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 uses observations from a training set T to find its closest neighbours in feature space to a given an unknown sample $\boldsymbol{x}$ a prediction value $\overline{y}$ [**?**]. The prediction for the k-NN classifier is usually calculated as an average or consensus vote, that is

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

The notion of 'closest' implies the use of some sort of metric. More often than not, feature vectors belong to to some subset $\mathbb{R}^n$, allowing us to apply 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

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

Other metrics such as Manhatten norm or hamming distance for discrete feature spaces [**?**]. 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 is presented in algorithm **??**. While this method is simple, computing

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: $\overline{y}(\boldsymbol{x})$

the distance between $\boldsymbol{x}$ and each $\boldsymbol{x}_{t_i} \in T$ can incur a large time overhead, especially for large training sets. In order to parallize this algorithm, 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 $i \neq j$ allowing us to carry out these computations on separate processes. Liang et al. (2010) takes advantage of this independence by splitting up the training examples into p partitions and send

these partitions to slave processes to compute the distances between feature vectors in our training example and a unknown sample $\boldsymbol{x}$. The k^{th} closet training examples are found within individual slaves and are collected within the master process. Combining the k^{th} closet acquired by each process locally, the master process then sorts the reduced list of training vectors to find the global k^{th} closet training examples [**?**]. Pseudo code for parallel KNN is shown in **??**.

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

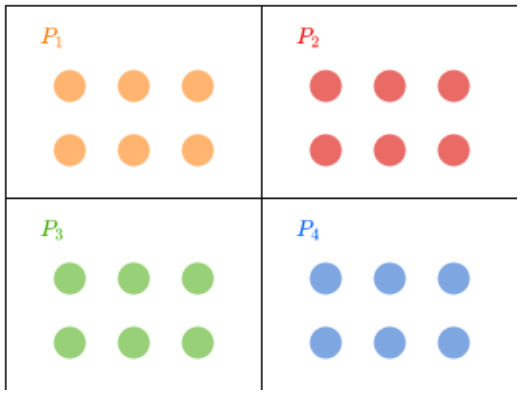
output: Predicted class $\overline{y}(\boldsymbol{x})$

$\{T_1, T_2, \ldots, T_p\} \leftarrow$ an equal partition of T;
for $T_i \in \{T_1, T_2, \ldots, T_p\}$ **concurrently do**
  $N_{k_i}(\boldsymbol{x}) \leftarrow$ the k^{th} nearest neighbors from T_i;
end
$N_k(\boldsymbol{x}) \leftarrow$ the k^{th} closest neighbors from $N_{k_1}(\boldsymbol{x}), N_{k_2}(\boldsymbol{x}), \ldots N_{k_p}(\boldsymbol{x})$;
$\overline{y}(\boldsymbol{x}) \leftarrow \sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} y_i$;

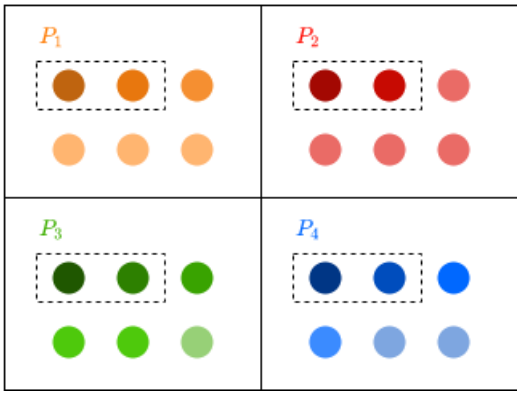
Result: $\overline{y}(\boldsymbol{x})$

This algorithm is shown pictorially below

Step 1) Partition the data and send partitions to processes P_1, P_2, P_3, P_4



Step 2) Determine the closest samples within each process



Step 3) Collect the k^{th} closest samples and order them in the master node get the global k^{th} closest samples

3. Parallel Multiclass Perceptron

- Similar to a definition is class the perception is an algorithm that uses an affine function as a threshold boundary to classify real valued feature vectors
- The output of a perceptron is simply given as

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

  where $\boldsymbol{w} \in \mathbb{R}^m$ with b acting as a bias

- A simple generalisation we can make to the perceptron is to allow the perceptron to predict from more than just two classes. This is known as the *multiclass perceptron*. A feature representation function maps a feature vector and its correspond label to a real valued vector. This output vector is then dot-producted with a weight vector $\boldsymbol{w}$, similar to the norm perceptron. The prediction value of an unknown feature vector for the multiclass perceptron can be computed as

$$\overline{y}(\boldsymbol{x}) = \arg\min_{y} \boldsymbol{w} \cdot f(\boldsymbol{x}, y)$$

- The iterative procedure to create our weight vector from the training set is also similar, although this time if we make an incorrect prediction we move the weight vector in the direction of $f(\boldsymbol{x}, y) - f(\boldsymbol{x}, \overline{y})$.
- Algorithm for the serial multiclass perceptron is shown in algorithm **??**. Note, for the sake of simplicity a learning rate parameter has been omitted with in the training algorithm

- One obvious way to parallize the algorithm from **??** is to split the training set T into p partitions. One can then train an initial weight on each of these partitions independently on the p processes and then take a weighted mixture of the weights from each of the different processes. The mixture coefficents will be given as the vector $\boldsymbol{\mu}$ where $\mu_i \geq 0$ and $\sum_i \mu_i = 1$. Algorithm for this naive parallel multiclass perceptron is shown in algorithm **??**.

Algorithm 3: Serial Multiclass Perceptron

input : Training data T, an initial weight vector $\boldsymbol{w}^{(0)}$

output: Updated weight vector

$k \leftarrow 0$;
while *until convergence* **do**
  for $(\boldsymbol{x}_i, y_i) \in T$ **do**
    $\overline{y} \leftarrow \arg\min_{y} \boldsymbol{w}^{(k)} \cdot f(\boldsymbol{x}, y)$;
    $k \leftarrow k + 1$;
    if $\overline{y} \neq y$ **then**
      $\boldsymbol{w}^{(k+1)} \leftarrow \boldsymbol{w}^{(k)} + f(\boldsymbol{x}, y) - f(\boldsymbol{x}, \overline{y})$;
  end
end

Result: $\boldsymbol{w}^{(k)}$

Algorithm 4: Naive Parallel Multiclass Perceptron

input : Training data T, an initial weight vector $\boldsymbol{w}^{(0)}$

output: Updated weight vector

$k \leftarrow 0$;
$\{T_1, T_2, \ldots, T_p\} \leftarrow$ an equal partition of T;
for $T_i \in \{T_1, T_2, \ldots, T_p\}$ **concurrently do**
  $\boldsymbol{w}^{(i)} = \text{SerialMulticlassPerceptron}\left(T, \boldsymbol{w}^{(0)}\right)$;
end
$\boldsymbol{w} = \sum_i \mu_i \boldsymbol{w}^{(i)}$;

Result: $\boldsymbol{w}$

- While **??** is easily understood and makes effective use of parallel resources without incurring too much over-head in inter-process communication, there is one large shortcoming of this algorithm. Even for linearly separable datasets T, algorithm **??** does not always converge to a separating hyperplane.
- In fact in Question ??? you yourself will be asked to provide a linearly separable dataset for which algorithm **??** will not converge.
- To salvage this parallel algorithm we can make sure each processors is communicating

its results before finalization.

- That is we can collect weights from processors after each single epoch, mix these the weights together and then redistribute this new weight vector to our processes again for a new epoch.
- This new iterative parameter mixing idea is shown in algorithm.

Algorithm 5: Parallel Multiclass Perceptron

input : Training data T, an initial weight vector $\boldsymbol{w}^{(0)}$

output: Updated weight vector

$\boldsymbol{w} \leftarrow \boldsymbol{w}^{(0)}$;
for $n \in \{1, 2, \ldots, N\}$ **do**
  for $T_i \in \{T_1, T_2, \ldots, T_p\}$ **concurrently do**
    $\boldsymbol{w}^{(i,n)} = \text{OnePerceptronEpoch}\,(T_i, \boldsymbol{w})$;
  end
  $\boldsymbol{w} \leftarrow \sum_i \mu_i \boldsymbol{w}^{(i,n)}$;
end

Result: $\boldsymbol{w}$

Algorithm 6: One Perceptron Epoch

input : Training data T_i, an initial weight vector $\boldsymbol{w}^{(0)}$

output: Updated weight vector

$k \leftarrow 0$;
$\boldsymbol{w} \leftarrow \boldsymbol{w}^{(0)}$;
for $(\boldsymbol{x}_i, y_i) \in T_i$ **do**
  $\overline{y} \leftarrow \arg\max_y \boldsymbol{w}^{(k)} \cdot f(\boldsymbol{x}, y)$;
  if $\overline{y} \neq y$ **then**
    $\boldsymbol{w}^{(k+1)} \leftarrow \boldsymbol{w}^{(k)} + f(\boldsymbol{x}, y) - f(\boldsymbol{x}, \overline{y})$;
    $k \leftarrow k + 1$;
end

Result: $\boldsymbol{w}^{(k)}$

- The obvious and main drawback of algorithm **??** over **??** this that there is much more inter-process communication overhead in reducing $\boldsymbol{w}^{(i,n)}$ and sending new values of $\boldsymbol{w}$.
- However, we will see that algorithm **??** will in fact converge for linearly separable training sets.
- We can show this in the following Theorem.

Theorem Assuming a training set T is linearly separable by a margin λ, let $k_{i,n}$ be the number of mistakes that occur ed on partition i of the training set during the n^{th} epoch of training. For any N, when training a perception with fixed parameter mixing in algorithm **??**

$$\sum_{n=1}^{N}\sum_{i=1}^{S}\mu_{i,n}k_{i,n} \leq \frac{R^2}{\lambda^2}$$

Using some notation from before, let $\boldsymbol{w}^{(i,n)}$ be the weight acquired on the n^{th} epoch of training by process i and let $\boldsymbol{w}^{(i,n)-k}$ be the weight vector for process i k iterations back. Let $\boldsymbol{w}^{(avg,n)}$ be the weighted mixture weight for the n^{th} epoch, that is

$$\boldsymbol{w}^{(avg,n)} = \sum_{i=1}^{S}\mu_{i,n}\boldsymbol{w}^{(i,n)}$$

we can easily enough show that

$$\boldsymbol{u}\cdot\boldsymbol{w}^{(i,n)} \leq \boldsymbol{u}\cdot\boldsymbol{w}^{(avg,n-1)} + k_{i,n}\lambda \tag{R1}$$

and

$$\left\|\boldsymbol{w}^{(i,n)}\right\|^2 \leq \left\|\boldsymbol{w}^{(avg,n-1)}\right\|^2 + k_{i,n}R^2 \tag{R2}$$

Using **??** and **??** we can inductively show the following results

$$\boldsymbol{u}\cdot\boldsymbol{w}^{(i,n)} \geq \sum_{n=1}^{N}\sum_{i=1}^{S}\mu_{(i,n)}k_{i,n}\lambda \tag{H1}$$

and

$$\left\|\boldsymbol{w}^{(i,n)}\right\|^2 \leq \sum_{n=1}^{N}\sum_{i=1}^{S}\mu_{i,n}k_{i,n}R^2 \tag{H2}$$

we can see that **??** implies $\left\|\boldsymbol{w}^{(i,n)}\right\| \geq \sum_{n=1}^{N}\sum_{i=1}^{S}\mu_{i,n}k_{i,n}\lambda$ since $\boldsymbol{v}\cdot\boldsymbol{w} \leq \|\boldsymbol{u}\|\,\|\boldsymbol{w}\|$ by the Cauchy Schwarz inequality and $\|\boldsymbol{u}\| = 1$. For the base case $\boldsymbol{w}^{(avg,1)}$. So using **??** and the

fact that $\boldsymbol{w}^{(avg,0)} = 0$ and

$$\boldsymbol{u} \cdot \boldsymbol{w}^{(avg,1)} = \sum_{i=1}^{S} \mu_{i,1} \boldsymbol{u} \cdot \boldsymbol{w}^{(i,1)} \geq \sum_{i=1}^{S} \mu_{i,1} k_{i,n} \lambda$$

To show **??**, the base case can be written as

$$\begin{aligned} \left\| \boldsymbol{w}^{(avg,1)} \right\|^2 &= \left\| \sum_{i=1}^{S} \mu_{i,1} \right\|^2 \boldsymbol{w}^{(i,1)} \\ &= \sum_{i=1}^{S} \mu_{i,1} \left\| \boldsymbol{w}^{(i,1)} \right\|^2 \\ &= \sum_{i=1}^{S} \mu_{i,1} k_{i,n} R^2 \end{aligned}$$

This expression is found by first applying Jensen's inequality followed by **??** as well as the fact that $\left\| \boldsymbol{w}^{(avg,0)} \right\|^2 = 0$. Now considering the general case, $\left\| \boldsymbol{w}^{(avg,N)} \right\|^2$

$$\begin{aligned} \boldsymbol{u} \cdot \boldsymbol{w}^{(avg,N)} &= \sum_{i=1}^{S} \mu_{i,N} \left(\boldsymbol{u} \cdot \boldsymbol{w}^{(i,N)} \right) \\ &\geq \sum_{i=1}^{S} \mu_{i,N} \left(\boldsymbol{u} \cdot \boldsymbol{w}^{(i,N-1)} + k_{i,N} \lambda \right) && \text{(Using **??**)} \\ &= \boldsymbol{u} \cdot \boldsymbol{w}^{(avg,N-1)} + \sum_{i=1}^{S} \mu_{i,N} \boldsymbol{u} \cdot \boldsymbol{w}^{(i,N-1)} k_{i,N} \lambda \\ &= \geq \left[\sum_{n=1}^{N-1} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} \gamma \right] + \sum_{i=1}^{S} \mu_{i,N} k_{i,N} \\ &= \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} \gamma \end{aligned}$$

Which thusly proves **??**. To show the inductive case for **??** we have

$$\begin{aligned}
\left\|\mathbf{w}^{(\text{avg},N)}\right\|^2 &\leq \sum_{i=1}^{S} \mu_{i,N} \left\|\mathbf{w}^{(i,N)}\right\|^2 \\
&=\leq \sum_{i=1}^{S} \mu_{i,N} \left(\left\|\mathbf{w}^{(\text{avg},N-1)}\right\|^2 + k_{i,N}R^2\right) \\
&= \left\|\mathbf{w}^{(\text{avg},N-1)}\right\|^2 + \sum_{i=1}^{S} \mu_{i,N} k_{i,N} R^2 \\
&\leq \left[\sum_{n=1}^{N-1} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} R^2\right] + \sum_{i=1}^{S} \mu_{i,N} k_{i,N} R^2 \\
&= \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} R^2
\end{aligned}$$

This shows **??**. Combining the results of **??** and **??** as well as the fact $\left\|\mathbf{w}^{(\text{avg},N)}\right\| > \mathbf{u} \cdot \mathbf{w}^{(\text{avg},N)}$

$$\begin{aligned}
\left[\sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n}\right]^2 \gamma^2 &\leq \left[\sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n}\right] R^2 \\
\sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} &\leq \frac{R^2}{\lambda^2}
\end{aligned}$$

This inequality tells us that if the weight are distributed uniformly then the number of mistakes is bounded and that there is enough convergence between iterations to guarantee convergence.

4. 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 for our model to learn off

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

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

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

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

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

Let's introduce 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

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

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 get around this? One method is to soundly combine models from different processes in the hopes of achieving a weight vector had the SGD was instead 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 $S_T(\boldsymbol{w})$ represent the SGD computation of a training T from an initial weight vector

$\boldsymbol{w}$, for example $S_{T_1}(\boldsymbol{w}_g) = \boldsymbol{w}_1$. To come up with a model combiner we need to think about how we can calculate

$$S_T(\boldsymbol{w} + \Delta\boldsymbol{w}).$$

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

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

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 [**?**] 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

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

The algorithm for parallel SGD is summarised in algorithm **??**

Algorithm 7: 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

$\{T_1, T_2, \ldots, T_p\} \leftarrow$ an equal partition of T;
for $T_i \in \{T_1, T_2, \ldots, T_p\}$ **concurrently do**
  Compute $S_{T_i}(\boldsymbol{w_g})$ and $M_{T_i}(\boldsymbol{w}_g)$;
end
for $i \in \{1, 2, \ldots, p\}$ **do**
  $\boldsymbol{w}_i \leftarrow S_{T_i}(\boldsymbol{w_g}) + M_{T_i}(\boldsymbol{w}_g) \cdot (\boldsymbol{w}_{i-1} - \boldsymbol{w}_g)$;
end

Result: $\boldsymbol{w}_p$

5. 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 performance. 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 hundreds 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 cores and 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 from home 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 typing `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 parallel code 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 find this code in the following git repository under the `src` folder `https://github.com/Michae1CC/STAT4402_TUTORIAL`. Exit out of vim by pressing `ctrl+c`, 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. These parameters include things such as how many cores to use, how much memory to use, what programs to execute, where to put the output should go 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 our job to the HPC, and then checkout or standard out and standard error files generated by the submitted job. The standard out and error files filenames and paths are specified by the `#SBATCH --output` and `#SBATCH --error` parameters in the batch file, respectively. As you can see from out `parallel_python.out`, we get an array sum of 4928.8167 which is what we would expect for an array of 10000 with elements that are generated using $U[0, 1]$. If you get any errors from this example, try checking your `parallel_python.err` to see what sort of errors are being thrown.

6. Questions

Q1) **a)** Suppose the training set

$$T = \{([0,1],0),([1,1],0),([2,1],0),([2,2],0),([5,8],1),([6,7],1),([5,7],1),([4,5],1)\}$$

was partitioned into

$$T_1 = \{([2,1],0),([2,2],0),([5,8],1),([6,7],1)\}$$
$$T_2 = \{([2,1],0),([2,2],0),([5,8],1),([6,7],1)\}$$

for a k-NN parallel algorithm using the 2 closest neighbours and the Manhattan distance metric to classify the an unknown sample $\boldsymbol{x} = [0,0]$ use two processes. What 2 closest neighbours would be returned by each of the processors?

b) Would the parallel k-NN algorithm still work if only $k-1$ closest neighbours was instead returned by each process?

c) i. Implement your own parallel k-NN algorithm from scratch in python, only using the `threading` and `queue` libraries to perform parallelism. Use `sklearn`s iris dataset to test your model, you can access this data set using the import `from sklearn import datasets` and then creating the dataset by using `datasets.load_iris()`.

ii. Assuming the completion of the question above, implementing these parallel algorithms yourself can be tedious and error prone. Forturnately many parallel Machine Learning algoirthms and models have been implemented for us already in `python`s `sklearn` library. Like `pytorch`, `sklearn` is free to use python library that implements many of the Machine Learning tools seen in this course. Use `sklearn`s `KNeighborsClassifier` class to train on the *breast cancer* data. Leave $\frac{1}{5}$ of the data aside to provide a test accuracy. Submit this script to `golith` using a job batch file. Use repeat training with $1-4$ threads and record your results. *Hint:* you will need to adjust the `n_jobs` parameter in the `KNeighborsClassifier` to chnage the number of threads used for running the k-NN algorithm.

Number of threads	1	2	3	4
Time (sec)	0.03754734	0.02993416	0.0252390	0.01833604

7. Answers

Q1)

a)

The neighbours to be returned by the first and second process would be $\{([2,1],0),([2,2],0)\}$ and $\{([2,1],0),([2,2],0)\}$ respectively.

7.0.1. **b)**

No, all k closest neighbours may randomly be given to a single process. If some of these k closest neighbours were neglected by this process then not all the k closest neighbours will be used in the final reduction step which may give differing results to the serial k-NN algorithm.

7.0.2. **c)**

7.0.3. **i)**

The code for the k-NN algorithm implemented from scratch is shown below

7.0.4. **ii)**

An example batch file for job submission is shown below

Code to training a parallel k-NN algorithm using `sklearn` is given below

The times for running the parallel algorithm with different numbers of processes is shown below