STA141C: Homework 2

Wangqian Miao

May 21, 2018

Environment

- 8GB RAM, Intel i5-6200U laptop.
- Python 3.6 on Windows.

1 Problem 1

1.1 Algorithm

When we want to calulate the distance from each row of `X_test` to other rows, we know that this process is independent and that is why we can apply the multiporcessing in KNN. The main method I use in this problem to realize parallelization is described as following.

1. Cut `X_test` into four parts with equal size.
2. Create the args (use `list in python`) for `go_nn` (prob. provided) with differen parameters.
3. Use `pool.starmap(go_nn, createArgs())` to realize parallelizetion, return a list.
4. Caculate the average result from 3. then we can find the final accuraccy.

1.2 Results and Aanalyse

Method	Time consuming	Accuracy(%)
Single thread	173.1689 s	79.40
By `Pool.map`(4 processes)	75.3.4427 s	79.40
By `Pool.apply`(4 processes)	110.3971 s	79.40

There exists some interesting result from the exprement.

1. Use `Pool.map` will accelate the code better than `Pool.apply` and it is what I use in the last.
2. Even though I use 4 processes in my code, I still cannot speed up the code 4 times as what it was.

2 Problem 2

2.1 Algorithm

In gradient decent, we can realize parallelization when we update $\boldsymbol{\omega}$. In the formula how we update the parameters, it is not hard to find that:

$$\nabla f(\boldsymbol{\omega}) = \sum_{i=1}^{N} \frac{-y_i}{1+\mathrm{e}^{-y_i \boldsymbol{\omega}^{\boldsymbol{T}} \boldsymbol{x_i}}} \boldsymbol{x_i} + \lambda \boldsymbol{\omega} \tag{1}$$

$$= \sum_{i=1}^{i=4} \left(\sum_{j=1}^{N/4} \frac{-y_{ij}}{1+\mathrm{e}^{-y_{ij} \boldsymbol{\omega}^{\boldsymbol{T}} \boldsymbol{x_{ij}}}} \boldsymbol{x_{ij}} \right) + \lambda \boldsymbol{\omega} \tag{2}$$

And that is why I can parallelize the code, the method is as following.

1. Cut `X_test,y_test` into four parts with equal size.
2. Create the args (use `list in python`) for `gradient` (as HW1) with differen parameters.
3. Use `pool.starmap(gradient, createArgs())` to realize parallelizetion, return a list.
4. Caculate the sum of result from 3. then we can find the update parameter $\boldsymbol{\omega}$.

2.2 Results and Aanalyse

The dataset `news20.binary.bz2`.

Method	Time consuming(s)	Training Accuracy(%)	Test Accuracy(%)
Vectorization by sparse matrix	764.1907	96.18	91.93
`Pool.map`(4 processes)	2420.9848	96.46	92.43

The dataset `data_files.pl`.

Method	Time consuming(s)	Training Accuracy(%)	Test Accuracy(%)
Vectorization by numpy_ndarray	1.4320	76.18	75.30
`Pool.map`(4 processes)	39.79	76.18	75.30

The observation is as following:

1. The parallelized code costs much more time than the single thread code. It needs 3 times as what it was to update parameters in each iteration for `news20.binary.bz2`, 30 times for `data_files.pl`.
2. The accuracy in the last does not change which means my algorithm is corect.

The reasons why it takes more time is as followed:

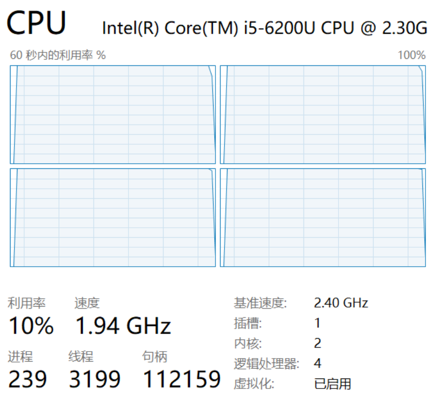
1. I realize vectorization in my code in HW1. When we use the sparce-matrix, ndarray and broadcast in `numpy`, it has helped us speed up a lot because they are all written in `C++, Fortran`.

2. The dataset `news20.binary.bz2` is quite large, even though I cut the data into four parts, it may come across cache collison when I use `Pool.map` which needs extra CPU clock cycles.

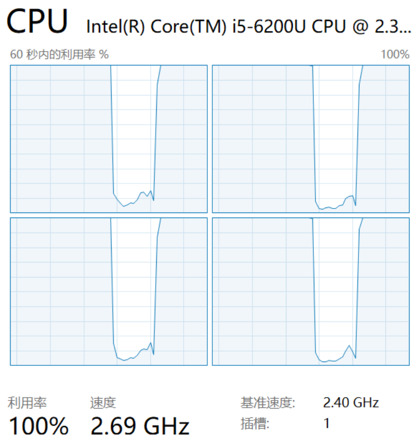
3. Using `with Pool(4)` will slow down the process, because it will open and close the process many times and waste a lot of time.

3 CPU status

The following figures show that when I use the parallelized code, the CPU is fully used.



(a) CPU status for problem 1: parallelization



(b) CPU status for problem 2: parallelization