STAT4402 Tutorial

Michael Ciccotosto-Camp

University of Queensland

1. Introduction

Most computational work done within science is impractical to perform on commercial laptops and desktops, typically due to the extremely high memory and processing demands. Hence, almost every university and industry has its own high-performance computer to carry out such strenuous problems. A lot of research within the realm of Machine Learning benefits from access to high performance machinery as most of them require matrix computation (which can be efficiently carried out on GPU clusters) and can be processed in parallel.

In this tutorial we will revisit two models covered in the first few weeks of lectures, these being the SDG linear regressor and KNN classifier. The naive implementation of both these algorithms are fairly inefficient, so we shall look at some ways in which these two methods can be decomposed and parallelised.

2. Parallel KNN

- The k^{th} Nearest-Neighbor (k-NN) methods use observations in the training set T closest in feature space to a given unknown sample $\boldsymbol{x}$ to directly find its corresponding prediction $\overline{y}$
- The prediction for the k-NN classifier is usually calculated as

$$\overline{y}\left(\boldsymbol{x}\right) = \sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} y_i$$

- The notion of 'clostest' implies the use of some sort of meteric. More often than not, feature vectors belong to $\mathbb{R}^n$ allowing us to use commonly used metrics to define distance between vectors in our feature space. For our purposes, we shall use the

Euclidean norm as a measurement of determining how close two feature values are to each other. The Euclidean norm is simply defined as

$$d(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_{i=1}^{n} (x_i - y_i)^2 \right)^{\frac{1}{2}}$$

- When a unknown sample $\boldsymbol{x}$ is to be classified, a k-NN classifier computes the distance between $\boldsymbol{x}$ and the other points within the training set T. The training data is then sorted by distance and the k^{th} closests training samples are then used to predict $\boldsymbol{x}$.
- A simple K-NN algorithm works as follows

```
Algorithm 1: Serial k-NN
  input  : Training data $T$, an unlabelled sample $\boldsymbol{x}$ and a value $k$
  output: Predicted class $\overline{y}(\boldsymbol{x})$
1 Computes distance $d(\boldsymbol{x}, \boldsymbol{x}_{t_i})$ for each $\boldsymbol{x}_{t_i} \in T$;
2 $N_k(\boldsymbol{x}) \leftarrow$ the $k^{th}$ closest $\boldsymbol{x}_{t_i}$ determined by $d(\boldsymbol{x}, \boldsymbol{x}_{t_i})$;
3 $\overline{y}(\boldsymbol{x}) \leftarrow \sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} y_i$;
  Result: $\overline{y}(\boldsymbol{x})$
```

- While this method is simple, computing the distance between $\boldsymbol{x}$ and each $\boldsymbol{x}_{t_i} \in T$ can incur a large time overhead, especially for large training sets
- One important observation is that computing the distances $d(\boldsymbol{x}, \boldsymbol{x}_{t_i})$ and $d\left(\boldsymbol{x}, \boldsymbol{x}_{t_j}\right)$ where $\boldsymbol{x}_{t_i}, \boldsymbol{x}_{t_j} \in T$ and $i \neq j$ may be done completely independently of each other meaning these computations may be carried out on separate processes.
- CITATION takes advantage of this independence to have distance computation carried out on different processes.

This algorithm is shown pictorially below

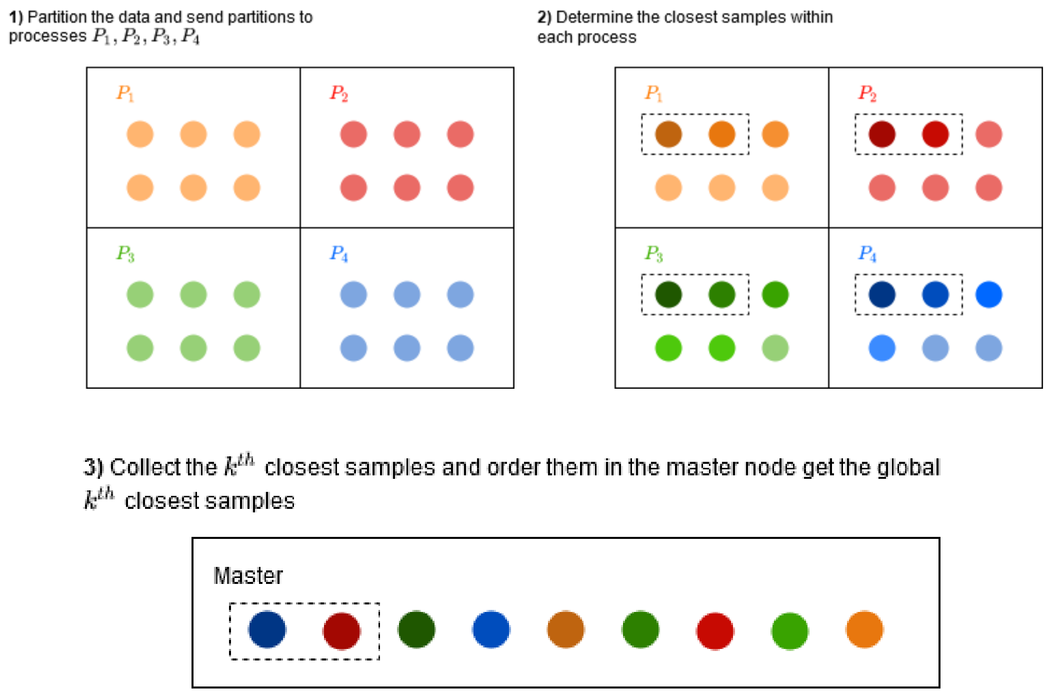
3. Parallel SGD

- As before let T a set of training samples $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{m} = \{z_i\}_{i=1}^{m}$
- Let $C(\boldsymbol{w})$ be the cost function

$$C(\boldsymbol{w}) = \frac{1}{N} \sum_{i=0}^{m} C_{z_i}(\boldsymbol{w})$$

Algorithm 2: Parallel k-NN

```
input  : Training data $T$, an unlabelled sample $\boldsymbol{x}$, a value $k$ and the number of processes to
         perform the algorithm $p$
output: Predicted class $\overline{y}(\boldsymbol{x})$
1 $\{T_1, T_2, \ldots, T_p\} \leftarrow$ an equal partition of $T$;
2 for $T_i \in \{T_1, T_2, \ldots, T_p\}$ concurrently do
3     $N_{k_i}(\boldsymbol{x}) \leftarrow$ the $k^{th}$ nearest neighbors from $T_i$;
4 end
5 $N_k(\boldsymbol{x}) \leftarrow$ the $k^{th}$ closest neighbors from $N_{k_1}(\boldsymbol{x}), N_{k_2}(\boldsymbol{x}), \ldots N_{k_p}(\boldsymbol{x})$;
6 $\overline{y}(\boldsymbol{x}) \leftarrow \sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} y_i$;
  Result: $\overline{y}(\boldsymbol{x})$
```


1) Partition the data and send partitions to processes P_1, P_2, P_3, P_4
2) Determine the closest samples within each process
P_1
P_2
P_3
P_4
3) Collect the k^{th} closest samples and order them in the master node get the global k^{th} closest samples
Master

- For gradient descent algorithms, we wish to find a weight vector $\boldsymbol{w}^*$ that minimizes our cost function, that is

$$\boldsymbol{w}^* = \underset{\boldsymbol{w} \in \mathbb{R}^n}{\arg\min} \sum_{i=0}^{m} C_{z_i}(\boldsymbol{w})$$

- We shall also introduct the notation $G \triangleq \frac{\partial C}{\partial \boldsymbol{w}}$ and $G_z \triangleq \frac{\partial C_z}{\partial \boldsymbol{w}}$ to simplify gradient notation as well as $H \triangleq \frac{\partial G}{\partial \boldsymbol{w}}$ and $H_z \triangleq \frac{\partial G_z}{\partial \boldsymbol{w}}$ to simplify Hessian notation.

- At each step of the SGD, a sample $z_j = (\boldsymbol{x}_j, y_j)$ is uniformly selected from the training set to update the existing weight vector as

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta_t G_{z_j}\left(\boldsymbol{w}_t\right)$$

- where η_t is just the learning rate at iteration t.
- Say a process performs SGD of a data set T_1 to get from a weight $\boldsymbol{w}_g$ to weight $\boldsymbol{w}_1$. When processing another training set T_2, a sequential SGD algorithm would have started at weight $\boldsymbol{w}_1$ to reach a possibly different weight vector $\boldsymbol{w}_h$.
- To parallelize the SDG algorithm we wish to start computing of training set T_2 on weight vector $\boldsymbol{w}_1$ while simultaneously running the training set T_1 on weight vector $\boldsymbol{w}_g$, but $\boldsymbol{w}_1$ is not know until SGD is finished with T_1. So how do we ge around this?
- One method is to soundly combine models from different processes, in the hopes of achieving a weight vector if SGD was simply run sequentially.
- This requires adjusting the computation of T_2 to account for the staleness $\boldsymbol{w}_1 - \boldsymbol{w}_g$ in the initial model.
- To do so, the second model performs its computations instead on $\boldsymbol{w}_g - \Delta\boldsymbol{w}$ where $\Delta\boldsymbol{w}$ is an unknown symbolic vector
- This allows the second model to run in parallel and not have to wait until $\boldsymbol{w}_1$ is produced from the succeeding model.
- Once the first process is done, the second process takes $\Delta\boldsymbol{w}$ to be $\boldsymbol{w}_1 - \boldsymbol{w}_g$
- This technique can be easily extended to an arbitrary number of processors.
- Let $S_T\left(\boldsymbol{w}\right)$ represent the SGD computation of a training T from an initial weight vector $\boldsymbol{w}$, for example $S_{T_1}\left(\boldsymbol{w}_g\right) = \boldsymbol{w}_1$
- To come up with a model combiner we need to think about how we can calculate

$$S_T\left(\boldsymbol{w} + \Delta\boldsymbol{w}\right)$$

- Assuming S_T is differentiable at $\boldsymbol{w} + \Delta\boldsymbol{w}$, we get following by consider the Taylor series of S_T about the point $\boldsymbol{w} + \Delta\boldsymbol{w}$

$$S_T\left(\boldsymbol{w} + \Delta\boldsymbol{w}\right) = S_T\left(\boldsymbol{w}\right) + S_T'\left(\boldsymbol{w}\right) \cdot \Delta\boldsymbol{w} + \mathcal{O}\left(\left|\Delta\boldsymbol{w}\right|^2\right)$$

- We will introduce the notation $M_D \triangleq S'_T$ as the model combiner. In the equation above the model combiner captures the first order information of how a change in $\Delta \boldsymbol{w}$ will effect the SGD

- When $\Delta \boldsymbol{w}$ is sufficiently small, one can neglect higher order terms and only use the model combiner to combine models from different processes.

- From CITATION we can show that for a sequence of input examples $z_1, z_2, \ldots, z_n$ the model combiner can be computed as

$$M_D(\boldsymbol{w}) = \prod_{i=1}^{n} \left(\boldsymbol{I} - \eta_i \cdot H_{z_i}\left(S_{T_{i-1}}(\boldsymbol{w})\right)\right)$$

  where $S_{T_0}(\boldsymbol{w}) = \boldsymbol{w}$. This result can be easily shown by applying the chain rule to $S_T(\boldsymbol{w}) = S_{T_n}\left(S_{T_{n-1}}(\ldots(S_1(\boldsymbol{w})))\right)$.

- Thus to create a parallelized SGD, each of the p processors start with the same initial global weight vector $\boldsymbol{w}_g$ to compute its local model $S_{T_i}(\boldsymbol{w}_g)$ and model combiner $M_{T_i}(\boldsymbol{w}_g)$ in parallel. A subsequent reduction phase computes $\boldsymbol{w}_i$ by adjusting the input of processor i by adjusting by the staleness introduced in the $i-1$ processor

$$\boldsymbol{w}_i = S_{T_i}(\boldsymbol{w}_{\boldsymbol{g}}) + M_{T_i}(\boldsymbol{w}_g) \cdot (\boldsymbol{w}_{i-1} - \boldsymbol{w}_g)$$

- The algorithm for parallel SGD is summaries below

Algorithm 3: Parallel SGD

```
input  : Training data $T$, an initial weight vector $\boldsymbol{w}_g$ and the number of processes to perform the
         algorithm $p$
output: Updated weight vector
1 $\{T_1, T_2, \ldots, T_p\} \leftarrow$ an equal partition of $T$;
2 for $T_i \in \{T_1, T_2, \ldots, T_p\}$ concurrently do
3     Compute $S_{T_i}(\boldsymbol{w}_{\boldsymbol{g}})$ and $M_{T_i}(\boldsymbol{w}_g)$;
4 end
5 for $i \in \{1, 2, \ldots, p\}$ do
6     $\boldsymbol{w}_i \leftarrow S_{T_i}(\boldsymbol{w}_{\boldsymbol{g}}) + M_{T_i}(\boldsymbol{w}_g) \cdot (\boldsymbol{w}_{i-1} - \boldsymbol{w}_g)$;
7 end
  Result: $\boldsymbol{w}_p$
```

4. Introduction to High Performance Computing

- So all these parallel algorithms seem great and all, but they won't actually benefit us anything if we don't have the computing power to run them!

- While commercial bought laptops usually have more than one core in them, their usage is taken up by processes running in the background, so if you ran anyone of these algorithms on your own machine, chances are you won't see much of an improvement in performace

- You might be asking now, *what sort of computers can run these algorithms to actually see performance improvements?* Well I'm glad you asked! The answer High Performance Computers. High Performance Computers (or just HPCs) are very large machines consisting of hundereds or even thousands of cores to run scientific or analytic programs on. The processes on HPCs are monitored by a special operating system so that any jobs that you submit to be run of a HPC gets exactly the number of processes and amount of memory you've asked for (provided that HPC system is able to provide those resources).

- As a student taking STAT4402 you should have access to UQ's `getafix` HPC.

- To log into `getafix` we can simply `ssh` via terminal (you will need to be on campus to do this or use a VPN).

- If you're working you'll need to run a few commands before we can start coding. `getafix` does not have many python libraries installed for users, so we will need to install them ourselves

We can start by creating a folder to put all of our code in, changing to that directory and creating our first python file using vim.

If you're unsure of what any of these commands do, you can simply look that the manual page for that command by type `man` followed by the command into your terminal. For example `man mkdir` will bring up the manual page for the `mkdir` command used above. As simple example, let's populated our newly created `parallel_sum.py` with a python script that will sum together the contents of a numpy array in parallel.

You can exit out of vim by pressing `ctrl+c`, then typing `:wq` into the prompt at the bottom and then hitting enter. This is great but how do we submit this script to be executed by the HPC. It wouldn't make sense to execute it in the login node we are current in since we wouldn't be enjoying the immense computing power offered by the HPC. To submit jobs to the HPC we need to create a job file, commonly refereed to as a batch file, which specifies a whole bunch of parameters and conditions that the HPC will use to run our code, including how many cores to use, how much memory to use, what programs to execute, where to put the output etc. An example batch file to run our `parallel_python.py` file is shown below. Saving this batch file as `parallel_python_batch.sh` we can submit to our job to the HPC.