CS182 Assignment 7 (Computational): Model Merging

Assigned Tuesday, March 13th

Due Tuesday, April 3rd, by 11:59pm

Model merging is a technique for building complex structured models of some process from data. The idea behind model merging is to begin with a completely unstructured model and then to incrementally modify it to reflect regularities in the data. In doing this, there is a tradeoff between the simplicity of the model and how well it accounts for the data.

In this assignment you will apply a simplified version of the model merging algorithm to the task of learning right-regular grammars.

1 Right-regular grammars

- A *grammar* is a tuple $G = (Nonterminals, Terminals, Rules, StartSymbol)$, where $Nonterminals$ or $Terminals$ are sets of symbols used to write the rules in $Rules$, and $StartSymbol$ is a unique start symbol from $Nonterminals$. Each *nonterminal* symbol can be expanded according to any rule in the grammar that has that symbol on the left side. *Terminal symbols* cannot be expanded and appear only on the right side of rules in the grammar.
- A *right-regular* grammar is one in which every rule looks either like

  $$A \rightarrow x_1\ x_2\ \ldots\ x_n\ B$$

  or like

  $$A \rightarrow x_1\ x_2\ \ldots\ x_n$$

  where the x_i are terminal symbols, and A and B are nonterminal symbols. That is, each rule has one nonterminal symbol on the left side; its expansion on the right side contains any number of terminal symbols followed by at most one nonterminal symbol.

An example grammar G_0 has $Nonterminals$ = {S,Y}, $Terminals$ = {the, dog, ran, away, down, the, street}, $StartSymbol$ = S, and $Rules$ consisting of the following:

S → the dog ran Y
Y → away
Y → down the street

G_0 produces exactly two distinct sentences.

2 The model-merging algorithm

A simplified greedy version of the model-merging algorithm presented in class can be applied to learn a right-regular grammar G that produces a given set of sentences $Data$. The algorithm has two main procedures:

- **Data incorporation**: Given a sentence from $Data$, add a set of rules to the grammar that generates this sentence from the start symbol. For example, the sentence 'the boy eats carrots' would be incorporated by adding the rule

$$\text{S} \rightarrow \text{the boy eats carrots}$$

- **Merging**: Find and perform a merge of grammar rules that improves the grammar by decreasing its *cost*; stop when no such merge can be found. The next two sections describe how merging works (Section 3) and how the cost of a grammar is calculated (Section 4).

The algorithm proceeds as follows:

1. Initialize the grammar by incorporating each sentence of $Data$, as shown above.
2. Find the cost reduction of each potential merge using your prefix and suffix tree data structures.
3. If at least one of the merges decreases the cost of the grammar on the data, perform the one that reduces the cost the most. If so, return to step 2. Otherwise, terminate.

3 Merging operations

We will incrementally improve the grammar by using two types of merge operations:

- A *prefix merge* operates on a set of rules with the same left side, and whose right sides share a common prefix. For example, the rules

$$\begin{aligned} N &\rightarrow a\ b\ c\ d \\ N &\rightarrow a\ b\ c\ e\ X \\ N &\rightarrow a\ b\ c\ f\ g \end{aligned}$$

  would be replaced, in a prefix merge, by:

$$\begin{aligned} N &\rightarrow a\ b\ c\ N' \\ N' &\rightarrow d \\ N' &\rightarrow e\ X \\ N' &\rightarrow f\ g \end{aligned}$$

- A *suffix merge* operates on a set of rules whose right sides share a common suffix. For example, the rules

$$\begin{aligned} N_1 &\rightarrow a\ b\ c\ d\ e \\ N_2 &\rightarrow f\ c\ d\ e \end{aligned}$$

  would be replaced, in a suffix merge, by:

$$\begin{aligned} N_1 &\rightarrow a\ b\ N' \\ N_2 &\rightarrow f\ N' \\ N' &\rightarrow c\ d\ e \end{aligned}$$

4 Calculating the effect of merging

The full-fledged Bayesian version of the model-merging algorithm is based on *maximizing* the *posterior probability* of the model given the data. In this assignment, however, we use a simpler *cost* function, which we attempt to *minimize*. Since we take a greedy approach to model merging, any merge that decreases the cost is accepted.

The *cost* of a grammar G given data D is defined as:

$$c(G) = \alpha s(G) + d(G, D)$$

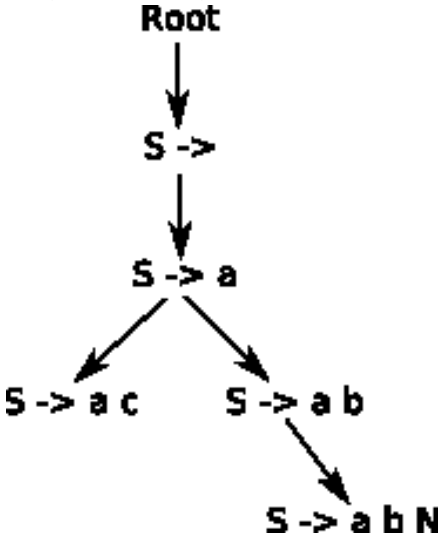
Here is an explanation of the terms in this function:

- $s(G)$ denotes the *size* of the grammar, and is equal to the number of symbols (i.e., terminals plus nonterminals) occurring on the right side of rules in the grammar, counting repeats. So, for example, the size of the simple grammar G_0 from Section 1 would be 8.
- $d(G, D)$ denotes the *total derivation length* of the data given this grammar, which is a reflection of how well the model accounts for the data. The derivation length of a sentence is defined to be the number of rules that have to be applied to the start symbol S to produce the sentence. The total derivation length of the data is the sum of the derivation lengths of all the sentences.
- α is a constant indicating the importance of the grammar size (relative to the total derivation length) and can be seen as controlling the algorithm's tendency to generalize. (A smaller grammar tends to generalize better to new examples, and a larger α leads to a greater cost being associated with large grammars.)

One of the good things about this cost function is that it allows us to efficiently calculate the effect of different merges without actually applying them and reparsing the data each time.

Affix trees

Affix trees (prefix and suffix trees) are data structures that make the cost change calculation much easier and more efficient. The idea is that a prefix tree has nodes that represent the beginning of a rule, and its leaves are the actual rules. Each node represents the first few symbols of a rule; some nodes represent full rules, while others only represent the beginnings of rules. For instance, if we have the rules $S \rightarrow abN$, $S \rightarrow ab$, and $S \rightarrow ac$, then we would construct the following prefix tree:



The advantage of this representation is that it allows us to easily compute both the potential merge points and the cost of the merge. The first part is easy: any node of the tree (excluding the root and leaves) is a potential merge point, where this prefix becomes its own rule, with a variable at the end of the rule that can expand into the children of this node.

The second part, that costs are easy to compute, is a bit harder to see. Suppose that at each node, we note down the number of rules that start with this prefix, and the total number of times those rules are used. Then, let's work with a particular node, with a prefix length of k, containing m rules. We will want to store in the node both the prefix and the set of rules that start with that prefix. By accessing the objects representing those rules, we can find out that they are each l_i long and used r_i times (where i is the number of the rule).

We can calculate how much the grammar cost changes if we perform a particular merge, and you are asked below to calculate how much it changes. We can calculate the cost of a merge from the relevant rules, and we can store the relevant rules in each node of the affix tree, so the result is that we can calculate the optimal merge just by looking at all the nodes in the affix trees (both the forward and backward ones).

Of course, the same trick can be done the other way around; it works exactly the same way, except that we arrange our tree by the ENDINGS of rules, rather than their beginnings. This is done so that we can extract the merges that come at the end of rules, just as the previous data structure allowed us to extract the merges at the beginnings of rules.

Problem 1. Cost change for prefix merges

Consider the set of rules

$$\begin{aligned} N &\rightarrow x_1\ x_2\ \ldots\ x_k\ y_{11}\ y_{12}\ \ldots\ y_{1l_1} \\ N &\rightarrow x_1\ x_2\ \ldots\ x_k\ y_{21}\ y_{22}\ \ldots\ y_{2l_2} \\ &\ldots \\ N &\rightarrow x_1\ x_2\ \ldots\ x_k\ y_{m1}\ y_{m2}\ \ldots\ y_{ml_m} \end{aligned}$$

(Don't be put off by the notation — it's just a precise way of expressing that these m rules share a common prefix of length k, followed by strings of varying length l_i.)

(a) Using this notation, write down the new rules that would result from applying a prefix merge.

(b) Calculate the change in size resulting from this merge (your answer should depend on k and m but not on the l_i).

(c) Assume now that the i^{th} rule in the original grammar was used r_i times when parsing the data. Calculate the number of times each rule in the modified grammar would be used.

(d) Use the result of part (c) to calculate the change in the total derivation length that would result from this merge.

(e) Show that the total change in cost is $\alpha(k + 1 - mk) + \sum_{i=1}^{m} r_i$.

Problem 2. Cost change for suffix merges

We can do the same thing for suffix merges. Consider the rules

$$\begin{aligned} N_1 &\rightarrow y_{11}\ y_{12}\ \ldots\ y_{1l_1}\ x_1\ x_2\ \ldots x_k \\ N_2 &\rightarrow y_{21}\ y_{22}\ \ldots\ y_{2l_2}\ x_1\ x_2\ \ldots x_k \\ &\ldots \\ N_m &\rightarrow y_{m1}\ y_{m2}\ \ldots\ y_{ml_m}\ x_1\ x_2 \ldots x_k \end{aligned}$$

Perform a similar analysis as in the previous problem and show that the change in cost after applying a suffix merge to these rules is $\alpha(m + k - mk) + \sum_{i=1}^{m} r_i$.

Problem 3. Implementing the model-merging algorithm

Use the skeleton code as the guide for implementation. (You must have a ModelMerge.main() method which can take in the arguments listed in the ModelMerge class comments; a basic one is modifiable. Also, to make grading easier, you should use MergeOutput.printMerge() to output each merge you actually perform. Everything else is modifiable, though we don't recommend deviating too far from the template.) Use your implementation to learn a grammar for a sample corpus. You may use the example data provided on the course website, or you may construct your own dataset.

To ease and speed grading, we are looking to force all correct solutions to induce the same grammar for a given ordered set of input. To do this, we have identified three potential (though probably rare) situations in which your algorithm might have a choice. We have the following fixes for each:

> Situation 1: Your best possible merge after new input produces a grammar of cost equal to not doing the merge
> Response: Don't merge
>
> Situation 2: After new input, merges of different lengths (like a prefix merge of length=2 vs a prefix merge of length=3) produce grammars of equal cost
> Response: Do the merge of shorter length
>
> Situation 3: After new input, merges of equal length, but opposite type (prefix vs suffix) produce grammars of equal cost (but better than the grammar without the merge)
> Response: Do the prefix merge
>
> (If you think of any other choice points, email us, and well specify the fix on the newsgroup.)

(a) Briefly describe your overall design, noting especially any modifications to the suggested classes.

(b) Experiment with several different values of α in the cost function. Give a qualitative description of the differences in the results they produce.

(c) Describe your Grammar.performMerge(Merge m) method. What special cases did you encounter? Why did it get complicated?

What to submit

- all your code
- a test run on some **small** examples that shows your algorithm performing merges and decreasing the grammar cost, in `test.txt`
- answers to the questions in this handout, in `answers.txt`

Submit all of the above electronically according to the standard procedures.