning curves for the TBL algorithm running on test data.

d. *(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 or devset data in any way. You can use any of the methods covered in this course. 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 and 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.