Distributed Expectation Maximization

Dartmouth College

Jake Leichtling ‘14 and Derek Salama ‘14

Computational Linguistics (COSC 73), Fall 2013

Professor Sravana Reddy

Motivation

Expectation-maximization (EM) is an iterative method for estimating the parameters of hidden markov models so as to maximize the probability of a set of emission sequences under the model. As we have seen in class, the algorithm is extremely useful for determining the most likely sequence of hidden states that produce an observable outcome—for example, tagging each word in a sentence with a part of speech. The algorithm employs dynamic programming to increase its efficiency but remains quite computationally expensive. By distributing the expectation step across multiple machines, EM can be sped up by a constant factor proportional to the degree of parallelism.[1] Our goal for this project is to provide a distributed EM implementation for researchers that can run on Amazon Elastic MapReduce (or any Hadoop cluster) with a minimal learning curve.

Previous Work

The EM algorithm fits naturally within the MapReduce framework for distributed computation. Even when running on a single machine, multi-threaded approaches to expectation maximization have shown significant speedups (Sand, Pederson and Mailund 2010). In *Data-Intensive Text Processing with MapReduce*, Jimmy Lin and Chris Dyer describe the theory behind a distributed EM algorithm. Further optimizations to this approach have been explored, such as the work on reducing memory requirements for latent variable models by Yang et. al.

Mahout, an open source library for scalable machine learning algorithms, provides an implementation of the Baum-Welch algorithm. We found Mahout unsatisfactory for several reasons. First and foremost, one must download and compile the complete Mahout source code

[1] See appendix for a more in depth description of the time and space complexity.

before using it. Secondly, the Baum-Welch algorithm seemed only accessible through the application programming interface, rather than through command line arguments. We hoped to provide a simple, precompiled Hadoop executable that could be run on an existing cluster with minimal setup or programming experience.

Implementation Overview

Our MapReduce implementation of distributed EM can be broken down into three principle Java classes: EMDriver, ExpectationMapper, and MaximizationReducer. As the name indicates, EMDriver contains the main method that drives the entire EM process. The main method first parses the command line arguments provided by the user to determine the locations of the input corpora, parameter, and output files. It parses the parameter files, randomly seeds the model with normalized probabilities, and outputs the seeded parameters to be used by EM. The main method then creates a MapReduce job configuration, specifying ExpectationMapper and MaximizationReducer as the map and reduce classes, respectively. Additionally, the driver indicates where the job will find the input and parameters, and where it should place its output.

The driver then starts the job and passes control to the ExpectationMapper class, which begins its work by calling the configure method. This method opens the parameter files pointed to by the job configuration and parses them into transition and emission probability dictionaries. Next, the map method performs expectation: Each map call receives a single line of input, for which it calculates the forward and backward matrices and the expected transition and emission counts. The map output collector, provided by the Hadoop framework, accepts each transmission and emission as a single record. The key for each record is the state on which the probability is conditioned, allowing the reducer, which receives all records with a given key, to normalize the probabilities for each state. Additionally, each mapper outputs a special record that contains the alpha for its input line.

Once all the map tasks have completed, the reducers are invoked. As mentioned previously, every reducer task receives an iterator for a given key containing all records collected with that key in the map phase. Each reducer combines the expected counts for its key and normalizes such that the probabilities of all transitions from the key state sum to one, and the same for all emissions from the key state. At this point, the reducer outputs a record for each transition and emission probability that will be used as a parameter for the next EM iteration. Additionally, the reduce task that receives the alpha records multiplies all of the alphas and writes the total alpha for the input corpora to a special file to be read by EMDriver.

After the reduce tasks have run, control returns to EMDriver, which reads the special alpha file. The main method compares this alpha to the alpha from the previous EM iteration, and if the difference between the two is less than the alpha convergence provided as a command line flag, then the EM process is finished! Otherwise, the main method configures another EM iteration, this time pointing the mappers to the parameters output by the previous iteration, and the process repeats.

When alpha convergence is reached, the main method of EMDriver optionally configures and runs a MapReduce job to produce a Viterbi tagging of the input corpora. There is no reducer for this job, and the mapper, which can be found in the ViterbiMapReduce class, calls a configure method identical to that found in the EM mapper to initialize its transition and emission probability dictionaries. Each map method call then performs the Viterbi algorithm for a given input line and outputs the line in its final form. We prefix each Viterbi tagged-line with its byte offset in the original input file so that the tagged output lines can be rearranged to their original input order.

Results

Our distributed EM program did not perform as well as we originally hoped. To evaluate our program, we performed a Viterbi part of speech tagging on the Penn Tree Bank corpus, then compared the results to the gold-standard tagging. The corpus consisted of 40,250 word tokens over 2,797 lines. The 36 unique part of speech tags correspond to 36 states, with 1332 possible transitions and 283,716 possible emissions. In addition to the corpus, we provided our program with all possible transitions and emissions. To find which part of speech tags corresponded with our hidden states, we mapped each state to the part of speech tag it corresponded most frequently with. With this mapping between part of speech tags and states, we checked how frequently the Viterbi output matched the gold standard tagging. Unfortunately, our algorithm only managed 19.73% accuracy. These results are well below the 40-62% accuracy Johnson achieved with expectation maximization on the same corpus. By constricting possible emissions to those observed, rather than the cross-product of all states and emissions, our accuracy improved to 32.0%. We believe the discrepancy between our results may be caused by our relatively naive bigram implementation—we had no special logic for punctuation, for example—but we were also constrained by time: Johnson typically ran 1000 EM iterations, taking an average of 2.5 days per run. We do not think these poor results indicate that our implementation is incorrect, as the Viterbi output for the "colorless green ideas sleep furiously" part of speech example was correct.[2]

We found two major impediments to the performance of our distributed algorithm: loading the prior model parameters and job iteration. Over ten iterations, each EM iteration averaged 15:03 minutes on an Elastic MapReduce cluster with 10 nodes.[3] To evaluate this speed, we ran the same forward-backward methods locally, and each iteration took only 1:21 minutes on the

[2] The "Part of Speech" model from Assignment 3, part 2.2.
[3] Each EC2 "Small Instance" has 1.7 GB of memory and 1.2 GHz cpu.

same corpus.[4] As noted in our implementation details, the primary input to each Map node was a line from the corpus, but each node also needed to load the model parameters from a file. The emissions file was 3.2 megabytes, which took significant time to open in the Amazon S3 distributed file system. In contrast, the local implementation was able to leave the model parameters in memory after each iteration, without dealing with costly filesystem I/O. When we ran EM with the biased emissions, the model parameter file was reduced in size to only 98 kilobytes, and the average iteration time was shrunk to 3:26 minutes.

The iterative nature of the EM algorithm also causes some friction within the Mapreduce framework, since each job has significant setup and teardown costs. If we worked with a lower-level distributed framework, we may have been able to skirt these fixed costs and reduce the amortized cost of each iteration. Finally, we could simply throw more machines at the problem. Although both the accuracy and performance of our algorithm were not what we had hoped for, this project has been a great learning experience about the trials and tribulations of distributed computing. We still believe that a finely-tuned distributed system could efficiently solve large EM problems, but distributed computing cannot simply be thrown at any algorithm and be expected to perform.

[4] 8 GB of memory, SSD hard drive, and 2.3 GHz quad-core.

Appendix: Time and Space Complexity of EM

Let S be the set of hidden states in the model, X be the set of all n-grams, and E be the set of emission sequences e_i. Let m be the length of the longest emission sequence. Each iteration of the algorithm is comprised of computing the forward matrix for each e_i, the backward matrix for each e_i, the counts for each n-gram in each e_i (n depends on the model at hand), and then combining the counts for all e_i in E. Computing the forward and backward matrices for a given e_i has a time complexity of $O(m \cdot |S|^2)$ and a space complexity of $O(m \cdot |S||)$. Calculating the counts of each n-gram for a given e_i has a time complexity of $O(m)$ and a space complexity of $O(|X|)$. Combining all the counts has a time complexity of $O(|E| \cdot |X|)$.

Note that the calculation of the matrices and counts for each e_i can be performed entirely in parallel, allowing us to reduce the time complexity of the most costly portion of the algorithm by a constant factor. If we parallelize the computation among k machines, the total time complexity for each iteration will be $O(\frac{m \cdot |E| \cdot |S|^2}{k}) + O(|E| \cdot |X|)$ and the memory requirement for each machine will be $O(m \cdot |S||) + O(|X|)$.

Works Cited

Jimmy Lin and Chris Dyer. 2010. Data-Intensive Text Processing with MapReduce. Morgan & Claypool Publishers.

Mark Johnson. 2007. Why doesn’t EM find good HMM POS-taggers. *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 296–305.

Andreas Sand, Christian N.S. Pedersen and Thomas Mailund. HMMlib: A C++ Library for General Hidden Markov Models Exploiting Modern CPUs.

Yi Yang, Alexander Yates, and Doug Downey. 2013. Overcoming the Memory Bottleneck in Distributed Training of Latent Variable Models of Text. *Proceedings of NAACL-HLT 2013,* pages 579–584.