Grand Valley State University

Machine Learning

CIS678

CPU-Accelerated Bayseian Inference Modeling

Author:
Jarred Parr

Supervisor:
Dr. Greg Wolffe

February 19, 2019

Abstract

Bayesian inference is a powerful tool for doing classification based on probability. In this project, it was discovered that due to the easily separable nature of the processes, Bayesian inference can be significantly sped up through common parallelization techniques. Everything from the data loading to the data processing can be batched and handled much more quickly than when run in a single-processor context. This aided significantly when applying the technique to the large training and test data provided.

1 Introduction

Upon discussion of this assignment and a cursory glance of the provided pseudocode, it was clear that this would be a computationally intensive process. Many layers of iteration would be required to achieve a favorable and accurate result. As gears began to turn, the idea of parallelism in some capacity immediately came to mind. After a brief exscursion down the path of a GPU-Accelerated CUDA approach, it was quickly determined that that would be far too time consuming to get right. Because of this, OpenMP was chosen to allow for things to continue at a favorable pace. This was a good choice and offered an overall speedup of about 7 when compared to the non parallel version. It is clear from this result that there was much more that could have been improved, but shrinking the overall runtime from minutes to seconds was a very satisfying outcome overall.

2 Program Architecture

Many considerations had to be made to ensure the program would run properly. The all-too-common tale of dabbling in parallelism is that data can sometimes get mangled as a result of colliding threads attempting to read or write from the same block of memory. To avoid this, many of the *#pragma omp critical* directives were utilized. These allowed for the compiler to detect the necessary critical sections and place the appropriate guards around the execution. The downside of a liberal usage of these directives is that it takes a significant bite out of your programs runtime. While it still runs faster than when single threaded, you lose a fair amount of your potential speedup when this is applied liberally throughout the program. This was

the first bottleneck which needed to be remedied. How this was solved will be discussed in the coming sections. Besides some of the more esoteric corners of the parallel runtime, the only other main components of the program that needed a more heavy handed consideration applied to them were that of the word and text processing. To avoid the use of library, the option to write a custom stemming algorithm was utilized. This allowed for the words with weird endings to be removed from the system with greater ease and stemmed to their root. This decreased the overall word count and provided a big boost to performance.

3 Growth Areas

Document and text processing in C++ is one of the least common things someone can see in the modern programming landscape. This is because C++ is absolutely atrocious at this task. List handling, text reading and parsing, even simply checking if a word exists in a string, are all very time consuming tasks. This project proposed significant growth in the way of using C++ in a practical setting such as this. My word stemming algorithm is as follows:

```
1  void document::stem_document() {
2    #pragma omp parallel
3    {
4    for (size_t i = 0; i < lines_.size(); ++i) {
5      for (const auto& c : topic_frequencies_) {
6        std::cout << c.first << " " << c.second << std::endl;
7      }
8
9      std::vector<std::string> words = split(lines_[i]);
10
11     std::string topic = words[0];
12
13     auto found = topic_frequencies_.find(topic);
14     if (found != topic_frequencies_.end()) {
15       #pragma omp critical
16       {
17       ++topic_frequencies_[topic];
18       }
19     } else {
20       #pragma omp critical
21       {
```

```
22          topic_frequencies_[topic] = 1;
23          }
24        }
25
26        for (size_t j = 1; j < words.size(); ++j) {
27          for (const auto& suffix : suffixes) {
28            #pragma omp critical
29            {
30            if (ends_with(words[j], suffix)) {
31              words[j].substr(0, words[j].size() - suffix.size());
32              }
33            }
34          }
35        }
36
37        // Tally frequencies
38        count_word_frequencies(words);
39
40        // Remake the words
41        #pragma omp critical
42        {
43        lines_[i] = join(words, " ");
44        }
45
46        auto cfound = classified_text_.find(topic);
47        if (cfound != classified_text_.end()) {
48          #pragma omp critical
49          {
50          classified_text_[topic] += lines_[i];
51          classified_text_[topic] += " ";
52          }
53        } else {
54          #pragma omp critical
55          {
56          classified_text_[topic] = lines_[i];
57          }
58        }
59      }
60
61      for (const auto& topic : topics) {
62        #pragma omp critical
63        {
64        auto found = word_frequencies_.find(topic);
65        if (found != word_frequencies_.end()) {
66          word_frequencies_.erase(topic);
```

```
67          }
68          }
69      }
70      }
71  }
```

Listing 1: The Crude Parallel Stemming Algorithm

Liberal use of custom subroutines like *split*() and *join*() were immensely helpful for working with high dimensionality data quickly. These algorithms were coded in the anticipation of needing to work with and reshape arrays on the fly, this ended up being a great asset to getting this project done in a reasonable timeframe. After much experimentation, trial and error, and frustration, the number of critical statements in the code was the minimal amount such that it would reliably run and product data and also would be able to maximize speed. In a future implementation it would be ideal to minimize read and write until the very end of the function. This decoupling of steps would allow for maximum parallel code running without compromising the potential speedup. This initially hypothesized result was sufficient enough though.

4 Data Considerations

Data was a big section of this project in more ways than one. It provided all of the training information and was reasonably cleaned. However, this did not mean it was perfect, and a lot of hypothesis testing was used to determine the appropriate trade-off between accuracy, speed of results, and ability for it to be coded in a reasonable manner. Because of this program being run in parallel a little bit of overhead was allowed in the way of additional processing on the data. As shown previously, the stemming algorithm was quite involved, but easy to separate, the removal of stop words from the text, however, was a bit more involved. It caused issues with the parallelism because splicing the segments ended up creating lots of copies and managing to slow down runtime significantly even in a parallel environment. Additional time would have been needed in order to facilitate the true optimization of this section of the code, but such time was not available at the time of implementation for the program. All words were stemmed down to their root and improper words were removed.

5 Results

Overall the algorithm was able to produce a total correct classification rate of roughly 81%. The total runtime for the non-parallel version was about 1 minute, and when the parallel code was added it averaged about 8.5 seconds for the final prediction. This was after the data was processed and compiled in accordance to the following fit algorithm:

```
1  void Bayes::fit() {
2    std::vector<std::string> vocabulary =
3        doc_->extract_keys<std::string, int>(
4            doc_->word_frequencies_);
5
6    // Sum of document class counts;
7    const int doc_sum = std::accumulate(
8        doc_->topic_frequencies_.begin(),
9        doc_->topic_frequencies_.end(),
10       0,
11       [](const std::size_t prev, const auto& el) {
12     return prev + el.second;
13   });
14
15   for (const auto& topic : doc_->topics) {
16     probability_map estimates;
17     const double class_proba = (double)doc_->
   topic_frequencies_[topic] / (double)doc_sum;
18     const std::string text = doc_->classified_text_[topic];
19     const std::vector<std::string> topic_words = doc_->split(
   text);
20     frequency_map word_count_in_topic = word_count(topic_words
   );
21     const double n = (double)doc_->count_words_in_line(text);
22     class_probabilities_[topic] = class_proba;
23
24     for (const auto& word : vocabulary) {
25       const double nk = (double)word_count_in_topic[word];
26       const double estimate{(nk + 1.0) / (n + (double)
   vocabulary.size())};
27       estimates[word] = estimate;
28     }
29
30     topic_word_probabilities_[topic] = estimates;
31   }
```

```
}
```

Listing 2: The Data Fit Algorithm

This fit algorithm processed the data and generated the necessary probability distributions for each word in each block of topic words. This created the necessary environment to call the parallel evaluator. The code for the evaluator is as follows:

```
void Bayes::evaluate() {
  std::cout << "running test suite" << std::endl;
  document d("../data/forumTest.data");
  auto new_data = d.lines_;
  std::array<std::string, 2> answer;
  double probability{0.0};
  std::vector<std::array<std::string, 2>> outcomes;
  double correct{0.0};

  for (const auto& line : new_data) {
    auto words = d.split(line);
    const auto topic = words[0];

    for (
    auto data = words.begin() + 1; data != words.end(); ++data
  ) {
      probability += topic_word_probabilities_[topic][*data];
    }
    const auto correct = std::max_element(
        class_probabilities_.begin(),
        class_probabilities_.end());

    answer[0] = correct->first;
    answer[1] = correct->second;

    outcomes.push_back(answer);
  }

  // Validate our coutcomes
  int idx = 0;
  for (const auto& line : new_data) {
    auto words = d.split(line);
    const auto topic = words[0];
    if (outcomes[idx][0] == topic) {
      ++correct;
    }

```

```
37        ++idx;
38      }
39
40      std::cout << "Percentage correct: " << correct / (double)
        outcomes.size() << std::endl;
41
42    }
```

Listing 3: The Data Evaluator Algortihm

The evaluator was by far the hardest part conceptually to understand. This is primarily because if any flaws in the previous data exist, this algorithm will immediately show those flaws in the data and a lot of time was spent debugging the generation and ordering of the data as a result.

6 Conclusion

This project stretched my knowledge of C++ to the very corners. And while my results were not what I was expecting, it definitely taught me a significant amount about how to properly apply the scientific method to a machine learning oriented problem, and also gave me some good criteria for what to do on the next project to improve efficiency overall. It is clear that the gains made from parallelism in machine learning can be quite significant in the way of speeding up model performance and tuning in a production environment. C++ is definitely not an easy way to simply prototype, but it is clear that if something is to be done well, and operate quickly, C++ is clearly the best choice for the task.