PARALLELIZED LATENT DIRICHLET ALLOCATION FOR MEDICAL FRAUD DETECTION

by

Joshua C. Ready

HONORS THESIS

Submitted to Texas State University
in partial fulfillment
of the requirements for
graduation in the Honors College
December 2020

Thesis Supervisor:

Gregory LaKomski

Tahir Ekin

ABSTRACT

This paper seeks to analyze topic modeling software that assists investigators search for potential fraudulent Medicare billing activities. Furthermore, it will document the research done to parallelize this software in order to reduce its computational processing time. This paper also seeks to study questions such as “do medical providers of a specific type have similar billing patterns across states and regions?” and “can semantic analysis be used to identify both common and outlier billing patterns for specific providers?” According to the Blue Cross, medical fraud costs the United States government $68 billion every year. [1] This project focused on improving the speed and usability of the existing software to combat this problem.

Terminology:

- LDA – Latent Dirichlet Allocation, a topic modeling technique that allows random datasets to be “clustered” based on content associations.
- CUDA – a parallel programming toolkit for Nvidia GPUs.
- API – application programming interface.
- CPU – the main processor on a computer, large instruction set but not many cores to allow for parallelization.
- GPU – graphics processing unit, a limited instruction set and massively parallelized processor.
- thread – an instance of software being processed, that runs in parallel with other threads.
- race condition – a computing error that occurs when two or more threads attempt to modify or access data at the same time, producing unpredictable results.
- mutex – mutual exclusion, a tool for programmers that allows them to control which threads can access data at any given time.

I. INTRODUCTION

This paper will be expanding on research done by Greg LaKomski, Tahir Ekin and Muzaffer Musal at Texas State University, who together wrote a paper titled "Latent Dirichlet Allocation for Medical Fraud Detection." [2] Their paper researched the use of unsupervised topic modeling software in an effort to search for hidden patterns in medical billing records that could indicate fraudulent activity. This software utilized a technique called Latent Dirichlet Allocation, which is capable of performing semantic analysis on datasets and clustering the contents into any number of topics. By applying this type of software to medical billing records and sorting doctors into topics based on the billing patterns, they were able to create a model of how a doctor of a certain type is likely to bill, assuming the majority of doctors bill honestly and correctly. A context analysis can then be performed between doctors and the appropriate model, and those doctors who have a billing pattern very different than the model can be further examined by fraud investigators. They applied the software to medical billing codes in the state of Vermont. They found that the technique worked, and that they could successfully identify doctors who were outliers in their professional specialty. The next challenge in employing this type of software on large datasets is that the computational complexity and time required grows rapidly as the size of the datasets increase. This paper will focus accelerating the software and decreasing the required computational time to make it applicable to larger datasets. To achieve this, Nvidia's CUDA API was incorporated into the existing C++ software. CUDA is an API that allows programmers the ability to parallelize their code and utilize the processing power of their GPUs. This paper will explore how LDA works to identify parts of the code that cause bottlenecks. Then, we attempt to decrease the computational time by parallelizing the code and evaluating the performance of the software.

II. LATENT DIRICHLET ALLOCATION

Before applying the CUDA API and modifying the software, it is important to understand how LDA works so that the program's output is not significantly altered. Latent Dirichlet Allocation was originally proposed by David Blei in 2003 for use in text mining. [3] Latent Dirichlet Allocation is an iterative, unsupervised topic modeling process where underlying patterns that may not be easily noticeable to a human can be exposed. The most common analogy for Latent Dirichlet allocation is to imagine that you have a number of books scattered on the floor, and you would like to organize them into a number of topics without knowing anything about the books or their contents. To create associations between books, each word is assigned a probability distribution for each topic, based on the number of appearances of that word in books that belong to certain topics. By repeatedly looking at each word in each document and reassigning its topic probabilities, eventually the process will organize the books into clusters based on having similar word patterns. The LDA process doesn't know anything about the topics before or after it does its work, it only sees that books are similar to each other. When the process starts, words and books are randomly assigned to topics. In the first iterations, the words and books will be reassigned to different topics frequently as the program works to redefine what the topics are. After many iterations, the words and books will eventually converge to their appropriate topics, and there will be less reassignment of topics. After enough iterations, little to no reassignment will occur. The number of iterations necessary is dependent on the size and number of books to be sampled. If LDA is applied to search for medical fraud detection, a doctor's billing codes can be thought of as the words in a book, and the doctors as the books. These terms may be used interchangeably in this paper.

The process is depicted graphically in the plate diagram in Figure 1 below, where the variables are as follows:

- α, η – the hyper-parameters, define the initial conditions.
- θ_d – the vector of topic proportions for doctor *d*.
- $Z_{d,n}$ – the current topic assignments for doctor *d*, procedure *n*.
- $W_{d,n}$ – unique procedure billed by doctor *d*, procedure *n*.
- *N* – the total number of unique medical procedures billed by all doctors.
- *D* – the total number of doctors in the dataset
- $\beta_{k,n}$ – the frequency of procedures related to topic *k*, procedure *n*.
- *K* – the total number of topics specified by user.

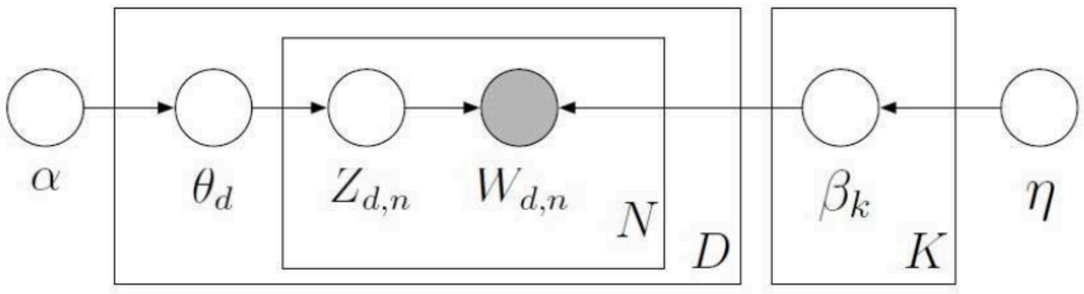

Figure 1: Graphical plate diagram showing the LDA process when applied to medical fraud detection.

The process can be summarized in this way:

1. The user supplies the dataset, the number of topics, and the number of iterations.
2. To start, random topic distributions are chosen for each billing procedure.
3. For every billing procedure *n* billed by doctor *d*, assign the procedure to a new topic by determining which topic of doctors billed that procedure the most.
4. Repeat this for every doctor and for the desired number of iterations.

It is important to note that in step 3, the process of choosing a new topic implements use of Bayesian statistical principles. In Bayesian statistics, previously known probability information is incorporated into the analysis, producing a more meaningful probability distribution. The previously known data is known as the prior, while the new probability distribution obtained after sampling the data is known as the posterior. In each iteration, the posterior is produced as a combination of the prior and the data sampled. The posterior then becomes the new prior, and the analysis is repeated. This process allows the topic probabilities to be effectively maximized because in each iteration the word to topic probabilities are based on prior information, allowing topics to redefined, but always remembering the previous assignments. This “memory” is essential in properly defining each topic model as more data is sampled.

Use of priors is also useful because it allows all previous sampling information to be retained, and new data can be ran through the model and clustered without the need to analyze the entire dataset again.

III. PARALLELIZATION USING CUDA

CUDA is a tool for parallelizing code and making use of GPUs in processing. GPUs have a limited number of instructions that they can execute, but they are equipped to do parallel calculations much faster than CPUs. By making use of GPU, simple calculations in the code that must be done on huge datasets can be hugely accelerated. CUDA only works on certain Nvidia GPUs and the users must have all of the CUDA tools installed on their machine. Setting up the CUDA environment is not to be underestimated, and difficulties were encountered when trying to get it working. Ultimately, use of a CUDA capable server owned by Texas State University was utilized, and all programs were tested and ran through a shell.

An analysis of the existing code to determine where the time-consuming parts of the code occur was performed and then the parts of the code that can be parallelized were identified. The time-consuming part of the code appears below.

```
for (int _lastIter = 0; _lastIter <= numIters; _lastIter++) {
        printf("Iteration %d ...\n", _lastIter);

     // for all z_i
     for (int nd = 0; nd < numDocs; nd++) {
        for (int n = 0; n < docvec[nd].length; n++) {
            // (z_i = z[m][n])
            // sample from p(z_i|z_-i, w)
            int topic = sampling(nd, n);
            z[nd][n] = topic;
        }
     }

}
```

Figure 2: The section from the original code that shows the main loop structures.

What this does is iterate through every billing code for every doctor and reassign each billing code to a new topic. This is done for a number of iterations specified by the user.

Given the current structure of the code, the time complexity is O(n^3). This means that as the dataset size *n* grows, the total processing time grows as a cube of *n*. When looking for code that can be parallelized, it is easiest to start with looking for loop structures, and so this section of the code is a prime candidate for parallelization. It is important to ensure that when the code is parallelized, the various threads do not create race conditions. A race condition occurs when two or more threads attempt to access and/or modify the same memory location, which can produce unpredictable results because the threads are "racing" and can execute in any order. We can see in the code below that the actual work that is done inside the inner-most loop is the sampling function. This function is the key part of the code that needs to be parallelized. We will seek to achieve parallelization at the word level in the code. That is, all iterations of the inner-most loop of the code above will happen in parallel. The number of parallel threads that will be spawned is equal to the number of words associated with each document. All of these threads must finish and return their results to the CPU before the next document can be iterated through. The sampling function shown below should be made into a CUDA function.

```
int model::sampling(int nd, int n) {
    // remove z_i from the count variables
    int topic = z[nd][n];
    int w = docvec[nd].words[n];
    wordsToTopic[w][topic] -= 1;
    wordsInDocToTopic[nd][topic] -= 1;
    totalWordsPerTopic[topic] -= 1;
    totalWordsInDoc[nd] -= 1;

    double Vbeta = vocabSize * beta;
    double Kalpha = numTopics * alpha;
    // do multinomial sampling via cumulative method
    for (int k = 0; k < numTopics; k++) {
      p[k] = (wordsToTopic[w][k] + beta) / (totalWordsPerTopic[k] + Vbeta) *
          (wordsInDocToTopic[nd][k] + alpha) / (totalWordsInDoc[nd] + Kalpha);
    }
    // cumulate multinomial parameters
    for (int k = 1; k < numTopics; k++) {
      p[k] += p[k - 1];
    }
    // scaled sample because of unnormalized p[]
    std::default_random_engine defEngine(time(0));
    std::uniform_real_distribution<double>dblDistro(0.0, 1.0);


    double u = dblDistro(defEngine) * p[numTopics - 1];

    for (topic = 0; topic < numTopics; topic++) {
      if (p[topic] > u) {
        break;
      }
    }

    // add newly estimated z_i to count variables
    wordsToTopic[w][topic] += 1;
    wordsInDocToTopic[nd][topic] += 1;
    totalWordsPerTopic[topic] += 1;
    totalWordsInDoc[nd] += 1;

    return topic;
}
```

Figure 3: The sampling function form the original code.

If this code were to be parallelized directly, race conditions would occur. This is because the wordsToTopic, wordsInDocToTopic, and totalWordsPerTopic vectors are used and modified by all threads. These vectors keep track of the current topics associated with each document and each word. One approach to remedy this is to use mutexes, however mutexes will limit the acceleration that can be achieved by CUDA because only one thread can access these vectors at a given time. Each thread will check the mutex variable before going into its critical section and will only execute if the mutex is free, essentially requiring the threads to hand off a baton to one another, giving permission to access the mutexed variables. The better approach is to relocate the lines of code that modify them outside the sampling function. This will require keeping track of the topic assignments for each word in each document before and after all CUDA threads are in execution.

Another issue faced is that CUDA does not work with vectors from the C++ standard template library, which is what most of the data structures in this code are. To solve this, pointers must be implemented for all vector indexing. The CUDA version of the sampling function appears below.

```
__global__ void cuda_sampling(int _numTopics, int _topic, int _w, double _alpha, double
_beta, double _Vbeta, double _Kalpha, int *dev_z, int *dev_wordsToTopic, int
*dev_wordsInDocToTopic, int dev_totalWordsPerTopic, int totalWordsInDoc, int *p)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int index = (col * y) + x;
    int stride = blockDim.x * gridDim.x;


    for(int i=index; i<n; i+=stride)
    {
        for (int k = 0; k < _numTopics; k++)
        {
            p[k] = (dev_wordsToTopic[_w][k] + _beta) / (dev_totalWordsPerTopic[k] +
_Vbeta) *
            (dev_wordsInDocToTopic[x][k] + _alpha) / (totalWordsInDoc[x] + _Kalpha);
        }


        for (int k = 1; k < numTopics; k++)
        {
            p[k] += p[k - 1];
        }

        double u = cudaRAND();


        for (topic = 0; topic < numTopics; topic++)
        {
            if (p[topic] > u)
            {
                break;
            }
        }


        z[_nd][_n] = topic;
    }

}
```

Figure 4: The CUDA version of the sampling function.

This code is very similar to the original, with the exception of the how the counter iterates through the loop. The loop is now inside the function rather than outside. To index the word in the document the thread ID is used, thus the need for the first 3 lines in the function. The counter is incremented by the value of stride, making the for loop execute once per block. Block sizing can be set when the function is called. Correct block sizing can have a large effect on processing time. The original code that called the sampling function must be changed to the code below. Note that cudaMallocManaged must be called for all arguments to cuda_sampling but was not shown for brevity.

```
cudaMallocManaged();

for (int _lastIter = 0; _lastIter <= numIters; _lastIter++) {
    printf("Iteration %d ...\n", _lastIter);

    // for all z_i
    for (int nd = 0; nd < numDocs; nd++) {
        cuda_sampling<<<1,docvec[nd].length>>>(numTopics, *z_after, w, alpha, beta,
Vbeta, Kalpha, *dev_z, *dev_wordsToTopic, *dev_wordsInDocToTopic, dev_totalWordsPerTopic,
totalWordsInDoc, *p);
    }

    cudaDeviceSynchronize();

}
```

Figure 5: The rewritten code loops from the original code.

Calling the sampling function in this way will spawn 1 block with a number of threads equal to the length of the current document being sampling. Note that this code does not show the updating of all the topic assignment vectors, which must be handled after the threads synchronize. The cudaDeviceSynchronize function will force the CPU to wait for all CUDA threads to finish before continuing, which is necessary to preserve LDA's functionality.

IV. CONCLUSION

Before attempting to parallelize the sampling function, other functions of the code were parallelized, but they were ultimately ineffective at accelerating the program because they are only called once. The sampling function in comparison is called for every word in every document. The CUDA version of the sampling function, however, was difficult to get to working properly. It produced unexpected output, such as all doctors being placed in the same topic, or the program never terminating. This is likely due to incorrectly setting the topic assignment values in the z vector. Other possible issues are the way the memory is being copied to and from the GPU, or the way the vectors are being indexed. Debugging and rewriting of the code is an on-going project. When the code is successfully parallelized, additional variables will be added to the LDA process, such temporal analysis. This will allow the program to not only examine what a medical provider is billing for, but also how often and when.

References:

1. Statistics. (n.d.). Retrieved December 06, 2020, from https://www.bcbsm.com/health-care-fraud/fraud-statistics.html

2. Ekin, T., Lakomski, G., & Musal, R. M. (2019). An unsupervised Bayesian hierarchical method for medical fraud assessment. Statistical Analysis and Data Mining: The ASA Data Science Journal, 12(2), 116-124.

3. Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent Dirichlet Allocation. Journal of Machine Learning Research, 3(Jan), 993-1022.

4. Lettier. (2019). Your Guide to Latent Dirichlet Allocation. Retrieved December 06, 2020, from https://medium.com/@lettier/how-does-lda-work-ill-explain-using-emoji-108abf40fa7d