5-2018

Improving IRWLS algorithm for GLM with Intel Xeon Family

Zhenzhi Xu
Purdue University

IMPROVING IRWLS ALGORITHM FOR GLM WITH INTEL XEON FAMILY

by

Zhenzhi Xu

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the Degree of

Master of Science

Department of Computer and Information Technology

West Lafayette, Indiana

May 2018

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Baijian Yang, Chair

Department of Computer and Information Technology

Dr. Tonglin Zhang

Department of Statistics

Dr. Byung-Cheol Min

Department of Computer and Information Technology

Approved by:

Dr. Eric T. Matson

Head of the Graduate Program

Dedicated to my father and my husband.

ACKNOWLEDGMENTS

I wish to gratefully acknowledge my thesis committee for their insightful comments and guidance and my family for their support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

MLE	Maximum Likelihood Estimation
GLM	generalized linear model
IRWLS	Iteratively reweighted least squares

ABSTRACT

Author: Xu, Zhenzhi. M.S.
Institution: Purdue University
Degree Received: May 2018
Title: Improving IRWLS algorithm for GLM with Intel Xeon Family
Major Professor: Baijian Yang

This study investigates utilizing the characteristics of Intel Xeon to improve the performance of training generalized linear models. The classic approach to find the maximum likelihood estimation of linear model requires loading entire data into memory for computation which is infeasible when data size is bigger than memory size. With the approach analyzed by Zhang and Yang (2017), the process of model fitting will be achieved iteratively through iterating each row. However, one limitation of this approach could be the iterative manner will impact performance when implementing it on Intel Xeon processor which delivers parallelism and vectorization. The study will focus on the tuning of application process and configuration on Xeon family processor based on the architecture of GLM model fitting algorithm.

CHAPTER 1. INTRODUCTION

This chapter will briefly introduce the thesis by discussing the problem statement and the questions raised about this specific problem. After that, it will specify the scope, significance, definition as well as assumption for the research of this thesis.

1.1 Problem Statement

The volume of data spawned worldwide grows over 50% per year (Reinders, 2017), thus the demand of interpreting data leads to the development of big data application that helps extract, manipulate, analyze and gain insights from the huge scale of data.

As one can see, many machine learning frameworks are proposed to optimize the performance of big data analytics application, like Spark ("Apache Spark - Lightning-Fast Cluster Computing", 2017). And meanwhile, at the hardware layer, architecture techniques are improved to satisfy the need of high performance as well. Intel Xeon family processor delivers massive parallelism and vectorization to support the most demanding high performance computing applications. The chips are designed to run scientific tasks and believed to dominate the market of machine learning applications. However, as different analytics applications have disparate architectures, it's extremely hard to identify the best configuration for the optimized performance.

The question of this study will focus on logistic regression model, one specific machine learning algorithm in generalized linear regression family, to do tuning of application process and configuration on Xeon family processor based on the architecture of this algorithm.

The Xeon family Processor is using AVX2, which is known as SIMD instruction which is specialized in performing the same operation on multiple data points simultaneously. Besides, the application's thread count is also an important factor to further explore. Increasing the application's thread count to maximum may be harmful as

it leads to increases in synchronization overhead and load imbalance (Heirman et al., 2014b).

As the behavior varies among different machine learning algorithms, this paper will only apply to specific workflow of machine learning analytics application. Besides, the experiment will be operated on Xeon family processor, however the result may be instructive for other SMT processors.

1.2 Research Question

The research question focuses on how to utilize the characteristic of Intel Xeon to improve the performance of training generalized linear models. This leads to two subquestions, which are:

- To utilize all the available cores for the computation
- To feed the data and organize the algorithm in the vectorized manner

1.3 Scope

The main goal delivered by this work is to study the attributes of Intel Xeon family, which is designed for scientific computation, and to find the opportunity of utilizing the hardware advantage of this specific processor for distributed big data analysis application.

This research will aim to improve the performance of application on vector extension instruction set by vectorizing the huge amount of input data and feeding them into processor for operating simultaneously. To determine the effectiveness of the proposed solution, the analytics speed and computational usage of classic approach of IRWLS and solvers from popular Python machine learning library will be recorded as baseline for comparison purposes.

Different data analytics application or workload will significantly impact the optimization process. The application in this thesis will be a logistic regression model which is used for explaining the relationship between one dependent binary variable and

selected features. With the determined model, the author will examine the impact of different workload size for the implementation of the proposed method and quantify the result with statistic result.

1.4 Significance

The demand of interpreting data leads to the rapid development of big data analysis model and hardware optimization for scientific computation support. This study seeks to implement the new statistical analytics model on Intel Xeon processor which is highly-anticipated to play an important role in large-scale high performance computation.

The main goal of this analysis will focus on address the factors in the hardware level that can impact the efficiency of data analysis and maintain the RAM-friendliness at the same tie. The proposed methods will be performed on Intel Xeon processor only, but it can also be implemented in other similar advanced processor like POWER 8 as well.

1.5 Definitions

AVX512 Instruction Set - These instructions represent a significant leap to 512-bit SIMD support. Programs can pack eight double precision or sixteen single precision floating-point numbers, or eight 64-bit integers, or sixteen 32-bit integers within the 512-bit vectors. (Detica, 2012)

logistic regression - The logistic regression is a widely used multivariate method for modeling dichotomous outcomes, in which the probability P of an outcome is related to a series of potential predictor variables.(Chen & Zhou, 2017)

1.6 Assumptions

The research is performed using the following assumption:

- The flight dataset fairly represents all and late flights during the past 22 years and over 95 percent of the data should be complete.
- The records in the dataset have no cause-effect relation which allows data partition for vectorization purpose and workload test.
- The solution using vectorization and parallelism will be adaptable or partially compatible when running on other multi-threading processor, like POWER8.

1.7 Limitations

The study is limited by the following conditions:

- The test is only conducted with specific programming languages while the performance may vary when using different languages.
- The dataset used in the experiment is stored and cleaned before testing. Thus, the consumed time will only cover the data-loading time and calculation time.

1.8 Delimitations

The study is conducted acknowledging the following delimitations:

- Algorithms other than logistic regression model are not included.
- The research will not be tested on processor other than Intel Xeon family coprocessor.
- The research is limited to the workload that is smaller than 20 GB
- The processing of data is not considered in this study.

1.9 Summary

This chapter introduced the research question along with its specifications. The classic implementation of logistic regression fails to take full advantage of the latest high performance technology in several aspects, like the way of data storage and the design of loop in the algorithm. Thus, this thesis will focus on the refinement of a specific machine learning algorithm in generalized linear regression family to see the improvement of performance one can achieve on many-core processor.

CHAPTER 2. REVIEW OF RELEVANT LITERATURE

The demand of interpreting data leads to the development of big data application that helps extract, manipulate, analyze and gain insights from the huge scale of data. To achieve higher performance of data analysis, many frameworks are proposed to facilitate big data analytics application and meanwhile, at the hardware layer, architecture techniques are enhanced for accelerating computation as well.

This chapter focuses on the overview of previous study in data analysis and parallel computing to understand the existing methodology of this area and the current challenge researchers are faced with. The chapter consists of three sections, starting with the summary of generalized linear models for big data, which will be the sample model in the paper. Then the second section presents the review of Spark architecture, which the experiment is performed on and followed by the third section about Intel Xeon family Co-processor that provides simultaneous multithreading for big data workload. It also gives a review about the papers that discuss tuning data analysis with multithreading processors and their achievement.

2.1 Generalized Linear Models

Ordinary linear regression is one of the fundamental and common models for finding the correlation between parameters in data analysis, however it's usually assumed to have a normal distribution that may be hard to achieve in some response. Thus, generalized linear models (GLMs) were proposed for achieving maximum likelihood estimates of the parameters with observations distribution according to some exponential family and systematic effects that can be made linear by transformation (McCullagh & Nelder, 1994). GLMs broaden the applicable distributions, like normal, gamma, binomial, binary, multinomial and Poisson, for ordinary linear regression model. Because the normal distribution is hard to achieve in industry, the approach of GLMs become widely-utilized in many fields for data analysis.

Rodriguez-Alvarez and Garrison (2016) used the generalized observation in a delay-Doppler map (DDM) for forecasting of tropical cyclone genesis and intensification. They optimized the samples using three methods, including maximum signal-to-noise ratio, minimum variance of the wind speed, and principal component analysis (PCA). PCA turned out to be the best performance in this particular scenario. Tanaka et al. (2016) examined the impacts of environmental factors on acoustical behavior on marine mammals using generalized linear model to avoid the effects of the time period and tidal change to the vocalization rate and assumed negative binomial distribution for build GML. Lee, Tak, and Ye (2011) developed a novel data-driven GLM for functional MRI analysis. Although independent component analysis (ICA) is widely-accepted for fMRI analysis, still recent studies show that actually ICA cannot guarantee the independence of simultaneous activity patterns in human brain. Thus, they proposed a data driven GLM based on the sparsity of signal. Fichte et al. (2016) applied GLMs to evaluate nuclear electromagnets pulses tests. Because the valid data sets for statical analysis are usually not achievable in industries like the nuclear field, they processed the data using GLM model in order to more sturdy and applicable prediction.

To obtain the optimization of GLMs based on the non Gaussian distributed response, several statistical methods could be used for finding the maximum likelihood estimates (MLE) for GLMs. The most popular ones are the Newton-Raphson, the Fisher Scoring, and the iteratively reweighted least squares (IRWLS) methods (Zhang & Yang, 2017). To start with each of these methods, one should initiate a guess of the solution and then iteratively calculate the next round by solving a weighted least squares problem based on the previous guess. finally the MLEs are obtained if the algorithm of method converges.

GLMs are a family of traditional regression models, including linear regression, logistic regression and Poisson regression. The logistic regression model is chosen in this study as the representative of GLMs family. The independent variables could be either continuous or categorical, which covers most of the response and dependent variable is categorical in logistic regression. This means This output of the model can be only two values which are zero and one. This attribute of the model fits many cases in industry in

real life, like automatic disease detection based on exploring the risk factors for some specific diseases. Alzheimer's disease is a regenerative brain disorder that affects elderly people and has got increasingly emphasis due to the aging of population. Barros and Silveira (2017) grouped features according to anatomical regions of brain and applied the method to MRI images from ADNI and compared the performance with the one of other sparse methods for AD. They proposed an evaluation using logistic regression for Alzheimer, which achieved both classification and the stability of the feature weight. After the analysis with logistic regression, they got the weight of each independent variables to figure out which are the determinant factors and did prediction according to the fitted model. Besides, the method proposed attained higher performance and more stable result.

Indra, Wikarsa, and Turang (2016) classified tweets into a set of topics using logistic regression. The information of current trends is critical for web-based applications like Tweeter. Hence, their research targeted to figure out the area of interests in real-time. Not like laboratory investigation, they handled many tasks, like ETL (extract, transform, and load), converting real tweets into feature vector consisted with words, before doing the machine learning process. They trained the model with 1800 labeled tweets ad evaluated with another 1800 ones and the result showed the accuracy to be 92%, which is very high for data analysis.

The traditional thoughts of data analysis is to use model for information exploits alone, but recently researchers are performing analysis with nested models. Liu, Fowler, and Zhao (2017) proposed to use support-vector machines (SVM) classification with spatial contest and include them into logistic regression to provide the probabilistic output. The approach was proven to gain higher accuracy when compared to two prominent families of spatial-spectral SVM classifiers, composite kernels and postprocessing regulation.

2.2 IRWLS

As discussed in the previous section, IRWLS is one of the most popular approach of finding the MLEs of GLMs. To achieve the goal, it starts from loading the whole

dataset to calculate an initial solution, and then obtains the next guess by re-iterating the whole set based on the previous round of iteration. The MLEs is finally attained when the algorithm converges. However, the calculation which depends on the whole dataset could be infeasible when the data is overwhelming for the RAM, especially for single machine which has comparatively limited memory size.

Suppose a dataset is loaded as an $m \times n$ dataframe, organized with several columns of features and one column of response associated with the feature, Let y defined as the m-dimensional vector of the expected response and (X plus a intercept) as the $m \times n$ matrix of variables. Then the relationship could be expressed as follow:

$$y = X\beta^T + \varepsilon, \varepsilon \sim N(0, \sigma^2 I_m) \tag{2.1}$$

The target of logistic regression training process is to find the MLEs of β, ε and the variance-covariance matrix of β.In the classic approach of IRWLS, one need to read the whole set to for training as:

$$\hat{\beta} = (X^T X)^{-1} X^T y \tag{2.2}$$

$$\hat{\varepsilon}^2 = y^T |I_M - X(X^T X)^{-1} X^T| y/(m-n) \tag{2.3}$$

$$\hat{V}(\hat{\beta}) = \varepsilon^2 (X^T X)^{-1} \tag{2.4}$$

So the computation is feasible when the matrix is small, but memory error will occur once the data size used in IRWLS exceeds the capacity of RAM.

With the approach proposed by Zhang and Yang (2017), the process of model fitting will be achieved through iterating each row so that IRWLS is no longer blocked by the memory limitation. The approach is proposed based on the loglikelihood function of (2.1) as

$$l(\beta, \sigma^2) = -\frac{m}{2}\log(2\pi) - \frac{m}{2}\log\sigma^2 - \frac{1}{2\sigma^2}(s_{yy} - 2s_{xy}^T\beta + \beta^T S_{xx}\beta) \tag{2.5}$$

where $s_{yy} = \sum_i^m Y_i^2$, $s_{xy} = X^T y = \sum_1^m x_i Y_i$, and $S_{xx} = \sum_1^m x_i x_i^T$. Thus, the $l(\beta, \varepsilon^2)$ can be calculated through scanning each row of observed data and accumulate the s_{yy}, s_{xy} and S_{xx}. Finally $\hat{\beta} = S_{xx}^{-1} s_{xy}$, $\hat{\varepsilon}^2 = (s_{yy} - s_{xy}^T S_{xx}^{-1} s_{xy})/(m-n)$, and $\hat{V}(\hat{\beta}) = \sigma^2 S_{xx}^{-1}$

2.3 Intel Xeon family Coprocessor

Besides the distributed data analytics engine, high performance computing is also taking vital role for accelerating computation performance. Nowadays, many computer architectures are using vector processing units for high performance of computation. The Intel Xeon Phi coprocessor is using AVX-512 which is the latest x86 vector instruction set and it's compatible with the previous SIMD (Single Instruction Multiple Data) schemes, like SSE and AVX. AVX-512 instruction is known as SIMD instruction which is specialized in performing the same operation on multiple data points simultaneously. Thus, data partitioning will help the algorithm to meet the concept of single instruction and multiple data in AVX-512. Besides, the application's thread count is also an important factor to further explore. Increasing the application's thread count to maximum may be harmful as it leads to increases in synchronization overhead and load imbalance (Heirman et al., 2014b).

One popular approach of utilizing parallelism of Xeon Phi is to directly use the Intel Math Kernel Library provided officially along with the coprocessor and automatic offloading to coprocessor. The MKL library popular now for scientific computing and gradually replacing the previous version of library. For example, MKL is now included in Anaconda, providing lower-level support for numpy which is the most useful python library for math computation. Without any coding, programmer can take the advantage of SIMD by just calling MKL under numpy to take care of resource allocation for parallelism. El-Khamra et al. (2013) evaluated the performance of R application on Xeon Phi with two testing workloads, including a widely-used R25 benchmark and a practical sample set in health informatics. The experiment shows that the speedup gained by using Xeon Phi coprocessor highly depends on the workload, usually for the small matrix sizes, the cost of offloading counteracts the benefits from parallelism. But when talking about

huge matrix sizes, the speedup could be dramatic. The highest speedup gained using MKL is approximately 60x with two coprocessors (240 threads in total) each with 40% of the selected workload. The experiment was conducted in a relatively black box manner as it's without any modification of the existing code but just substituted the previous library with MKL library. However it shows the unprecedented performance improvement achieved by parallelization from Intel Xeon Phi coprocessor.

Other than utilizing MKL library provided by Intel, researchers are also seeking the possibility of optimizing their codes to accommodate the SIMD operation mode in order to gain benefit of parallelism. According to the data analysis requirement, many of the scientific computation algorithms focus on a batch of small size matrix. Thus, one possible solution for performance improvement is to refactor these algorithms so that batches of matrices can be computed simultaneously.

Adelstein-Lelbach, Johansen, and Williams (2017) proposed an approach to solve independent banded matrix problems using SIMD architectures. Suppose there is a 3D Cartesian grid for calculation, the strategy used by MKL solver is to extract each vertical column of elements in the grid and solve them independently which perfectly utilizes the nature of task parallelism but fail to use vectorization of SIMD architecture. To improve it, each tile of vertical columns is extracted and all the columns in this tile are simultaneously solved, interleaving the computation of individuals. The solver supports four layout and tiling scheme combinations according to different partition methods and each of them shows different improvement due to different Intel architecture. This approach well demonstrates task parallelism and vectorization for matrix solving. Comparison is done on three Intel architectures with different cache, vectorization, and threading features: Intel Ivy Bridge, Haswell, and Knight's Landing (which is the latest version of Xeon Phi coprocessor). The approach turns out to achieve improvement on each of the test platform, attains 2x speedup over the MKL solver on Xeon platforms and gains approximately 12x speedup on Knight's Landing.

Vector memory provides sufficient data bandwidth for Processing Elements (PEs) through multiple memory banks. Some common structures can achieve ideal memory bandwidth due to their access patterns in vector memory, however, for some others, such

as sparse-matrix, independent histogram whose access locations are random and can not be predicted in advance, the benefit of vector-SIMD is not that obvious. The vector memory which only supports the common access patterns will cause a low utilization of the memory bandwidth and a long unhidden memory latency in these applications. Tan, Chen, Liu, and Wu (2017) proposed a model for gather and scatter operations on local vector memory for sampling. With the result, researchers got the distribution of access locations, the probability of access conflicts, and also the guidance for performance optimization.

Besides memory bandwidth, there are also other bottlenecks, including memory latency, workload imbalance and computation, which could be eliminated for higher performance. Elafrou, Goumas, and Koziris (2017) emphasized on the attributes of different computation architectures and proposed a low-overhead optimizer for sparse matrix-vector multiplication on the Intel Xeon Phi. They first designed heuristics that determine the bottleneck(s) of a matrix based on the estimated performance bounds. Then defined two classifiers, profiling-based and feature-based, to represent performance bottlenecks for each problem and classify if the problem should be handed-tuned or trained with machine learning approach. Finally, they evaluated the optimizer on Intel Knights Corner (which is the earlier version of Knight Landing coprocessor) and figured out it optimizing sparse matrix-vector multiplication appropriately for most of large matrices and resulted in significant speedups over the corresponding widely-used compressed sparse row implementation in the latest Intel MKL library.

There are several APIs in the market for developers to use for optimizing the code according to lower-level architecture. Thus, besides refining the manner of data input, researchers are also utilizing these APIs for improving parallelism. Ponte, Gonzlez-Domnguez, and Martn (2017) explored the parallelism of Xeon Phi coprocessor with OpenMP which is a parallel programming API to evaluate its performance in using SIMD directives. OpenMP provides explicit vector programming through SIMD directives that helps to optimize the existing code for the characteristics of Intel Xeon Phi coprocessor. Three different applications, including matrix multiplication, Poisson equations and Molecular Dynamics, were computed and optimized for the purpose of

vectorization. To illustrate, the vector loop of matrix multiplication is modified with previous matrix transposition and OpenMP SIMD clauses. They did the comparison of speedup of the auto-vectorization and the OpenMP optimized version to see the achievement by human intervention. The result turns out to vary a lot due to different problem size and application. OpenMP SIMD vectorization achieved up to 6.3x speedup compared to non-vectorized version of Poisson equation and 2x compared to auto-vectorization version. And it achieved approximately 4x speedup with respect to the auto-vectorization for Molecular Dynamics.

Intel Xeon Phi coprocessor delivers massive parallelism and vectorization to support the most demanding high performance computing applications. The chips are designed to run scientific tasks and believed to dominate the market of machine learning applications. However, as different analytics applications have distinct architectures, it's extremely hard to identify the best configuration for the optimized performance. The experiment by Ponte et al. (2017) shows us the dramatic difference due to input size which is similar to what's mentioned in MKL example and also application, thus researcher are looking at the possibility of delivering optimized solution automatically according to different scenarios.

Common practice for application designers to test performance of their application is to use a range of thread counts and see which thread count works best. Yet, this could be inaccurate because they overlook input set and phase behaviors.

Heirman et al. (2014a) were seeking the method of automatically find the optimum thread count at sub-application granularity by exploiting application phase behavior. They extended CRUST to take behavior of simultaneous multithreading on the Xeon Phi into account. The NAS Parallel Benchmarks are run with a specific input set on an Intel Xeon Phi 7120A system to illustrate the performance impact from applications. And one benchmark is chosen to be run with five different input sets of increasing size to demonstrate the effect of working set. The author explored the performance of using different per-core thread counts on an Intel Xeon Phi system, and showed how the optimum thread count varies across applications, when changing the input set of some applications. Then he integrated CRUST into the OpenMP runtime library; by combining

application phase behavior and leveraging hardware performance counter information it can reach the best static thread count for most applications. CRUST can automatically find the optimum thread count at sub-application granularity by exploiting application phase behavior at OpenMP parallel section boundaries, and uses hardware performance counter information to gain insight into the applications behavior. However, the CRUST is not compatible for nested parallelism, instead a more detailed analysis of parallelism at all nesting levels may be able to expose more detailed phase behavior.

As the behavior varies among different machine learning algorithms, the papers discussed above only apply to each specific workflow of machine learning analytics application. Besides, the experiment was operated on Xeon Phi processor, however the result of the experiments from other SMT processor like POWER8 could also be instructive.

Jia et al. (2016) explored the method to optimize SMT setting for Spark-based big data workloads on POWER8 dynamically for various machine learning algorithm and figured out the factors may affect the prediction-based dynamic SMT threading frameworks efficiency and decrease the performance improvement. In order to take the best use of the dynamic thread count adjusting, the applications should update the number of threads accordingly. The method in this paper proposes a framework to manage SMT configurations dynamically and embed it into Spark system. And the evaluations of this approach on a POWER8 system show that with the proposed method it can achieve up to 56.3% performance boosting and an average performance improvement of 16.2% over the default setting. However, there are still factors that may decrease the efficiency and performance improvement in the experiment of which the solution is still to be addressed, like prediction accuracy, sampling period delay, hardware penalty and prediction penalty. However, undeniably the prediction-based solution is truly qualitative leap for utilizing SMT in industry application scenario and makes SMT more practical and efficient for data analysis in real life.

2.4 Summary

This chapter reviewed three main components will be used in the study, including the regression model, the solver to fit the selected model and the manycore coprocessor which the study will perform experiment on.

The Row by Row approach of IRWLS attains the RAM-friendliness as well as the accuracy inherited from the classic IRWLS. This is crucial when the size of dataset is overwhelming for RAM and guarantees the feasibility and practicality of machine learning implementation.

When talking about data analysis, one another factor people care about is the performance. Performance can be optimized by utilizing manycore coprocessors like Intel Xeon family which is designed for parallelized computing required for scientific computation. However, the improvement is not that significant when implementing data analytics application directly on these hardware. Thus, researchers devoted a lot of efforts for optimization. One possible approach is to utilize Intel MKL library which serves between application and hardware to translate the code in a more parallel manner. And one can also refine his application with OpenMP which is a parallel programming API to evaluate its performance in using SIMD directives to achieve higher parallelization during its execution. Other than monitoring performance case by case, a black box approach was also proposed to learn the best configuration through job execution without understanding its internal logics.

CHAPTER 3. FRAMEWORK AND METHODOLOGY

3.1 Research Framework

The study will focus on examining the performance improvement of fitting logistic regression, when utilizing the parallelism and vectorization from Intel Xeon family coprocessor.

3.1.1 Testing Condition

The experiment will be performed with the following specifications:

- Anaconda-4.3.1 which provides packages along with python-2.7.13
- OpenMP 4.0 and Cython 0.25.2
- scikit-learn 0.19.0
- Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz with 4 cores and 8GB RAM
- 20 GB flight dataset to predict flight delay

3.1.2 Testing Procedures

First, the runtime and consumed memory with machine and core number will be recorded as the baseline of performance. The logistic regression algorithm provided by scikit-learn will be used.

Then, the refinement of parallelism will be done in several steps, including the multiprocessing with Python module, data partition and modification with OpenMP. The comparison of runtime and memory will be performed with the baseline and among different configurations of parallelism. The quantitative data will be collected and

analyzed for building the relation of this configurations with parallelism level on the processor.

After coming up with a refined model, the performance with streaming data will be examined for a further insight of how share memory in CPU helps or damage performance when facing different data pressure.

3.2 Unit & Sampling

This section will discuss about the sample used in the study and methodology of test will be done using this sample.

3.2.1 Hypotheses

The hypotheses will be:

- H_0: There is no positive effect of performance when running GLM model fitting on Intel Xeon coprocessor
- H_A: There is a positive effect of performance when running GLM model fitting on Intel Xeon coprocessor

3.2.2 Population

The population will be airline on-time performance, including origin airport, destination airport and duration, which helps to evaluate if a specific flight will delay. The reason of choosing it as target of population is because the application of machine learning in predicting trend in industries is a hot topic as shown in literature review. Also generating categorical result (delay or not in this case) fits the characteristics of logistic regression model.

3.2.3 Sample

The sample of log will be 20GB with the details of airline performance. To guarantee the randomness when examining the impact of different workload, the data will be shuffled before splitting.

3.2.4 Variables

To achieve the relation between the nature of application and their performance on Intel Xoen coprocessor, the following independent variables will be designed and manipulated,

- The different manner of algorithm
- The record number in each data partition
- The size and input method of workload for data analysis

3.2.5 Measure for Success

The refinement of logistic regression model does help the reduction of runtime and consumed memory when run on Intel Xeon coprocessor. Also, the RAM-friendliness is well-achieved.The speedups can be evaluated quantitatively.

3.2.6 Threats to Validity

The size of dataset is vital to the speedups compared to the baseline. Thus, the sample size used in this study just provides a snapshot of the performance improvement as it is relatively small when comparing to which in real case in industry. Also, the hardware used for testing is a four cores processor with RAM size to be eight GB, thus the statistics for more advanced processors like Intel Xeon Phi processors may vary.

CHAPTER 4. RESULTS AND DISCUSSION

This section will discuss the procedure of tuning IRWLS fitting process including,

- Examine the performance and accuracy of the Row by Row approach proposed by Zhang and Yang (2017) on Intel Xeon processor and profile the bottleneck in runtime
- Parallelize the procedure within Python, using multiprocessing library and data partitioning, to utilize the performance benefit provided by manycore processor
- After the attempt inside Python, the next step is to rewrite code with Cython and openMP to remove GIL, which is the security mechanism of Python implementation, for gaining multithreading
- After optimization for computation aspect, examine the performance of IRWLS fitting process for streaming data

The classic IRWLS approach for logistic regression and the model in scikit-learn will be used as benchmark for each performance analysis.

4.1 The concept of Row by Row IRWLS

According to the equation (2.5), instead of calculating over the whole dataset, IRWLS can also be achieved by scanning Row by Row which can efficiently solve the problem from the huge size of dataset. Firstly, it starts with calculating and accumulating the working sufficient statistics to attain the estimator of the logistic regression model from the whole observed dataset, and then iteratively redo the step until the algorithm converges.

To sum up, the work flow is as follow, With the implementation of the Row by Row IRWLS, the RAM limitation is removed and memory issue is well-resolved. However, speaking about an fitting process, accuracy and performance are the two main indicators that measure the overall quality of it. Moreover, to support the need of complex

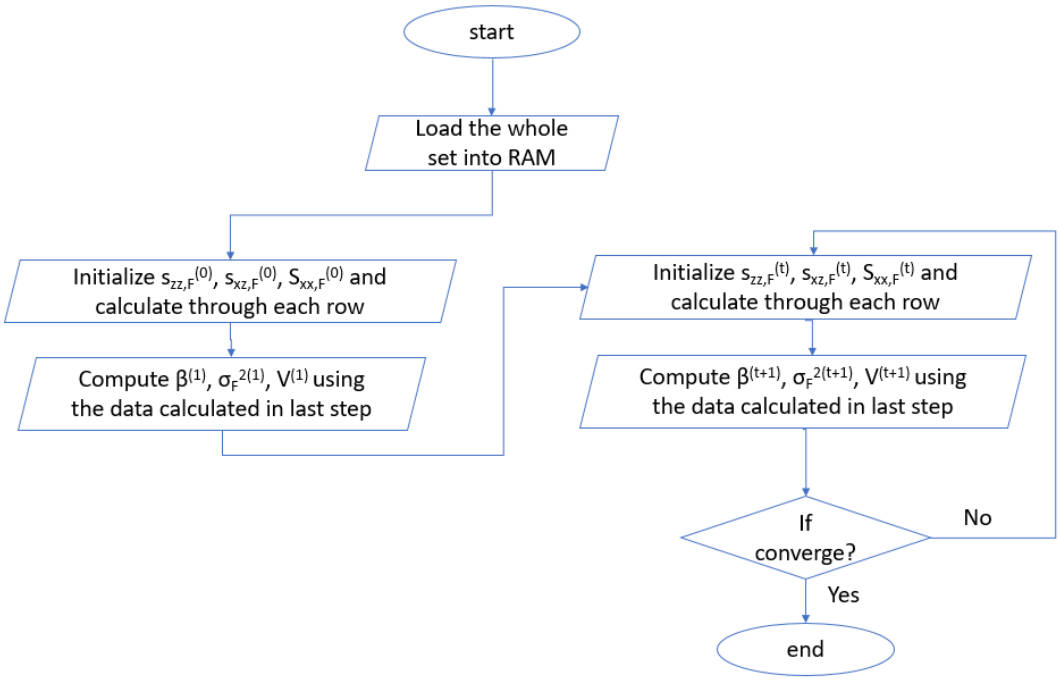

Figure 4.1. Flow Chart of Row by Row IRWLS

and massive scientific computation, many computer architectures are emerging and taking vital role for high performance computing. Thus, the experiment in the next part will focus the analysis of the accuracy and performance comparison of the new approach and several popular logistic regression models to have an initial assessment of the approach.

4.2 Initial assessment of Row by Row approach

Other than the classic IRWLS approach, scikit-learn is chosen as a benchmark to compare with the Row by Row IRWLS for several reasons:

- As both of the IRWLS approach in this assessment are implemented in Python, scikit-learn becomes the benchmark as it's the most popular Python machine library that offers various algorithm and widely accepted in industry
- The assessment focuses on the computation on single machine where scikit-learning is usually performed

- The logistic regression algorithm in scikit-learn is using several solvers which cover small data and big data situation respectively
- To handle huge data, the out of core technique of scikit-learn provides partial-fitting based on each chunk of streaming data

The dataset used to evaluate performance and accuracy is the 4.2 million records of one-year flight data in U.S.. The records provide massive details and several variables among them are selected for predicting if the flight delays more than 15 minutes. The chosen features are consisted with continuous variables and the prediction is target to fall in 1 or 0 which can be well resolved by logistic regression model.

Table 4.1. Summary of chosen variables

Variable	Definition
DISTANCE	The total distance between two airports
MONTH	The month that flight flies
DEP DELAY	If the flight delays when departs
AIRLINE ID	The airline ID of the flight
FL NUM	The flight number of the flight

All of the compared algorithms establish an an initial MLE of β in the first iteration and then adjust and optimize the β through following iterations. Thus, the accuracy and consumed time highly depends on the number of iteration performed inside each algorithm. The Figure 4.2 through 4.4 is based on the behavior of the default setting of scikit-learn approach and the earliest stable accuracy from IRWLS, according to the experiment result, classic IRWLS presents comparatively high accuracy and performance while the Row by Row solution loses the advantage of performance after the refinement. Both of the two versions reach 90.32% for accuracy after their first iteration of training, but the Row by Row solution uses approximately 10000X time compared to classic version. Meanwhile, the two solvers from scikit-learn are exhibiting different behaviors due to their properties.

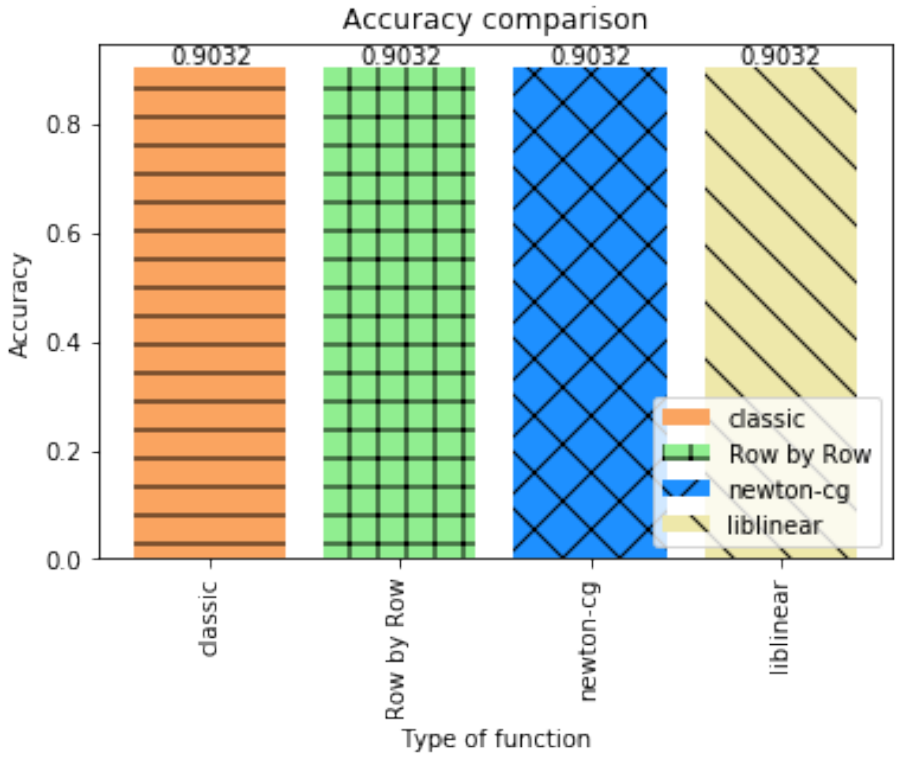

Figure 4.2. Initial assessment of accuracy with scikit-learn

Liblinear (Fan, Chang, Hsieh, Wang, & Lin, 2008) is the default algorithm from scikit-learn and performs very fast for large-scale classification problem. It uses coordinate descent method which solves the multivariable problem by keeping most of the variables, solving one variable, in this case each β_i in $\{\beta_1, \beta_2, ..., \beta_N\}$, at a time until finishing one iteration of the whole set of β. it avoids large matrix calculation and as a result, the computational complexity is much lower than Newton's method. The accuracy reached by default setting of Liblinear for this specific experiment is identical to the other three solvers after 19 round of iterations.

Newton-cg (Wang, Sun, & Toh, 2010) stands for Newton Conjugate Gradient, which is modified from Newton's method and uses conjugate gradient to solve the computational difficulties come from Hessian matrix inversion inside Newton method. This solver resembles IRWLS in structure and its feature of iteration, and outputs the accuracy of 90.32% in the above experiment.

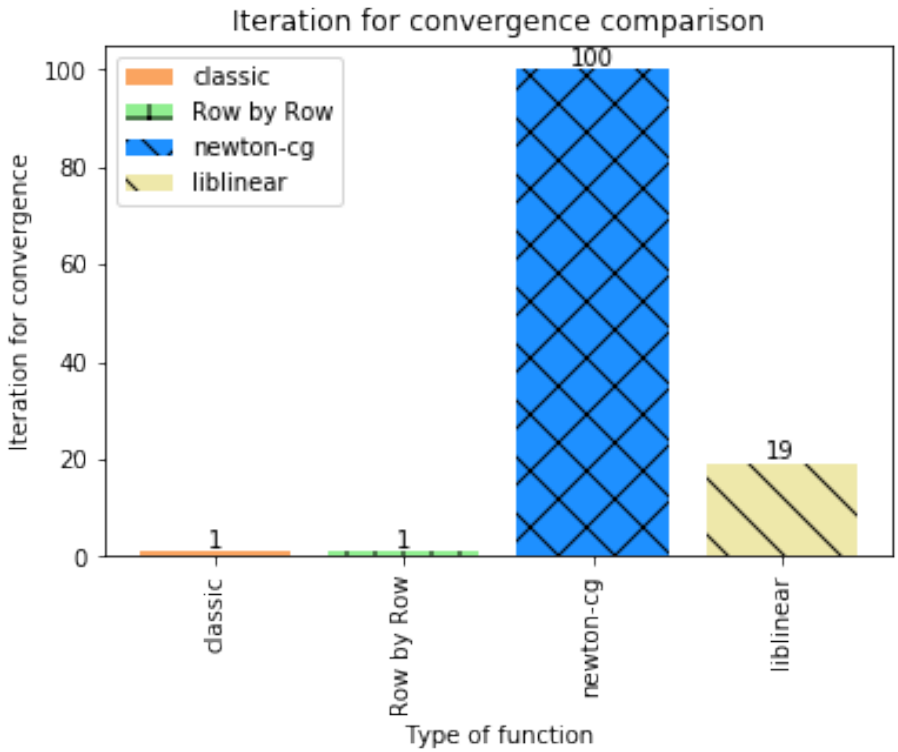

Figure 4.3. Initial assessment of iternationwith scikit-learn

The experiment demonstrates the advantage of IRWLS through training result but also exposes the drawback after implemented in Row by Row fashion with Python. Both of the IRWLS approaches exhibit high accuracy after the first iteration when most of the other approaches need more than 10 iterations to stabilize their result. However, the classic version of IRWLS is fully implemented with numpy-MKL matrix calculation which is compiled with C in backend and supported by BLAS (Basic Linear Algebra Subprograms). The BLAS prescribes a set of routines for commonly used linear algebra operations, like dot products and matrix multiplication, which are exactly what is needed in IRWLS. Thus, with the support of numpy, the huge $m \times n$ (with m stands for record number and n for feature number) matrix calculation is highly optimized and can be solved within few seconds. Contradictorily, iterating each row and calculate the $1 \times n$ matrix for m times will dramatically decrease the ability of parallelism offered by numpy. And with the thread-safe mechanism called GIL (Global Interpreter Lock) of Python, the

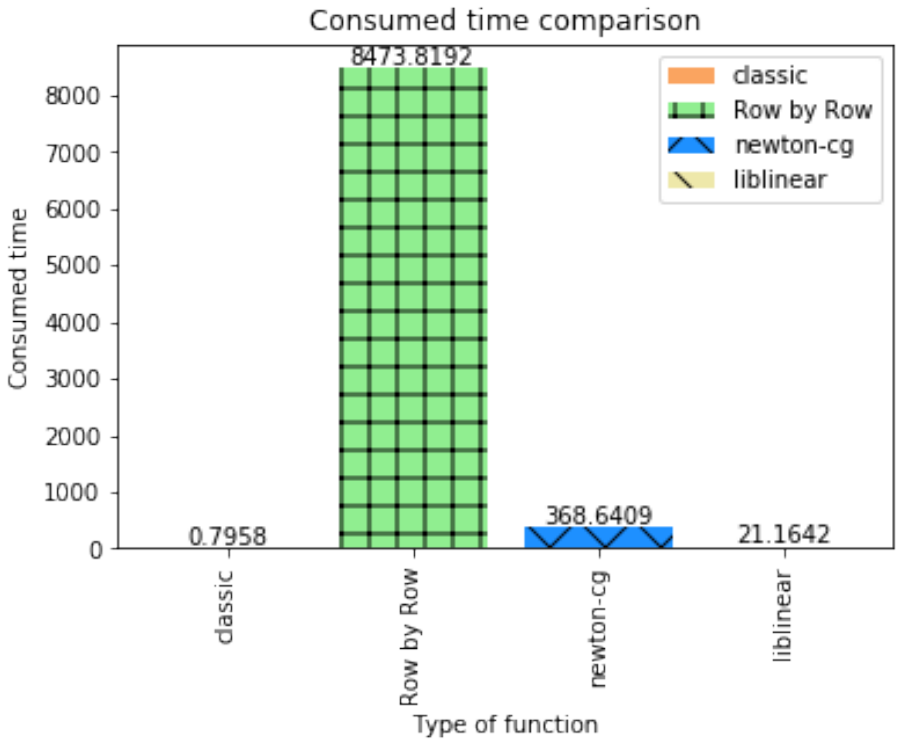

Figure 4.4. Initial assessment of consumed with scikit-learn

small matrix of each row is processed sequentially which fails to get the benefit from manycore processor. The cpu resource monitoring proves this statement as 4 cores (400% cpu) of the machine are fully occupied for classic version of IRWLS while only 100% cpu is in operation for Row by Row IRWLS.

In this initial assessment, classic IRWLS exhibits high accuracy and low time cost as the solver of logistic regression for the following reasons,

- reaches comparatively high accuracy in the first several rounds of iterations
- utilizes the parallelism from numpy-MKL for matrix calculation

however, due to the limitation of Python, it loses efficiency after the refinement for memory issue. Thus, the upcoming sections will be organized with several optimization methods based on Row by Row approach of IRWLS to seek the opportunity of high efficiency along with memory safety.

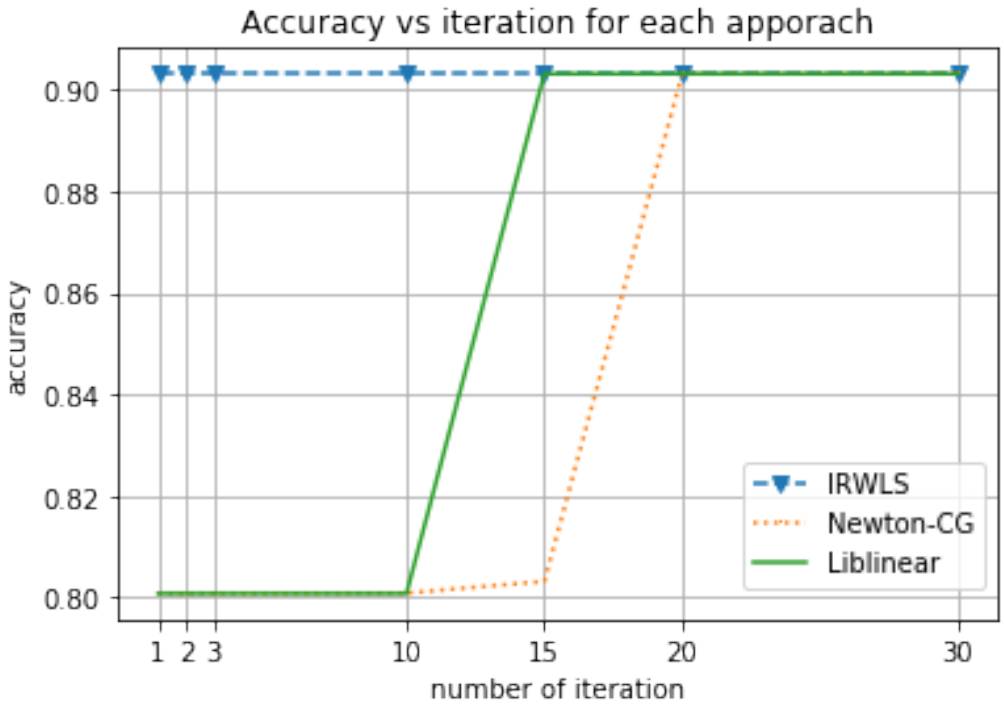

Figure 4.5. Accuracy Per Iteration compared with scikit-learn

4.3 Multiprocess for many-core processors

Multiprocessing is a Python module which supports creating subprocesses in order to fully leverage the manycore processor. The **Pool** object from multiprocessing creates a number of process pool according to machine specification and then feed partitioned data along with the to-be-parallelized Python, in this case the Row by Row IRWLS, to utilize all the cores provided by hardware. Each of the sub-process works on a sub-dataset, and calculates the corresponding $S_i = s_{zz\_i}, s_{xz\_i}, S_{xx\_i}$. After all the processes in all pools finished, S_i is accumulated and β is calculated eventually. Since there could be massive S_i depending on the number of processes created, the $\sum_i^n S_i$ is calculated recursively to reduce the computational complexity to $O(N \log N)$.

Similar to **Pool**, **Process** also conveys the multiprocessing attribute to the Python code but it only defines the number of processes to receive the data partition and code which comes more statically compared to **Pool**.

Algorithm 4.1 Calculate β using Multiprocess.pool for one iteration Row by Row IRWLS

Input: The observed dataset D, the number of core C and the target partition number N
Output: $\hat{\beta}$

Initiate the number of pool that equals to C: pool = Pool(C) and divide D into N partitions
While there is unprocessed data partition **do**
  Map the data partition and Row by Row IRWLS algorithm into the available pool
  While not reach the end of the partition **do**
    Calculate $S_i = s_{zz\_i}, s_{xz\_i}, S_{xx\_i}$ through each row
  end for
  Append the result to the list of S
end for
Calculate the sum of $S = \sum_i^n s_{zz,F}, \sum_i^n s_{xz,F}, \sum_i^n S_{xx,F}$
Calculate $\hat{\beta} = {S_{zz,F}}^{-1} s_{xz,F}$

Algorithm 4.2 Calculate β using Multiprocess.process for one iteration Row by Row IRWLS

Input: The observed dataset D, the number of core C
Output: $\hat{\beta}$

Initiate the number of process that equals to C and divide D into C partitions
For Each: Process in C **do**
  Feed the data partition and Row by Row IRWLS algorithm
  For Each: not reach the end of the partition **do**
    Calculate $S_i = s_{zz\_i}, s_{xz\_i}, S_{xx\_i}$ through each row
  end for
  Append the result to the queue of S
end for
Calculate the sum of $S = \sum_i^n s_{zz,F}, \sum_i^n s_{xz,F}, \sum_i^n S_{xx,F}$
Calculate $\hat{\beta} = {S_{zz,F}}^{-1} s_{xz,F}$

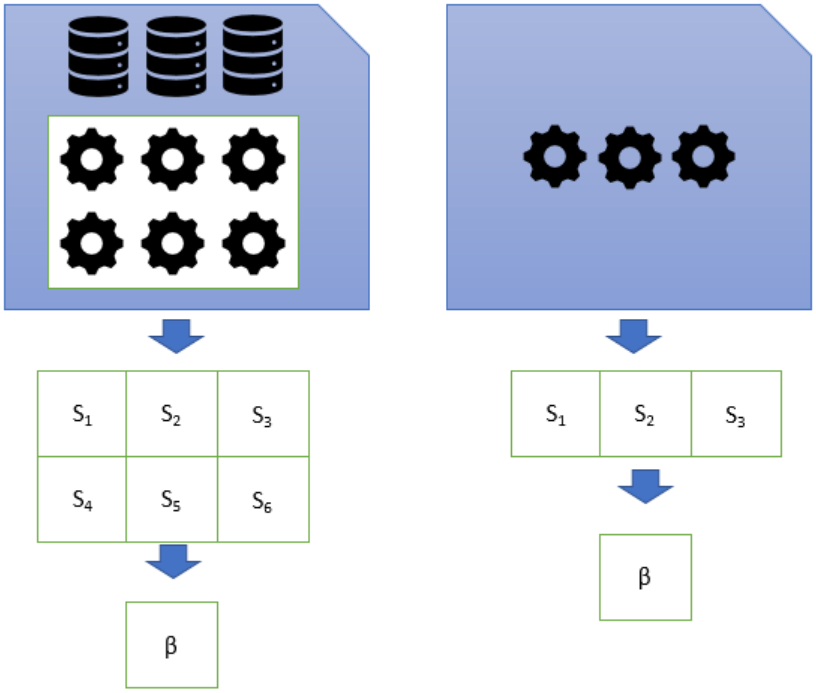

Figure 4.6. Two multiprocess schema offered by multiprocessing library

Table 4.2. Comparison of consumed time using and before multiprocessing.process

Parallel Type	Total Time	Subprocess Time
Original Row by Row	2508.0840 s	
Row by Row with Pool	512.4808 s	504.3569 s
		505.9989 s
		508.5978 s
		511.5340 s
Row by Row with Process	643.3401 s	643.9057 s
		635.6812 s
		643.1564 s
		638.5798 s

The experiment is performed on 0.8 million records for one iteration. Undoubtedly, from the Table 4.2, with the help of **Process**, the Python code is successfully launched among all the cores from Xeon manycore processor and consumed time decreases by approximately 74% as the original process is divided into four

subprocesses and work in parallel. Similarly, **Pool** gets 80% speed boost but both of them fail to solve the root cause of the slow performance of the Row by Row IRWLS.

While the multiprocessing library helps to schedule jobs to available processors for parallelism which improves the performance by the factor of number of cores, still the computation power of each core is to be leveraged.

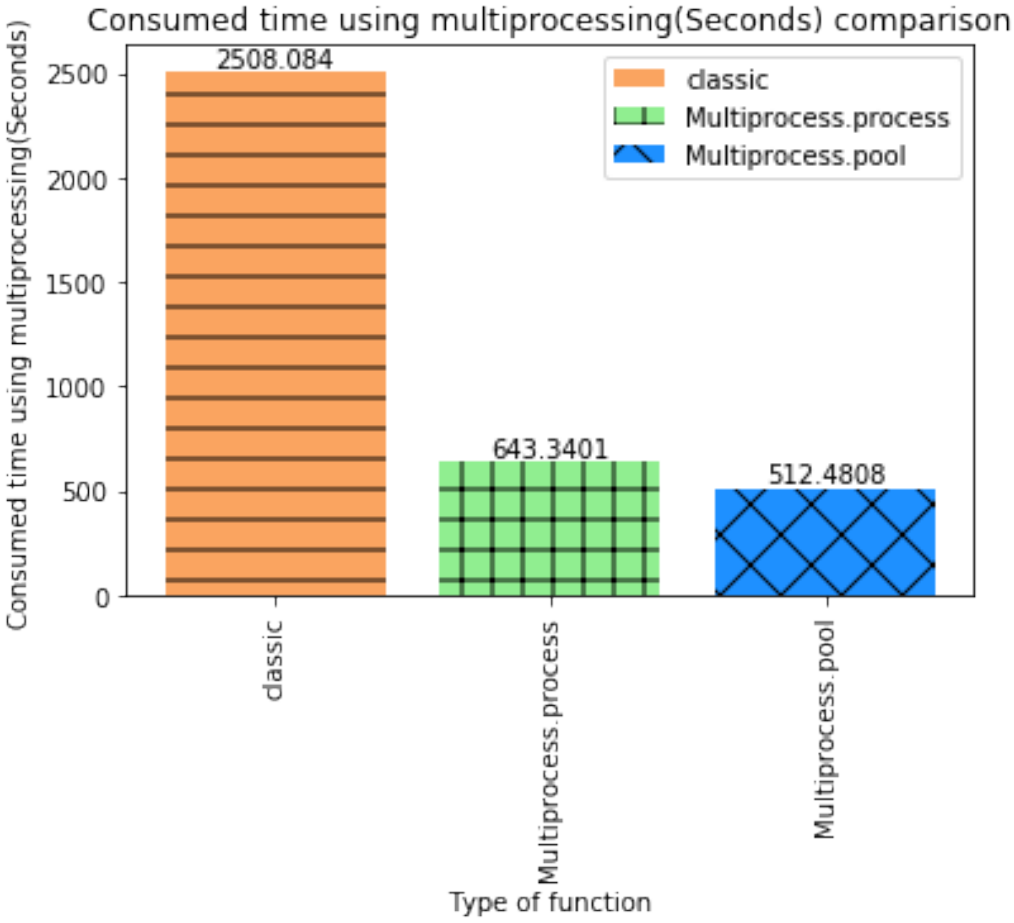

Figure 4.7. Multiprocessing Module for speed up

4.4 Data partition

To address the bottleneck of the new approach, the Python code should be profiled in order to provide the statistics describing the frequency and time consumed by each

program component execution for the approach. The two tables below list the five functions that consumes the most execution time inside Row by Row and classic IRWLS. The total consumed time is up to 4800 seconds versus 0.5 seconds, and the training set is both consisted with 0.8 million records with five selected features.

Table 4.3. Function execution statistics for Row by Row IRWLS (which consumed 2508.0840 seconds in total)

function	ncalls	tottime	percall	cumtime	percall
isinstance	421600011	110.298	0.000	136.848	0.000
pandas/core/series.py	10400000	81.148	0.000	428.942	0.000
IRWLS_Matrix	800000	67.514	0.000	2146.785	0.003
pandas/core/ops.py	8800000	65.376	0.000	1256.313	0.000
numpy.core.multiarray.dot	2400001	40.551	0.000	282.577	0.000

Table 4.4. Function execution statistics for classic IRWLS (which consumed 0.1870 seconds in total)

function	ncalls	tottime	percall	cumtime	percall
pandas/core/internals.py	4	0.074	0.018	0.074	0.019
IRWLS_Matrix	1	0.039	0.039	0.160	0.160
numpy.core.multiarray.dot	4	0.020	0.005	0.058	0.014
numexpr/necompiler.py	11	0.016	0.001	0.019	0.002
numpy.core.multiarray.concatenate	2	0.010	0.005	0.010	0.005

Obviously, calling the core function per row also leads to spending much time in initiating parameters repeatedly which is extremely time consuming in Python. High level language like Python does not require static typed parameter and costs extra time to examine the data type when invoking the parameter. In this specific case, the **isinstance** function occupies 6% of the total time which makes it enter the top 5 list of expensive function.

Besides, the **dot** function from numpy occupies the most time for tottime(which stands for the total time in the function excluding call to subfunction). The core function

IRWLS_Matrix which used for calculating $s_{zz,F}$, $s_{xz,F}$ and $S_{xx,F}$ spends the most time and all its subfunctions from invocation till exit as shown in cumtime. That's because during the calculation for extracting $s_{zz,F}$, $s_{xz,F}$ and $S_{xx,F}$ from each row, IRWLS_matrix is call and massive matrix multiplications are called inside it, which leads to the even frequent **dot** function. Unfortunately, cutting the matrix into tiny pieces avoids fully utilizing performance superiority of Numpy in handling expensive matrix and brings the expense of Python data initialization and typing.

All the above result in the 13,000X increment in execution time compared to the classic solution even if both of them are literally computing on identical data and output the result are with same quality.

In the previous section, the data is split into partitions to feed into parallel processes and run serially within the the process. However, if the size of data is successfully accepted by the process, it can definitely do matrix calculation based on the received data. Thus, the Row by Row approach is updated to Matrix by Matrix approach to make the use of vectorization and parallelism from Numpy.

To achieve so, the original data (with m records) is divided into P partitions and convert the m times calculation on $1 \times feature$ matrix into P times operation of $(\frac{m}{P}) \times feature$ matrix.

Table 4.5. Function execution statistics for 640 per partition IRWLS (which consumed 4.1649 seconds in total)

function	ncalls	tottime	percall	cumtime	percall
isinstance	658761	0.175	0.000	0.218	0.000
numpy.core.multiarray.dot	3751	0.094	0.000	0.501	0.000
IRWLS_Matrix	1250	0.138	0.000	3.565	0.003
pandas/core/series.py	16250	0.130	0.000	0.692	0.000
pandas/core/ops.py	13750	0.102	0.000	2.030	0.000

Take partitioning with 640 records for example, the execution time of the initial iteration of IRWLS decreases by 99.8% and successfully delivers result in 4.1649 seconds. From the Python profiler, the initialization of variables and **dot** from the core

function still dominate the execution time but the frequency and elapse time drop dramatically. Also, one another interesting fact is that, the cumulated time (including the subprogram) per each call of core function is almost identical with the 640 times larger matrix. The vectorization offered by Numpy boosts the performance when operated on expensive matrix manipulation and that's the reason of the efficiency of classic IRWLS approach as the operation takes place on the largest matrix of this case.

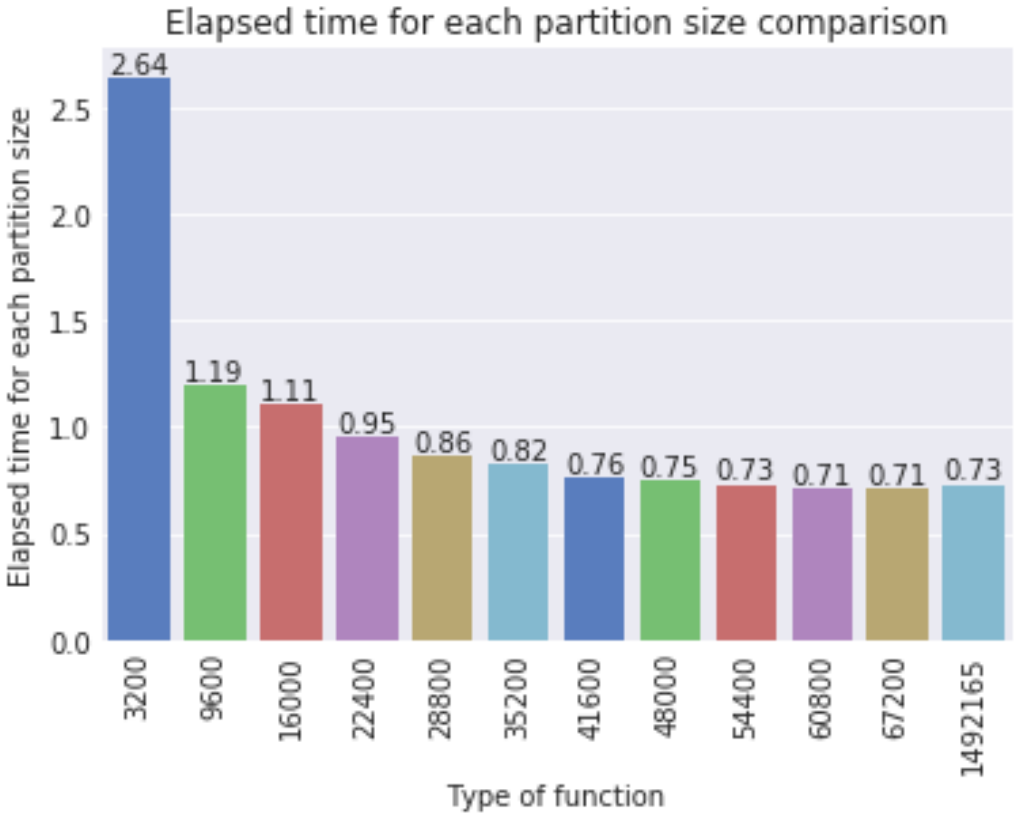

Figure 4.8. Optimal Partition size

However, it doesn't indicate one should always load the whole set for the best performance based on manycore processor, instead, the execution time converges much earlier. To illustrate, Figure 4.8 is the consumed time based different partitions of original dataset. The time drops by 60% from 3200 records to 9600 records per partition, and then deceases slowly and keep being approximately 0.18 seconds from 67200 records per partition till the whole set. Thus, to make the full use of Numpy and guarantee memory

safe at the same time, one strategy could be utilizing Matrix by Matrix IRWLS with an optimal partition size.

4.5 Cython with openMP

In the above sections, the parallelism is achieved via using multiprocessing and data partition. The Python module multiprocessing circumvents the Global Interpreter Lock in order to get parallelism among all the available cores and data partition helps the Python code utilize the parallelism and vectorization provided by Numpy. And one another popular solution of Python optimization is to embed Cython to replace the expensive and un-parallelized component in order to get the performance boost.

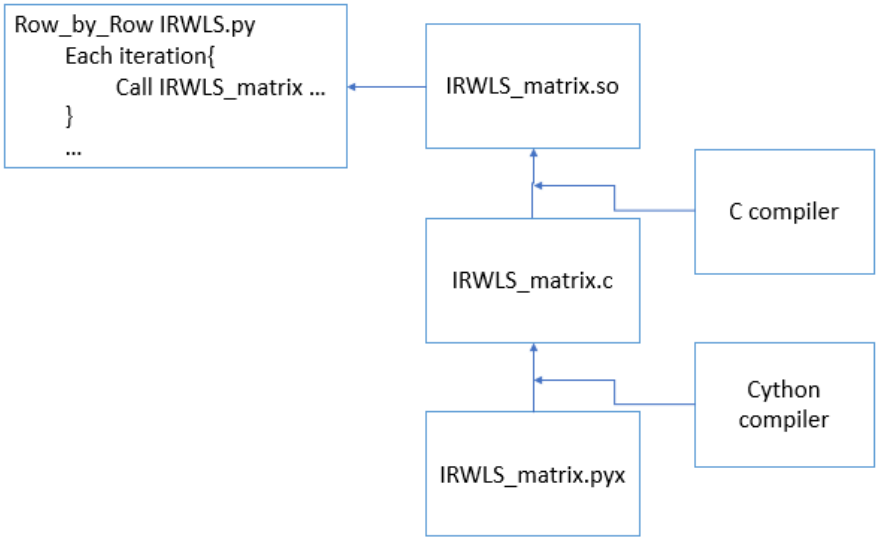

Figure 4.9. workflow of Row by Row IRWLS with Cython

As discussed above, there is GIL in Python preventing multithreading from executing code concurrently and as a result, the Python code cannot fully utilize the manycore processor.

Also, unlike lower level language (c, c++), Python is kind of interpreted language. Python code is compiled to fundamental instructions to be interpreted by Python virtual

machine and makes it flexible to execute on any platform. However, this VM design is a double-edged sword as it runs much slower than native compiled code which is translated directly to machine code. As one can see in the profiling data in last section, Python spends 6% time in assigning data type in Row by Row IRWLS as new variables emerge for the computation in each row in each iteration. Unlike C or C++, the variables in Python are all objects without any specific type. This brings the flexibility for coding but increases the burden for interpreter to figure out the type and related low level operation set. Thus, the design of dynamic dispatch makes the performance slow while a compiled program with static typed variables will skip all these works Python needs to do.

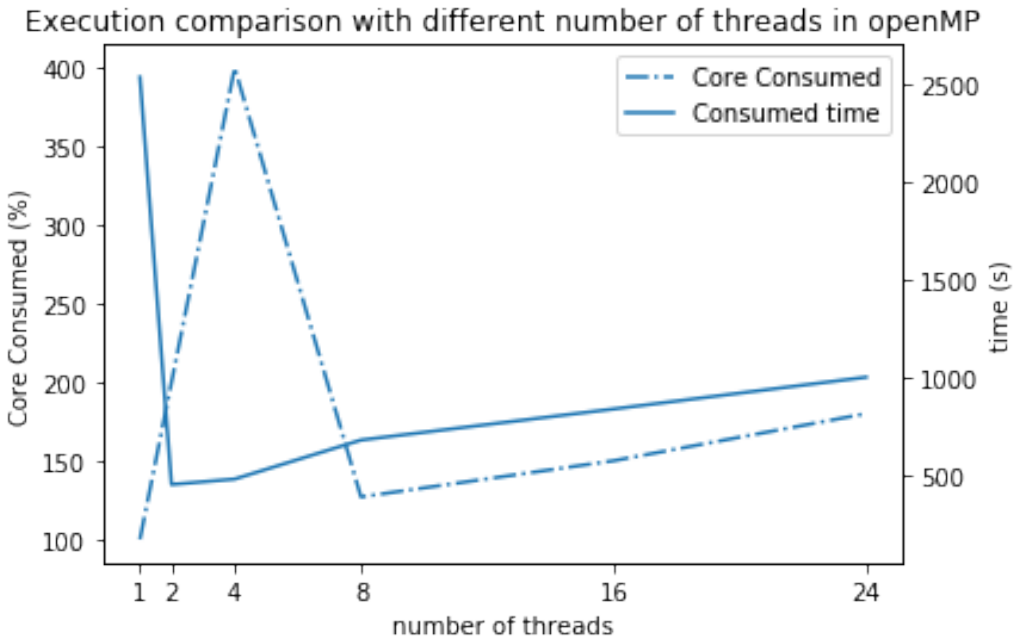

Figure 4.10. Performance comparison with different number of threads in openMP

With all the above, there for sure is a chance to improve the performance of Row by Row IRWLS by replacing the core algorithm with Cython. The core algorithm is written in .pyx file then compiled as C program. To run the Row by Row IRWLS, the Cython version of the algorithm convert the Python object into C data, then compute in compiled C and finally return the result as Python object back to main code to proceed the upcoming execution (See Figure 4.9).

After the implementation of Cython embedded Row by Row IRWLS with default setting of thread number, all the four cores of the machine are utilized and the total time drops by 81% which is similar to the result of Python module multiprocessing. Also, As shown in Figure 4.10, the performance and CPU occupation vary when with different number of threads in openMP. The figure shares the behavior of setting thread number to [1, 2, 3, 8, 16, 24]. Each thread will occupy a hardware thread When the number of thread is set to be equal or less than the number of hardware thread, and will occupy approximately 10% of one single CPU when the setting is larger. The performance boosts when implementing multithreading with numthread as 2 or 4 but becomes slower when configuring more threads which come with overhead and resource competing.

4.6 Apply Row by Row approach with streaming data

With the discussion in the previous sections, the speed of Row by Row IRWLS is increased with respective optimization approach and the data partition method beats the other two as it successfully works in parallelism and vectorization which fully leverages the computation resource.

Table 4.6. Summary of chosen variables for out of core logistic regression

Variable	Definition
DISTANCE	The total distance between two airports
MONTH	The month that flight flies
DAY OF MONTH	The exact day of month that flight flies
DEP DELAY	If the flight delays when departs
AIRLINE ID	The airline ID of the flight
ORIGIN AIRPORT ID	The airport that flight departs
DEST AIRPORT ID	The airport that flight flies to
TAXI OUT	The time between departure and wheels off
CRS DEP TIME	Scheduled departure time
CRS ARR TIME	Scheduled arrival time
DEP DEL15	If the flight departs late for more than 15 minutes

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-2-bf1df041b894> in <module>()
      9     start = time.time()
     10     datapath = "/opt/hadoop/data/"
---> 11     df = pd.read_csv(datapath+"merged4G.csv")
     12
     13     df.dropna(axis=1, how='all', inplace=True)

/opt/hadoop/anaconda2/lib/python2.7/site-packages/pandas/io/parsers.pyc in parser_f(filepath_or_buffer, sep, delimiter, header,
 names, index_col, usecols, squeeze, prefix, mangle_dupe_cols, dtype, engine, converters, true_values, false_values, skipinitia
lspace, skiprows, nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates, infer_datetime_format,
 keep_date_col, date_parser, dayfirst, iterator, chunksize, compression, thousands, decimal, lineterminator, quotechar, quotin
g, escapechar, comment, encoding, dialect, tupleize_cols, error_bad_lines, warn_bad_lines, skipfooter, skip_footer, doublequot
e, delim_whitespace, as_recarray, compact_ints, use_unsigned, low_memory, buffer_lines, memory_map, float_precision)
    653                     skip_blank_lines=skip_blank_lines)
    654
--> 655         return _read(filepath_or_buffer, kwds)
    656
    657     parser_f.__name__ = name

/opt/hadoop/anaconda2/lib/python2.7/site-packages/pandas/io/parsers.pyc in _read(filepath_or_buffer, kwds)
    409
    410     try:
--> 411         data = parser.read(nrows)
    412     finally:
    413         parser.close()
```

Figure 4.11. Memory error when loading the 4GB dataset into Python dataframe

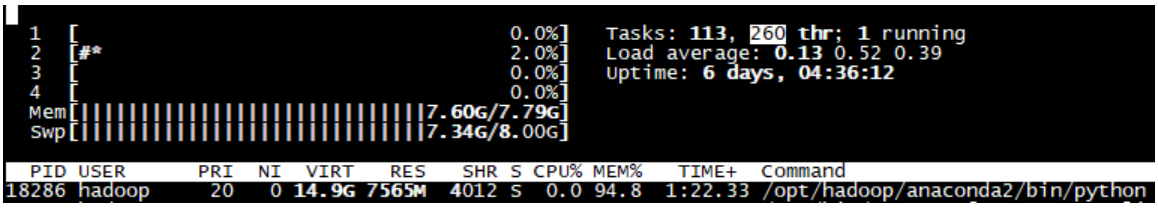

Figure 4.12. Memory saturation when loading the 4GB dataset into Python dataframe

In this section, the practice of big data training on logistic regression model will be done using the data partitioned IRWLS and scikit-learn approach. The classic IRWLS approach is eliminated in large scale training as it requires to load the whole dataset into RAM. As shown in Figure 4.11, not yet proceeding to IRWLS, the training process gets stuck when loading a 4GB csv as a Python dataframe into RAM of 8GB. From the interaction process viewer in terminal in Figure 4.12, one can see that the process of classic IRWLS approach occupies 14.9GB from both RAM and swap but dies because of memory saturation.

Like the classic IRWLS, the scikit-learn model used earlier for comparison is also not feasible when the data is much bigger than the size of RAM. To solve the out-of-core problem, SGDClassifier in scikit-learn is implemented to partially fit each chunk of dataset loaded into RAM and eventually finish the fitting until the end of the observed data. Unlike its in-memory fitting process, the out-of-core version of this approach doesn't support the configuration of iteration but limit the number of iteration in SDG to be one instead.

Again the flight data will be used as observed data in this case. The size of training data will be approximately 4GB, 8GB, 12GB, 16GB and 20GB. The records from year 2010 will be used to measure the rating of prediction. The size of dataset is overwhelming for the RAM size which is merely 8 GB, thus the machine will load and fit a chunk of original dataset and then release the memory to load the upcoming chunk. 64000 lines of records from the training dataset are loaded into RAM each time for both of the two approaches for fitting in this experiment.

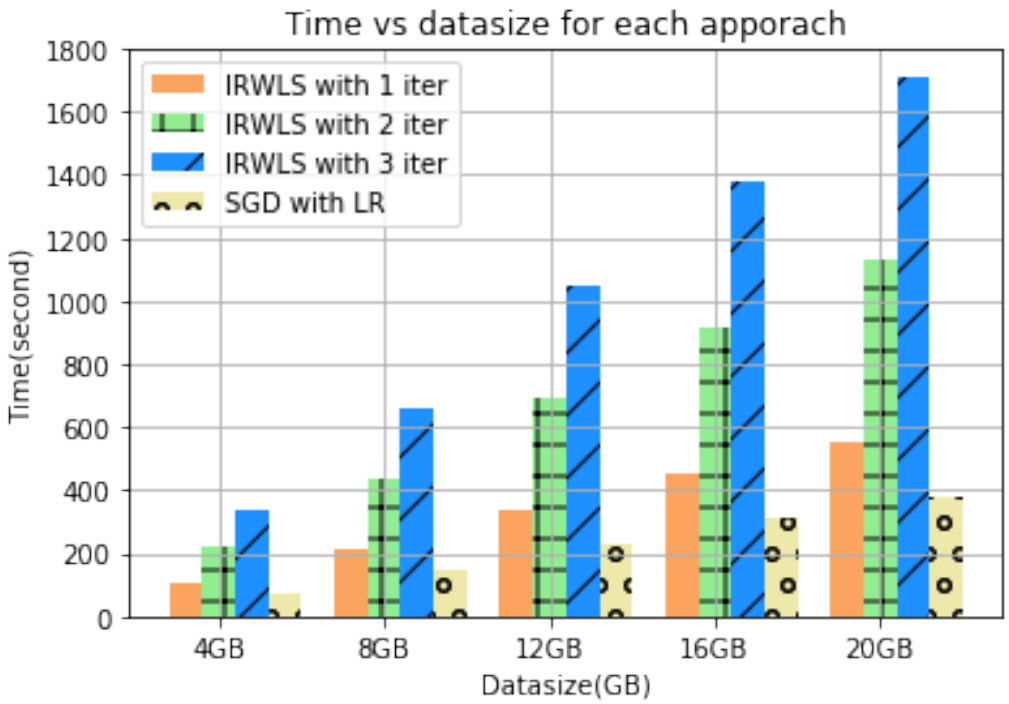

Figure 4.13. The time from different size of training data for IRWLS and SGDClassifier

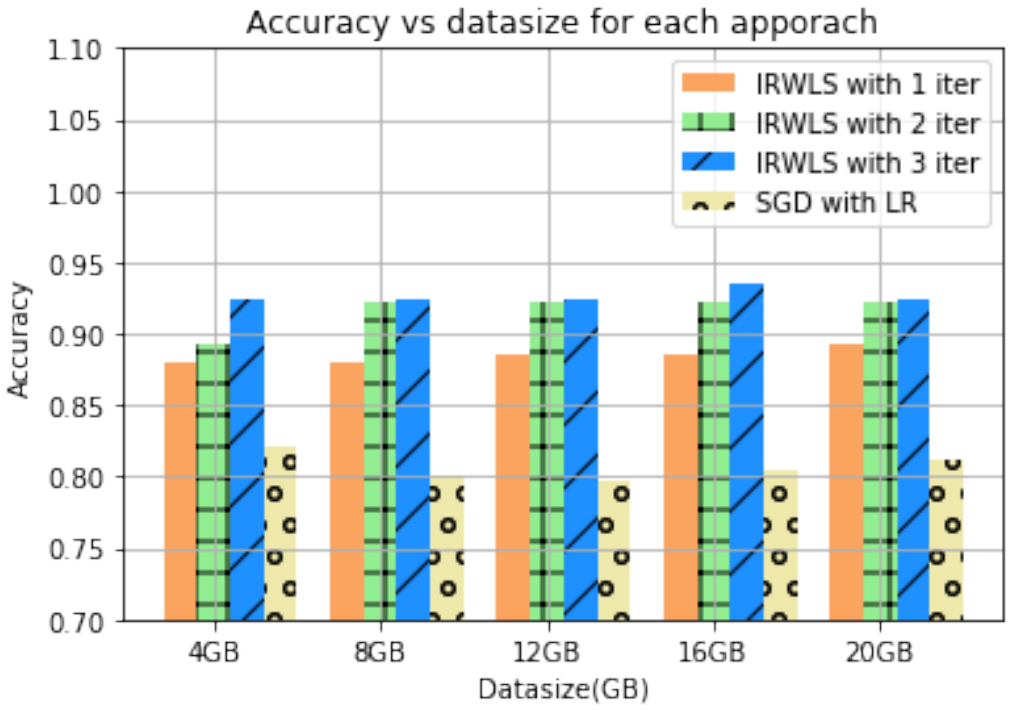

Figure 4.14. The accuracy from different size of training data for IRWLS and SGDClassifier

Table 4.7. Accuracy for each trial on 20GB data with IRWLS and SGDClassifier

Trial	IRWLS	SGDClassifier
1	0.9236	0.8190
2	0.9236	0.8190
3	0.9236	0.8227
4	0.9236	0.8232
5	0.9236	0.8213

As illustrated in Figure 4.13, SGDClassifier beats IRWLS by 30% of the consumed time on each datasize for one iteration of fitting process, and about 67% and 80% respectively for two and three iterations of IRWLS. That's mainly because the SGDClassifier solves the training process with gradient descent based on randomly selected records. However, IRWLS scores higher when talking about the accuracy of

training. Similar to the accuracy behavior in the initial assessment, the accuracy of IRWLS always reaches approximately 90% in its first iteration and slightly goes up to 92% and maintains this high accuracy after the second iteration according from Table 4.7. The accuracy from SGDClassifier is comparatively low which is about 80% for each datasize. Besides, due to the randomness of example, the accuracy of SGDClassifier may slightly vibrate even with the same dataset and same feature selection. Thus, if one is seeking a training method provides both high accuracy and high performance, IRWLS with one iteration will be very advantageous.

Also, as the new implementation of IRWLS only records the unstructured array of **S** in the RAM, the occupation of RAM keeps being around 3% as the in-memory observed data is merely 64000 records and the extra variables from IRWLS are small enough to be omitted. While for the compared approach, more than 20% of RAM is occupied during execution.

In a nutshell, the refined approach of IRWLS solves the streaming and out-of-core condition which is not achievable by the classic approach and keeps the comparatively high accuracy. Although applying iterations on large-scale data with IRWLS greatly increase the execution time as it needs to compute unstructured array **S** based on each record, the number of iteration can be minimized as the accuracy of IRWLS usually reaches acceptable rating in early iteration.

CHAPTER 5. CONCLUSION

The classic IRWLS exhibits high accuracy and performance as the solver of logistic regression, but the computation procedure of it requires computation over the whole dataset concurrently which is infeasible when being executed over large scale data. To solve this problem, Zhang and Yang (2017) discussed the Row by Row approach of IRWLS to iterate and compute over each row so that one can reduce the burden come from memory size. Based on the optimization of experiment conducted, several conclusions can be drawn as follow,

- Compared with several popular logistic regression solvers, the model fitted by IRWLS produces better rating of accuracy and is able to reach and keep high accuracy in its early iteration. But due to the nature of Python, the original Row by Row approach cannot utilize the parallelism of manycore processor like Intel Xeon family CPU.

- With the implementation of multiprocessing module to manually assign each process with partitioned job, the performance increases by the factor of involved core number. But due to the Row by Row style of data feeding, the matrix calculation is relatively inexpensive which fails to fully utilize the benefit of vectorization from the architecture of processor.

- The execution time decreased when embedding Cython and openMP into the Row by Row IRWLS to manipulate the number of thread. But similar to the multiprocessing approach discussed previously, the data feeding is the blocker of a even powerful performance boost.

- Partitioning the data as chunk and feeding into IRWLS significantly decreases the consumed time for logistic regression training process and maintains the advantage of high accuracy and memory-friendly of Row by Row approach. Also, the size of partition greatly impact the workload of matrix computation fed to processor and

impact the extent of vectorization. Thus, the chunk size selection is important for fully using the resource of each core.

- The refined version of Row by Row approach can be successfully launched when the scale of data fits and can't fit the memory size. And with optimal chunk size, it delivers the result with high performance and low memory occupation percentage.

The revised IRWLS approach solves the out-of-core issue and gets performance boosting on single processor but the utilization of distributed parallel computing is still to be studied. Also, the experiments are all performed on a 4-core machine which may not significantly displays the advantage of manycore processor. The future work will be performed with more involved CPU(s) from single processor or distributed machine to explore and optimize the fitting approach of machine learning algorithm with advanced high computing architecture.

REFERENCES

Adelstein-Lelbach, B., Johansen, H., & Williams, S. (2017). Simultaneously solving swarms of small sparse systems on simd silicon. *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 1128-1137.

Apache spark - lightning-fast cluster computing. (2017, Sep). Retrieved from `https://spark.apache.org/`

Barros, H., & Silveira, M. (2017, July). Atlas based sparse logistic regression for alzheimer's disease classification. In *2017 39th annual international conference of the ieee engineering in medicine and biology society (embc)* (p. 501-504). doi: 10.1109/EMBC.2017.8036871

Chen, C., & Zhou, J. (2017, July). Data association via logistic regression model for multiple target tracking problems. In *2017 20th international conference on information fusion (fusion)* (p. 1-6). doi: 10.23919/ICIF.2017.8009733

Detica, B. (2012, July). The big data refinery: Distilling intelligence from big data. Retrieved from `https://www.baesystemsdetica.com/uploads/resources/BigDataRefineryWhitepapersinglepages19.06.12.pdf`

Elafrou, A., Goumas, G. I., & Koziris, N. (2017). Performance analysis and optimization of sparse matrix-vector multiplication on intel xeon phi. *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 1389-1398.

El-Khamra, Y., Gaffney, N., Walling, D., Wernert, E., Xu, W., & Zhang, H. (2013, Oct). Performance evaluation of r with intel xeon phi coprocessor. In *2013 ieee international conference on big data* (p. 23-30). doi: 10.1109/BigData.2013.6691695

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., & Lin, C.-J. (2008). Liblinear: A library for large linear classification. *Journal of machine learning research*, *9*(Aug), 1871–1874.

Fichte, L. O., Knoth, S., Potthast, S., Schaarschmidt, M., Sabath, F., & Stiemer, M. (2016, July). Application of generalized linear models to evaluate nuclear emp tests. In *2016 ieee international symposium on electromagnetic compatibility (emc)* (p. 748-753). doi: 10.1109/ISEMC.2016.7571742

Heirman, W., Carlson, T. E., Van Craeynest, K., Hur, I., Jaleel, A., & Eeckhout, L. (2014a). Automatic smt threading for openmp applications on the intel xeon phi co-processor. In *Proceedings of the 4th international workshop on runtime and operating systems for supercomputers* (pp. 7:1–7:7). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2612262.2612268` doi: 10.1145/2612262.2612268

Heirman, W., Carlson, T. E., Van Craeynest, K., Hur, I., Jaleel, A., & Eeckhout, L. (2014b). Undersubscribed threading on clustered cache architectures. In *High performance computer architecture (hpca), 2014 ieee 20th international symposium on* (pp. 678–689).

Indra, S. T., Wikarsa, L., & Turang, R. (2016, Oct). Using logistic regression method to classify tweets into the selected topics. In *2016 international conference on advanced computer science and information systems (icacsis)* (p. 385-390). doi: 10.1109/ICACSIS.2016.7872727

Jia, Z., Xue, C., Chen, G., Zhan, J., Zhang, L., Lin, Y., & Hofstee, P. (2016, Sept). Auto-tuning spark big data workloads on power8: Prediction-based dynamic smt threading. In *2016 international conference on parallel architecture and compilation techniques (pact)* (p. 387-400). doi: 10.1145/2967938.2967957

Lee, K., Tak, S., & Ye, J. C. (2011, March). A data-driven spatially adaptive sparse generalized linear model for functional mri analysis. In *2011 ieee international symposium on biomedical imaging: From nano to macro* (p. 1027-1030). doi: 10.1109/ISBI.2011.5872576

Liu, W., Fowler, J. E., & Zhao, C. (2017, March). Spatial logistic regression for support-vector classification of hyperspectral imagery. *IEEE Geoscience and Remote Sensing Letters*, *14*(3), 439-443. doi: 10.1109/LGRS.2017.2648515

McCullagh, P., & Nelder, J. A. (1994). *Generalized linear models*. Chapman & Hall.

Ponte, C., Gonzlez-Domnguez, J., & Martn, M. J. (2017, July). Evaluation of openmp simd directives on xeon phi coprocessors. In *2017 international conference on high performance computing simulation (hpcs)* (p. 389-395). doi: 10.1109/HPCS.2017.65

Reinders, J. (2017, Jun). *Intel avx-512 instructions.* https://software.intel.com/en-us/blogs/2013/avx-512-instructions. Intel.

Rodriguez-Alvarez, N., & Garrison, J. L. (2016, Feb). Generalized linear observables for ocean wind retrieval from calibrated gnss-r delay - doppler maps. *IEEE Transactions on Geoscience and Remote Sensing*, *54*(2), 1142-1155. doi: 10.1109/TGRS.2015.2475317

Tan, H., Chen, H., Liu, S., & Wu, J. (2017, July). Modeling and evaluation for gather/scatter operations in vector-simd architectures. In *2017 ieee 28th international conference on application-specific systems, architectures and processors (asap)* (p. 143-148). doi: 10.1109/ASAP.2017.7995271

Tanaka, K., Nishizawa, H., Mitamura, H., Kittiwattanawong, K., Ichikawa, K., & Arai, N. (2016, Oct). Effects of environmental factors on vocalization pattern of dugongs

revealed by generalized linear model. In *2016 techno-ocean (techno-ocean)* (p. 54-57). doi: 10.1109/Techno-Ocean.2016.7890747

Wang, C., Sun, D., & Toh, K.-C. (2010). Solving log-determinant optimization problems by a newton-cg primal proximal point algorithm. *SIAM Journal on Optimization*, *20*(6), 2994–3013.

Zhang, T., & Yang, B. (2017). *Maximum likelihood in generalized linear models for big data.* (Unpublished Manuscript)