Vivek Sharath

Project Write Up

Description of Problem

Kobe Bryant is without a doubt one of the greatest basketball players we have ever seen. From hitting some of the most improbable shots, to being the leader of 5 NBA championships, Bryant has inspired millions (myself included). This project addresses a common sentiment in the Machine Learning Community: can't we just automate all this analysis? This project aims to provide users with an easy to use platform to allow for a deep dive into how Bryant played, specifically how he shot the ball. Users can visualize shots Bryant took over his career or build machine learning models to find some kind of pattern in the way Bryant shot with just a touch of a button.

Function 1: Visualization Job

Ever wonder what Bryant's shooting record is against the lackluster Boston Celtics? Or maybe you want to see how he did in the playoffs against the overhyped Utah Jazz. Maybe you just want to see the 4th quarter shots he has hit against the knock off version of the Lakers: LA Clippers. All this can be done and more by doing a Visualization Job with specifications to your liking.

This command gives all the 3-Pointers Kobe took against the Utah Jazz in the playoffs.
{"command":"visualize", "specs": ["Opponent:UTA", "ShotType:3PT Field Goal", "Playoffs:1"]}

Function2: ML Job

NBA analytics and Machine Learning go together like peanut butter and jelly, like Shaq and Kobe, like parallel programming and Go, like UChicago and stress. As mentioned before, we will be working with 25,698 records and several attributes. The almighty class label is a binary variable that specifies if Bryant Made the shot (1) or if Bryant Missed the shot (0). This project has 4 models to choose from: Logistic Regression, Decision Trees, Artificial Neural Networks, and KNN.

Here is an example of specifying a ML Job. As you can see, the setup is pretty similar to a visualization job. The difference is the "command" and the "model" part (duhh). Here is an example of a logistic regression model with the paramater C and solver. We want all possible combinations between these two paramaters so that we get a logistic regression model with C = 1 &solver = "newton-cg", C = 2 &solver = "newton-cg", C = 3 &solver = "newton-cg", C = 1 &solver = "lbfgs, C = 2 &solver = "lbfgs", C = 3 &solver = "lbfgs".

{"command":"ml", "model": "logistic regression", "params": ["C", "solver"], "param_range":[[1,3], ["newton-cg", "lbfgs"]]}

Vivek Sharath

Implementation

Data

Bryant was an absolute gunslinger during his 20 year NBA career. Believe it or not, but Bryant has shot the ball a whooping 30,698 times! (One of my personal favorites was when he hit a fadeaway, one legged three pointer against Dwyane Wade to win the game back in 09). The data we will be using for this project comes from kaggle. The data has seemingly endless information for every single shot Kobe has ever taken such as: the x,y coordinates the shot was taken, the team Bryant played against, the year the shot was taken, whether the shot was made or not, and even how much time was remaining on the clock when the shot was taken. These are just a few of the many attributes that are provided for every shot Bryant has ever taken. Unfortunately, this dataset is missing 5,000 values (for competition reasons). For the purposes of this project, we will be working with the 25,698 given records.

Data Decomposition: Visualization Job

Our visualization job requires that we analyze each and every record in the dataset. Unfortunately, this takes a long time with just one thread. We can split the entire dataset into N chunks for N go routines to analyze. Each go routine will loop through their block of data and check if the record meets all the criteria that the user specifies. If the criteria is not met, the record is not plotted. If the criteria is met, the record is plotted.

Data Decomposition: ML Job

This is where things got a little tricky. For each ML Job, we are given a model with various parameters. The combination of all parameters and their ranges was not easy to parallelize. The solution was to have one thread determine all the possible combinations of parameters and their ranges and save the info to an array. So if the ml job was
{"command":"ml", "model": "ann", "params": ["hidden_layer_sizes", "activation", "solver"], "param_range":[[1,6], ["tanh", "relu"], ["sgd", "adam"]]}
we would have 24 different combinations. We can assign N combinations to N threads. If there are more combinations than threads, we simply spawn goroutines in the amount of combinations. When a thread works on a combination, the thread calls a python script called ml.py. This ml.py script takes in the model name and parameters specified in this combination, runs 3-fold cross validation on the model, and returns the accuracy. The go program analyzes each incoming model accuracy and outputs to the user the model with the best accuracy.

Functional Decomposition

I decided to go with pretty standard functional decomposition. I had a main thread that reads in the kind of usage the user wants, and then calls on the reader function. The reader function can increase as the number of threads increases. The reader function reads from the standard input of commands and enqueues those commands to a lock free queue (more info about this below). The reader function also dequeues tasks and calls on the appropriate functions based on the

task. If the task is a visualization job, the reader function calls the visualization worker funcion. If the task is a ML job, the reader calls the ML worker function. The reader function exits back to the main function once the queue is empty. The main function ends once all worker functions are done completing their tasks. Communication between worker functions and the main function is done through a waitgroup. Communication within worker functions is done using channels.

Challenges

There were several challenges with this project. The first was figuring out how in the world to plot using golang. There was a learning curve for this as I was introduced to new syntax. With the help of this youtube video: https://www.youtube.com/watch?v=ihP7IQivA6M&t=1683s, I was able to get the ball rolling on plotting. Once I got the basics down, building appealing graphs became easier. However, if I had allocated my time a little better, I could have done a better job with making the visualization more appealing. I underestimated the amount of time this project was going to take.

Another challenge, as mentioned earlier, was splitting up different combinations of models for threads to work on. My implementation was a little ugly to be honest (code in proj3.go). I used a handful of nested for loops in order to get the results that I needed. Furthermore, I couldn't parallelize this so I was pretty disappointed. There may be a better way to parallelize determining all the combinations of models, but I went with what worked given the time constraint.

Another challenge that I ran into was trying to do Machine learning with Go. I gave up on this almost instantly. I decided to take the easy route and just have a python script actually fit the model to the data, while the go script does all the pre-processing.

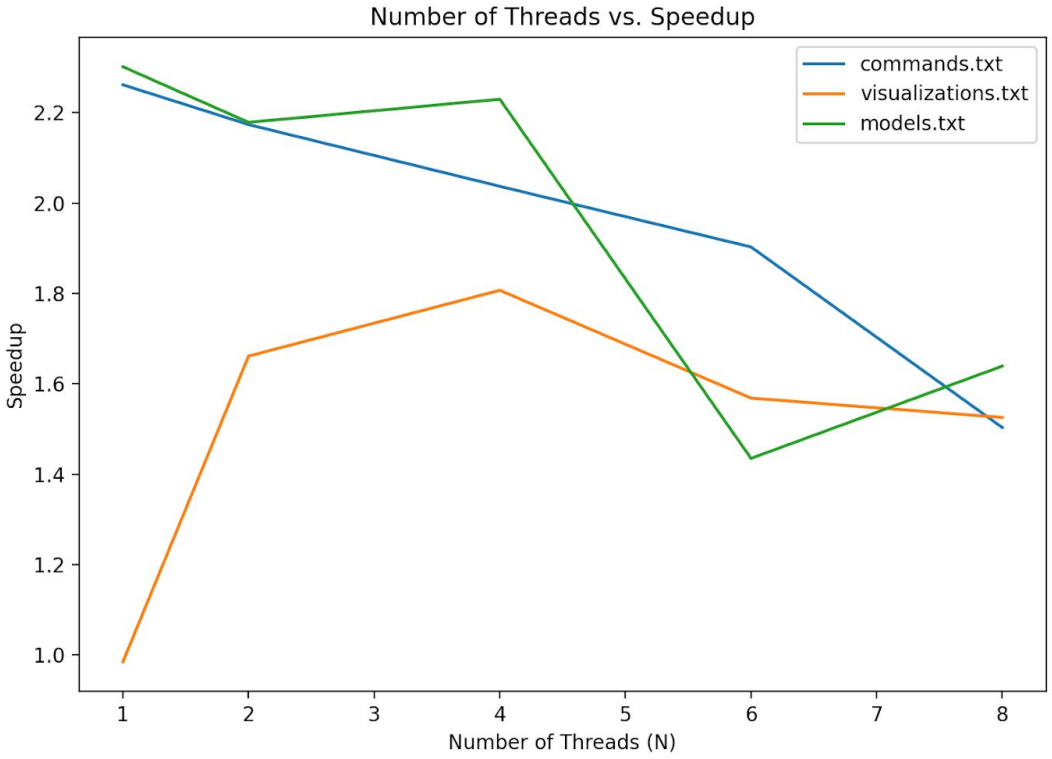
Another challenge (I feel like DJ Khaled saying another so much) was communicating between a python script and a Go script. I looked far and wide for a solution on stackoverflow and finally found something that ended up working. Unfortunately, just sending one model for the python script to analyze took quite some time. When I ran the python script with an arbitrary model on the same dataset, the results came back pretty fast. When using go to call the python script, the results came back a lot slower. There might be a better way to communicate between a go script and a python script and I am a little disappointed I couldn't find that solution. Fortunately despite the delay, the python script does eventually return the results to the Go Script which took me a long time just to figure out how to implement.

Machine Specs

The testing machine I used is a 2016 Macbook Pro running on macOS Mojave version 10.14.6. The Processor is a 3.3 Ghz Intel Core i7. This laptop has 2 cores with 16GB memory.

Vivek Sharath

Graph interpretation



Overview

I ran experiments on 3 files. Models.txt only had ML jobs, Visualizations.txt only had visualization jobs, and commands.txt had a combination of ML and visualization jobs. Furthermore, I computed Speedup by varying threads N = [1,2,4,6,8]

Analysis

For models.txt and commands.txt, performance seemed to decrease as we added more threads. This might be a result of having too much overhead. Just calling a python script took a lot of computing power. This coupled with excessive threads can cause extra traffic and poorer results. What is also interesting is that performance did not improve between 4 and 6 threads (6 threads and up had 2 readers). This was very different from proj2 where adding an extra reader helped greatly with performance. Visualizations.txt saw improvements as we increased the threads, but went down when we introduced a second reader with 6 threads. This could also be due to having too much overhead with an extra reader thread and more worker threads. Text files with models seemed to have better speedup due to the fact that just getting the results of one model from a python script takes a very long time. Parallelizing this makes a big difference in performance because we have less idle thread. Finally, the speedup across all text files were not as good as the speedup seen in project2. This due to the inability to take care of bottlenecks such as determining all the combinations of model parameters specified by the user, waiting for

all goroutines to return with a model accuracy, reading from standard input sequentially, and the great deal of pre-processing in the main function (more info on this below).

Hotspots

The hotspot for this program is without a doubt the two worker functions. For the visualization worker function, we take care of this hotspot by having N threads loop through N chunks of the dataset and plot accordingly. For the ml worker function, we assign N threads to take care of N chunks of different combinations of models. The data decomposition and execution here definitely helped with performance as seen below

Bottlenecks

As mentioned before, the main bottleneck that we could not parallelize was figuring out all the combinations for a model. I ended up having just one thread do all the computing here and then N goroutines to work on different model combinations. Another bottleneck was synchronizing all the threads to output their model accuracy because we want to display to the user the best model. The performance of this depends on how fast the slowest thread is in determining a model's accuracy. The last bottleneck was the preprocessing of data at the start of the program. Whether we run the program sequentially or parallel, we have to sequentially read in the very large json file and create a mapping of struct fields to numbers.

Advance Feature - Lock Free Queue

The advanced features that I added to this project was using a lock-free queue which can be found starting on line 83. The code for this was built on the code Professor Samuels provided us on the Midterm (problem 6). The change here turned the queue into a data structure that held strings instead of ints. This was done using the unsafe.pointer. More details of the implementation of this can be found in proj3.go, line 83.