Parallel TF-IDF Vectorization for Text Classification (Rough Draft)

***Abstract*—TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic that represents the significance of a word in a document relative to its corpus (collection of documents) it is included in. It is a widely used algorithm used for preprocessing in text classification. However, its normal, sequential, algorithm takes quite a long time to vectorize and compute TF-IDF scores across and entire corpus of documents. There are also not many TF-IDF implementations in C++, the most widely used TF-IDF implementations are in Python, which is much slower than C++. In this project, I present a parallel TF-IDF vectorization, computation, and classification implementation written in C++. The goal of this project is to provide an efficient and quick TF-IDF algorithm, and to compare my parallel implementation to my sequential implementation. I was successful in creating the parallel TF-IDF vectorization, computation, and classification. Source code is included in this project, and the performance evaluation in Python is included as well.**

Index Terms—TF-IDF, parallel, vectorization, computation, classification, C++, efficient, quick

Introduction

Natural language processing (NLP) is a vital field in today's world. One of the key techniques used in NLP is Term Frequency-Inverse Document Frequency (TF-IDF), which helps identify important words within a document, while filtering out less relevant words. Search engines such as Google utilize TF-IDF to rank web pages, filtering billions of somewhat relevant words to ensure users receive the most meaningful results. However, computing TF-IDF at this scale is extremely computationally expensive, especially when processing large datasets, such as internet search results.

A common approach to handling this challenge is to precompute and store TF-IDF scores in a database. However, maintaining such a vast database is resource intensive. An alternative solution is to compute TF-IDF dynamically when needed. Unfortunately, most existing implementations of TF-IDF are sequential and written in Python, which limits their efficiency and performance, especially for large-scale applications.

This project focuses on implementing a parallelized TF-IDF classifier in C++ to significantly improve computation speed and efficiency. It has been shown that C++ can be up to 100 times faster than Python for certain computational tasks due to its optimized memory management and lower-level control over hardware resources. By leveraging parallel computing, this implementation aims to further enhance TF-IDF performance by distributing the workload across multiple processing cores.

Beyond search engines, TF-IDF is widely used in spam detection, sentiment analysis, and document clustering, where faster computation can lead to real-time processing improvements. Traditional implementations suffer from bottlenecks in matrix operations, memory usage, and sequential execution, which can be mitigated using optimized C++ techniques. This project will integrate the Armadillo library, a highly efficient linear algebra library, to handle matrix operations and accelerate TF-IDF computation.

By implementing TF-IDF classification in C++ with parallelization, this project aims to demonstrate significant performance improvements over traditional approaches. The expected outcomes include faster execution times, efficient memory usage, and scalability for large datasets, making this approach highly suitable for real-world NLP applications

(Keeping these two paragraphs for later reference) The job of TF-IDF is to filter out these somewhat relevant words, whilst keeping the relevant ones accessible. However, keeping a database of TF-IDF scores for every item on the internet is extremely resource expensive. One solution is to compute the TF-IDF every time. This is also extremely costly, since most common TF-IDF implementations are sequential and in Python. In this project, I will focus on the implementation of a parallel TF-IDF in C++.

It has been shown that C++ can be up to one hundred times faster than Python when performing some computing tasks. Not only can the parallel TF-IDF be quicker and more efficient than traditional implementations of the TF-IDF classification, but the sequential implementation in C++ can also increase efficiency and performance times as compared to the traditional implementation.

Problem Statement

I have achieved the main goal for this project: Implement a parallel TF-IDF text classifier in C++, and benchmarking it against my sequential TF-IDF text classifier in C++. I will be using the sequential TF-IDF text classifier as my control. As far as I know, there is no publicly available parallel TF-IDF implementation in C++. With this project I seek to be able to implement key components of the commonly used text classification algorithm into a parallelized, C++ implementation. Once I have equivalent implementations (sequential and parallel), I will run tests on the two different implementations to see if there is any performance or efficiency differences between them. I will be looking mainly at speed, and briefly discussing text classification accuracy.

Motivation

In my research for this project, I could not find any parallel TF-IDF implementations written in C++, either in literature or production.

Discuss implementations of sequential TF-IDF found in C++. And importance of increasing speed and efficiency of TF-IDF here.

Related Work

My parallel C++ implementation of TF-IDF is primarily based on using atomics and avoiding locks. (Discuss avoiding locks here)

Implementation

Breakdown of tasks I completed on this project:

- Write a parallel and sequential TF-IDF vectorization and computation in C++
- Implement a text classification using cosine similarity
- Implement data structures as classes for the Corpus, Document, and Category
- Create a testing environment to extract benchmarks and performance results
- Compare benchmarks between the parallel and sequential implementations
    - Measure performance
    - Measure accuracy of classification

    - Test multiple corpus’ against multiple unknown documents

Plan for Implementation

Summarize the above list here

Anticipated & Encountered Challenges

Anticipated

- Memory leaks
- Overwritten memory

Encountered

- Memory leaks
- Overwritten memory
- Challenges finding an accurate classification algorithm
- Issues with copying memory from classes from one thread to the next

Testing

To accurately compare the results against the sequential C++ implementation, I ran the same input of training data and unknown documents for the parallel C++ implementation.

Discuss how it was tests and such here.

Evaluation

Discuss the evaluation here, not currently ready since not all data has been processed.