論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Bi-directional Exploration of High-performance Computing and Machine Learning: Mutual Benefits
著者(和文)	郭 箭
Author(English)	Jian Guo
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第11033号, 授与年月日:2019年3月26日, 学位の種別:課程博士, 審査員:松岡 聡,遠藤 敏夫,額田 彰,脇田 建,横田 理央
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第11033号, Conferred date:2019/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Bi-directional Exploration of High-performance Computing and Machine Learning: Mutual Benefits

Jian Guo

A Dissertation Presented to
Tokyo Institute of Technology
in Candidacy for the Degree
of Doctor of Science

Recommended for Acceptance
by the Department of
Mathematical and Computing Science
Adviser: Professor Satoshi Matsuoka

June 2018

Abstract

High-Performance Computing (HPC) puts together multiple computer technologies like computer architecture, algorithms, programming and operating system to meet increasing requirements in computing power and speed. It is used to effectively and quickly handle complex computational problems. Machine Learning is a sub-discipline of artificial intelligence (AI), that is closely related to mathematical optimization as well as computational statistics. By use of Machine Learning, computer systems are able to "learn" some patterns from objects like a human. Machine Learning models make judgments or predictions on unknown instances of the same problems. HPC and Machine Learning are two independent fields of computer science respectively, which have been developed in their own fields for decades. They have achieved great results and have made great contributions to the academia and industry.

In recent years, we can see that HPC and Machine Learning benefit each other rather than profiting unilaterally. There is a mutually beneficial relationship between the HPC and Machine Learning on some topics. In this thesis, we extend and strengthen the mutual benefit between HPC and Machine Learning to broader fields by bi-directional research exploration: speeding up Machine Learning with HPC's help, and benefiting the productivity and efficiency of HPC by use of Machine Learning.

For improving the overall efficiency of Machine Learning, we accelerate the feature extraction of bioacoustics data by employing a Just-in-Time (JIT) compiler technique, Numba, which implements automatic parallelisation and performs other optimizations, as well as a GPU-accelerated library going by the principle of modifying Python-based baseline as little as possible. With small modifications in the baseline code of feature extraction, our optimized feature extraction yields maximum 86x speedup over the baseline without losing programming efficiency. In order to reduce the entire time cost of Machine Learning modeling for handcrafted

feature-based classification, we speed up the training of SSAE with a soft-max layer for bird sounds classification by use of HPC hardware and software. Our experiment shows that using GPUs (K20, P100) results in up to 10x speed-up compared to CPU when training with different feature dimensions of bird sounds.

For helping HPC by improving the overall productivity and efficiency of HPC systems, we propose a Machine Learning based data-driven approach to predict run time-underestimated jobs in HPC systems by learning complex hidden patterns between different HPC applications and different features of computing resource usage, as well as user behavior from job logs. The experiment results show that our prediction models are able to predict run time-underestimated jobs with different accuracy at different checkpoints times in HPC systems, which would benefit HPC users by reducing time and monetary loss. It also helps HPC systems free up computing resources to a certain extent and allows users and administrators take the appropriate action (to terminate those on-going jobs that are predicted to be run time-underestimated). All in all, our Machine Learning-based prediction model would be able to improve the overall productivity and efficiency of the HPC system. In the preliminary simulation, we evaluate benefits of the prediction model with a metric named Saved-Lost Rate (SLR), most of SLRs are around 0.05-0.12, 0.337 is the best one. If we set the checkpoints for different applications to their best points, we can estimate that our prediction model would save 24962 hours totally based on the result of preliminary simulation from the existing database.

All in all, in this thesis, we perform the bi-directional research exploration between HPC and Machine Learning: speeding up Machine Learning with HPCs help; benefiting the productivity and efficiency of HPC by use of Machine Learning. Through our three use cases, we can clearly conclude that HPC hardware and software approaches are able to supply powerful computing resource for acceleration and scaling of Machine Learning. Meanwhile, Machine Learning can benefit HPC in

improving the overall productivity, efficiency and etc. This is the role of interaction between HPC and Machine Learning: mutual benefit and promotion.

Acknowledgements

Firstly, I would like to express my special appreciation and thank my advisor Prof. Satoshi Matsuoka. I would like to thank for his kind guidance, suggestion, and advice as well as his great patience during my Ph.D. studies. Secondly, I want to thank all Matsuoka Laboratory members for their keen support throughout my student life at Tokyo Institute of Technology. Then, I would like to express my greatest thanks to Akihiro Nomura-san, who provided me with a lot of help and access to the TSUBAME - the supercomputing system, which benefits my research greatly. I would like to express my greatest gratitude to my good friend - Kun Qian who provided baseline components to perform my joint research work and conducted experiments for our interdisciplinary research including the accelerating feature extraction and autoencoder training on HPC system. Without their precious support, I haven't conducted this research. I also want to thank the committee members in charge of my thesis: Prof. Satoshi Matsuoka, Prof. Toshio Endo, Prof. Ken Wakita, Prof. Akira Nukada and Prof. Rio Yokota, for their precious time and feedback that resulted in constantly improving and polishing my thesis. I would like to thank many friends and colleagues at Tokyo Tech: Hamid, Kevin, Artur, Alex, Jens, Ryan, Xu, Nagasaka, Matsumura, Haoyu, Shweta, and others that I apologize for not mentioning, for their professional support and their friendship. I would like to thank Ministry of Education, Culture, Sports, Science and Technology (MEXT) and AIST- Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory (RWBC-OIL), which offer me the scholarship and RA to support my study at Tokyo Institute of Technology. Finally, and most importantly, I really appreciate my parents for their support and raising in the recent years. Without their support, this thesis could never be completed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

1.1.1 High Performance Computing

High-Performance Computing (HPC) puts together multiple computer technologies like computer architecture, algorithms, programming and operating system to meet increasing requirements in computing power and speed. HPC is used to effectively and quickly handle complex computational problems such as numerical analysis of physics equations, industrial simulations, climate modeling, molecular simulation etc. [1]. HPC focuses on developing parallel algorithms and parallel computing systems to fully utilize the performance of computer hardware to meet the needs of various computing tasks. The terms HPC and supercomputing are sometimes used interchangeably [2].

1.1.2 Machine Learning

Machine Learning is a sub-discipline of artificial intelligence (AI), that is closely related to mathematical optimization as well as computational statistics. By use of Machine Learning, computer systems are able to "learn" some patterns from objects like a human. Based on the patterns and features that have been learned, Machine

Learning models make judgments or predictions on unknown instances of the same problems [3].

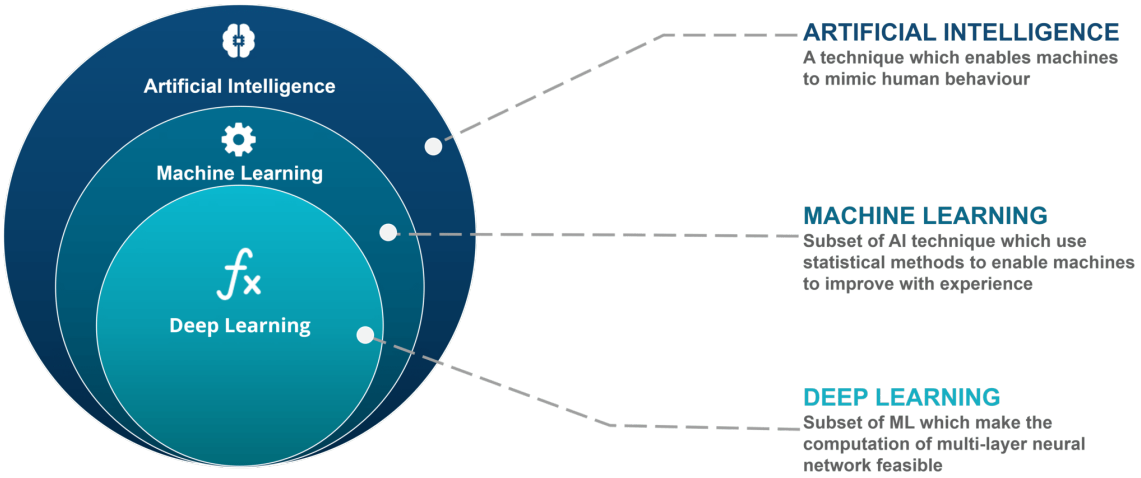


Figure 1.1: AI-vs-ML-vs-Deep-Learning

Deep Learning is a sub-discipline of Machine Learning. In contrast to Machine Learning, Deep Learning is a method of statistical learning that automatically extracts features or attributes from a raw dataset. Deep Learning does this by utilizing deep neural networks which have many hidden layers and bigger training dataset, and hence requires more powerful computational resources. With Deep Learning methods, the algorithm constructs representations of the data automatically. Figure 1.1 [4] presents the relationship between AI, Machine Learning, and Deep Learning.

1.1.3 The Relationship between HPC and Machine Learning

HPC and Machine Learning are two independent fields of computer science respectively, which have been developed in their own fields for decades. They have achieved great results and have made great contributions to the academia and industry.

In recent years, Deep Learning, as a subset of Machine Learning, has become the fastest growing technique in the world. Deep Learning architectures such as Convolutional Neural Network (CNN), Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) have been applied to different fields like Computer Vision, Speech Recognition, Natural Language Processing and so on. Most of this progress is due to the powerful ability of Deep Learning to learn subjects from larger amounts of data, which benefits from the bleeding edge computing power from HPC [5][6][7].

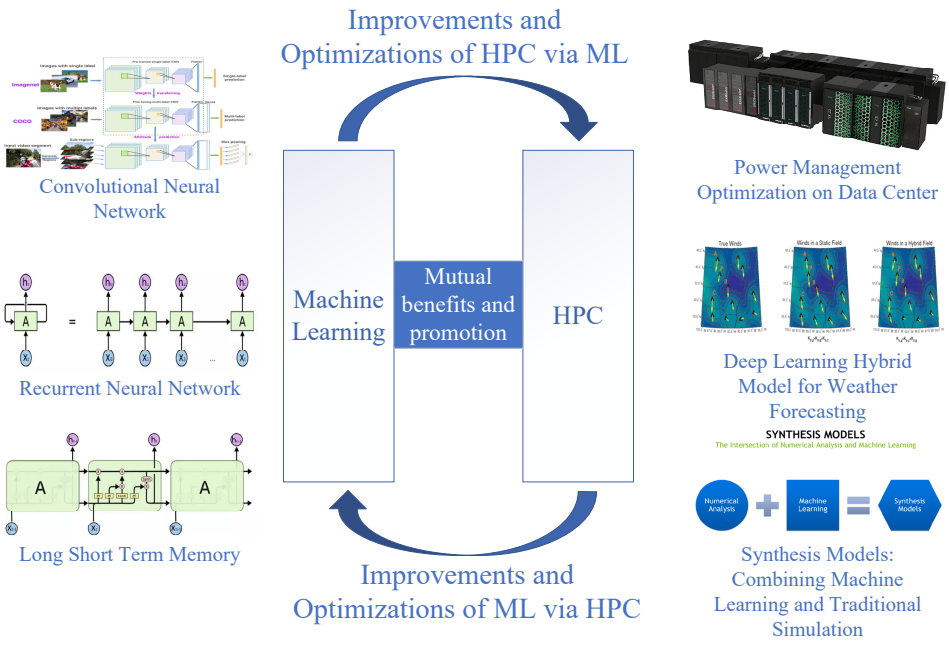


Figure 1.2: Mutual Benefits Between HPC and Machine Learning

On the other hand, there are some fields where traditional HPC researchers start experimenting with Machine Learning and Deep Learning techniques to advance scientific discovery. For instance, in meteorology modeling researchers take advantage of Deep Learning-based image recognition for weather forecasting [8] and climate modeling [9]. In addition, Nvidia collaborates with research institutions on

"synthesis models", which combines Machine Learning and traditional HPC-based simulations for improving accuracy, accelerating time to solution, and significantly reducing costs. Some preliminary achievements have been yielded: 1. Speeding up the predicting of molecular energetics by 300,000x; 2. Reducing analysis time of the gravitational lensing from weeks to 10 milliseconds; 3. Making the generation of bose einstein condensate 14 times faster than traditional models; 4. Improving accuracy of disruption prediction on sustaining fusion from 85% to 90% [10]. Moreover, Google also has developed a Machine Learning-based framework that predicts PUE within 0.4% error for a PUE of 1.1, which proved that Machine Learning can be used for improving energy efficiency for data centers [11].

Based on recent trends, we can see that HPC and Machine Learning benefit each other, as Figure 1.2 shows, rather than profiting unilaterally.

1.2 Motivation

In the previous section we showed that there is a mutually beneficial relationship between the HPC and Machine Learning on some topics. In this thesis, we would like to extend and strengthen this mutual benefit. First, for improving Machine Learning with HPC, we focus on speeding up feature extraction in conventional Machine Learning, as well as accelerating the training of Autoencoder by use of HPC software and hardware in two use cases of bioacoustics engineering respectively. In the opposite direction – improving HPC with Machine Learning - we build a Machine Learning-based model to predict job run time underestimation in HPC system for improving the overall efficiency and productivity of HPC systems.

1.2.1 Improving Machine Learning with HPC

Large processing capabilities of HPC machines allow significant speedups in accelerating Deep Learning, because of the suitability of such processing architectures for the matrix and vector computations used in training of CNN and RNN/LSTM, etc.

However, Deep Learning is not a master key to solving all problems in Machine Learning-related fields. First, the problem of data size. Deep Learning needs big training data size and when the training data is small, Deep Learning algorithms do not perform well [5]. Second, the purposes of tasks: taking Facebook as an example, the AI infrastructure in Facebook needs to handle a wide variety of daily tasks. For Machine Learning modeling, some models take just minutes to train, while others may take days or even weeks. For example, News feeding and Ads are the daily business in Facebook, which use more than 100 times computing resources than other algorithms. Hence, Facebook uses "traditional, conventional Machine Learning" algorithms whenever possible, and only adopts to Deep Learning, CNN, RNN and LSTM etc., when absolutely necessary [12]. Next is problem of interpretation. There are studies that can explain initially why Deep Learning works well in computer vision and natural language processing [5] [13] [14]. However, Deep Learning does not provide the possibility to understand which features, and why, are important. In addition, Deep Learning needs high-end infrastructure to train in reasonable time. Indeed, latest research shows that the training of a CNN on ImageNet can be completed in hours or even minutes [15]. However, such achievements require hundreds or even thousands of high-end GPUs, a fast and stable internal network within a large number of HPC cluster nodes, as well as a lot of researchers who have good knowledge of HPC and Deep Learning. At present, the institutions that can offer these costly resources are very rare. As

Figure 1.1 shows, Machine Learning is a subset of AI, and Deep Learning is a subset of Machine Learning [16] [3].

Figure 1.3 shows the trends about the application of Machine Learning (including Deep Learning) in the field of bioacoustics research on Google Scholar from 2012 to 2017. Although more and more researchers are trying to experiment with deep learning in recent years, however, until 2017, using of conventional Machine Learning is still the mainstream in research field of bioacoustics.

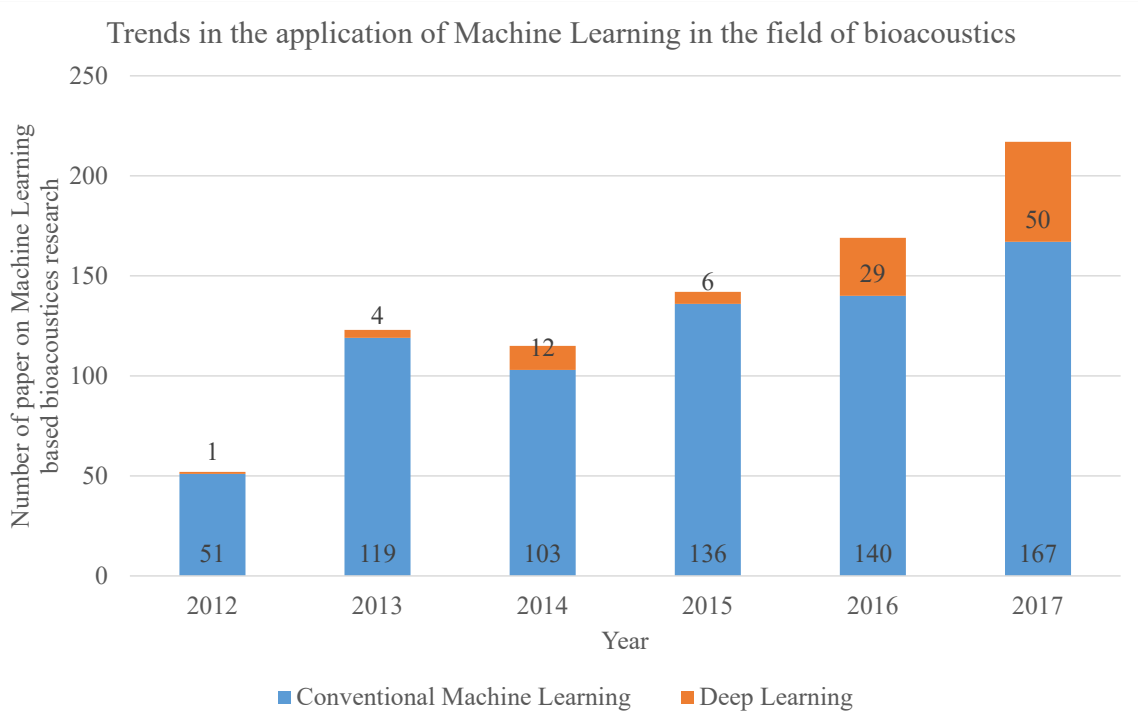


Figure 1.3: The trends about the application of Machine Learning in the field of bioacoustics

In general, Machine Learning workflows contain four parts as Figure 1.4 [17] shows;

As Figure 1.5 shows, the training dataset representations are "handcrafted" as a set of features in Machine Learning algorithms, which requiring processes such as feature selection and extraction. However, researchers have to design different features for different problems, which needs writing customized functions to extract different features from raw bioacoustics data. Those customized features includes a lots of

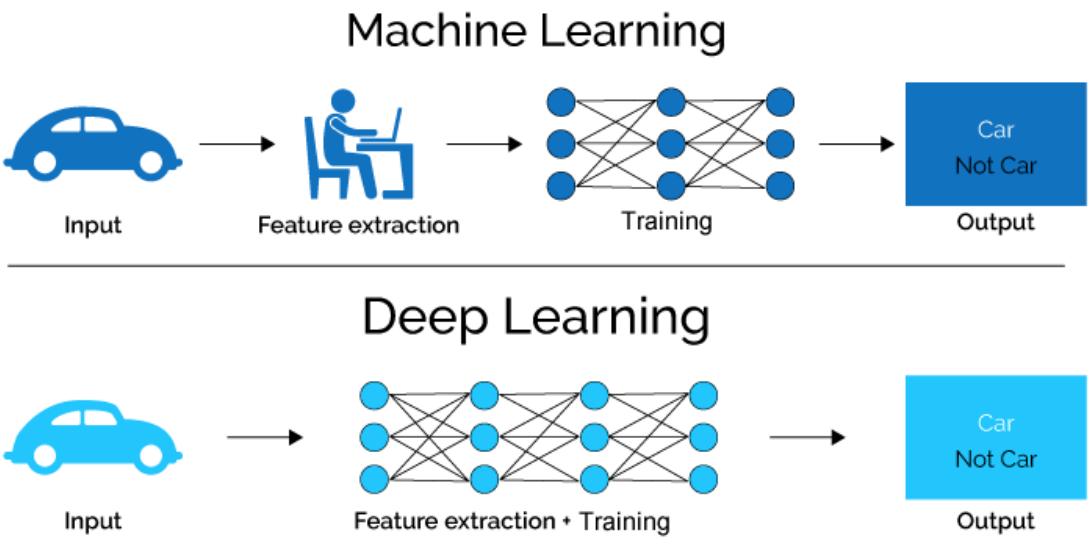


Figure 1.4: Machine Learning Workflows

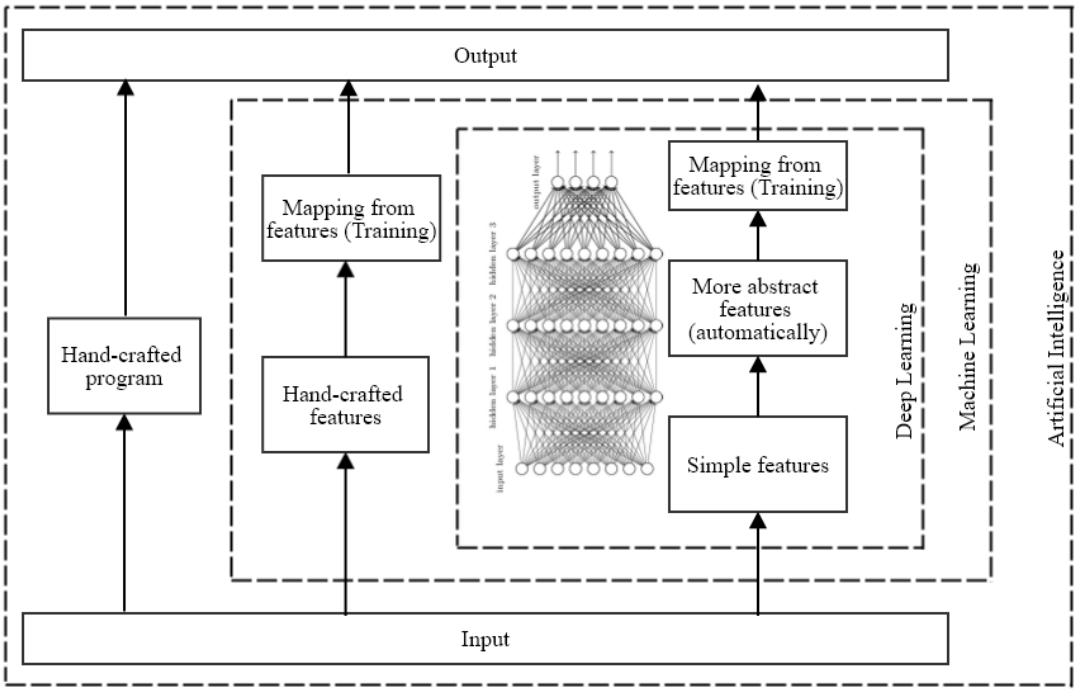


Figure 1.5: Difference between Machine Learning and Deep Learning

different mathematical computation to raw data, among them, some computations have no data dependencies to raw data, which cannot be extracted directly from existing feature library [18].

Meanwhile, to improve programming efficiency and allowing easy integration into popular Machine Learning models, most of Machine Learning projects are programmed in Python (including the baseline code of feature extraction in this research) [19, 20]. There are exiting libraries for extracting common features in Machine Learning [18], but they are not applicable to every type of applications. Hence, it is often required that we create customized functions using Python that are not available in any optimized library. This results in low performance for such applications (feature extraction). In other words, accelerating feature extraction means speeding up the entire Machine Learning modeling procedure.

For HPC-based optimization, programming or code efficiency is not only a matter of how much time the program need to occupy computing resource, but also how much time the programmers spends on writing the applications [21]. For example, for programming Nvidia GPU, CUDA C is the first choice to get the best performance. Compared to C and CUDA, Python is much more productive in programming efficiency [22]. Anaconda and Intel have been working on optimizing the performance of Python for many years, which has helped make Python with optimized computational packages become more competent for HPC applications (such as NumPy, SciPy, Scikit-learn, Numba) [23] [24]. Figure 1.7 [25] shows the summary from a study that looked at how long it took to write code for strings processing in various languages. To this end, one of our goals is to achieve good performance while keeping programming efficiency high. To do so, we will try to improve the performance of existing high-level code with minimal modifications, rather than rewriting the program using a low-level language.

As an example of accelerating training, we focus on a specific type of Machine Learning algorithm called Autoencoder. An Autoencoder "compresses" (encodes) input data into an abstract representation, and then "decompress" (decodes) it back to matches the original input data as much as possible. This is typically done for

dimensionality reduction. In addition, an extended version of Autoencoder – named Sparse Autoencoder (SAE) is presented, in which the hidden layer actually may has more or less hidden units than inputs data, but only a small set of the hidden units in the hidden layer will be activated at the same time. Both of Autoencoder and SAE are adopted for unsupervised learning, dimension reduction, and learning generative models of data. In addition, the Stacked Sparse Autoencoder (SSAE), which is consist of multiple hidden layers (more than 2 layers generally) of SAE, in which the outputs of each layer will be the inputs of the next layer. The SSAE trained with a softmax layer has been used for classification and regression problems.

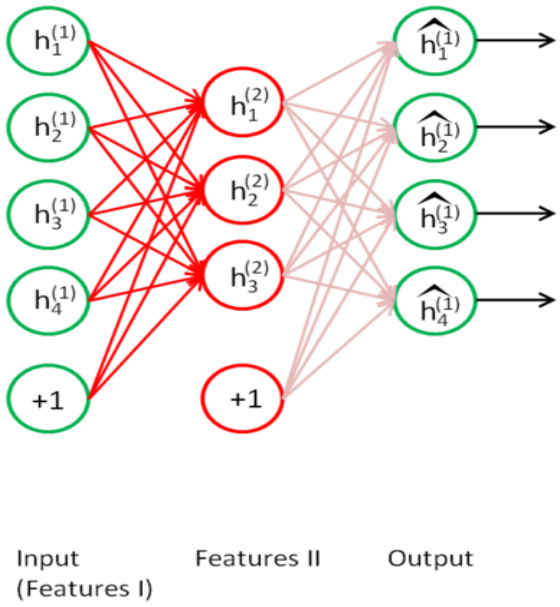


Figure 1.6: An Example of Autoencoder

For accelerating the training of Autoencoder by use of HPC software and hardware, we take the training of a SSAE as an example, which is composed of the following components: 1. Feed forward, i.e. calculating the output of the network given a set of weight A and weight B; 2. Loss functions calculation; 3. Back propagation, i.e. calculating the gradients of the objective; 4. Optimize weight A and weight B given cost and gradients; 5. Go back to step 1 until convergence [26]. This shows that the training of autoencoders is a time-consuming task.

1.2.2 Improving HPC with Machine Learning

In modern HPC systems, users are usually requested to estimate the job execution time for system scheduling when they submit a job. In the case of underestimation, the system will terminate the on-going jobs when their estimated run time expires. In this case, users will lose their processed data, and can no longer get the final results they need. Therefore, most users have to resubmit their jobs again and run them again from the beginning, which is a costly situation for users and systems since they waste time and system resources. This significantly hinders the productivity of HPC users and machines.

Predicting jobs run time underestimation can mitigate waste of time and system resources by taking early action for such jobs. For instance, if an ongoing job execution is predicted to be run time-underestimated based on its characteristics, system administrators or an automated agent can explicitly send a notification to the user who submitted the job. Then, the user can fix the problem by killing the job and resubmitting it after parameter adjustment, which will be able to benefit HPC users by saving time and money, and also improve the efficiency of the HPC system.

1.3 Problem Statement

1.3.1 How Can Bottlenecks in Machine Learning Applications be Accelerated using HPC?

As an example of accelerating feature extraction, we focus on Snore sound data analysis.

Snore sound (SnS) data is a kind of bioacoustics data which contains very important information for diagnosis and evaluation of sleep-related breathing

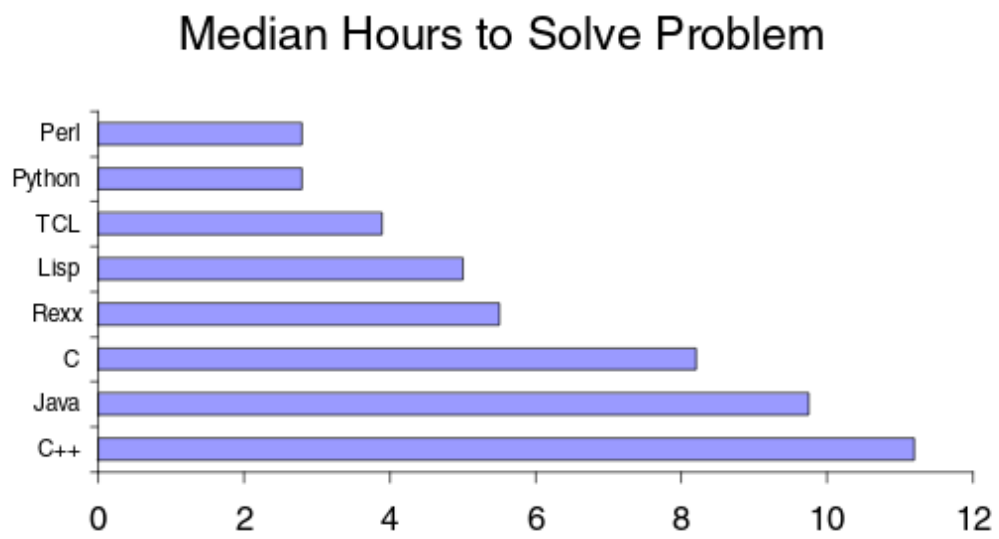


Figure 1.7: How long it takes to write a string processing application in various languages.

disorders with high prevalence, such as Primary Snoring and Obstructive Sleep Apnea (OSA) – a serious chronic sleep disease that affects around 3% 7% of male and 2% 5% of female in both developed and developing countries [27], commonly is diagnosed in middle-aged males [28].

According to existing SnS data we have and the latest report from a hospital cooperating with us, we know that each OSA subject's data will be recorded for 8 hours SnS data with 2 microphones (around 10.4 GB/Day). The feature extraction baseline for this process takes around 180 seconds to extract features from 20 minutes of data, which means that the doctor and engineer need to wait around an hour to get the extracted features from one subject for further steps such as feeding features into Machine Learning algorithms or visualization programs. With the increasing number of collected SnS data from subjects, Feature Extraction is becoming the bottleneck, preventing quick analysis of patient data. What is more, the trade-off between performance optimization and programming efficiency may also force the programmer to select different optimization strategies.

As an example of accelerating training, we focus on bird sound data classification.

Bird sounds, regarded as the "speech of birds", are significant not only for ornithologists in studying birds' species and their mating activities [29], but also for the measurement of climate changes, and biodiversity of a reserve by ecologists [30]. Machine learning researchers can contribute by analyzing bird-sound data efficiently. The autoencoders are useful in bird sound related research since a SSAE can be trained as a classifier to classify labeled bird species.

Various sound recording devices are deployed to collect recordings of bird life in bird gathering areas, especially during the season of bird migration [31]. There are a number of bird sound recordings that cannot be recognized by trained Machine Learning models, because the sound-based bird research is regional, time domain, and seasonal and climate-factored [32] [33]. In training SSAE for bird sound classification, there are some parameters like the sparsity parameter of average activation of hidden units, the number of hidden units and the learning rate would affect the final classification result of birds' species. Researchers have to spend a lot of computing time to retrain models multiple times for parameter adjustments whenever it is required to add sound recordings of unlabeled birds with exiting datasets. This continuous retraining requires a significant amount of compute time and resources.

The research question 1 is: How can feature extraction of SnS data and training of Autoencoder be accelerated for speeding up Machine Learning using HPC?

1.3.2 How Can Machine Learning Improve Efficiency and Productivity of HPC systems?

Modern HPC systems are built with an increasing number of CPU/GPU cores, memory, and storage space. At the same time, HPC applications have been rapidly growing in complexity. Moreover, not all users have enough experience working reasonably with HPC systems. Writing and executing programs on an HPC system

requires more experience and knowledge than on a PC. Those subjective and objective reasons make the prediction of job run time underestimation become a very complicated problem.

The research question 2 is: How can Machine Learning be used to predict job run time underestimation in order to reduce the waste of time and computing resources for improving overall productivity and efficiency of HPC systems?

1.4 Approaches and Contributions

In this thesis, as shown in Figure 1.8, we extend and strengthen the mutual benefit between HPC and Machine Learning by bi-directional research exploration: speeding up Machine Learning with HPC's help, and benefiting the productivity and efficiency of HPC by use of Machine Learning. Through our research and related research, we can clearly conclude that HPC hardware and software approaches are able to supply powerful computing resource for acceleration and scaling of Machine Learning. Meanwhile, Machine Learning can benefit HPC in improving the overall productivity, efficiency and etc. Through our work, we extend the mutually beneficial bond between HPC and Machine Learning to broader fields.

1.4.1 Improving Machine Learning by Accelerating Feature Extraction and Training with HPC Software and Hardware

For improving the overall efficiency of Machine Learning, we accelerate the feature extraction of bioacoustics data by employing a Just-in-Time (JIT) compiler technique, Numba, which implements automatic parallelisation and performs other optimizations, as well as a GPU-accelerated library going by the principle of modifying Python-based baseline as little as possible. With small modifications in

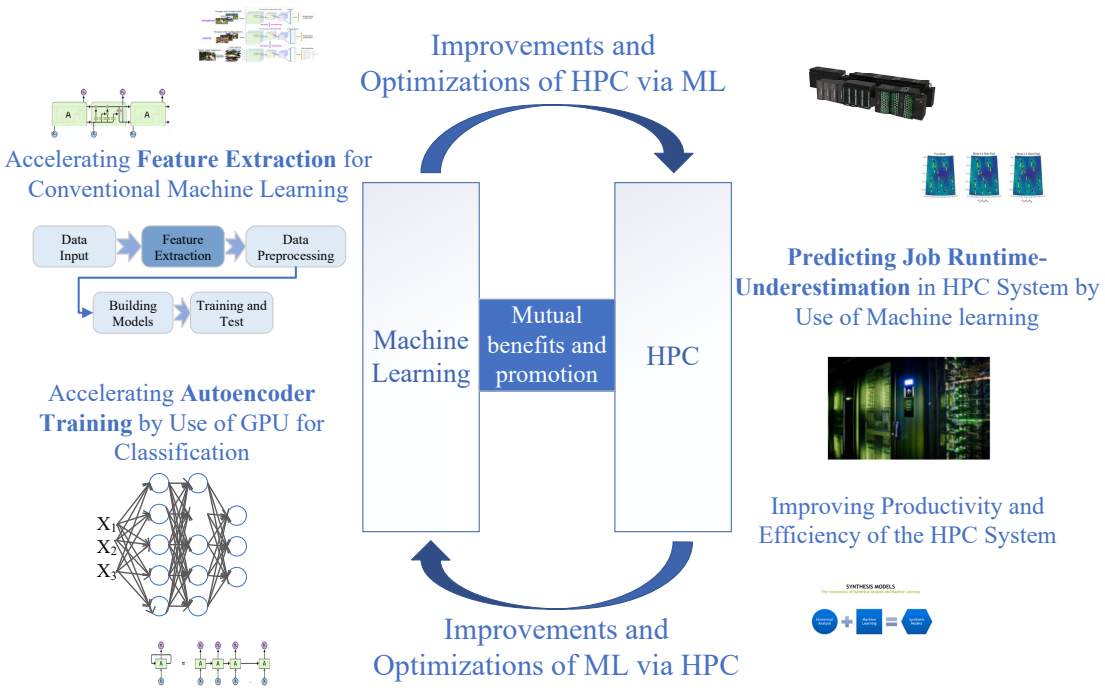


Figure 1.8: The Contribution of This Thesis: To Extend and Strengthen the Mutual Relationship between HPC and Machine Learning

the baseline code of feature extraction, our optimized feature extraction yields maximum 86x speedup over the baseline without losing programming efficiency.

In order to reduce the entire time cost of Machine Learning modeling for handcrafted feature-based classification, we speed up the training of SSAE with a soft-max layer for bird sounds classification by use of HPC hardware and software. Our experiment shows that using GPUs (K20, P100) results in up to 10x speed-up compared to CPU when training with different feature dimensions of bird sounds.

1.4.2 Improving the Overall Efficiency and Productivity of HPC System by Predicting the Job Run Time-underestimation with Machine Learning

For helping HPC by improving the overall productivity and efficiency of HPC systems, we propose a Machine Learning based data-driven approach to predict run time-underestimated jobs in HPC systems by learning complex hidden patterns between different HPC applications and different features of computing resource usage, as well as user behavior from job logs. The experiment results show that our prediction models are able to predict run time-underestimated jobs with different accuracy at different checkpoints times in HPC systems, which would benefit HPC users by reducing time and monetary loss. It also helps HPC systems free up computing resources to a certain extent and allows users and administrators take the appropriate action (to terminate those on-going jobs that are predicted to be run time-underestimated). All in all, our Machine Learning-based prediction model would be able to improve the overall productivity and efficiency of the HPC system. In the preliminary simulation, we evaluate benefits of the prediction model with a metric named Saved-Lost Rate (SLR), most of SLRs are around 0.05-0.12, 0.337 is the best one. If we set the checkpoints for different applications to their best points, we can estimate that our prediction model would save 24962 hours totally based on the result of preliminary simulation from the existing database.

1.5 Thesis Outline

This thesis is divided into five chapters, and is organized as follows:

Chapter 2: Background and Related Work

We first introduce background knowledge and some previous work on the topics, followed by short introducing techniques we used in these studies.

Chapter 3: Improving Machine Learning by Accelerating Feature Extraction and Training with HPC Software and Hardware

First, we give a brief introduction about what the feature extraction is and how to do feature extraction for bioacoustics data; then how to utilize a series of HPC-based optimizations to accelerate this process; after that, we show the configurations for our HPC-based optimizations, and present the results with detailed analysis. In the end, we conclude that by slightly modifying the original code, our feature extraction programs yields 41x-86x speedup over the Python-based baseline.

Second, we start with a short description on bird sound database and features we use in our work. Then, we introduce our SSAE for bird sound classification. Finally, we give our hardware and software configurations of experiment and show the results about speed-up of training SSAE.

Chapter 4: Improving the Overall Efficiency and Productivity of HPC System by Predicting the Job run time-underestimation with Machine Learning

Beginning with how to collect necessary data, we present the details about feature engineering, data preprocessing, and Machine Learning algorithms for building prediction models for job run time-underestimation. We also introduce the precision, recall, average precision and precision-recall curve are fairer metrics for evaluating model performance on imbalanced datasets, which have been ignored in previous works. After a series of analysis, modeling, simulations and tests, the results show our approach is able to predict run time-underestimated job with different accuracy at different checkpoints time, which would benefits HPC users

and system administrators as well as the overall productivity and efficiency of HPC systems.

Chapter 5: Conclusion and Future Work

Finally, we summarize the contributions made by our work and discuss possible directions for future research.

Chapter 2

Background and Related Work

2.1 Accelerating Feature Extraction and Training of Machine Learning for Bioacoustics Research

2.1.1 Bioacoustics and SnS Data

Bioacoustics is a cross-disciplinary science that combines bioinformatics and acoustics. Usually it refers to the investigation of sound production, dispersion and reception in animals (including humans) [34]. Identifying and labeling bioacoustics data is a difficult task [35], which requires manually and carefully comparison by specialists and leads to limited data that can be used for supervised learning.

In Machine Learning-based bioacoustics-related research, feature extraction is a critical part in the whole Machine Learning process, because it will abstract and retain important information from the raw bioacoustics data for subsequent Machine Learning algorithms. With increasing numbers and size of bioacoustics data, and constantly new features have been invented, which often require a lot of math calculations. Extracting features from those raw acoustics data become a computation intensive and time-consuming task. The quality and quantity of

available bioacoustics data have a great influence on current bioacoustics-related studies [36] [37] [38]. Thus, accelerating feature extraction for Machine Learning-based bioacoustics research by use of HPC would be able to effectively reduce the time cost of feature extraction, which will increase the overall efficiency of doctors, scientists, and engineers in Machine Learning-based research [39].

Actually, this a further and extensions from our previous works in [40] [41]. The bioacoustics data used in this work is collected from 31 patients who suffer from Obstructive Sleep Apnea (OSA), a serious chronic respiratory disease. There are some related researchers focused on bioacoustics-based features studies of SnS data which collected from OSA subjects in recent years [42] [43]. Bioacoustics features-based analysis of SnS data is a kind of non-invasion measurement. By using of this method, doctors will be able to local obstruction sites and collapse degree in the upper airway (UA) [44] [45].

2.1.2 NumPy, Numba and CuPy

NumPy

Python has been widely used, in the many fields of Scientific Computing and Machine Learning, so programming in Python allows rapid prototyping and programming like Matlab. Python offers a high-level programming abstraction, however, can only achieve low performance without additional libraries. NumPy is a powerful extension package for scientific computing with Python, which supports multi-dimensional, large arrays and matrices-base scientific computing, in pace with a number of advanced math functions to perform various calculations on these arrays. The baseline of feature extraction of bioacoustics data in this work is in NumPy based Python program.

Numba

To optimize the performance of feature extraction in Machine Learning, in principle, we modify baseline Python code as little as possible with Numba. Numba is an LLVM-based Just-in-Time (JIT) compiler which will compile Python syntax to machine code [46]. Therefore, we proceed to update the original code. Unlike most JIT compilers for interpreted languages, Numba neither trace nor replace the interpreter in Python. Instead, Numba asks users to use special decorators to replace original Python function which will be JIT compiled when it is first called with a new type signature at run time. Numba can speed up the execution time/running time of applications which are programmed in Python. After a few modifications (adding special decorators to original Python code), matrix-oriented and math-heavy original Python will be JIT compiled to native machine codes with high performance similar in performance to C and Fortran languages. Figure 2.1 [47] shows how does Numba work.

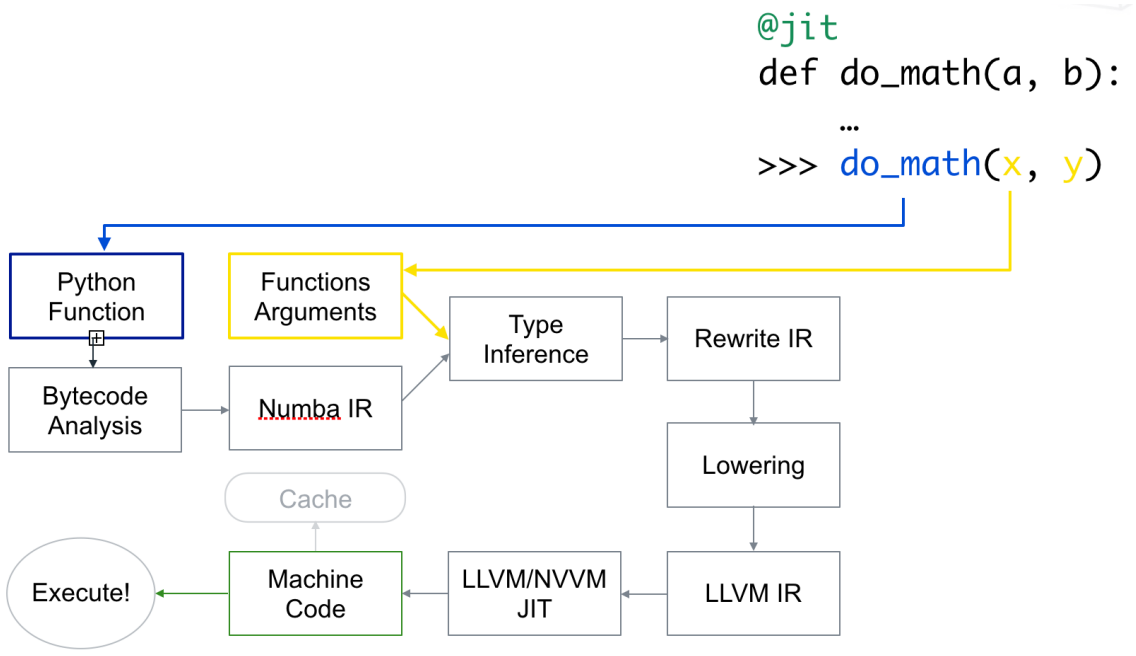


Figure 2.1: How does Numba work

CuPy

Recently, the way of improving the performance of the computing hardware is changing from increasing the frequency to increasing the number of cores gradually. Hence, multi-cores CPU architectures and GPU with high numbers of processing elements can be used for everyone. Therefore, parallel programming for using of multi-cores CPU and GPU (or multi-cores CPU and GPU clusters) with HPC becomes more and more important. Meanwhile, for GPU and HPC related programming (MPI, OpenMP), it is well known that CUDA and C based programs have the best performance comparing to other programming languages like C++, Python, etc. (in same configurations). However, efficiently parallel programming based on multi-core architectures and GPUs is generally difficult. There are many aspects to consider, including load balancing, communication and data exchange between different processes and synchronization, as well as building performance model and testing. To gain some more speed up easily, we use the aforementioned GPU accelerated NumPy-like library CuPy. Since CuPy has same API and is highly compatible with NumPy, for using of CuPy, firstly we just need to transfer NumPy arrays to CuPy arrays (memory copy HtoD), then replace NumPy functions with CuPy functions in Python code. CuPy has already supported most of NumPy functions, methods, and data type. In CuPy, all universal functions for element-wise operations, reduction and Linear algebra functions (including various products such as dot, matmul, etc.) are accelerated by CUDA-based libraries by automatically wraps and compiles the Python code to make a CUDA binary.

2.1.3 Stacked Sparse Autoencoder and Bird Sounds

The Autoencoder is a type of neural network which is used for unsupervised learning problems. It compresses the input data into a shorter code, and then decompress

that code into something that keep original information as much as possible [48][49]. Figure 2.2 shows the original architecture of a Autoencoder.

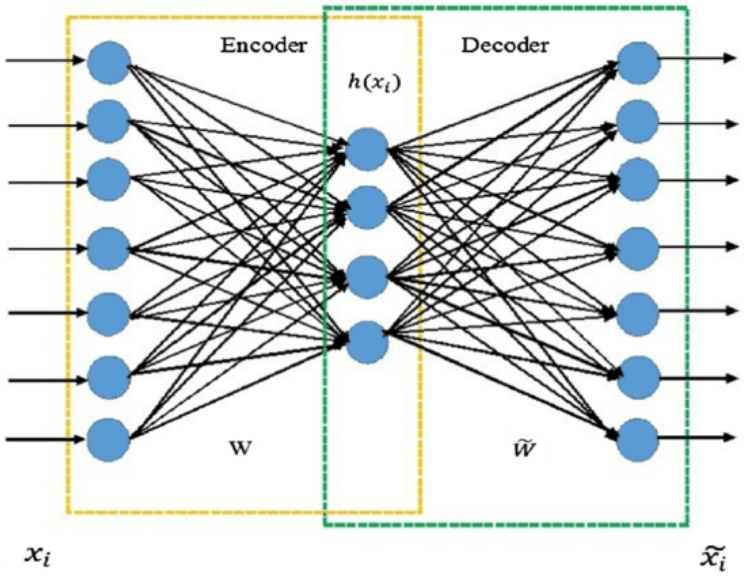


Figure 2.2: The Architecture of Autoencoder

Moreover, an improved version - Sparse Autoencoder (SAE) is proposed, in which, the hidden layer actually may has more or less hidden units than inputs data, but only a small set of the hidden units in the hidden layer will be activated at the same time [50]. Both of them are employed for dimension reduction, unsupervised learning and learning generative models of data [51].

The Stacked Sparse Autoencoder (SSAE) is consist of multiple hidden layers (more than 2 layers generally) of SAE, in which the outputs of each layer will be the inputs of the next layer. The SSAE trained with a softmax layer has been used for classification and regression problems [50]. Other than using GPUs to accelerate CNNs for computer vision and RNN/LSTM for NLP, We examine whether or not training an SSAE with a softmax layer can benefit from using GPU. We take bird sounds classification as an example and train the SSAE with a softmax layer for this task, we quantitatively compare training speed-ups of SSAE using CPU and GPU.

Bird sounds, regarded as the "speech of birds", are significant not only for ornithologists in studying birds' species, mate, activities, etc. [29], but also for the measurement of climate changes, and biodiversity of a reserve by ecologists [30]. With the increasingly collected amount of bird sound data by experts and amateurs, researchers in the highly active areas of 'deep learning' recently, which can contribute by digging such large amount of bird sound data efficiently. However, training SSAE is a time-consuming task for traditional CPU-based computing, which makes it difficult to handle the bird sound data. In this work, we train a SSAE with a softmax layer and explore the potential in performance employing the GPU in training the SSAE.

2.2 Machine Learning Predictions for Underestimation of Job Run Time

2.2.1 TSUBAME 2.5

TSUBAME2.5 [52] is GPU supercomputer located at Tokyo Institute of Technology, that operated from November 2010 to July 2017, including its ancestor TSUBAME2.0. TSUBAME2.5 is renowned as holding the title greenest supercomputer in the world in the Green500 List [53] in November 2010 and June 2011. The system consists of 1408 compute nodes, each of which has three NVIDIA Tesla K20X GPUs (upgraded from NVIDIA Tesla M2050 on September 2013), two CPUs and SSDs as local scratch storage. The nodes are interconnected with dual-rail InfiniBand QDR on a full fat-tree network. All nodes run SUSE Enterprise Linux 11 and compute jobs are managed by PBS Professional 11. Figure 2.3 shows the system architecture of TSUBAME 2.5.

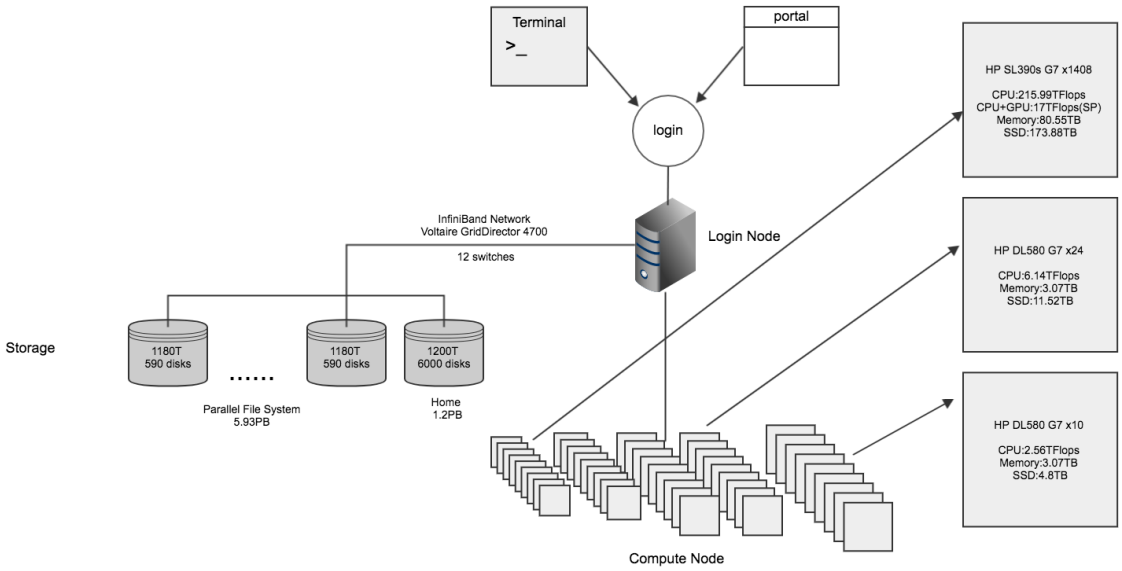


Figure 2.3: System Architecture of TSUBAME 2.5

2.2.2 Imbalanced Datasets

Whether abnormal detection or job status prediction, in general, the number of correct instances (majority class) should be much more than the number of incorrect instances (minority class) in a dataset, which leads to an imbalanced dataset just like our TSUBAME 2.5 dataset presented in Table 2.1. We take TSUBAME 2.5 as an example to explain the job scheduling system and the importance of job run time-underestimation in an HPC system.

We collect job history data from PBSs log, associated CPU and GPU usage information as features from Ganglia [54], and application information from accounting logs of each job. For data cleaning work and data reliability, we dropped all log job instances that contain NaN which may be caused by a recording error when collecting data. The size of the logs from Jan. 1, 2015 to Dec. 31, 2016 is about one million job instances, which are enough for our research experiments. Table 2.1 shows real world data we collected by Ganglia and PBS from TSUBAME 2.5, with a summary of the entire job data set and subsets categorized by scientific

application name. In Table 2.1, we found that, regardless of the whole dataset or a given subset, both are imbalanced datasets. That is, at least one of the classes accounts for only a small minority of the data. Aside from subset named "WRF", the rest are extremely imbalanced subsets. For example, subsets named "Bio: BLAST", "Bio: MEGADOCK" and "MD:Desmond MD", the minority class (labeled 1, having run time-underestimation) are less than 1%, 1.45% and 2.2% respectively on those subsets. The majority class (labeled 0, not having run time-underestimation) occupies overall 94.82% over the entire data set. In extremely imbalanced data set, accuracy is not a very useful metric for evaluating the classification performance, in which the majority class would be very easily to get high values of overall accuracy [55]. In that case, we have seen that accuracy is misleading metric to use when working with an imbalanced data set.

Table 2.1: Imbalanced Subsets Categorized by Scientific Application Name

Name of Dataset	Number of Instance	Number of Instance with label 0 (Job run time-correctly estimated)	Number of Instance with label 1 (Job run time-underestimated)
Whole data	987,123	935 926 (94.82%)	51 197 (5.18%)
Ab-Initio: PHASE	454	384 (84.59%)	70 (15.41%)
Bio: BLAST	6 367	6352 (99.76%)	15 (0.23%)
Bio: MEGADOCK	228 728	225 416 (98.54%)	3 312 (1.45%)
CAE: Abaqus	170	143 (84.11%)	27 (15.89%)
CAE: CST MW-Studio	944	861 (91.2%)	83 (8.8%)
CAE: Fluent	467	434 (92.94%)	33 (7.06%)
CAE: LS-DYNA	1 712	1 583 (92.46%)	129 (7.54%)
CAE: MSC Marc	28	28 (100%)	0
CFD: OpenFOAM	4 876	4480 (91.88%)	396 (8.12%)
MATLAB	3 127	2 832 (90.57%)	295 (9.43%)
MD: AMBER	45 554	44 382 (97.43%)	1 172 (2.57%)
MD: CHARMM	103	79 (76.7%)	24 (23.3%)
MD: Desmond MD	4 510	4411 (97.8%)	99 (2.2%)
MD: GROMACS	212 817	205 528 (96.57%)	7 289 (3.43%)
MD: NAMD	3 530	2 871 (81.33%)	659 (18.67%)
MD: Tinker	20 889	20 125 (96.34%)	764 (3.66%)
MD: lammps	1 285	1 147 (89.26%)	138 (10.74%)
MPI	124 828	112 321 (89.98%)	12 507 (10.02%)
Others	4 365	4 044 (92.65%)	321 (7.35%)
Python	208 892	199 368 (95.44%)	9 524 (4.56%)
QM: Gaussian	31 116	29 462 (94.68%)	1 654 (5.32%)
QM: OpenMX	3 786	3 459 (91.36%)	327 (8.64%)
QM: Quantum Espresso	5 276	4482 (84.95%)	794 (15.05%)
QM: VASP	69 445	58 541 (84.3%)	10 904 (15.7%)
RISM	102	102 (100%)	0
Vis: POV-RAY	1 967	1 833 (93.19%)	134 (6.81%)
WRF	1 785	1 258 (70.47%)	527 (29.53%)

So far, we have realized very clearly that predicting job-run time underestimation a binary classification problem on an extremely imbalanced dataset. Almost all classifications that will predict every sample as the majority class (run time correctly estimated jobs in this work) can still achieve very high accuracy [56]. We can see that no matter what algorithms it is based on, and no matter what the data subset is, the majority class always has very high scores on all metrics. Therefore, for building models on extremely imbalanced datasets, the overall accuracy is not an appropriate performance metric of evaluation models. Minority classes are more important than the predictions of majority classes in classification problem with an imbalanced dataset. Therefore, we propose that taking precision, recall on minority classes, average precision (AP) and precision-recall curve (PRC), rather than overall accuracy to evaluate similar works.

2.2.3 Related Works

Gathering and analyzing job logs collected from HPC systems or large-scale cluster is a widely studied topic in computer science literature. In recent years, there have been many Machine Learning technique based studies on analyzing job logs focusing on anomaly detection, failure prediction, run time prediction, and so on. For instance, Machine Learning and data mining algorithms are implemented for failure diagnosis in a large scale internet system [57] and more recently in HPC for failures prediction in [55]. Despite the popularity and huge progress in Machine Learning algorithms and softwares, the area of HPC job state prediction through accurate modelling of power or CPU utilization series seems to be unexplored. The complexity of HPC Systems and HPC applications is one of the main obstacles for building accurate models. Authors in [58] presented a Machine Learning based Random forests classification model for predicting unsuccessful job executions. In modern supercomputing centers, successful or health jobs occupy a very large part of job databases. However, authors

used the overall accuracy as an evaluation metric in those works, which cannot truly reflect unsuccessful execution results.

To the best of our knowledge, our work is the first to analyze job data and build models according to different HPC scientific applications using Machine Learning. We discover that different features of computing resource usage, with different weights on the prediction of job run time-underestimation, ended up affecting HPC applications' run time.

Chapter 3

Improving Machine Learning by Accelerating Feature Extraction and Training with HPC

The contents of this chapter were published in 2016 and 2017 in form of two publications [59, 60]. As we explained in section 1.2, conventional Machine Learning was still the mainstream in bioacoustics field until 2017.

Comparing to Deep Learning, in conventional Machine Learning, hand-crated feature extraction is necessary. In this case, researchers have to design different features for different problems, which needs writing customized functions to extract features from raw bioacoustics data. Those customized features includes a lots of different mathematical computation, which cannot be extracted directly from existing feature library [18].

Meanwhile, to improve programming efficiency and allowing easy integration into popular Machine Learning models, most of Machine Learning projects are programmed in Python (including the baseline code of feature extraction in this research) [19, 20]. There are exiting libraries for extracting common features in

Machine Learning [18], but they are not applicable to every type of applications. Hence, it is often required that we create customized functions using Python that are not available in any optimized library. This results in low performance for such applications (feature extraction).

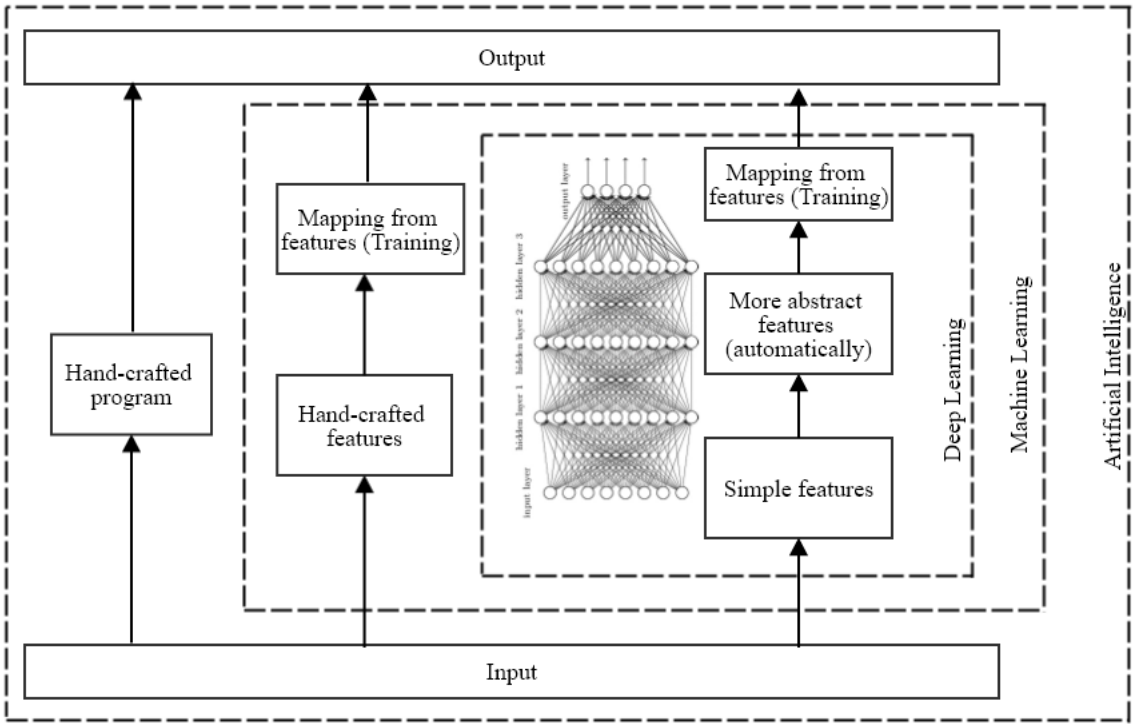


Figure 3.1: Feature Learning in Convolutional Neural Network

The conventional machine learning model is fast, and can also converge even with small samples. What is more is its interpretability. Specifically, in bioacoustics engineering, financial related fields, etc., most raw data consists of time series data or time domain bioacoustics data. In general, researchers and engineers need to extract the different features from the raw data corresponding to different tasks and purposes, then input the extracted features into the machine learning algorithms for training the model. We have introduced in 1.2 that feature extraction is the most critical and time-consuming step in conventional machine learning tasks. In other words, accelerating feature extraction means speeding up the entire machine learning modeling procedure.

In this work, with the principle of modifying Python based baseline code as little as possible, we further the work in [61], and present accelerating bioacoustics data feature extraction for snore sound (SnS) data. This is done via Python based JIT compiler techniques Numba, and a GPU accelerated Numpy-like library CuPy. At the beginning, we present a brief introduction about what is feature extraction and how to do it for bioacoustics related research. Next, we show how to utilize HPC-based optimizations like Numba techniques and CuPy library to accelerate this process; then, we show our experiment software and hardware configurations, detailed analysis of results and discussion will be given for conclusion. By slightly modifying the original code, our feature extraction programs yields 41x-86x speedup over the Python-based baseline.

3.1 Accelerating Feature Extraction of Bioacoustics Data in Machine Learning

3.1.1 Bioacoustics Features

We extract totally 20 bioacoustics features from snore sound data (SnS) which has been collected from patients who are suffering from a kind of serious chronic breathing condition named Obstructive Sleep Apnea (OSA) [62]. A few doctors, engineers and researchers have been focusing on bioacoustics features-based analysis of SnS in recent years [42] [43]. Bioacoustics features-based analysis of SnS data is a kind of measurement approaches without body invasion. By using of this method, doctors need to find out obstruction sites and collapse degrees in upper airway [63] [44] [64] [45].

These 20 bioacoustics features are Fast Fourier Transform (FFT)-based acoustics features, that can be used for helping doctors to diagnose, remedy as well as study

OSA diseases efficiently with visualization, which significantly reduce the suffering of patients during diagnosis. However, extraction those features from SnS data is time-consuming, doctors have to wait for hours for getting the extracted features from one patient's SnS data after data recording from patients in the sleep labs. In order to study how to accelerate features extraction, we extract acoustic features from all segmented frames which can reflect the spectrum distribution and the frequency. We split those 20 bioacoustics features into 4 feature groups (FG) in this work.

Spectral Features at 1000Hz

In the first feature group (FG 1), there are eight acoustic spectral features in each 1000 Hz-band (i. e., $f_{mean(1)}$, $f_{mean(2)}$, $\cdots$, $f_{mean(8)}$), that will be extracted to reveal the important information about the spectrum structure of each sub-band in SnS data. The FG 1 can be represented in formula 3.1.

$$f_{mean(k)} = \frac{\sum\limits_{f_i=1000*(k-1)}^{1000*k} f_i * S_{f_i}}{\sum\limits_{f_i=1000*(k-1)}^{1000*k} S_{f_i}} k = 1, 2, \cdots, 8 \tag{3.1}$$

Power Ratio at 800Hz

The power ratio at a frequency of 800 Hz, as the feature group 2, which can be used for distinguishing different obstruction sites in upper airway from SnS data [65]. We set Power Ratio at 800Hz with formula 3.2.

$$PR_{800} = \lg\left(\frac{\sum\limits_{f_i=0}^{800} S_{f_i}^2}{\sum\limits_{f_i=800}^{f_c} S_{f_i}^2}\right) \tag{3.2}$$

Center, Peak, and Mean Point of the Spectrum

We extract the center, peak, and mean point of the spectrum (f_{center}, f_{peak}, f_{mean}) as features, which are presented in formula 3.3, formula 3.4 and formula 3.5. In formula 3.3, the S_{fi} is used for representing the absolute amplitude of SnS spectrum at the frequency of f_i Hz. The f_c is used to represent the cut-off frequency of SnS spectrum, that has been set to 8k Hz in this work. In formula 3.5, the f_{mean} is used for reflecting the structure of a SnS spectrum. We set those 3 features (f_{center}, f_{peak}, f_{mean}) as feature group 3:

$$\text{s.t.} \sum_{f_i=0}^{f_{center}} S_{f_i} = \sum_{f_i=f_{center}}^{f_c} S_{f_i} \tag{3.3}$$

$$\text{s.t.}\, S_{f_{peak}} = \max\{S_{f_i}, f_i = 0, \cdots, f_c\} \tag{3.4}$$

$$f_{mean} = \frac{\sum_{f_i=0}^{f_c} f_i * S_{f_i}}{\sum_{f_i=0}^{f_c} S_{f_i}} \tag{3.5}$$

Sub-band Energy Ratios

Similar to $PowerRation_{800}$, we extract sub-band energy ratios (SER) as feature group 4 which can be used for reflecting the spectrum distribution of SnS [66]. In this work, we totally extract eight SERs (from $SER_{(1)}$ to $SER_{(8)}$) as full 8000 Hz SnS spectrum from every SER in the 1000 Hz-band. Therefore, these 8 features will be calculated and be extracted with formula 3.6:

$$SER_{(k)} = \frac{\sum_{f_i=1000*(k-1)}^{1000*k} S_{f_i}^2}{\sum_{f_i=0}^{f_c} S_{f_i}^2} k = 1, 2, \cdots, 8 \tag{3.6}$$

Each feature is one dimension, and 20 features (i.e. 20 dimensions) will be extracted from each frame. All extracted features, as low-order features, will be used to calculated and generated higher-order features which can be recognized Machine Learning algorithms or be reviewed by doctors for helping diagnosis. In this work, we just focus on accelerating the feature extraction of low-order features.

3.1.2 Methods for Accelerating Features Extraction of Bioacoustics Data

For HPC-based optimization, programming or code efficiency is not only a matter of how much time the program need to occupy computing resource, but also how much time the programmers spends on writing the applications. For example, for programming Nvidia GPU, CUDA C is the first choice to get the best performance. Compared to C language and CUDA, Python is much more productive in programming efficiency, but it is a low performance without additional libraries. Python is widely used, especially in the field of scientific software, so coding in Python allows simple programming of prototypes similar to the interactive platform like MATLAB. Using of appropriate high-performance-optimized libraries can boost acceleration of performance bottleneck parts of Python code significantly without losing programming efficiency.

In this work, our objective is to achieve accelerating bioacoustics data feature extraction on HPC and GPU accelerators in principle of modifying Python code as little as possible. First, for the case of only using CPU, we explain the basic procedure of bioacoustics features extraction with a NumPy based baseline program. Then we optimize our baseline with Numba, which is a JIT compiler technical for Python programs [46]. After that, if GPU is available, we further optimize this work with a GPU accelerated NumPy-like library, CuPy, an open-source Python-based scientific computing library accelerated with NVIDIA CUDA, and it can automatically use

CUDA-based HPC libraries like cuBLAS, cuRand, cuSolver, cuSPARSE, and cuFFT to make full use of the CUDA-enabled GPU architecture for accelerating Python code [67].

Basic Procedure of Bioacoustics Features Extraction with NumPy (baseline)

In most bioacoustics feature projects, the basic procedure of bioacoustics feature extraction can be represented as figure 3.2. In this work, all SnS bioacoustics data (all around 20 30 minutes long) are saved as WAV format with a 44.1K sampling rate, which means that each piece of data has around 60 92 million samples. In bioacoustics feature extraction, the features are extracted in frames which are composed of samples. Therefore, one must do framing on all the samples. The frame length dependents on the applications and feature algorithms. In this work, we use 512 samples as one frame (frame length is 512). Next, in order to recover some of the lost bioacoustics information within the framing window, overlapping is used to keep longer time domain information (for example, some random transient events may be closer to the center of the closest frame). We use 50% overlap, which means that the second half of the N frames (256 samples in this work) will be overlapped with the first half of N+1 frames (256 samples in this work) as figure 3.3 presented. Then, overlapped frames are converted into individual spectral components and thereby provides frequency information about the bioacoustics by Fast Fourier Transform (FFT). Finally, feature algorithms extract features from those spectral components following FFT, and as a result output the extracted features.

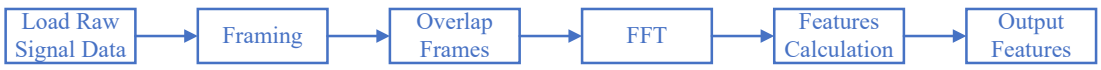


Figure 3.2: Procedure of bioacoustics Features Extraction (baseline)

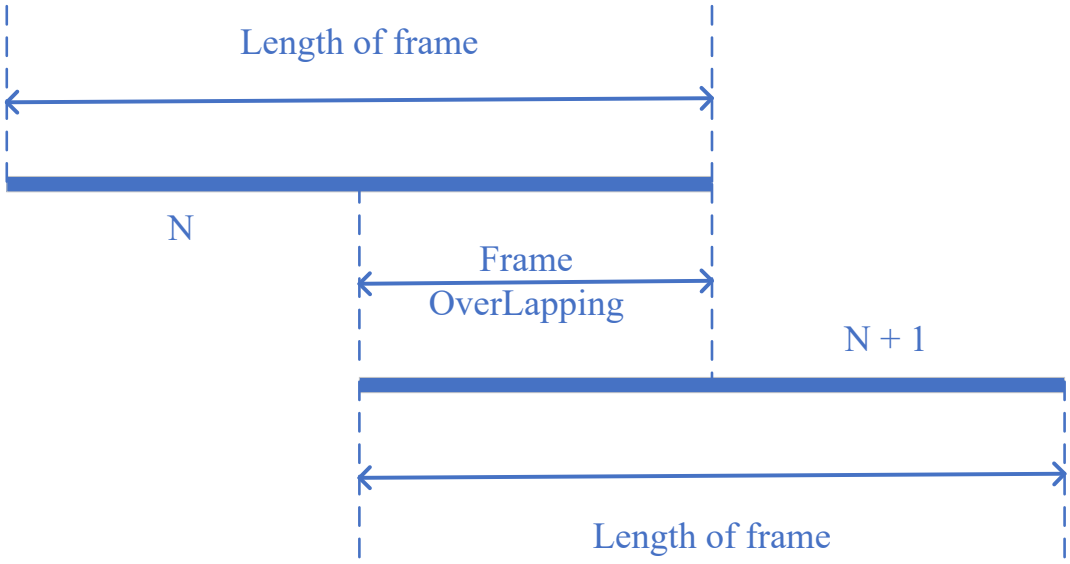


Figure 3.3: Overlapping Frames

The baseline is a typical feature extract program following the procedure presented in figure 3.2. It is a serial computing program and written in Python with NumPy. NumPy is a Python library, which has become the core library for scientific computing in Python as a matter of fact. In recent years, most Python related scientific computing programs are based on NumPy, mainly because it supports multi-dimensional, large arrays and matrices-base scientific computing with a number of advanced math functions to perform various calculations on these arrays.

The Workflow of Optimization

We need to check the baseline code and profile its performance for founding out bottlenecks in the program. Then, according to different bottlenecks, giving different optimization methods and verify the effect step by step.

Basic Optimization

We use "cProfile" profile the baseline code at first, then, we read the baseline code. As we mentioned, it is a serial computing program and written in Python with NumPy.

The workflow of the baseline is as below: 1. A data file is opened from the hard disk storage, 2. Load one frame of data, and close the data file 3. A series of feature extraction are performed for this frame and output extracted features. This flow is repeated until the last frame in raw data has been processed. Usually, one raw data is around 24 minutes of WAV or binary files which contains hundreds of thousands of frames. Frequently opening, reading and closing a file from a hard disk is a very inefficient. The first step of optimizing is avoid frequent reading and writing of hard disk, so we load the all samples into memory at once. Next, we perform framing and overlapping with the same configuration from the original baseline containing the for-loop, instead of performing framing and overlapping one frame at a time from hard disk. We call those as "baseline on memory" method.

In NumPy, an Universal functions (ufunc) provides a "vectorized wrapper for performing mathematical operations between arrays with different shapes. Among them, most of built-in universal functions are implemented in compiled C code, which leads to efficient algorithm implementations [68]. We adjust the baseline code to ufunc arrays and make most of mathematical operations occur between NumPy arrays (we call it as "ufunc" in this thesis).

Accelerating with Numba

In order to optimize the performance of feature extraction, in principle, we modify original Python code as little as possible with Numba. Numba is a JIT compiler for Python, the most common way to use Numba is through its "njit" decorator that pass that attempts to automatically parallelize and perform other optimizations on (part of) a function. Numba reads the Python bytecode for a decorated function and combines this with information about the types of the input arguments to the function. It analyzes and optimizes user code, and finally uses the LLVM compiler library to generate a machine code version of function, tailored to your CPU

capabilities [46]. Setting the "parallel" option to "True" for "njit" decorator enables Numba attempts to run automatically parallelize code on all CPU cores.

For Numba-based method, first, we use the same steps in basic optimization. Then we use NumPy's FFT function to convert all overlapped frames into individual spectral components contains the for-loop. Next, we annotate a Python decorator (@njit) on feature extraction functions. The decorator substitutes the function object with a special object that triggers the compilation when called. The following code illustrates the usage of the @njit decorator on feature extraction function of bandration:

```
@numba.njit()
def bandration_new1(wave_data):
    nf = len(wave_data)
    result = []
    for i in prange(nf//nl):
        r = np.power(np.abs(wave_data[(i*nl):((i+1)*nl)]), 2)
        ap = np.sum(r[:nl//2])
        for i in prange(1,9):
            bp1 = (i-1) * 1000
            l1 = round(1 + (bp1/fs*nl))
            bp2 = i * 1000
            l2 = round(bp2/fs*nl)
            result.append((np.sum(r[:l2]) - np.sum(r[:l1])) / ap)
    return result
```

Accelerating with CuPy

To gain some more speed up, we use the aforementioned GPU accelerated NumPy-like library CuPy. Since CuPy has same API and is highly compatible with NumPy, for using of CuPy, we just need to transfer NumPy arrays to CuPy arrays (memory copy HtoD), then replace NumPy functions with CuPy functions in Python code.

First, we still need to load all the samples into memory at once, using the same method as we performed for the Numba based optimization. Though framing and overlapping on CPU showed to be a little time consuming, utilizing local memory is a faster choice than relying on the I/O between CPU and GPU. Hence we decided to keep framing and overlapping on CPU. Then, we reshape all overlapped frames into frames matrices, where the rows became frames. We copy the frame matrices from local memory to device memory and replace all NumPy's arrays and operations with CuPy. In CuPy, all universal functions for element-wise operations, reduction and Linear algebra functions (including various products such as dot, matmul, etc.) are accelerated by cuBLAS. CuPy will automatically wraps and compiles the Python code for building CUDA binary execution. Compiled CUDA binary execution files can be cached and be reused in subsequent runs.

3.1.3 Experiment Evaluation and Result Analysis

In this section, we introduce the basic information of data sets for experimentation. After introducing our hardware and software environment, we present the results with single SnS data to investigate the speed up on different features groups. We also show the results with different sizes of SnS data to examine the overall performance improvement of our optimization for feature extraction.

Experiment Setup

All the SnS data is offered by the department of Otolaryngology at Beijing Hospital in authorized using [69] . Overall, the aforementioned 20 acoustic features are extracted from 31 SnS WAV files. All SnS data is recorded as WAV files with 44K sampling rate; in other words, 1 second contains 44100 recorded samples. Details are presented in Table 3.1. We should be aware that the number of samples in one frame (frame length) is an issue for professionals in the field of bioacoustics data analysis. In

general, 256/512/1024 samples per frame are the most common options for most bioacoustics projects. In this work, the baseline is set at 512 samples as a frame length for framing. We use SnS data No.1 as an example to explain how we perform framing and overlapping, as well as test the acceleration of our methods. In data No.1, there are 62,973,952 samples in the raw WAV file. Thus we get 122,996 frames with 512 as frame length after framing.

Table 3.1: Details about all SnS Data

ID	Samples	Size(MB)	ID	Samples	Size(MB)	ID	Samples	Size(MB)
1	62973952	245,922	2	67517440	263,740	3	63571968	248,328
4	70332416	274,736	5	87945216	343,536	6	59038720	230,620
7	82373632	321,772	8	64582656	252,276	9	48999424	191,404
10	31785984	124,164	11	60652544	236,924	12	43146240	168,540
13	65515520	255,920	14	62135296	242,716	15	41151488	160,748
16	77259776	301,796	17	63079424	246,404	18	43756544	170,924
19	74427392	290,732	20	65698816	256,636	21	71034880	277,480
22	61619200	240,700	23	45182976	176,496	24	31515648	123,108
25	62876672	245,612	26	71384064	278,844	27	64488448	251,908
28	70273024	274,504	29	50573312	197,552	30	76018688	296,948
31	33886208	132,368						

The longest SnS data is No.5 (87, 945, 216 samples, contained in 33 minutes) and the shortest one is No.24 (31, 515, 648 samples, contained in 12 minutes). Rest of them are around 24 minutes, and in total the 31 SnS data fragments amount to around 8.1 GB.

The hardware and software configuration of our experimental environments are given in detail in Table 3.2 and 3.3.

Experiment Results

As we defined in the previous section, we divide these 20 bioacoustics features into 4 feature groups (FGs). We profile each part of run time of these FGs like file I/O times, preprocessing and feature extraction separately using build-in time functions.

	Environment for CPU-based Optimizations (NumPy, Numba)
CPU	Intel Processor i7-3820 3.60 GHz (4 Cores, 8 Threads)
Memory	16 GiB DDR3 1600 (4x4GiB)
OS	Windows 10 v17763.134
Software	Anaconda Python 3.6 with NumPy 1.15.1 (Intel MKL)
	Numba 0.40

Table 3.2: Environment for CPU-based Optimizations (NumPy, Numba)

	Environment for GPU-based Optimizations (CuPy)
CPU	Intel Xeon CPU E5-2630 v4
GPU	NVIDIA Tesla P100 with 16 GiB HBM2 PCI-e
Memory	256 GiB
OS	CentOS 7.5.1804 with CUDA 10.0
Software	Anaconda Python 3.6 with NumPy 1.15.1 (Intel MKL)
	Numba 0.40, CuPy 5.0.0

Table 3.3: Environment for GPU-based Optimizations (CuPy)

The baseline load and extract features one frame by one frames from the original SnS data. As we know from Figure 3.2, feature extraction requires frame data processed by overlapping and FFT. However, the baseline program executes FFT repeatedly whenever different FGs are extracted. Feature extraction frame by frame is consistent with the logic of bioacoustics data processing, however, it is a very inefficient method for parallel computing.

We valuate the performance of our optimizations from comparing originally baseline with basic optimization. First we run the originally baseline code without any change, Table 3.4 shows that,

Then, since we have already know that the baseline is a serial computing program. It loads and extracts features from hard disk data file one frame by one frame, which is a very inefficient way. Loading data into memory is the necessary first step for further optimizations. We execute each method 3 times and keep the fastest result for comparison. Table 3.4 shows that, we just load the data into memory and keep the other unchanged, which save 23.40 seconds on FP64 (double precision) and 33.94 seconds on FP32 (single precision).

Table 3.4: Execution time of original baseline and baseline on memory with one data

	FG1	FG2	FG3	FG4	FFT	Preprocess	Load(I/O)	Total (seconds)
FP32 (Baseline)	31.946	9.602	266.461	208.525	1.867	0.777	2.912	522.09
FP64 (Baseline)	36.183	14.013	78.125	51.212	3.97	2.714	3.2149	189.431
FP32 (Baseline on memory)	22.201	7.812	254.48	198.992	2.781	1.571	0.312	488.149
FP64 (Baseline on memory)	26.118	10.871	73.406	49.895	3.25	1.917	0.574	166.031

According to Table 3.4, we can see that the execution time of feature extracting of FG 3 and FG 4 are the most time-consuming parts in the entire process, and more special is, feature extraction on those 2 groups in FP32 is much slower than FP64. After reading the baseline code, we find that there are a lot of point-wise arithmetic calculations which contain both NumPy and built-in numbers for cumulative sum operation. In this case, NumPy will convert numbers to Python built-in data type, according to official document of Python and NumPy, native Python 3 only has "int, float, bool and complex" 4 built-in data types. On the other hand, in NumPy, NumPy.float = NumPy.float64 = built-in float (NumPy.float32 is implemented by NumPy not native Python). For some reason, NumPy.float32 is hard-coded to take the slower path, while NumPy.float64 gets a chance to take the faster path [70]. The overhead of data type conversion on a large data set and the data type conversion mechanism in NumPy lead to the low-efficient point-wise arithmetic calculations (between NumPy and built-in numbers), and also make the FP32 is slower than the FP64 in this case.

Due to the above reasons, looping point-wise arithmetic calculations contain both NumPy and built-in numbers is not a efficient way. Figure 3.4 shows that comparing "ufunc optimization with the baseline on memory, the time cost on FG3 and FG4 has been reduced significantly reduced. The total execution time of FP32(baseline on memory) has been reduced from 488 minutes to 18.362 seconds

since we rewrite the point-wise arithmetic calculations (between NumPy and native Python built-in numbers) to arrays with built-in universal function, which avoids the huge overhead of data type conversion between NumPy and native Python. For FP64-based comparation, "ufunc also shorts the execution time of feature extract from 166 seconds to 34.491 (speed up 4.8x). Compared to FG3 and FG4, the execution time reduction of FG1 and FG2 is not very obvious, because the algorithm complexity of FG1 and FG2 is lower than FG3 and FG4, and most of the calculations in that FG1 and FG2 have already based on arrays.

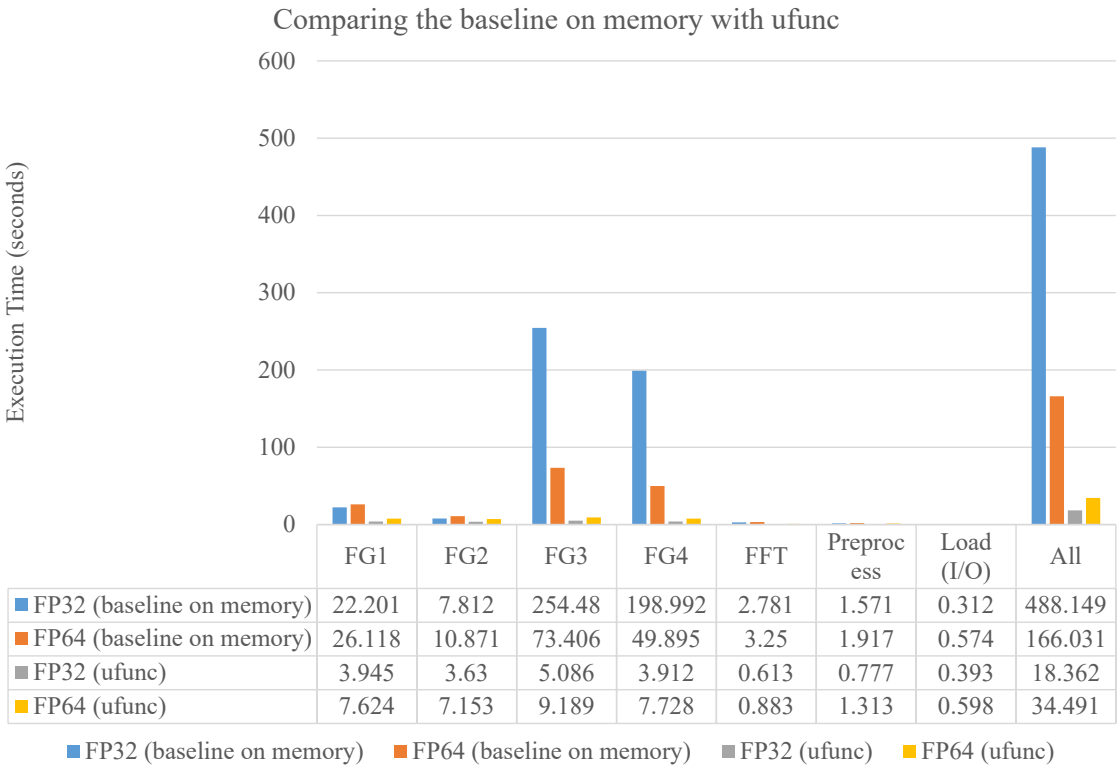


Figure 3.4: Comparing the baseline on memory with ufunc

In Numba-based optimization, we start from "njit"-based optimization. In this mode, Numba will attempt to automatically convert Python code to machine code. Then, we set "parallel=True" ("njit+parallel") to enforce Numba to take advantage of all the cores on one machine. Since Numba is using LLVM to compiler Python code to machine code [46], compilation will takes a few seconds. For fair comparison, we

extracted feature extractions for the same file twice, and record the second execution time which EXCLUDED compile time. We use the same counting method to CuPy-based optimizations, because CuPy also needs time to initialize and compile CUDA.

Numba determines the specialized types of all values within a function being compiled. Until now, although Numba has already supported lots of array operations that have parallel semantics for parallelize, type inference can fail if arguments or globals have Python types unknown to Numba, or if functions are used that are not recognized by Numba. So we would better keep a clear loop structure in the code for Numba compiler and do not use functions or function parameters that Numba does not yet support.

As shown in Table 3.5 and 3.4, we can see that Numba achieves much more performance improvement than ufunc optimization. Whether FP32 or FP64, comparing with ufunc optimization, "njit" significantly reduces the time of feature extraction on all FGs. Specifically, "njit" shorts the time cost of FP32 from 18.36 seconds to 5.94 seconds, also decreases execution time of FP64 from 34.491 seconds to 8.19.

Furthermore, by setting "parallel=True" enforces Numba to take advantage of all the cores on one machine, we can see that the performance of Numba-based optimization is improved further. Specifically, the running time of FP32 reduced from 5.94 seconds to 3.085 seconds, similarly, the time cost of FP64 to 8.192 from 4.56. Especially in FG1-4, "njit+parallel" method costs 0.281, 0.273, 0.314, and 0.291 seconds respectively, which approximately speedup the FGs 3-4 times over "njit" only method. There are no difference in the time cost of loading (I/O), preprocessing and FFT, since all of them are CPU serial execution without any optimization.

Furthermore, we optimize the baseline with CuPy and try to yield better speed-up with the GPU. Here we mainly take results from FP64 as comparative

Table 3.5: Execution time of njit and njit+parallel optimization with one data

	FG1	FG2	FG3	FG4	FFT	Preprocess	Load(I/O)	Total (seconds)
FP32(njit)	0.868	0.893	1.338	0.893	0.814	0.76	0.383	5.949
FP64(njit)	1.11	1.107	1.581	1.081	1.208	1.527	0.578	8.192
FP32(njit+parallel)	0.253	0.249	0.252	0.224	0.656	1.015	0.436	3.085
FP64(njit+parallel)	0.281	0.273	0.314	0.291	1.0768	1.733	0.592	4.56

reference since the baseline was in FP64. Firstly, since the time spent executing preprocessing on host memory is less than the time spent on executing it on device memory (and transferring data between CPU and GPU), we keep execution on CPU and host memory. Therefore, we can see from Figure 3.6 that our CuPy method makes no difference in time as Numba on I/O and preprocessing. However, it is worth noting that CuPy yields much better performance than Numba (both of njit and njit+parallel) on the rest of the tasks. Specifically, as Figure 3.6 showed, CuPy's FFT function (FP64) only costs 0.064 seconds to complete computation on all frames of one SnS data. This resulted in a 15x speed-up over NumPy's FFT function (it has been fully optimized by Intel). Moreover, CuPy's accelerated method costs just 0.07 seconds on FG 1, thus resulting in a 3x speed-up over Numba FP64 (njit+parallel) method. Similarly, for FG 2, CuPy costs 0.0171 seconds – this is 16 times faster than FP64 (njit+parallel). CuPy costs merely 0.025 seconds on FG 4 respectively (a 12x speed-up on FG 4 over the Numba FP64).

However, for FG3, CuPy optimization is slower than Numba (njit+parallel). The first reason is that we adopted different algorithm GPU. In the baseline and Numba-based methods, there is a feature that needs element-wise cumulative summing on arrays and element-wise multiply by another array, after that, each of element of result also has to do a "if" operation, GPU is not good at this kind of task. We decompose this feature into 2 parts, we first calculate element-wised cumulative summing and other calculations on arrays by CuPy build-in functions, and then we do "if" and

output the feature by use of CPU after exporting the result of cumulative sum back to Host memory.

Therefore, for FG3, "njit+parallel" will be better choice. We replace the FG3 function in CuPy with "njit+parallel" function. As Table 3.6 shows, we combine the best performance methods together for the best overall performance, the combination of "CuPy+Numba" costs only 2.196 seconds to finish all the feature extraction with on SnS data.

Table 3.6: Execution time of CuPy(CuPy+Numba) optimizations on P100 with one SnS data

	FG1	FG2	FG3	FG4	FFT	Precess	Load (I/O)	Total(seconds)
FP32(CuPy on P100)	0.0362	0.0107	1.1408	0.0193	0.0427	0.8497	0.4176	2.517
FP64(CuPy on P100)	0.0707	0.0171	1.1705	0.025	0.0646	1.17	0.5963	3.114
FP64(CuPy on P100+Numba)	0.068	0.0165	0.327	0.027	0.0721	1.071	0.615	2.196

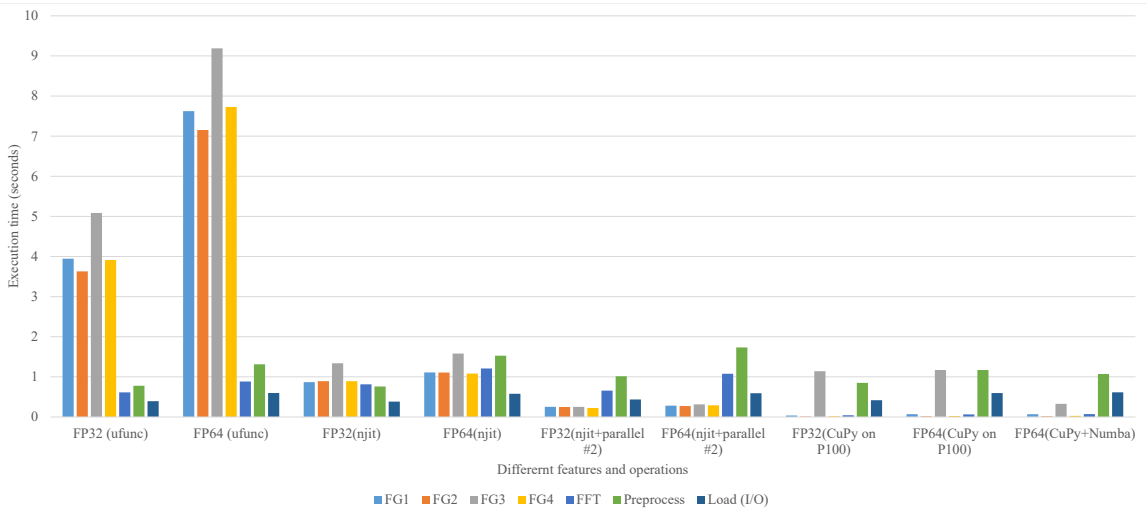


Figure 3.5: Comparing all optimization methods with execution time on one SnS data

A horizontal comparison as Figure 3.5 and 3.6 showed, the time spent in execution obviously decrease with different optimizations. We can see that loading one test data into memory for avoiding frequently I/O from hard disk can speed up the baseline 1.14x. Then, Using NumPy built-in ufunc functions to vectorized all calculations can also make the baseline of feature extraction 5.492 times faster. After that, by use

of Numba, "njit" and "njit+parallel" methods are approximately 23 times and 41 times faster than the baseline respectively. Finally, combination of "CuPy+Numba" optimization is 86.26 times faster than the CPU-base baseline (The result of FG3 in combination optimization is based on the configuration of GPU 3.3).

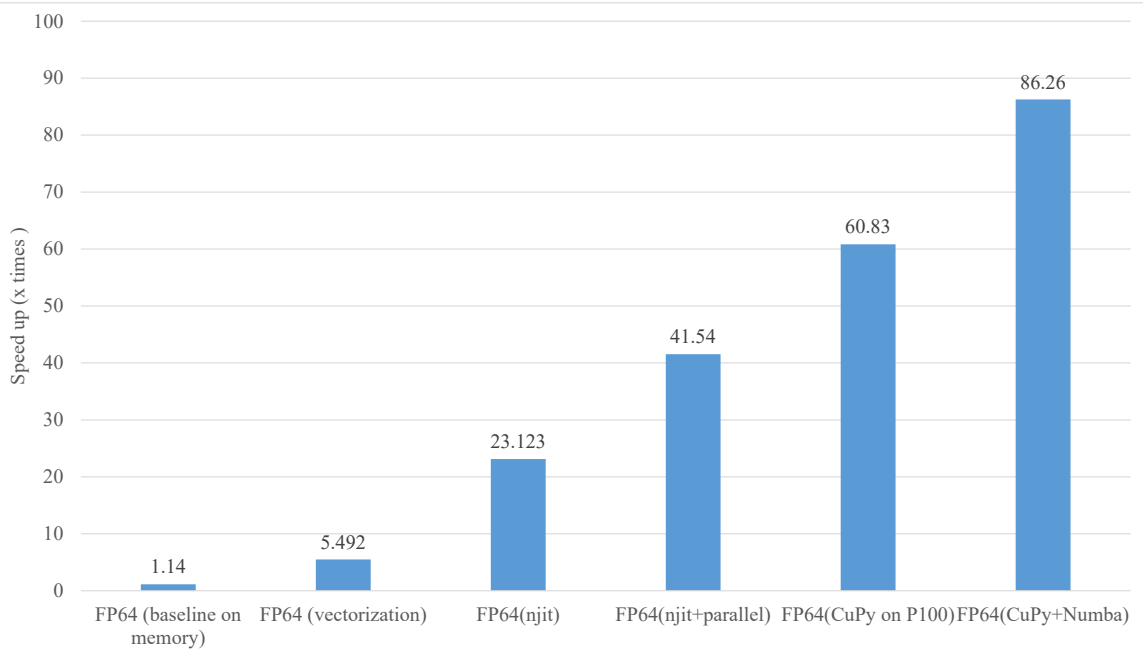


Figure 3.6: Speedup of Feature Extraction on All Optimizations Comparing with Baseline on one SnS Data

As Figure 3.7 showed, the bottlenecks were calculations of different FGs (97% of total run time), among them, FG3 and FG4 were the biggest two bottlenecks, which occupied 44% and 30% of total run time respectively. At the same time, the I/O and preprocess cost less than 2% of total time in the baseline. After using the combination method, the run time of feature extraction on FGs has been greatly reduced. The bottlenecks are preprocess, I/O (occupy 49% and 28% respectively) now, optimizing those two time-consuming parts will be our future work.

Table 3.7 shows that our CuPy-based optimization has met the memory band on P100 GPU.

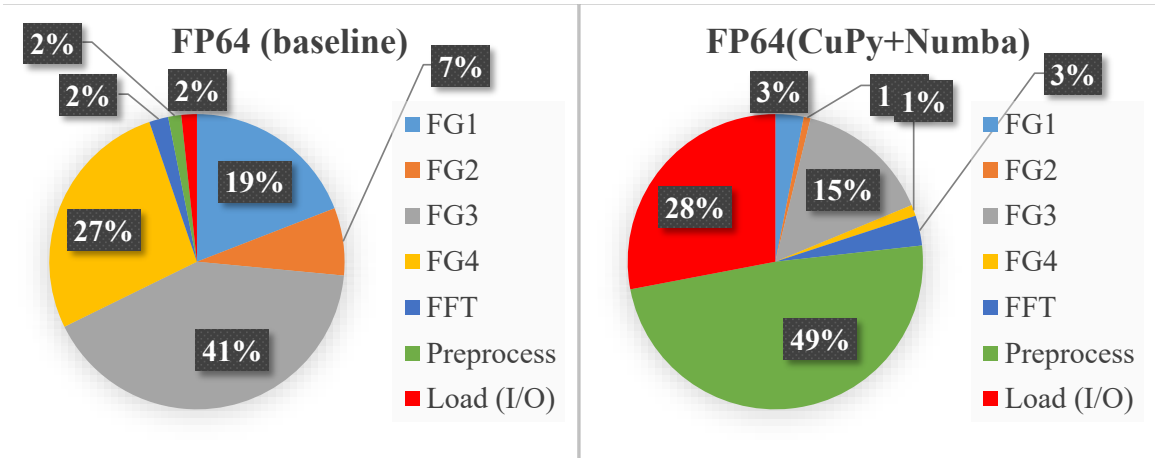


Figure 3.7: Bottleneck before and after optimization

Table 3.7: Memory Band on CuPy+Numba Optimization

	cupy_power	fill	cupy_true_divide	cupy_sum	cupy_subtract	cupy_add
Read Throughput(Avg)	149.50GB/s	161.09MB/s	211.45GB/s	117.25GB/s	220.30GB/s	153.63GB/s
Wrtie Throughput(Avg)	149.46GB/s	217.36MB/s	94.752GB/s	3.1632GB/s	92.263GB/s	108.01GB/s
Total	298.96GB/s	378.45MB/s	306.202GB/s	120.412GB/s	312.563GB/s	261.64GB/s
	cupy_log10	cupy_multiply	fft	cupy_copy	cupy_absolute	
Read Throughput(Avg)	129.60GB/s	246.18GB/s	249.30GB/s	213.69GB/s	345.10GB/s	
Wrtie Throughput(Avg)	96.283GB/s	245.17GB/s	249.25GB/s	269.43GB/s	172.53GB/s	
Total	225.882GB/s	491.35GB/s	498.55GB/s	483.12GB/s	517.63GB/s	

In general, we can see clearly that, our CPU-based optimization Numba based optimization always achieve about 41 times speed-up by the baseline. Likewise, the combined optimization method is about 86.26x speed-up on one SnS data. For all 31 SnS data (8.1 GB), baseline costs around 86 minutes to finish feature extraction on all of them. While our combined optimization method complete the same work in just 68 seconds.

Currently, in the Department of Otolaryngology, Beijing Hospital, there are 2 sleeping labs for collecting SnS data. In each lab, one OSA patient will be recorded 8 hours of whole night SnS data, one of 8 hours whole night SnS data is about 5.2 GB, 2 labs will collect 10.4 GB (5.2 GB x 2) whole night SnS data per day. With the baseline program, doctors and engineers need to wait for at least 90-100 minutes everyday for getting features from 2 SnS data (Our 8.1 GB experiment data costs 86 minutes, 10.4 GB data definitely needs more time than our experiment costed).

Apart from daily SnS data, there are some SnS data sent by other partner hospitals weekly since more and more people are beginning to notice this disease and start seeking treatment, the size of SnS database grow gradually because of above reasons. All of those SnS data needs to be extracted features firstly, then feeding to Machine Learning algorithms for assisting doctors to diagnose the OSA. According to economic and manpower situations, renting or purchasing multiple HPC nodes, and put those SnS data into nodes by data parallelism. Then with our optimizations, the time cost of feature extraction will be greatly reduced, which means that the efficiency of entire process of Machine Learning for aiding OSA diagnosis will benefit from the accelerating feature extraction.

3.1.4 Programming Efficiency for HPC-based Optimizations

As we mentioned in section 1.2, not all the computer-related practitioners understand parallel computing and how to use CUDA C to reconstruct existing programs. Since the baseline was programmed in NumPy-based Python. Therefore, one of our goals is to achieve good speedup performance with minimal modifications, rather than rewriting the program using a low-level language. Through our practice, we observe different methods for different problem types and summarize experience through experiment, which will promote HPC to a wider range of applications. We compare the programming efficiency of Python-based optimizations with the baseline code. Our comparison based on follow aspects: 1. Number of lines of code, 2. How many weeks we spent on programming, 3. Is there any special required changes? 4. Speed-up compared to baseline.

As Table 3.8 shows, ufunc is the easiest to implement by using NumPy. Programmers who is familiar with NumPy can easily write array-based ufunc calculations. However, the performance is low comparing to other methods.

Table 3.8: Programming Efficiency on Different Optimizations

	Lines	Time cost (weeks)	Required changes	Speed-up over baseline on One SnS Data(FP64)
Baseline	166			1x
NumPy (ufunc)	137	1	Array-based Universal functions in NumPy	5.5x
Numba (njit)	147	1	Adding "njit" decoration; Clear loop structure; Numba-supported functions only	23.1x
Numba (njit+parallel)	147	2	Adding "njit" and "parallel" decorations; Clear loop structure; Numba-supported functions only; Explicit loops; No cross iteration dependencies in loop (extra rules explained below)	41.5x
CuPy	144	1	Using NumPy-based ufunc code; Changing function names from NumPy to CuPy; Data transferring between Host/Device	60.8x
Numba+CuPy	149	3	All changes are required by Numba and CuPy	86.3x

The "njit" and "njit+parallel" methods are based on Numba, which needs code having a clear loop structure for Numba compiler. Especially for "njit+parallel" method, programmers have to follow some special rules such as: 1. Explicit parallel loops (change "range" with "prange") specify that a loop can be parallelized. 2. Users have to make sure that the loop does not have cross iteration dependencies except for supported reductions. 3. Can only be used on constant loops. 4. Within a nested loop, index names of different levels of loops cannot be repeated.

Using CuPy is as easy as using NumPy, first we need data transferring between Host and Device memory, which can be done by examples like "array_gpu = cupy.asarray(array_cpu)" and "array_cpu = cupy.asnumpy(array_gpu)". Then we just need to change the function name from NumPy to CuPy in our NumPy-based ufunc code, then CuPy will automatically convert it into CUDA code that is executed on the GPU. Although CuPy provides a convenient interface, our code needs to follow the principles of GPU programming. As we explained, the reason why CuPy-based FG3 is not faster than CPU-based "njit+parallel" version since its special implementation. According to the experiment results in Table 3.8, we think that the CuPy-based optimization is the best trade-off between speed and programming efficiency for extracting SnS features.

All in all, we can see clearly that, by using Python-based toolkits, we can improve the performance of existing high-level code with minimal modifications, rather than rewriting the program using a low-level language, which make it easier to use HPC in various fields, of course, including Machine Learning.

3.1.5 Summary

Today, conventional Machine Learning still plays an irreplaceable role in academic and industrial fields like bioacoustics engineering, financial related fields, and so on. Therefore, we would like to improve the overall efficiency of conventional Machine Learning by accelerating feature extraction – considered to be the most important art in Machine Learning. Specifically, we accelerate bioacoustics data feature extraction on HPC and GPU accelerators by employing Numba, a JIT compiler technique, as well as CuPy, a GPU accelerated Numpy-like library going by the principle of modifying Python code as little as possible.

All in all, the final results of our experiments shows the Numba based optimization achieve around a 41x speed-up over the baseline. Likewise, CuPy+Numba combination achieves around a 86x speed-up. In conclusion, our optimization methods can greatly accelerate feature extraction, which will further shorten run time cost and improve the overall efficiency of Machine Learning in practical applications. Dotted box in Figure 3.8 marks the contribution of this section.

3.2 Accelerating the Training of Autoencoder by use of HPC Software and Hardware

In the previous subchapter, we present accelerating the feature extraction part in a SnS data-based Machine Learning application by use of HPC software and

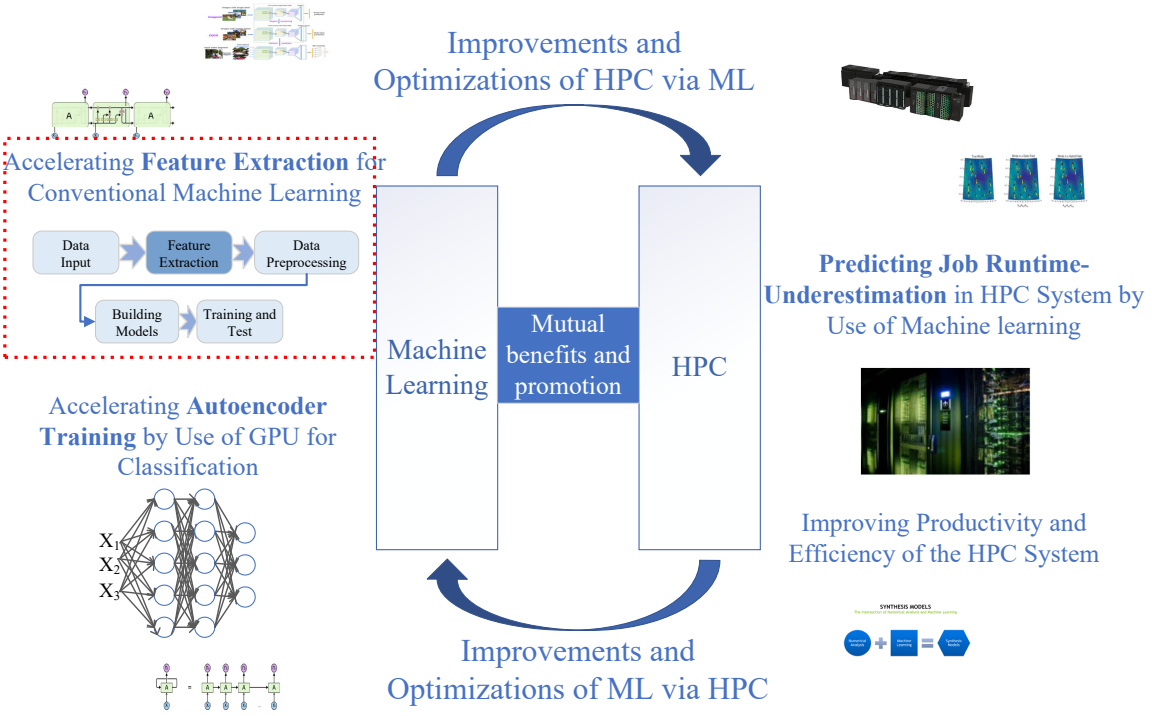


Figure 3.8: Extend and Strengthen the Mutual Relationship between HPC and Machine Learning

hardware. However, since the training part of this SnS data-based application is copyright, we cannot access the specific settings and parameters of the training part in this application. Then, we decide to use bird sound for accelerating the training part, because it is also a kind of bioacoustics data, and we can use feature extraction toolkit [18] to extract common feature sets ('eGeMAPS', 'IS09_emotion' and 'IS13_ComParE') from bird sound data, which was proved to be efficient in bird sound recognition [18][71].

In this chapter, we propose a Stacked Sparse Autoencoder (SSAE) and quantitatively compare the speed up of CPU, and GPU on training this stacked SSAE for classification of bird sound data. First, we give a short description on bird sound database and features we use in our work. Then, we introduce our SSAE trained with a softmax layer for bird sound classification. Finally, we show the

configurations of experiment and show the experiment results about speed-up of training SSAE.

The Autoencoder is a type of neural network which is used for unsupervised learning problems. It compresses the input data into a shorter code, and then the Autoencoder decompress that code into something that keep original information as much as possible [48][49]. Moreover, an improved version - Sparse Autoencoder (SAE) is proposed, in which, the hidden layer actually may has more or less hidden units than inputs data, but only a small set of the hidden units in the hidden layer will be activated at the same time [50]. Both of them are employed for dimension reduction, unsupervised learning and learning generative models of data [51].

The Stacked Sparse Autoencoder (SSAE) is consist of multiple hidden layers (more than 2 layers generally) of SAE, in which the outputs of each layer will be the inputs of the next layer. The SSAE trained with a softmax layer has been used for classification and regression problems. Other than using GPUs to accelerate CNNs for computer vision and RNN/LSTM for NLP, We examine whether or not training an SSAE with a softmax layer can benefit from using GPU. We take bird sounds classification as an example and train the SSAE with a softmax layer for this task, we quantitatively compare training speed-ups of SSAE using CPU and GPU.

Bird sounds, regarded as the "speech of birds", are significant not only for ornithologists in studying birds' species, mate, activities, etc. [29], but also for the measurement of climate changes, and biodiversity of a reserve by ecologists [30]. With the increasingly collected amount of bird sound data by experts and amateurs, researchers in the highly active areas of 'deep learning' recently, which can contribute by digging such large amount of bird sound data efficiently. However, training SSAE is a time-consuming task for traditional CPU-based computing, which makes it difficult to handle the bird sound data. In this work, we train a

SSAE with a softmax layer and explore the potential in performance employing the GPU in training the SSAE.

3.2.1 Database and Features of Bird Sounds

Work Flow

As Figure 3.9 shows, the audio recordings of the bird sounds are processed by signal processing techniques to extract acoustic features, which represent the salient characteristics of the birds' sounds. These features act as the 'input' of autoencoders for the subsequent training in our consideration. The autoencoders and their 'output' can be further used as supervised, unsupervised, or semi-supervised learning elements, which contribute to the bird sounds study.

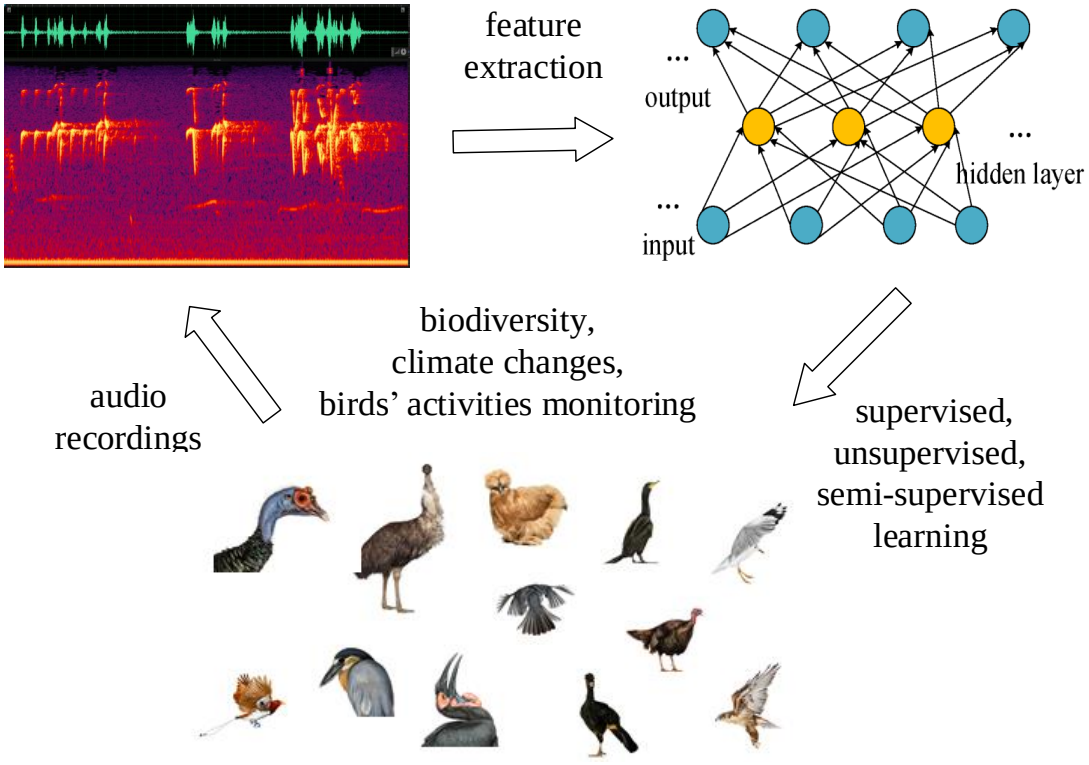


Figure 3.9: *The Work Flow of Training Autoencoders for Bird Sounds Classification*

Feature Extraction and Database

The bird sound data are provided by the Museum fur Naturkunde Berlin (MNB), Berlin, Germany. The original database contains bird sounds of 273 species (subspecies) in 6488 audio recordings. We removed the ones with less than 20 audio recordings, and separated the sounds from entire 86 species of birds (5060) into training (3070 recordings), development (972 recordings), and test (1018 recordings) set. The minimum, maximum, and average time duration of these recordings is 0.330 s, 59.033 s, and 2.835 s, respectively. In each set of data, sounds from 86 species of birds are included.

Table 3.9: openSMILE Feature Sets.

Feature sets	Dimensions
'eGeMAPS'	88
'IS09_emotion'	384
'IS13_ComParE'	6 373

Here we use a toolkit named openSMILE [18] to extract three different kinds of feature sets ('eGeMAPS', 'IS09_emotion' and 'IS13_ComParE') from bird sound data, which was proved to be efficient in bird sound recognition [72, 73]. Features base on Low Level Descriptors (LLDs) with functionals form the input for training the SSAE. We use those three different dimensions of feature sets to train our autoencoder for comparing the GPU based speed-up with different feature dimensions. The feature number per audio instance are given in Table 3.9. These features were designed by audio experts, including pitch, formants, Mel-Frequency Cepstral Coefficients (MFCCs), spectral parameters, energy/amplitude related parameters, to name but a few. The sets represent different feature dimensions scaling at 10, 10^2, and 10^3, respectively.

3.2.2 Training and Accelerating a Stacked Autoencoder for Bird Sound Classification

The Autoencoders are feedforward neural networks, which aim at minimizing the reconstruction errors between the inputs $\boldsymbol{X}$, to the outputs $\widetilde{\boldsymbol{X}}$ as:

$$\boldsymbol{J(X, \widetilde{X}) = ||X - \widetilde{X}||^2,} \tag{3.7}$$

where $\boldsymbol{X}$, $\widetilde{\boldsymbol{X}}$ represents the inputs, and outputs of the autoencoders respectively. In a simplest case, when feeding $\boldsymbol{X}$ into a one-hidden-layer autoencoder, a new map will be generated as:

$$\boldsymbol{Y = \sigma_1(WX + b),} \tag{3.8}$$

where $\boldsymbol{W}$ is the weight vector, and $\boldsymbol{b}$ is the bias. $\boldsymbol{\sigma}$ is the activation function. $\widetilde{\boldsymbol{X}}$ is reconstructed as:

$$\boldsymbol{\widetilde{X} = \sigma_2(\widetilde{W}Y + \widetilde{b}).} \tag{3.9}$$

Thus, 'autoencoders' can learn some higher level features from the original inputs, i. e. lower level features.

The Stacked Sparse Autoencoder (SSAE) is consist of multiple hidden layers (more than 2 layers generally) of SAE, in which the outputs of each layer will be the inputs of the next layer. Formally, as Figure 3.10 shows, consider a SSAE with N layers. The input X(k) is raw input data, The original information of raw data is contained within h(k)(m), which are the activation (or weights) of hidden units. This vector h can represent the input in terms of higher-order features. The features generated from SSAE can be used for classification problems by feeding h(k+1) to a softmax classifier after fine tuning.

In this section, we train a SSAE with 2 hidden layers for bird sound classification. First, we train a SAE on the raw inputs to learn features on the raw input as Figure 3.10 showing [74].

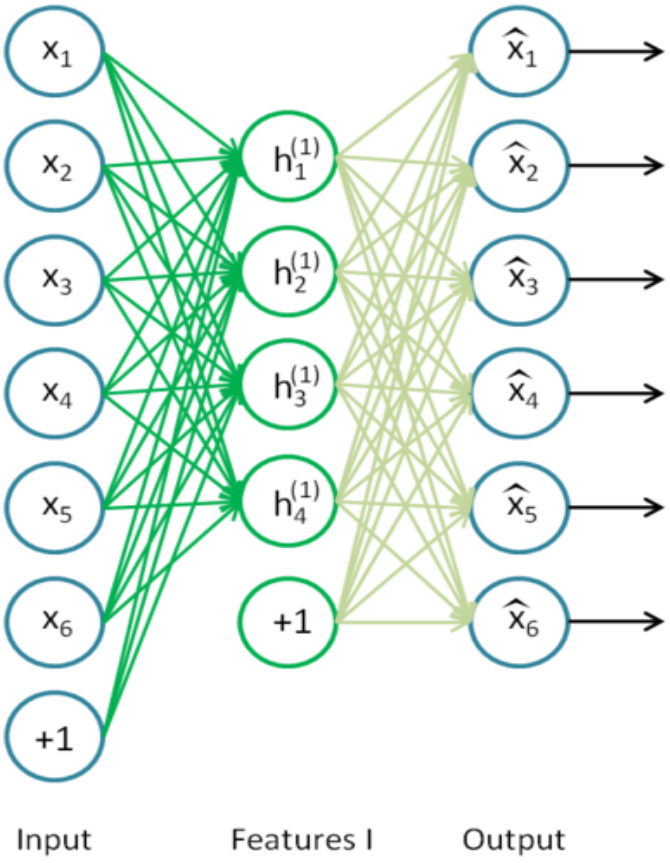


Figure 3.10: Training the SSAE: Step 1

Next, as Figure 3.11 showing [74], we feed the raw input into trained SAE, to obtain the first-level feature activations (or weights) h(1)(k) for the inputs x(k). Then feeding the first-level primary features to the second SAE as the "raw input" to learn second-level features h(2)(k) on the first-level features.

Following, we need to import the first-level features into the second SAE to obtain the second-level feature activations h(2)(k) for each of the first-level features h(1)(k). After that, we treat the second-level features as "raw input" to a softmax classifier, to train it as the second-level features for target labels as Figure 3.12 showing [74].

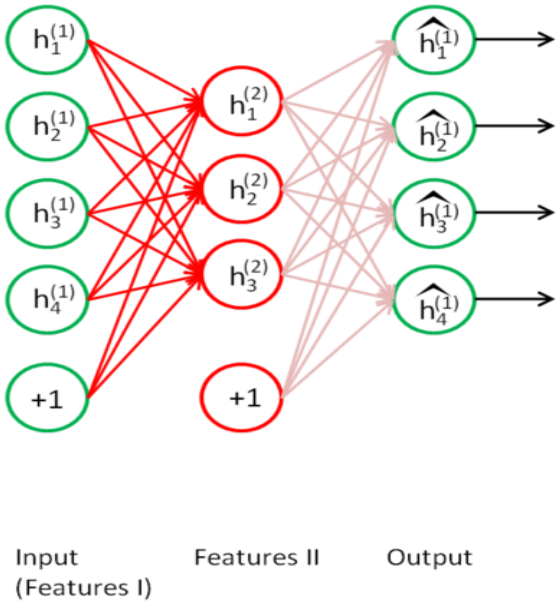


Figure 3.11: Training the SSAE: Step 2

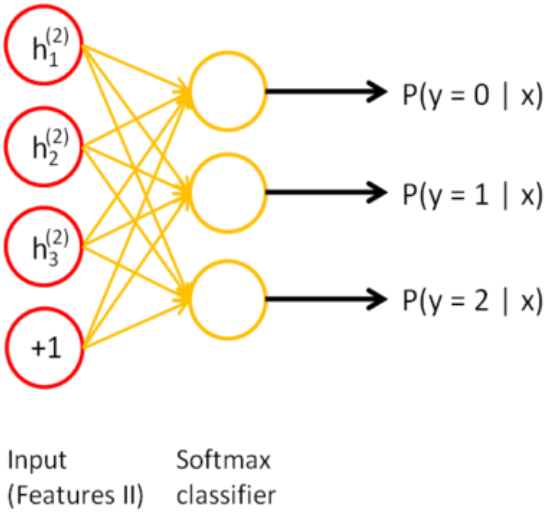


Figure 3.12: Training the SSAE: Step 3

Finally in Figure 3.13 [74], we combine all those three layers together to build a SSAE with 2 hidden layers with a softmax classifier layer as a classifier for classifying.

3.2.3 Experiment and Result

For some cases in training SSAE, there are more (rather than fewer) hidden units than inputs in their hidden layers. In this case, only a small number of the hidden

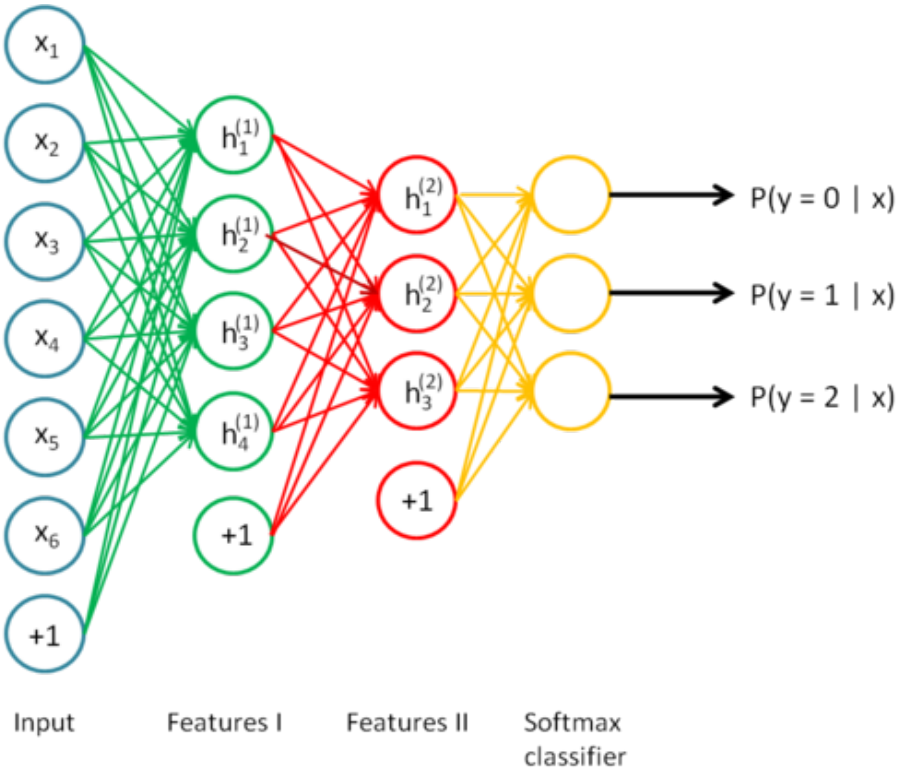


Figure 3.13: Training the SSAE: Step 4

units will be activated at the same time [50]. In this work, we set up the first and second hidden layers have 2048 and 1024 hidden units respectively, we train it with a learning rate of 0.1, the batch size of 256, a softmax layer with cross-entropy as cost function and the maximum iterations of 300.

Table 3.10: Hardware and Software Configuration

	Hardware and Software Configuration
CPU	Intel Core i7-3930K 3.2GHz (6 Cores, 12 Threads)
GPU 1	Tesla K20Xm 6 GiB GDDR5 with CUDA 8.0
GPU 2	Tesla P100 PCIE 16 GiB HBM2 with CUDA 8.0
Software	Python 3, MXNet 0.8.0, CuDnn 5.0

We conduct our experiments with hardware configurations shown as Table 3.10. Results are shown in Table 3.11. We can see that, the higher the feature dimension, the more time taking on training SSAE whether CPU or GPU. Specifically, the GPU 1 (a K20) can achieve around 7-8x times and the newer GPU 2 (a P100) can achieve

even almost 10x times the speed-up vs the CPU Intel Core i7-3930K (6 Cores, 12 Threads) when training with different feature dimension. The GPU will be one's first choice in future Autoencoder training.

Table 3.11: Time costing of CPU and GPU Training of SSAE

	Training Time			Speed-up over CPU
	CPU	K20x	P100	(K20x/P100)
Feature set 1 (eGeMAPS)	24.16	3.42	2.71	7x/8.9x
Feature set 2 (IS09_emotion)	30.04	4.15	3.27	7.2x/9.1x
Feature set 3 (IS13_ComPareE)	111.69	14.87	11.39	7.5x/9.8x

3.2.4 Summary

In this chapter, we propose a Stacked Sparse Autoencoder (SSAE) trained with a softmax layer and quantitatively compare the speed-up of CPU, and GPU on training this SSAE for bird sound data. Our experimental results show that the GPU outperformed the CPU on speed-up training SSAE about 10x faster, specifically, when the feature dimension, or data number is increased. Future work includes implementing end-to-end deep learning solutions like CNN and RNN for bird sound data classification, and train it on multiple GPUs. Dotted box in Figure 3.14 marks the contribution of this section.

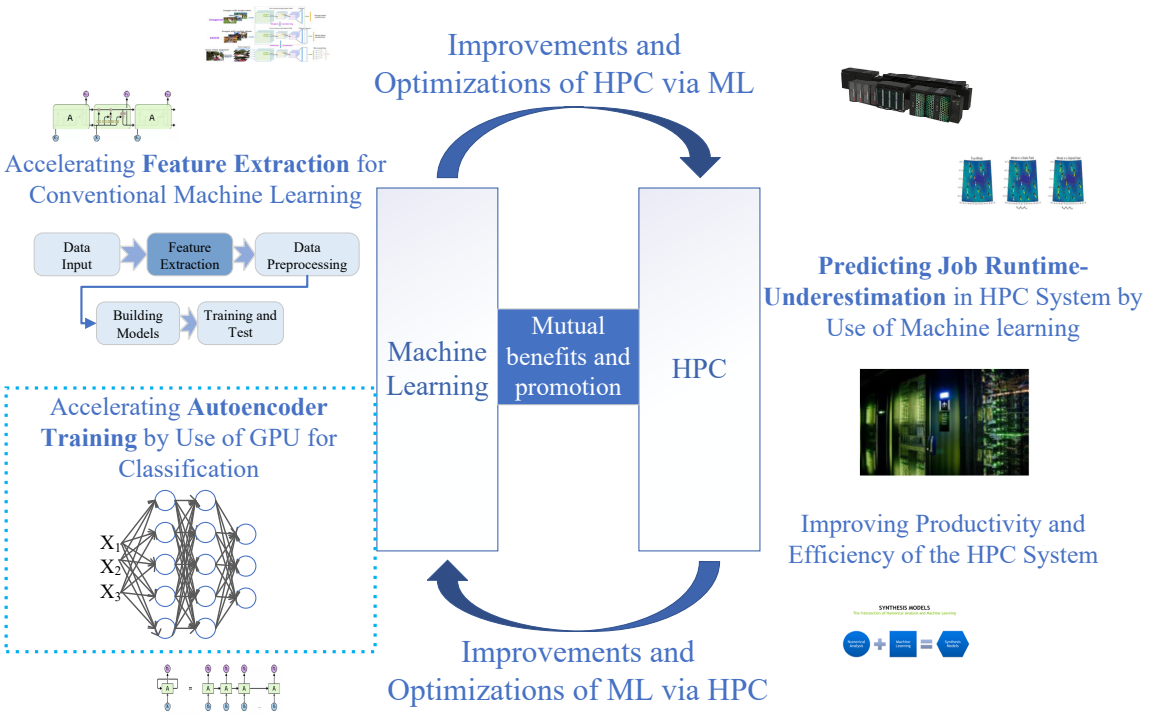


Figure 3.14: Extend and Strengthen the Mutual Relationship between HPC and Machine Learning

Chapter 4

Improving HPC system by Predicting the Job Run Time-underestimation with Machine Learning

The contents of this chapter were published in 2018 [75].

As we mentioned in chapter 1, usually HPC users are requested to estimate the run time of their jobs for system scheduling when they submitting jobs to HPC systems. Predicting jobs, especially those which may not finish before its allocated time expires, can mitigate wastes of time and system resources by taking early actions for run time-underestimated jobs. For instance, system administrators or an automated agent can explicitly send a notification to the user who submitted this job. The user can fix it by killing the job and resubmitting it after parameter adjustment. We propose a data-driven approach that is, one that actively observes, analyzes, and logs jobs for predicting underestimation of job run time on HPC systems. Using data produced by TSUBAME 2.5, we apply Machine Learning algorithms to recognize patterns

about whether the underestimation of job run time occurs. Figure 4.1 illustrates the overview of job run time underestimation prediction using Machine Learning models.

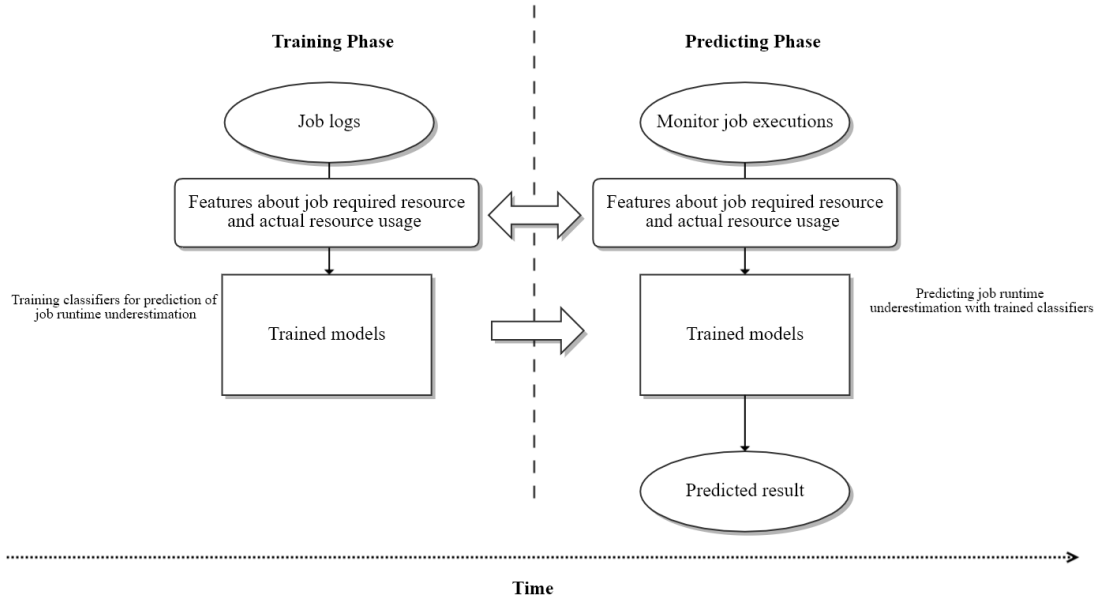


Figure 4.1: The Overview of Predicting Model for Job Run Time-Underestimation

For training predicting models, predicting models are trained with job features which are extracted as the mean values of computing resource utilization such as CPU usage, Memory usage etc. from raw time series data by sampling once every 60 seconds. Machine Learning algorithms extract features and characterize the hidden patterns of associated with labeled instances of job run time-underestimated, and build a models based on these hidden patterns so that the model can predict job run time-underestimation from the features in instances of job logs.

For prediction, the trained models can be used to predict unknown label instances about whether a job has run time underestimation. Furthermore, the trained Machine Learning models can also perform on-line prediction for predicting on-progress jobs. Since we extract same features from on going jobs through the same sampling frequency and method [58].

4.1 Workflow of Building Machine Learning Models for Predicting Job Run Time-underestimation

In this chapter, we introduce our approach from an overview description of designing and building workflow, followed by how to collect necessary data and feature engineering used for preprocessing the dataset. In the following we present the design and implementation of our machine learning-based prediction methods and the evaluation of our approach. Finally, we gave our detailed analysis based on experiment results and discussion. The overall workflow of our approach is depicted in Figure 4.2

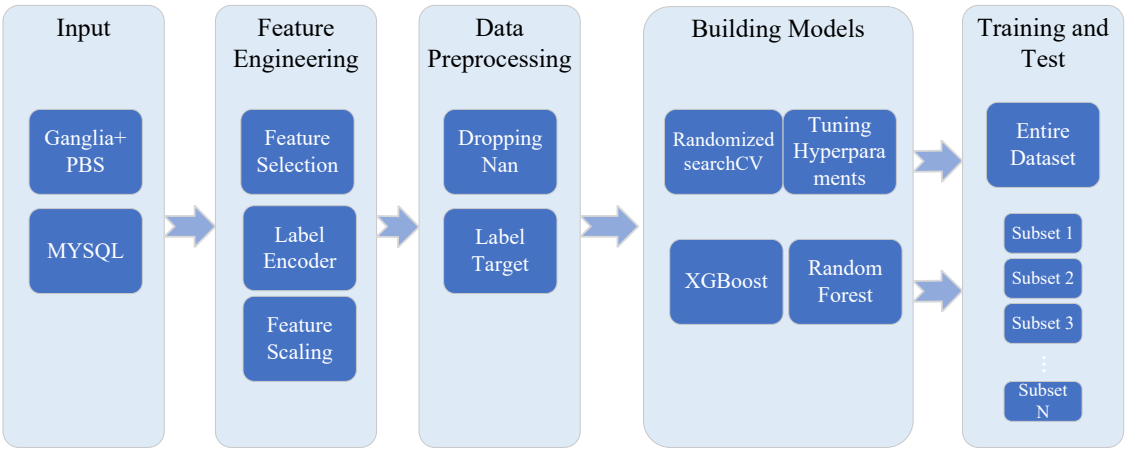


Figure 4.2: Overall Workflow about How to Build Machine Learning based Models

4.2 Data Collecting

Machine Learning algorithms learn from data. So the first step for training a Machine Learning model is gathering data. This step is very important, since the quality and quantity of data directly affect how good the Machine Learning model will be. Even

if data is good enough, we still need to make sure that it is in a useful scale, format and even that meaningful features are included.

In related work [76][77][78], computing resource utilization like CPU usage, Memory usage etc. have been proved to be related to job run time, job status, abnormal occurrence on HPC systems. Besides, paper [79][80][81][82] show that job's requesting compute resources (nodes, processors, memory, CPU cores etc.) when users submitting their job to an HPC system will correspondingly affect predicting results as well. Requesting compute resources and associated computing resource utilization usage are collected as logs of each job, which can be used to build prediction models for job run time underestimation. So, how to efficiently collect high quality and useful data is the first step in this research. PBS [83] and Ganglia [54] are used to collect job logs data as mentioned above.

4.2.1 Data Collecting from PBS

In general, accessing an HPC system is done through a system of login nodes. These nodes are reserved solely for the purpose of managing one's files and submitting jobs to the batch system. Acceptable activities include editing/creating files, uploading and downloading files of moderate size, and managing one's batch jobs. Users may also compile and link small-to-moderate size programs on the login nodes. An example of a login node can be found at Figure 4.3. Since CPU and memory usage are severely limited on the login nodes, meanwhile there are typically many users on the login nodes at any one time. Therefore, extensive calculations would degrade the responsiveness of those nodes. Submitted jobs are queued and then run as resources become available. The scheduling policies in place on the system are an attempt to balance the desire for short queue waits against the need for efficient system utilization. To run a batch job, users need to put the commands into a text file instead of typing them at the prompt. Then user submit this file to the PBS,

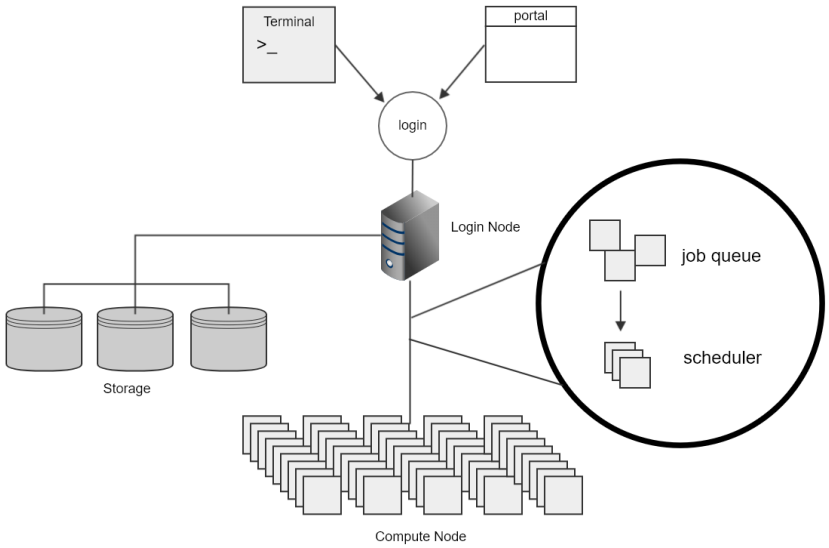


Figure 4.3: Job Submitting and Scheduling in HPC system

which will run it as soon as resources become available. The output of a job would normally go into a log file. Users can check the status of their job interactively and/or receive emails when it begins and ends execution. The overview of processing in an HPC system is shown in Figure 4.3

As we discussed in Chapter 2.2, TSUBAME 2.5 use the PBS to perform job scheduling. The PBS allows users to submit jobs requesting the resources (nodes, processors, memory, GPUs etc.) that they need to scheduler. We collect these jobs requesting the resources information from accounting logs of each job through the PBS. Table 4.1 shows the logs data of each job we collected by PBS from TSUBAME 2.5.

4.2.2 Data Collecting from Ganglia

On the other hand, the computing resource utilizations such as CPU usage, Memory usage etc. are also necessary data for building prediction models in our research. We use Ganglia to recorded statistics covering metrics such as CPU and memory utilization for many nodes in HPC systems. Ganglia is an open-source distributed

Table 4.1: List of Job Requests Information collected by PBS

Features	**Description**	**Features**	**Description**
queue	Job class name	req_pl	Requested priority
userhash	Anonymized user name	grouphash	Anonymized project name
nhosts	Calculated number of host involved	is_array	1 if the job is a part of array (parameter survey) job
req_mem	Requested memory amount	req_et	Requested option to extend maximum run time
req_ncpus	Recorded number of CPU used	req_gpus	Requested number of GPUs per node
req_walltime	Requested run time (wallclock time)	app	Recorded application which run inside of the job
queue_time	Timestamp of job submit	start_time	Timestamp of job submit
end_time	Timestamp of job finish	exit_state	Recorded exit status of job script
year	fiscal year	month	month

monitoring software for HPC systems, which has been widely used in thousands of grids and clusters around the world currently, and it can scale to handle clusters with several thousand of nodes easily [54]. Figure 4.4 shows the working principle diagram of Ganglia.

Computing resource utilizations during job running are saved as time series logs data of each job by Ganglia Table 4.2 shows the logs data of each job we collected by Ganglia from TSUBAME 2.5.

4.2.3 Sampling Time Series Data

Since the mean value is the most basic and commonly used average indicator, which make it a useful measure for time series data. Especially, if we needs a single number as a "typical" value which represent a set of known numbers, the mean value of this set of numbers does this best [84]. The disadvantage of arithmetic mean is that it is more susceptible to extreme values.

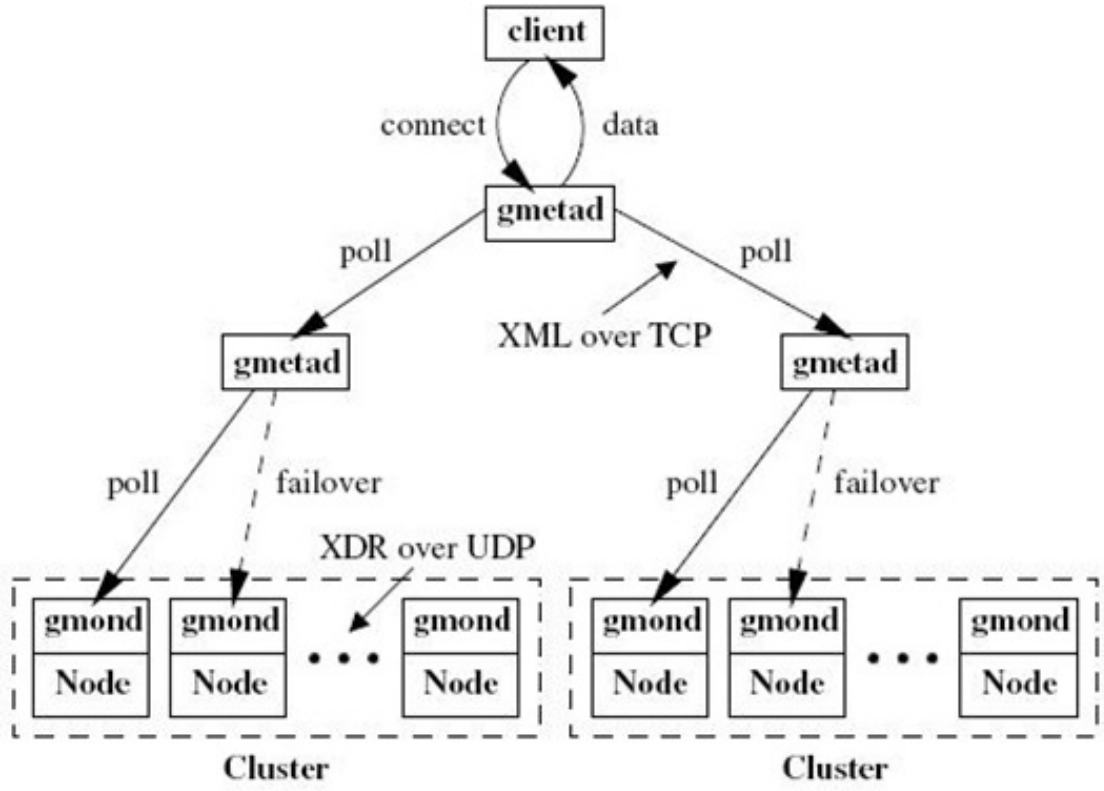


Figure 4.4: Ganglia Architecture

For every job, we extract the arithmetic mean value of computing resource utilizations such as CPU usage, Memory usage etc. from raw time series data collected by Ganglia by sampling once every 60 seconds. Since the raw time series data is uniformly sampled, the normalized weighting for each time is 1.0. Therefore, we use arithmetic mean value instead of time-weighting mean value.

4.3 Feature Engineering

Feature engineering refers to the process of using related knowledge to convert raw data into feature vectors as the input of Machine Learning algorithms. It is the most important initial step in Machine Learning, which directly affects the performance of models and usually requires a lot of time. Typical feature engineering process

Table 4.2: List of Computing Resource Usage Collected by Ganglia

Features	**Description**	**Features**	**Description**
used_cpupercent	Recorded CPU usage	used_mem	Recorded memory usage
used_ncpus	Recorded number of CPU used	used_vmem	Recorded memory address space usage
num_gpu_used	Number of GPU per node which is actually used	gpu_utilization	Recorded average GPU utilization per node (0.0 3.0)
used_nodesec	Used run time on per node	used_cputime	Used run time per CPU
used_walltime	Recorded run time (wallclock time)		

includes data clearing, feature extraction, feature selection, and so on [85]. The main purpose of extracting features is for interpreting in the context of a problem from raw data, which might help when solving the problem [86]. The quantity and quality of features will directly affect on whether the performance of Machine Learning model is good or not. Therefore, getting enough useful features from the raw data is the first step in building good models for solving our problem.

Here we have to mention that there are some people say that the deep learning does not require feature engineering anymore, for deep learning, researchers just need to feed raw data into deep neural networks, and the result will be available after training and testing is misleading. Actually, the deep learning use automated feature engineering algorithms to calculate and generate lots of candidate features, and to learn those candidate features with different types of neural networks [87]. Specifically, deep learning shows good results in computer vision and speech processing which process unstructured data such as images or audio. However, in our research, relevant features we collect from TSUBAME 2.5 are structured data instead of unstructured data. Therefore, we still use conventional Machine Learning techniques in this research since.

4.3.1 Feature Selection

From the previous sections, we know that the raw data about compute resources usage was time series data of extreme size. Directly feeding raw time series data into Machine Learning algorithms will produce unacceptable compute overhead, which will lead to serious delay between data collection and building model as well as wasting computational resource. Instead of using raw time series data, we selected a set of relevant features from the raw job logs data for building prediction models by normalizing and converting them to MySQL database. In Machine Learning tasks, selecting suitable and useful features is also an essential step to make the model is easier to be interpreted. Additionally one can enjoy decreasing the training times and avoid overfitting by reducing dimensionality of features [88].

In this research, our purpose is building a Machine Learning technique-based model that can predict whether a job is underestimated on its run time. Therefore we have to remove redundant or irrelevant features such as "***used_cputime***", "***used_nodesec***", "***used_walltime***", "***start_time***" and "***end_time***" for preventing knowledge leaking. This is a preliminary study in which we try to reveal complex patterns hidden in utilization of computing resources (may caused by system noises, and users cannot predict those noises), user behaviors, and different applications on an HPC system. Those features are redundant, which have a large impact on the prediction of job run time.

4.3.2 Label-Encoder

So far, we have selected enough feature variables as the training set and also have made corresponding labels as the test set. However, there are a few more important things needing to be addressed before we create the training Machine Learning model.

First is Label-encoding. For most Machine Learning-based modeling, the data that will be feed into Machine Learning algorithms must be in numerical type. However,

Table 4.3: 5 Instances with Selected 18 Features as Training Set and Test set

Training set (X)							
used_cpupercent	used_mem	used_ncpus	used_vmem	req_mem	req_ncpus	req_walltime	req_gpus
975	974532	3	8.95E+07	1.07E+09	3	3540	3
1733	4.8451e+06	12	5.49226e+06	2.14748e+09	12	10800	0
1196	1.12E+06	192	6.83E+08	2.15E+09	192	86400	48
896	828776	24	3.41E+08	2.15E+09	24	86400	3
1197	2.64E+06	12	9.20E+07	2.15E+09	12	86400	1

Training set (X)										Test set (y)
req_pl	req_et	nhosts	is_array	gpu_utilization	num_gpu_used	group	queue	user	app	Label: 1 or 0
0	0	1	0	81.0646	3	5	2	5	20	0
0	1	1	0	0	0	11	2	35	8	0
0	1	16	0	48.8958	3	166	5	368	11	0
0	1	1	0	169.295	3	10	5	473	11	0
0	1	1	0	49.8069	3	5	5	185	11	1

there are some feature columns that are non-numerical type in TSUBAME 2.5 log data. For instance, the column ***queue*** is list including [G, H, L128F, S, S96, X] which represents varying queues in TSUBAME 2.5 HPC system. In addition, the column ***userhash*** and ***grouphash*** keep hash values from 1 100 users and 421 user groups. Label-encoder can be used to transform non-numerical variables (as long as they are hashable and comparable) to numerical variables. For example, LabelEncodeing can turn [G,S96,G,H,S96] into [1,2,1,3,2], but then the imposed ordinality means that the average of G and H is S. In this work, we used Labelencoder to transform feature variables in columns ***userhash***, ***grouphash*** and ***queue*** from categorical variables to numerical variables. The example code as below shows how to use LabelEncoder class in scikit-learn transform non-numerical labels to numerical labels [89].

```
>>> from sklearn import preprocessing
>>> le = preprocessing.LabelEncoder()
>>> le.fit(["G", "S96", "G", "H", "S96"])
LabelEncoder()
>>> list(le.classes_)
['G', 'H', 'S96']
>>> le.transform(["S96", "S96", "H"])
```

```
array([2, 2, 1]...)
>>> list(le.inverse_transform([2, 2, 1]))
['S96', 'S96', 'H']
```

4.3.3 Feature Scaling

Second is feature standardization. Based on Table 4.3, we can see that the range of values of training set columns varies widely. For instance, in column ***used_mem***, values range from single units to millions of units. Meanwhile, in column ***used_cpupercent***, the values range from 0 to hundreds of thousands. In contrast with these two columns, the column ***is_array*** is bool type (0 or 1). For some Machine Learning algorithms, if the range of values in training set columns are in wide variation, gradient descent function will not work properly without feature Scaling, which will lead to the function cannot be converged [85].For example, most of distance-based classifiers calculate the distance between example points. If one of the feature column has a broader range of values, gradient descent function will be stuck by this particular feature. Therefore, the range of all features columns should be scaled in roughly the same range. To explain this, we present a visualization example for comparing the feature data distribution with feature scaling or without it. As Figure 4.5 showing, in the scaled version we can see that the orders of magnitude are roughly the same across all the features. The scaled set outperforms the unscaled one.

Feature standardization is one way to perform feature scaling, which would make the values of each feature column have zero-mean and unit-variance. It has been widely used in many Machine Learning algorithms (e.g., SVM, logistic regression, and neural networks) for feature scaling. Concretely, first we need to calculate the mean values and standard deviation for each feature column. After subtracting the

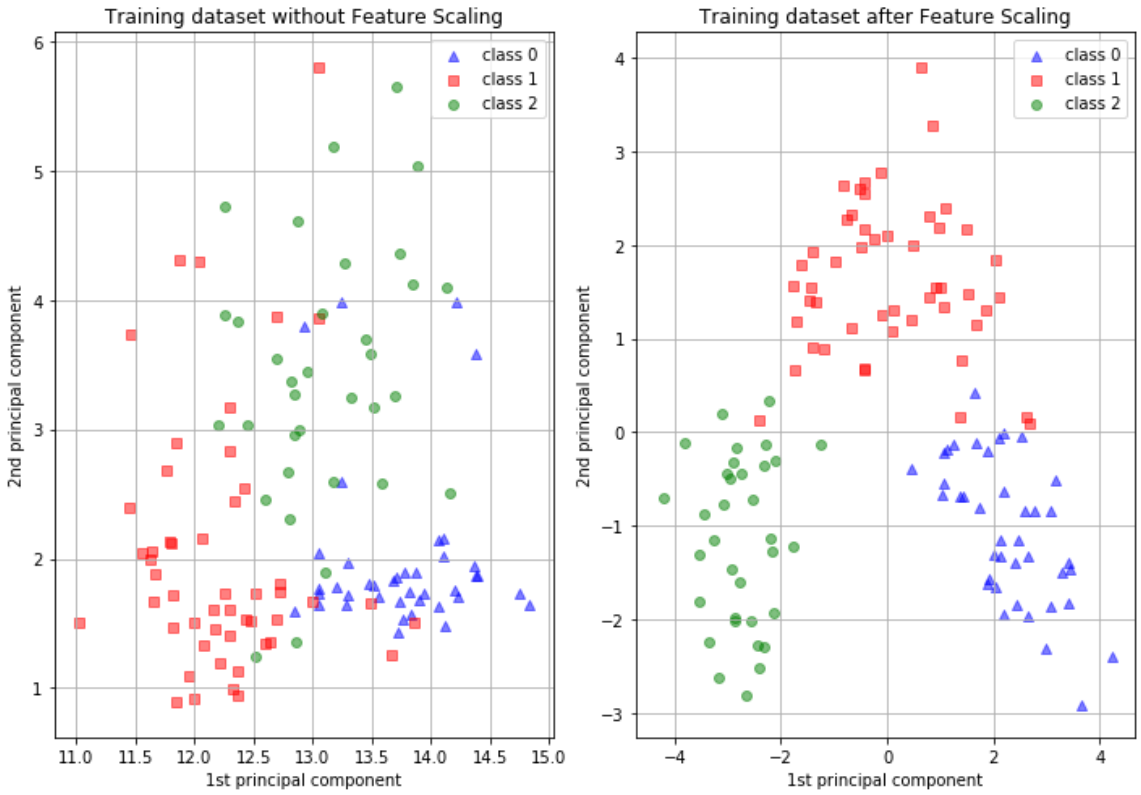


Figure 4.5: Importance of Feature Scaling

mean value from each feature value, we need to divide the values of each feature by its standard deviation. (since mean is already subtracted) which can be presented in the following formula:

$$\boldsymbol{x'} = \frac{\boldsymbol{x} - \bar{\boldsymbol{x}}}{\boldsymbol{\sigma}}$$

Where $\boldsymbol{x}$ is the original feature vector, $\bar{\boldsymbol{x}}$ is the mean of that feature vector, and $\boldsymbol{\sigma}$ is its standard deviation [90]. We give 5 job instances - with 18 selected features which as Table 4.3 showed.

One important thing to note, in most of Machine Learning cases, usually the dataset will be split into training, validation, and testing sets respectively. If we do feature scaling to calculate the mean value and standard deviation over the whole dataset, we will introduce "future information" into the training set; in other words, the mean and standard deviation that should haven't been know yet will be contained.

Therefore, we have to perform feature scaling just over the training set and keep the mean and standard deviation. Then do feature scaling on validation and test sets with those kept mean and standard deviation previously.

4.4 Data Pre-processing

4.4.1 Data Cleaning by Dropping NaN Data

Data cleaning is also an important step in Machine Learning projects. In Real-world, data-gathering methods are often loosely controlled, resulting in out-of-range values, impossible data or missing values, etc. Thus, the representation and quality of data is first and foremost before running an analysis [91]. After preliminary inspection, we find that there are missing values (NaN value) in our TSUBAME 2.5 job logs database, which may be due to errors in collecting software or methods. In order to help improve the quality of the data and predicting results, we clean our data by data Dropping NaN Data. For this work, we dropped all log job instances that contain NaN which may be caused by recording error when collecting data. The size of the logs from Jan. 1, 2015 to Dec. 31, 2016 is about 1 million job instances, which are enough for training and testing our models. Table 2.1 shows the summary of the entire job data set and subsets categorized by scientific application name.

4.4.2 Labeling Target

Additionally, we needed to create the target variable as the test set $\mathbf{y}$ which is then compared with the results produced with the training set $\mathbf{X}$. We label the test set by the following formula:

$$y' = j.used\_walltime - j.req\_walltime$$

where j.used_wall time is actual run time of a job, and j.req_wall time is user estimated time of a job. If $\boldsymbol{y' < 0}$, we label this job as 0 in the test set y, which means that the actual run time of this job does not exceed the user's estimated time when its user submitted it. Relatively, if $\boldsymbol{y' >= 0}$, this job will be labeled as 1 in the test set, which indicates run time-underestimation. In this case, this job will be terminated by the HPC system immediately before its completion. The purpose of our work is to predict whether a job is run time-underestimated after job submission. l

4.5 Machine Learning Algorithms for Building Prediction Models

We compare two popular supervised machine learning algorithms: Random Forest [92] and XGBoost [93]. The reason we chose to compare these two algorithms is that there are decision tree based models (both based on ensembles of decision trees) that solve tabular data very well, and have certain properties that a deep net does not have (e.g. ease of interpretation and invariant to input scale, and much easier to tune). Both of these methods are widely used as they outperform other distance-based algorithms like logistic regression, support vector machine, and kNN in data science.[94][93][95][96][56][97]

4.5.1 Random Forest

In general, Random Forest is a bagging learning method method that builds a multitude of decision trees when training phase, and then outputting prediction results of the individual decision trees [98] [99]. When training decision trees deeply, it tends to over-fitting the training dataset with low bias but high variance. Random Forest tries to reduce the variance by training different parts of the same training dataset and averaging out the results of multiple deep decision trees.

Bagging learning trains and fits multiple decision trees by repeatedly selecting a random sample with replacement of the training dataset and averaging the prediction results from all the individual trees [100].

4.5.2 Extreme Gradient Boosting (XGBoost)

XGBoost is a scalable tree boosting system that implements the gradient boosting decision tree algorithm. This is a generalized gradient boosting implementation that includes a regularization term, used to combat over-fitting, as well as support for arbitrary differential loss functions. The underlying principle behind XGBoost is Gradient Boosting, which itself relies heavily on Gradient Descent. It has been widely used by data scientists and provides state-of-the-art results on many problems [93].

4.6 Evaluation Metrics for Imbalanced Data Sets

We have realized very clearly that we are dealing with a binary classification problem on an extremely imbalanced dataset. There are 2 ways that are given by data scientists and researchers to deal with imbalanced data set. First is to collect more minority class data or to re-sample the imbalanced dataset by over-sampling (e.g. adding copies of instances from the minority class) or by under-sampling (e.g. deleting instances from the majority class). We cannot do either of these strategies, because over-sampling will increase the size of the data set thereby greatly extending training time, and under-sampling may lose important information as a consequence of dropped data. Second is to change the performance metrics. There are metrics that have been designed to get fair performance evaluation when working with imbalanced classes.

4.6.1 True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN)

In the field of Machine Learning, a confusion matrix for binary classification shows four different outcomes: true positive (TP), false positive (FP), true negative (TN), and false negative (FN). The actual values form the columns, and the predicted values (labels) form the rows. The intersection of the rows and columns show one of the four outcomes. [101]. The four outcomes are basic metrics in classification problems and can be formulated in Table 4.4.

Table 4.4: Contingency Table about Four Outcomes of Statistical Tests

	Predicted Positive	Predicted Negative
Actual Positive	True Positives (TP)	False Negatives (FN)
Actual Negative	False Positives (FP)	True Negatives (TN)

We explain these basic metrics in our research as below:

True Positives (TP): TP are the cases when the actual label of instance was True and the model also predict this instance as True. In this research, the case where a job is actually run time-underestimated and the model classifies the case as run time-underestimated falls under TP.

True Negatives (TN): TN are the cases when the actual label of instance was False and the model also predict this instance as False. In this research, the case where a job is NOT run time-underestimated and the model classifies the case as NOT run time-underestimated falls under TN.

False Positives (FP): FP are the cases when the actual label of instance was False, but the model predict this instance as True. In this research, the case where a job is NOT run time-underestimated however the model classifies the case as run time-underestimated comes under FP.

False Negatives (FN): FN are the cases when the actual label of instance was True, but the model predict this instance as False. In this research, the case where there is a run time-underestimated job however the model classifies the case as NOT run time-underestimated comes under FN.

4.6.2 Accuracy, Precision, Recall and F1-score

As we discuss in Section 2.2, accuracy is not a reliable metric for evaluating the performance of a classifier with imbalanced data set, because it will always yield very high accuracy if the data set is imbalanced. Precision, recall and their derivative metrics allow more detailed analysis for binary classification problem with imbalanced data. Precision can be calculated with formula 4.2; Recall is defined with formula 4.3. Figure 4.6 shows the definition of precision and recall in our research.

$$\boldsymbol{Accuracy} = \frac{\boldsymbol{TP} + \boldsymbol{TN}}{\boldsymbol{TP} + \boldsymbol{TN} + \boldsymbol{FP} + \boldsymbol{FN}} \tag{4.1}$$

$$\boldsymbol{Precision} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FP}} \tag{4.2}$$

$$\boldsymbol{Recall} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FN}} \tag{4.3}$$

4.6.3 Average Precision (AP) and Precision-Recall Curve (PRC)

Average Precision (AP)

The Average Precision (AP) is a single number metric used to summarize the overall performance of a model with an imbalanced data problem [102].

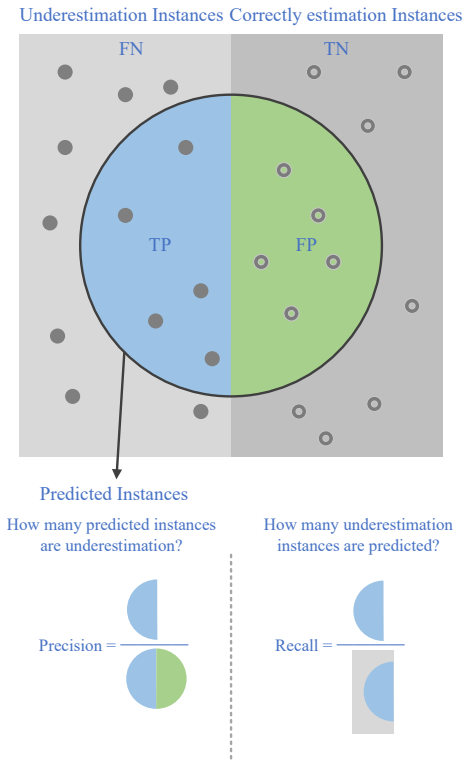


Figure 4.6: Precision and Recall

This is the average of the precision obtained every time a new positive sample is recalled. AP essentially comes from prediction scores, which corresponds to the area under the precision-recall curve. Concretely, the AP is the precision value that averaged over all values of recall between 0 and 1:

$$\int_0^1 \boldsymbol{p(r)dr}$$

This is equivalent to taking the area under the PRC. Put into practice, the integral can be calculated by summing a set of precision numbers with each possible threshold,

then multiplied by the change in recall:

$$\sum_{t=1}^{N} \boldsymbol{P(t)\Delta r(t)}$$

In this formula, N is the total number of instances in dataset, P(t) is the precision at threshold t, and delta r(t) is the change in recall that happened between thresholds t-1 and t. The value of AP is between 0 and 1; the higher the value, the better.

Precision-Recall Curve (PRC)

Precision-recall curve (PRC) is a better choice for imbalanced datasets [103]. The relationship between recall and precision can be observed is one kind of trade-off in PRC with different thresholds. Figure 4.7 shows two examples of the PRC and its space. The left figure in 4.7 shows a ideal case that has no overlap for the positive class and negative class, respectively. The ideal case means that the model is able to discriminate positive and negative of all instances with 100% recall and 100% precision. The PRC will passes through the upper right corner if it is a ideal model (be able to predict 100% precision and 100% recall). The right figure in 4.7 presents two Precision-recall curves examples, generally speaking, the closer a PRC is to the upper right corner, the better performance of the model.

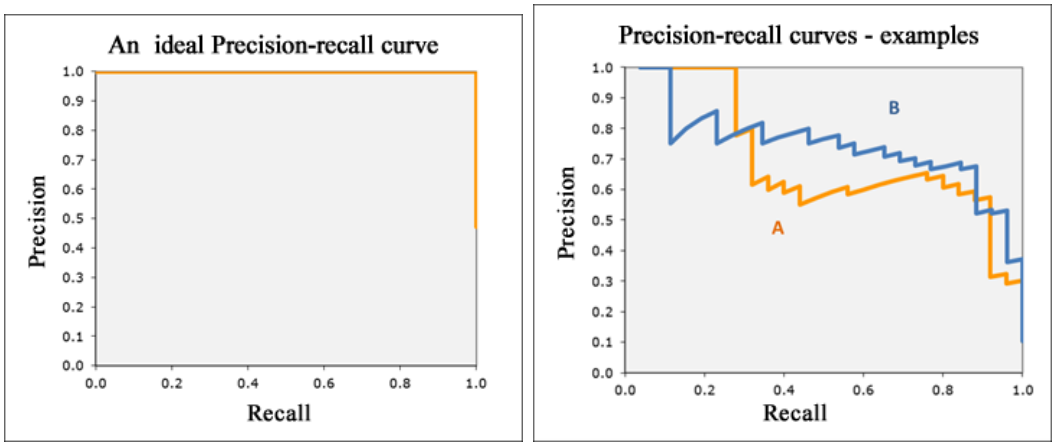


Figure 4.7: Two examples of Precision-Recall Curves (PRC) and PRC space

4.6.4 Precision and Recall Scores as A Function of The Decision Threshold

Plotting decision threshold with precision and recall can simply help to select the threshold value that gets the best precision-recall trade-off for different models [104]. To plot this decision threshold graph, for each instance, we first need to get the raw probability that an instance is predicted to be in a class. Next, for all instances in the training set, we obtain the raw probability scores that a given instance is predicted to be in a class. Then, we compute precision and recall for all possible thresholds based on those scores. Finally, we are able to plot precision and recall as functions of the threshold value. With the help of this relationship, we can simply select the threshold value that gives one the best precision-recall trade-off for different purposes. The following python code shows how to define a function to draw the decision threshold graph:

```
import numpy as np
import matplotlib.pyplot as plt
from xgboost import XGBClassifier
from sklearn.metrics import precision_recall_curve


xgb_clf = XGBClassifier()
# Train a XGBoost classifier
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=
                                   0.3)
# Split X, y into X_train, X_test, y_train, y_test
xgb_clf.fit(X_train, y_train)
precision, recall, threshold = precision_recall_curve(y_test,
                                   xgb_clf.predict_proba(X_test)[:,1]
                                   )
```

```
# Using trained classifier to fit y_test and calculate precision,
                                    recall, threshold by
                                    precision_recall_curve()

def plot_precision_recall_vs_threshold(precisions, recalls,
                                    thresholds):
    plt.figure(figsize=(8, 8))
    plt.title("Precision and Recall Scores as a function of the
                                        decision threshold on Entire
                                        Dataset")
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.ylabel("Score")
    plt.xlabel("Decision Threshold")
    plt.legend(loc='best')
# Define the plot function
plot_precision_recall_vs_threshold(precision, recall, threshold)
plt.show()
```

4.6.5 Using Right Metrics for Our Tasks

In all classifiers, a trade off will always occur between precision and recall. We hope to train a classifier that gives high precision over the minority class (label 1, a job having run time-underestimation), while maintaining reasonable precision and recall for the majority class. In the case of modeling on an extremely imbalanced dataset, quite often the minority class is of great significance. For our imbalanced binary classification problem, we will take advantage of the combination of the above-mentioned evaluation metrics to diagnose our model.

As AP is a single value, it is an ideal first metric to use to evaluate the overall performance of a classifier in the evaluation stage. After the best classifiers were

chosen with AP, we used PRC to evaluate precision vs recall in the minority class. From the user's point of view, False Positives are more dangerous and costly than False Negatives when predicting run time-underestimation. Following this logic, precision will be more important than recall in this case. On the other hand, seeing job run time-underestimation from the eyes of an HPC system administrator, saving system resources as much as possible takes priority. Thus in this case, recall will be more important than precision. The function of decision threshold helps to select the threshold value that get the best precision recall trade-off for different models.

4.7 Experiment Evaluation and Result Analysis

In this section, we describe the procedure about how to train, tune and test Machine Learning models with TSUBAME 2.5 job datasets for predicting job run time underestimation, and how to evaluate these predicting models with suitable metrics.

4.7.1 Experiment Setup

Software Environment

We introduce the hardware and software environment we use for building and tests. Currently, our work mainly focuses on building models and result analysis. There is no direct relationship with speed up, so the hardware information related to acceleration will not be introduced at this moment. The software, packages and their version are presented in Table 4.5. We use Anaconda 5.1 as a modeling platform. Numpy and Pandas are used to process data. Scikit-learn is used to build the random forest based models [105]. The XGBoost library is compiled from binary code and installed to Anaconda for building XGBoost based models.

Table 4.5: Software, Libraries and Version

Software Libraries	**Version**
Anaconda	5.1
Python	3.6
Scikit-learn	0.19.1
XGBoost	0.71
Numpy	1.14
Pandas	0.23

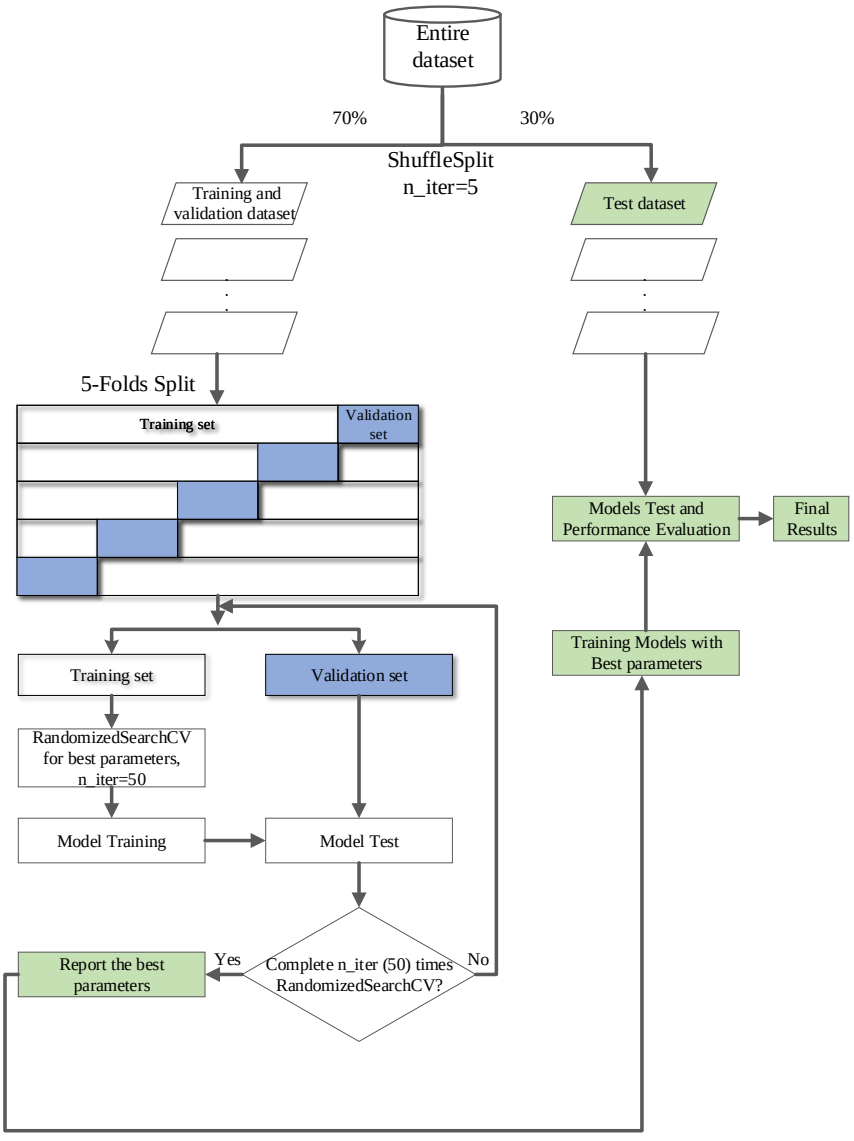


Figure 4.8: Experimental Design and Process

Procedure of Modeling and Evaluation

Building a good Machine Learning model needs rigorous steps to ensure its reliability. Firstly, we build our models according to the procedure of Figure 4.8 and evaluate models with the procedure of Figure 4.9. AP is a single value and it is the first metric used to evaluate the overall performance of a classifier in the evaluation stage. After the best classifiers were chosen with AP, we used PRC to evaluate specific precision and recall, because different types of users have different needs about precision and recall. The function of decision threshold helps to select the threshold value that get the best precision recall trade-off for different models.

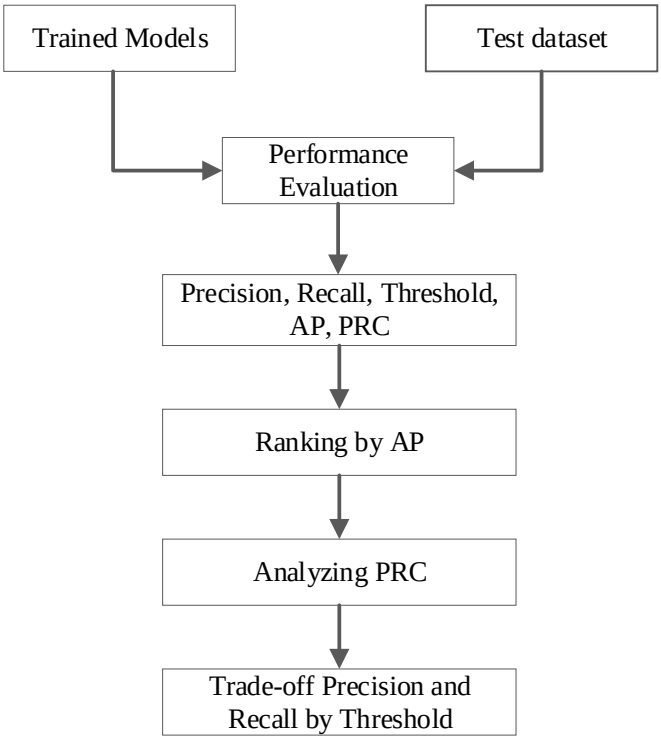


Figure 4.9: Evaluation Process for Dataset

Data Splitting with and ShuffleSplit Cross-Validation (Random permutation CV) and K-fold Cross-Validation (k-fold CV)

At first, we create a model for the entire dataset, then we split the entire dataset into subsets categorized by scientific application name and build sub-models for them, respectively. Regardless of whether we build a model with the entire dataset or with subsets, we split these dataset into training, validation and test sets at the beginning as Figure 4.8 showing. The training set is used for fitting Machine Learning models; the validation set is used for parameter tuning; the test set is used to evaluate the final results of prediction models. After performing the final model on the test set, the model and parameters cannot be tuned any further. Ideally, the test set should be used only at the end of the data analysis [106]. Since we split the entire job dataset into subsets, there are some subsets in which the absolute number of minority class samples is too small (see Table 2.1). Therefore, we use the ShuffleSplit Cross-Validation (Random permutation CV) to split 70% of the data as training and validation sets, and keep the rest 30% as a test set with 5 iterations at first [105]. Random permutation CV may return different results for each call of StuffleSplit. We make the results identical by setting $\boldsymbol{random_state}$ to an integer. The Random permutation CV method will randomly sample one's entire dataset during each iteration to generate a training set and a test set. The test size and training size parameters control how large the test and training test sets should be for each iteration. Since the user is sampling from the entire dataset during each iteration, values selected during one iteration could be selected again during another iteration. Then, for the 70% of the training and validation sets, we performed 5-fold cross-validation (5-fold CV) with RandomizedSearchCV for tuning hyper-parameters.

Next, we build a model with the best hyper-parameter combination which is decided after 50 RandomizedSearchCV iterations, and evaluate this model with the

test set. We will obtain one set of results (including precision, recall, threshold etc.) in each ShuffleSplit iteration, which means that we will have 5 sets of results after 5 ShuffleSplit iterations for each dataset. Next, for each dataset and model combination, we record 5 sets of precision-recall pairs, which can be calculated after 5 ShuffleSplit iterations for each dataset. A pair of precision and recall corresponds to one PRC. There are 5 PRC's and one mean PRC in each graph for showing the specific precision and recall relationship on every subset and every model.

Equivalent to getting 5 sets of precision-recall pairs, we record 5 AP values from 5 ShuffleSplit iterations and calculate the mean value of these 5 AP values as the first metric to evaluate the overall performance of the model on each dataset. Specifically, for one dataset, we train and test XGBoost and RF models on it with our procedure, respectively. Both XGBoost and RF model gets 5 AP values and 1 mean AP value among the 5 values. The mean AP value of XGBoost and RF is used to evaluate the overall performance of models, and the 5 AP values are used to evaluate the robustness of the model by box plot. Therefore, we not only need to evaluate how high the mean AP value is, but also to observe the stability of those 5 AP values.

After ranking models with AP, we plot precision and recall scores as a function of threshold to help select the threshold value that gets the best precision-recall trade-off for different models. Plotting precision and recall scores as a function of threshold also helps to select the threshold value that gets the best precision recall trade-off for a user or administrator's different purposes.

Tuning Hyper-parameters for Prediction Models

Meanwhile, most Machine Learning algorithms have several hyper-parameters that will affect the performance of models. Tuning hyper-parameters is an indispensable step to improve a model's performance, which often improve its accuracy or other metrics, like precision and recall, by 3-5%, depending on the algorithm and dataset.

In some cases, parameter tuning may improve the accuracy by around 50% [97]. In this study, we train our model and tune hyper-parameters via LOOCV with the RandomizedSearchCV function from scikit-learn [89]. The RandomizedSearchCV is an estimator used for optimizing hyper-parameters with different parameter settings. Comparing to GridSearchCV, instead of attempting all parameter values, but rather a set of random number of parameter settings are selected from the specified distributions. We set 5-fold cross-validation with n_iter to 50, and we also set AP as the scoring metric in RandomizedSearchCV. Parameter settings and optimized parameters are presented in Table 4.6

Table 4.6: Hyper-parameters Settings of XGBoost, Random Forest and the Best Hyper-parameters after Tuning in this Study

Machine Learning Algorithm	Hyper-parameters	Best Parameters after tuning
XGBoost	n_estimators: How many boosted decision trees to be used	n_estimators=200
	learning_rate: The learning rate	learning_rate=0.2
	max_depth: Maximum depth of each decision tree	max_depth=18
	max_delta_step: Setting it to a finite number (say 1) will help convergence	max_delta_step=2
	booster: Select one booster (gbtree, gblinear or dart)	booster=gbtree
Random Forest	n_estimators: How many boosted decision trees to be used	n_estimators=160
	min_weight_fraction_leaf: The minimum weighted required to be at a leaf node	min_weight_fraction_leaf=8
	max_features: Splitting the number of features (auto, sqrt, log2 or None)	max_features=18
	criterion: The function is used for measure the quality of a split	criterion="entropy"

4.7.2 Result and Analysis of Classification with Entire Dataset

We trained and tuned classifiers with the XGBoost and the RF on the entire dataset with procedure at Figure 4.8. We used the best chosen classifiers based on 5-fold CV on the training set (70% entire dataset) to predict the test set (30% entire dataset) 5 times with ShuffleSplit. Table 4.7 and Table 4.8 shows that the XGBoost and

the RF have an extremely similar overall performance result. The result consists of similar values of run time-underestimation prediction (in terms of overall precision and recall) in the entire dataset. As we estimated, the precision and recall of the majority class are very high on both algorithms (0.98, and as high as 0.99). In contrast, all metrics on the minority class are lower than those on the majority class. However, the overall precision and recall achieved very high scores on both algorithms (all around 0.97), due to combining absolute quantity and relative quantity subsets into an entire imbalanced dataset. There is a slight difference in precision and recall between the two algorithms; XGBoost outperforms the RF in precision by 0.02, while decreases the RF's recall by 0.01. Thus, the precision, recall on the minority class, AP and PRC are fairer metrics when evaluating model performance.

Table 4.7: Prediction Results with Entire Dataset by XGBoost

Class	Precision	Recall	F1-score	Total
1	0.8	0.7	0.74	15273
0	0.98	0.99	0.99	280864
Avg/Total	0.98	0.97	0.97	296137

	Predicted Positive	Predicted Negative
Actual Positive	10677 (TP)	4596 (FN)
Actual Negative	2749 (FP)	278115 (TN)

Table 4.8: Prediction Results with Entire Dataset by Random Forest

Class	Precision	Recall	F1-score	Total
1	0.78	0.71	0.74	15304
0	0.98	0.99	0.99	280833
Avg/Total	0.97	0.97	0.97	296137

	Predicted Positive	Predicted Negative
Actual Positive	11178 (TP)	4287 (FN)
Actual Negative	3186 (FP)	277486 (TN)

Figure 4.10 shows the 5 PRC's and 5 AP's generated by XGBoost and RF respectively on the entire dataset. The PRC and AP of XGBoost and RF yield similar performance results.

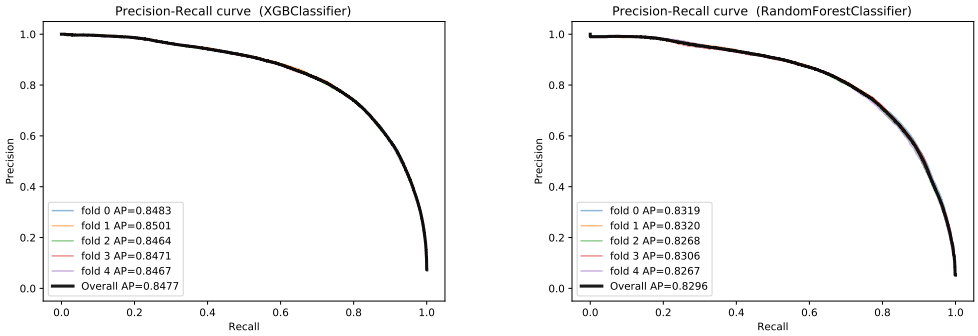


Figure 4.10: Precision-Recall Curves by XGBoost and Random Forest on Entire Dataset

4.7.3 Results and Analysis of Classification with Subset Dataset Categorized by Scientific Application Name

In most HPC systems, there are a huge number of jobs submitted by thousands of users who are potentially grouped into hundreds of user groups. In relevant research about job logs analysis, researchers usually divide logs into subsets with different rules or purposes for seeking hidden patterns from those logs [107][58][108] . In this research, our main purpose is predicting whether a job may or may not finish before its run time estimated by its user. The run time is mainly affected by many factors, such as user behaviors and computing resource usage in the HPC environment. (In this study, we do not consider human intervention from users or administrators, nor random hardware failures). The entire job dataset was split into subsets categorized by scientific application name for mining potential patterns which may affect run time of HPC applications. According to Table 2.1, there are almost one million job logs

based on 27 pre-installed HPC applications in TSUBAME 2.5 (except those in the unlabeled "others" class). Figure 4.11 shows the 27 applications and their areas.

Evaluating by AP

We used XGBoost and RF to build prediction models with the optimized hyper-parameters presented in Table 4.6 and run them through each subset by 5-fold Random permutation cross-validators respectively. The evaluation results of AP are plotted as a box plot in Figure 4.12.

Figure 4.12 shows the 5 AP's described in the boxplot for each of the 25 subsets, after taking "others" as a subset and removing "RISM", "CAE: MSC", from all training datasets. This was because there is no instance of run time-underestimation (labeled 1, minority class, underestimation) in their subsets. We sort them in alphabetical order. In general, we can see that there are little differences in mean performance between the two classifiers on the same subset. In some subsets, the mean AP of XGBooot is higher than the mean of RF, whereas some are not (essentially, a 50/50 split). On subsets "MD: AMBER", "MD: GROMACS", "MD: NAMD", "MD: Tinker", both XGBoost and RF models achieve very high and stable AP's. Although these 2 classifiers also have very high AP's and the mean value of th AP on "CAE: Abaqus" and "Vis: POV-RAY", some AP's are not stable enough. Even so, the AP in the these subsets are greater than or equal to 0.9, so we can still say that our models achieve very good overall prediction performance about job run time underestimation on the aforementioned applications, that is "CAE: Abaqus", "MD: AMBER", "MD: GROMACS", "MD: NAMD", "MD: Tinker" and "Vis: POV-RAY", in an HPC environment.

On the other hand, regardless of using the XGBoost or RF models, the boxplot shows that the difference among the 5 AP's is significant on "Bio: BLAST", "CAE: CST MW-Studio", "CAE: Fluent", "CAE: LS-DYNA", "MD: CHARMM" and "MD:

Applications	Description
Ab-Initio: PHASE	Macromolecules Calculation and Simulation
Bio: BLAST	In bioinformatics, BLAST for Basic Local Alignment Search Tool is an algorithm for comparing primary biological sequence information, such as the amino-acid sequences of proteins or the nucleotides of DNA sequences.
Bio: MEGADOCK	a protein-protein interaction prediction software for heterogeneous supercomputing environments
CAE: Abaqus	It is a software application used for both the modeling and analysis of mechanical components and assemblies (pre-processing) and visualizing the finite element analysis result. A subset of Abaqus/CAE including only the post-processing module can be launched independently in the Abaqus/Viewer product.
CAE: CST MW-Studio	It is a specialist tool for the 3D EM simulation of high frequency components.
CAE: Fluent	It is the most-powerful computational fluid dynamics (CFD) tool available
CAE: LS-DYNA	A general-purpose finite element program capable of simulating complex real world problems.
CAE: MSC Marc	A powerful, general-purpose, nonlinear finite element analysis solution to accurately simulate the product behavior under static, dynamic and multi-physics loading scenarios.
CFD: OpenFOAM	A C++ toolbox for the development of customized numerical solvers, and pre-/post-processing utilities for the solution of continuum mechanics problems, including computational fluid dynamics (CFD).
MATLAB	An environment tuned for iterative analysis and design processes with a programming language that expresses matrix and array mathematics directly.
MD: AMBER	Amber is a suite of biomolecular simulation programs.
MD: CHARMM	CHARMM is a set of force fields for molecular dynamics, and the name for the molecular dynamics simulation and analysis computer software package.
MD: Desmond	A software package developed at D. E. Shaw Research to perform high-speed molecular dynamics simulations
MD: GROMACS	A molecular dynamics (MD)package mainly designed for simulations of proteins, lipids and nucleic acids.
MD: NAMD	Nanoscale Molecular Dynamics (NAMD, formerly Not Another Molecular Dynamics Program) is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems.
MD: Tinker	A computer software application for molecular dynamics simulation with a complete and general package for molecular mechanics and molecular dynamics, with some special features for biopolymers.
MD: lammps	An acronym for Large-scale Atomic/Molecular Massively Parallel Simulator.
MPI	A standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures.
Others	
Python	Python is an interpreted high-level programming language for general-purpose programming.
QM: Gaussian	A computational chemistry software package to speed up molecular electronic structure calculations
QM: OpenMX	A program for extended structural equation modeling.
QM: Quantum Espresso	A quantum mechanical software package for nanoscale modeling of materials
QM: VASP	A computer program for atomic scale materials modelling, e.g. electronic structure calculations and quantum-mechanical molecular dynamics.
RISM	The RISM module allows the user to calculate the site-site radial distribution functions g(r) and pair correlation functions c(r) for a multi-component molecular liquid.
Vis: POV-RAY	An open source ray tracing tool capable of making high quality images of 3-dimensional scenes.
WRF	The Weather Research and Forecasting (WRF) Model is an numerical weather prediction system designed for both atmospheric research and operational forecasting applications.

Figure 4.11: HPC Applications and Their Areas in TSUBAME 2.5

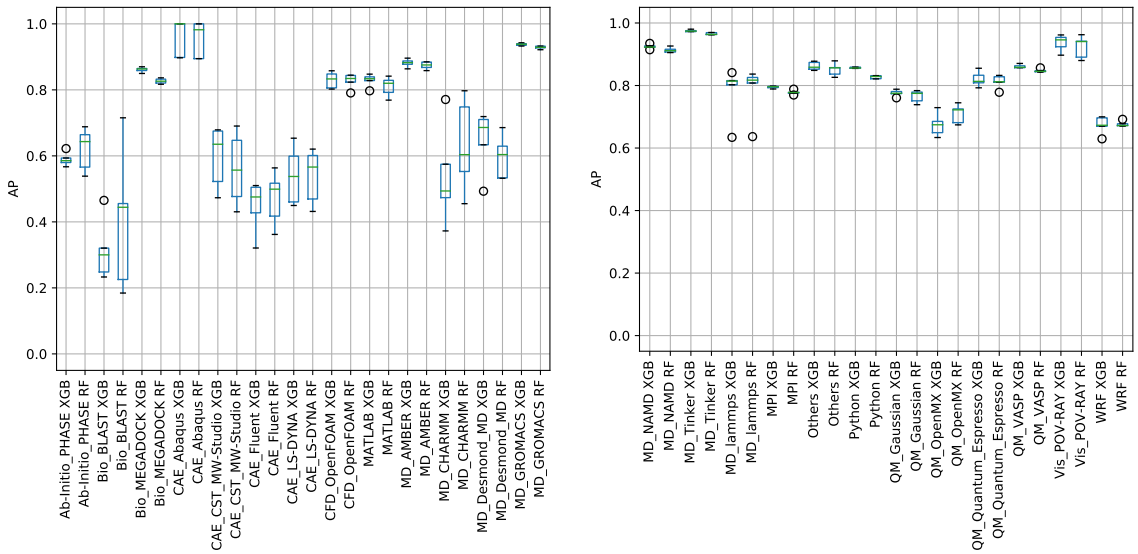


Figure 4.12: The Box-whisker-Plot of AP after Running Through Subsets 5 times with ShuffleSplit

Desmond MD". We believe that one of the reasons in making scattered distribution of these 5 AP's is due to imbalanced subsets. According to the percentage of the minority class of each application (as was also shown in Table 2.1), we can see that the absolute number and the relative percentage of the minority class are much lower than those of other subsets. In particular, in application "Bio:BLAST", the absolute number (15) and the relative percentage (0.45%) of the minority class are much lower than those on other subsets. As we mentioned in the section on data splitting, we split every dataset into a training validation set (70%) and a test set (30%). The test set can only keep 30% of 15 instances of the minority class, in the case of "Bio:BLAST", it may be just 5 minority class instances which is too small to achieve stable results. We can find a similar situation in "CAE: CST MW-Studio", "CAE: Fluent", "CAE: LS-DYNA", "MD: CHARMM" and "MD: Desmond MD", where the absolute number of job instances of the minority class are 83, 33, 24, 99 respectively. Obviously, the distributions of 5 AP's on each of these subsets are scattered as well. We can see that the "CAE: Abaqus" even yields a perfect AP of 1 once, however, the distributions

of 5 AP's are still slightly scattered due to its absolute number (27) of the minority class. It is called a small disjuncts problem [109] [110].

For the rest of the subsets, their absolute number of the minority class, at least, are more than 100. The distributions of the 5 AP's are more convergent than those that have a small disjuncts problem, which means that models are more stable. For most cases in this study, the percentage of the minority class almost has no impact on the accuracy performance. However, we found that, the absolute number of the minority class affects the stability of the model.

Evaluating by PRC

After evaluating the overall performance with the single value metric AP, we need to evaluate the model's accuracy. As we mentioned previously, an ideal model with high precision and high recall will yield many results, with all results labeled correctly. However, in the real world, this ideal model is almost non-existent. In general, a curve above another curve has a better performance level. In other words, the closer a PRC is to the upper right corner, the better the model is. In order to evaluate evaluating performance (precision and recall) while stability, we draw the 5 PRC's and the mean curve of these them generated by XGBoost and RF on each subset by 5-fold Random permutation cross-validator respectively.

As Figure 4.13 4.14 4.15 show, our models yields very good both of precision and recall on "MD: Tinker", "MD: NAMD" and "MD: GROMACS". Specifically, PRC's are very close to the upper right corner (1, 1), precision and recall are around 90% or higher than 90% at the same time, which means that our models are returning accurate job prediction on both of having or not having run time underestimation, as well as returning a majority of all run time underestimated jobs. Especially, "MD: Tinker" by XGBoost achieves the best precision, recall and AP with 90%, 93% and 0.965 respectively. Simultaneously, we also find that PRC's are not often zigzag in

shape and almost have no going up and down on these applications, which means that our models not only achieve good precision and recall, but perform stably as well. These models are able to seek potential patterns which may affect run time of HPC applications.

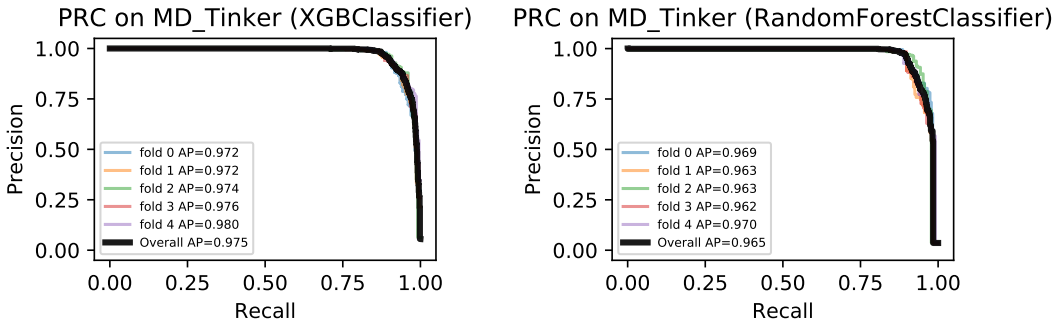


Figure 4.13: Precision-Recall Curves on "MD: Tinker" by XGBoost (left) and Random Forest (right)

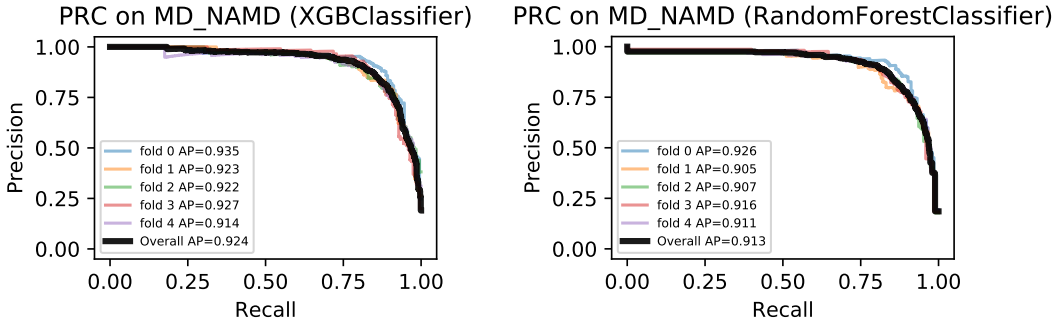


Figure 4.14: Precision-Recall Curves on "MD: NAMD" by XGBoost (left) and Random Forest (right)

Figure 4.16, 4.17 and 4.18 present PRCs on "CAE: Abaqus", "Vis: POV-RAY" and "MD: AMBER". We can see that our models also yield good precision and recall (both of precision and recall are around 80%∼90%), although there is a little zigzag behavior that frequently goes up and down. We think that it is due to the small disjuncts. This is because the absolute numbers on the minority class are 24 and 134 respectively on "CAE: Abaqus" and "Vis: POV-RAY". These models are able to

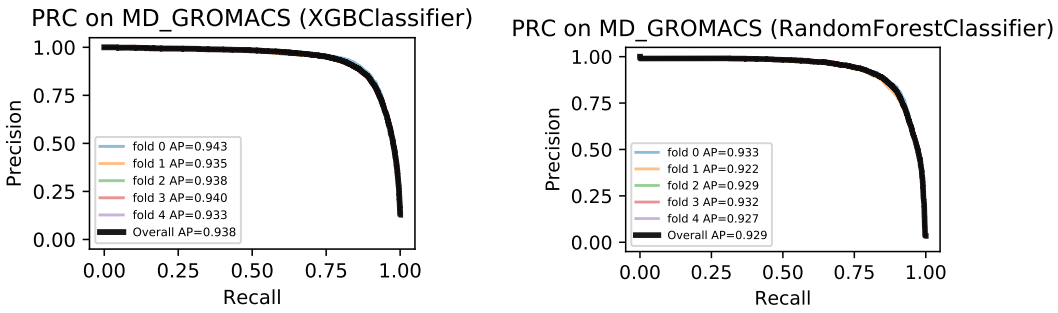


Figure 4.15: Precision-Recall Curves on "MD: GROMACS" by XGBoost (left) and Random Forest (right)

seek potential patterns which may affect run time of HPC applications. However, the stability of these models needs to be further enhanced.

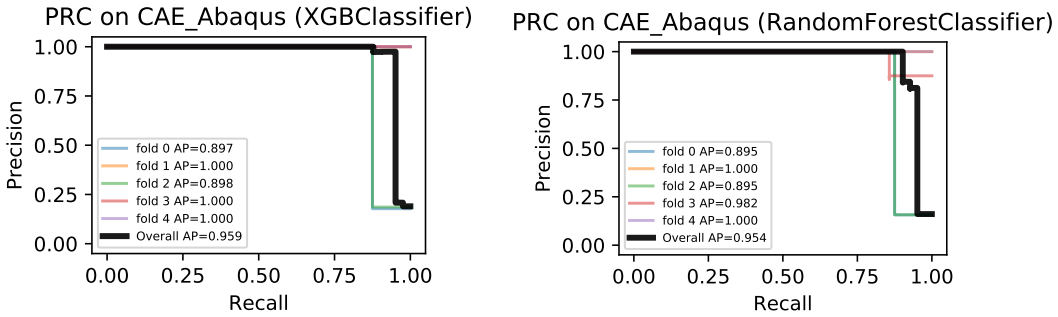


Figure 4.16: Precision-Recall Curves on "CAE: Abaqus" by XGBoost (left) and Random Forest (right)

For those models with moderate performance including "Bio: MEGADOCK", "CFD: OpenFOAM", "MATLAB", "MPI", "Python", "QM: Gaussian", "QM: Quantum Espresso", "QM: VASP" and "WRF", although they are still imbalanced datasets with low relative percentage on the minority class, their absolute number (at least 295 instances on the minority class) are much higher than those thay have small disjuncts. From Figure 4.19 4.20 4.21 4.18 4.22 4.23 4.24 4.25 4.26 4.27, we can see that their performance is moderate but stable. Although these models are able to seek potential patterns which may affect run time of HPC applications and are stable, their performance accuracy (both of precision and recall) needs to be further enhanced as well.

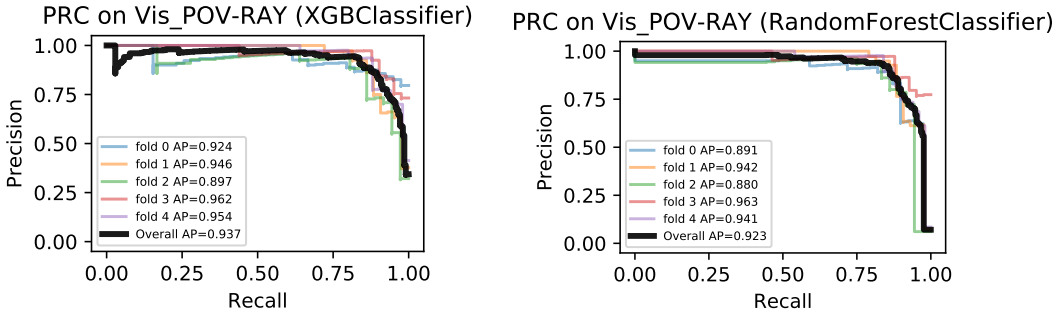


Figure 4.17: Precision-Recall Curves on "Vis: POV-RAY" by XGBoost (left) and Random Forest (right)

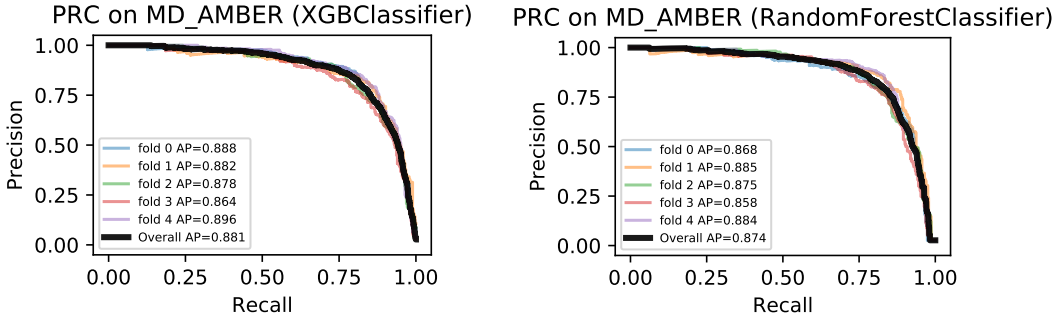


Figure 4.18: Precision-Recall Curves on "MD: AMBER" by XGBoost (left) and Random Forest (right)

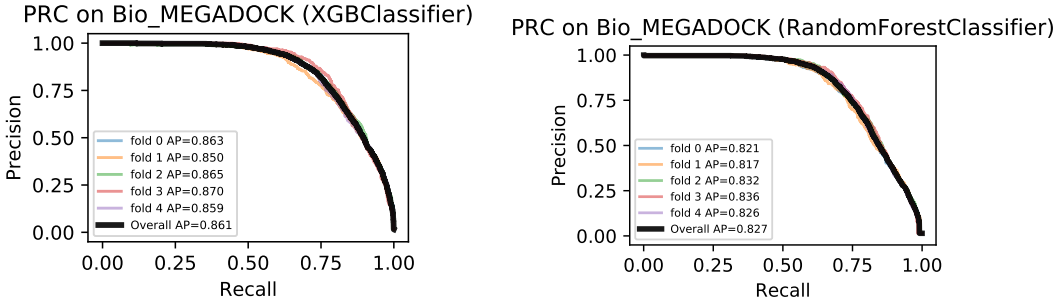


Figure 4.19: Precision-Recall Curves on "Bio: MEGADOCK" by XGBoost (left) and Random Forest (right)

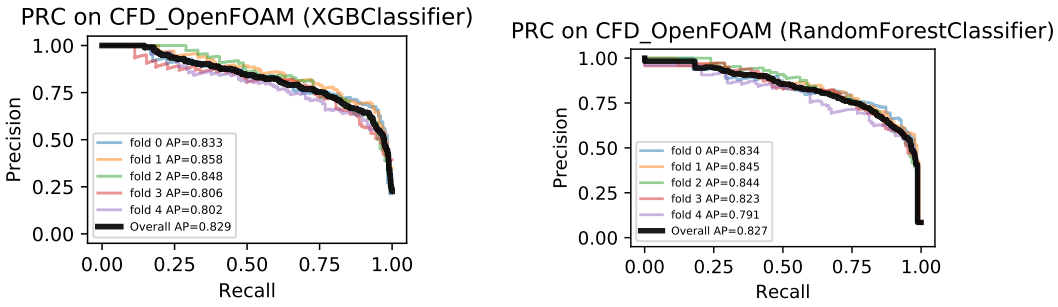


Figure 4.20: Precision-Recall Curves on "CFD: OpenFOAM" by XGBoost (left) and Random Forest (right)

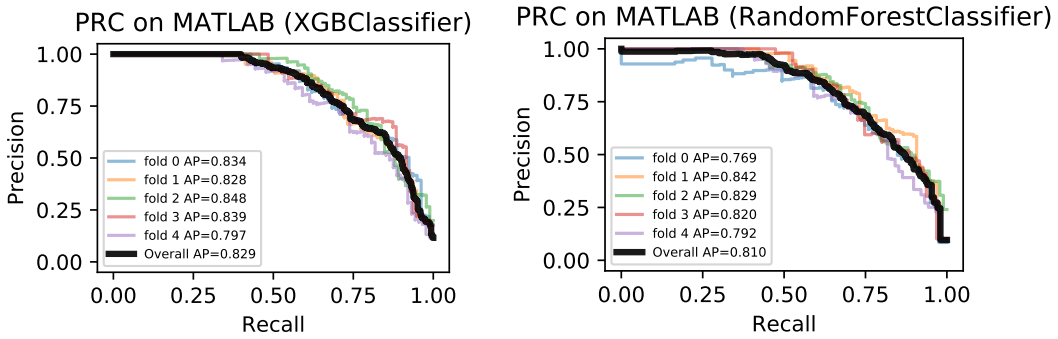


Figure 4.21: Precision-Recall Curves on "MATLAB" by XGBoost (left) and Random Forest (right)

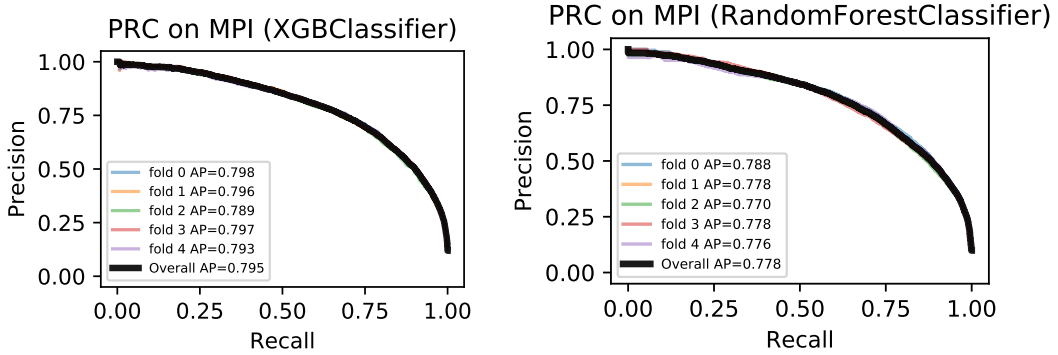


Figure 4.22: Precision-Recall Curves on "MPI" by XGBoost (left) and Random Forest (right)

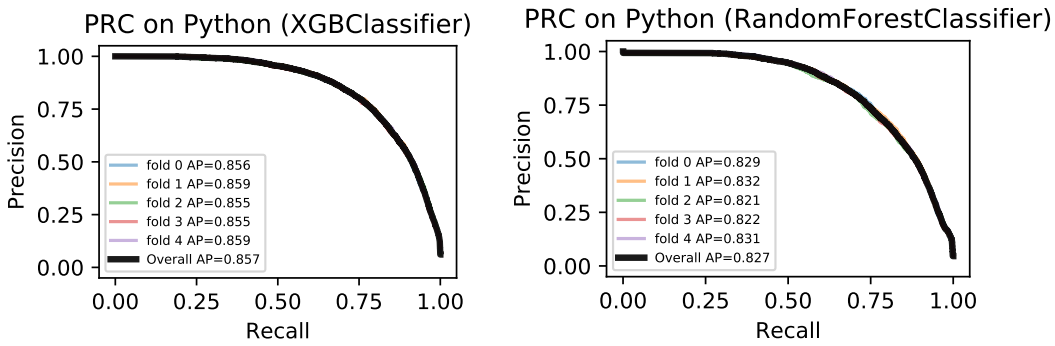


Figure 4.23: Precision-Recall Curves on "Python" by XGBoost (left) and Random Forest (right)

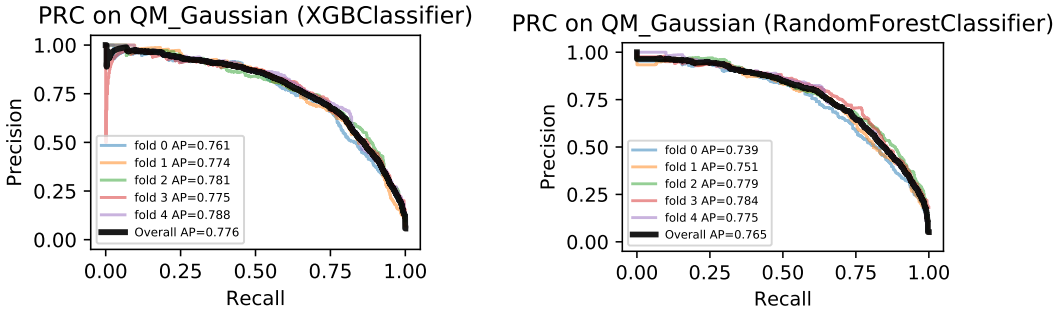


Figure 4.24: Precision-Recall Curves on "QM: Gaussian" by XGBoost (left) and Random Forest (right)

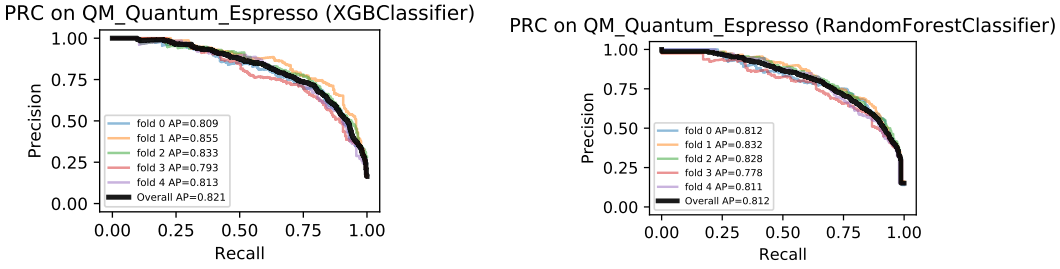


Figure 4.25: Precision-Recall Curves on "QM: Quantum Espresso" by XGBoost (left) and Random Forest (right)

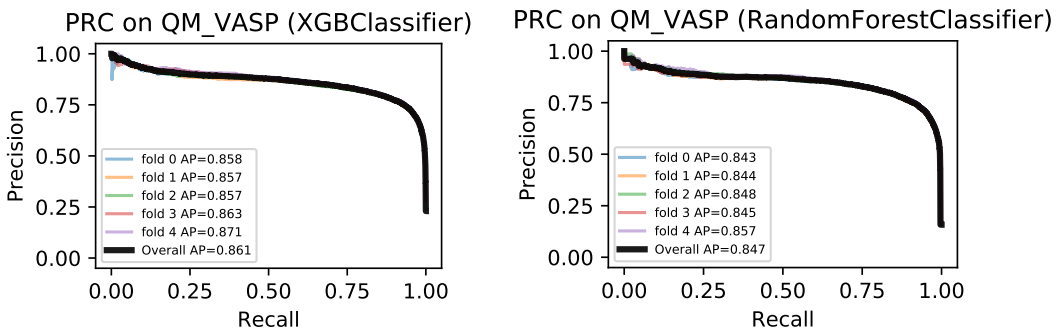


Figure 4.26: Precision-Recall Curves on "QM: VASP" by XGBoost (left) and Random Forest (right)

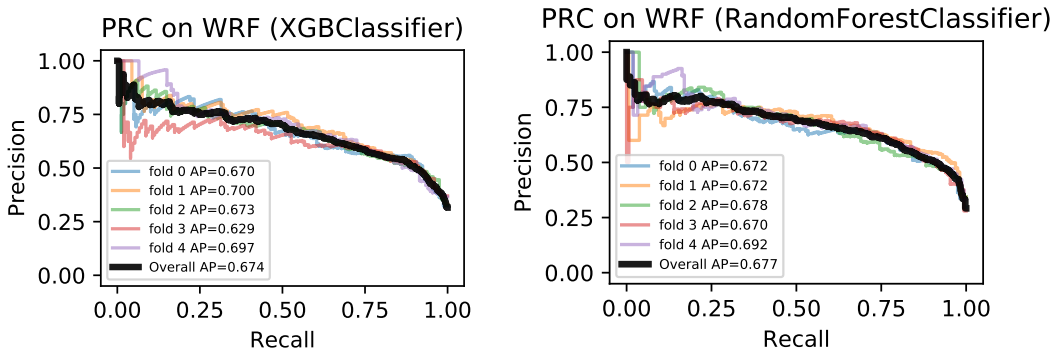


Figure 4.27: Precision-Recall Curves on "WRF" by XGBoost (left) and Random Forest (right)

For the rest of application models, their PRCs have been showed in figure 4.28 4.29 4.30 4.31 4.32 4.33.

We can clearly see that PRC's are often zigzag in shape on "Bio: BLAST", "CAE: CST MW-Studio", "CAE: Fluent", "CAE: LS-DYNA", "MD: CHARMM" and "MD: Desmond MD", as these PRC's are far away from the top right corner (1, 1). In other words, these models can neither guarantee performance stability nor output acceptable precision or recall. As we explained previously, the instability is largely due to small disjuncts. However, the poor precision and recall prove that these models cannot seek potential patterns, which may affect run time of these application. Hence these models need to be modified.

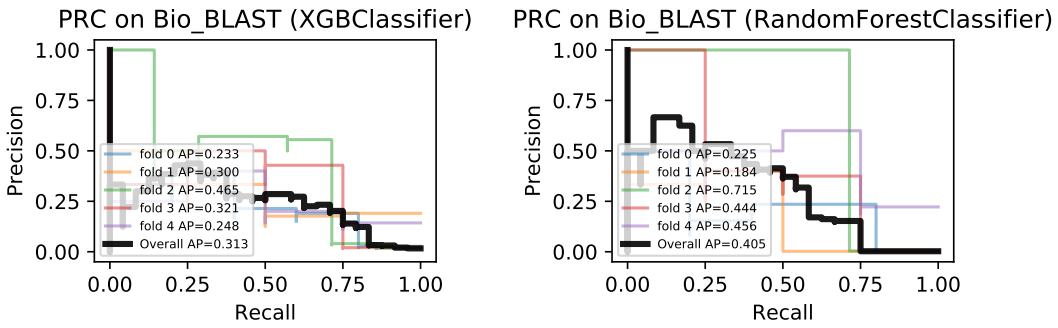


Figure 4.28: Precision-Recall Curves on "Bio: BLAST" by XGBoost (left) and Random Forest (right)

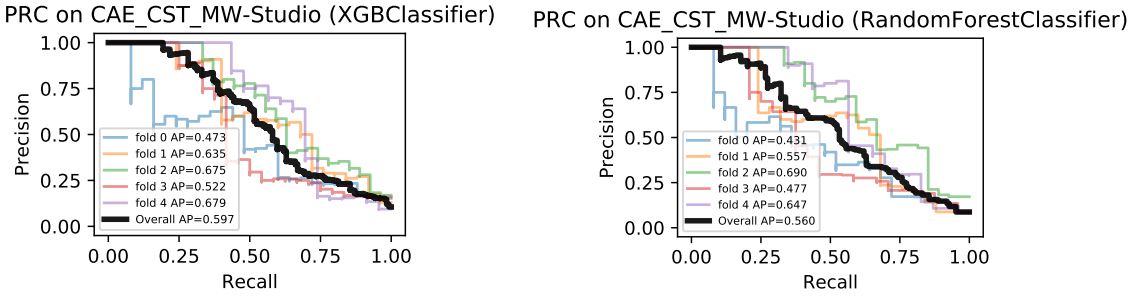


Figure 4.29: Precision-Recall Curves on "CAE: CST MW-Studio" by XGBoost (left) and Random Forest (right)

Evaluating by Plotting Decision Threshold with Precision and Recall

After ranking models with AP, we plot precision and recall scores as a function of the threshold to help select the threshold value that gets the best precision-recall trade-off for different models. Given the default settings of XGBoost and RF model, the default

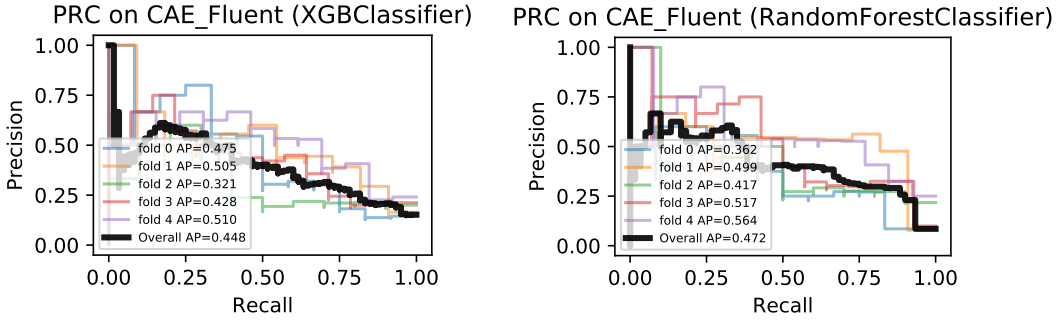


Figure 4.30: Precision-Recall Curves on "CAE: Fluent" by XGBoost (left) and Random Forest (right)

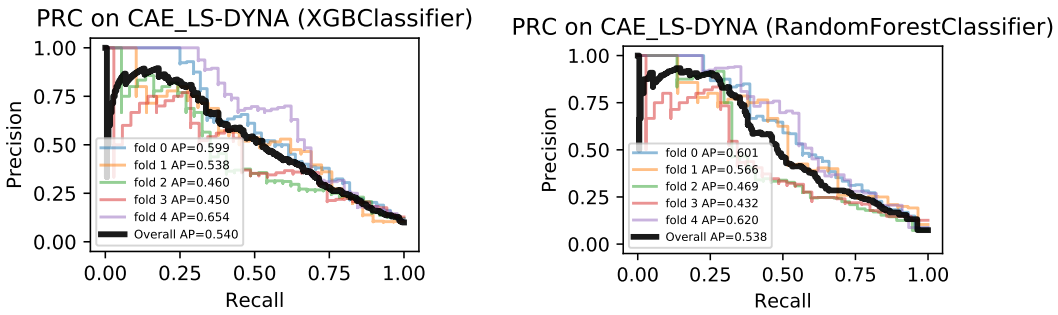


Figure 4.31: Precision-Recall Curves on "CAE: LS-DYNA" by XGBoost (left) and Random Forest (right)

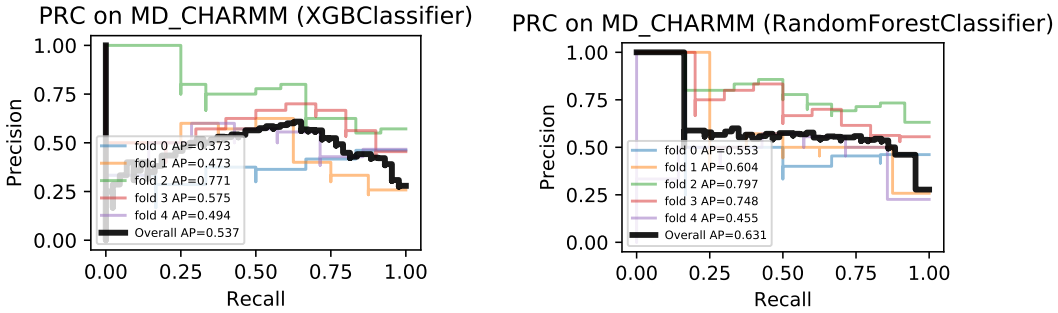


Figure 4.32: Precision-Recall Curves on "MD: CHARMM" by XGBoost (left) and Random Forest (right)

value of the threshold is 0.5. The default output of every model, particularly precision, recall and derivatives, including PRC and AP, are calculated based on this default threshold (0.5). However, in general, HPC users may need higher precision than recall, because they just want to make sure that jobs with run time-underestimation prediction are indeed run time underestimated jobs that need to be killed early.

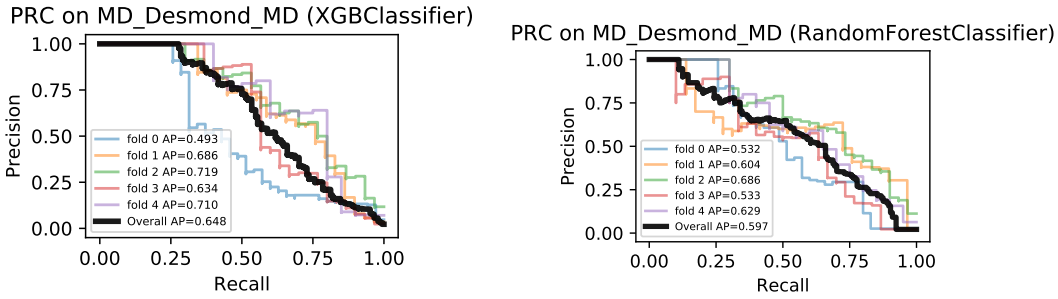


Figure 4.33: Precision-Recall Curves on "MD: Desmond MD" by XGBoost (left) and Random Forest (right)

On the contrary, an HPC system administrator just wants to save more computing resources and urge users to kill more suspicious run time underestimated jobs as early as possible. Thus, a higher recall would help them to detect more of these types of jobs. Plotting precision and recall scores as a function of threshold helps to select the threshold value that gets the best precision-recall trade-off for their respective purposes.

Taking models "Vis: POV-RAY" and "QM: VASP" as examples, we draw their precision and recall as a function of threshold and show how it works in Figure 4.34 4.35. The rest of them can be found at Appendix A. In every graph of Figures 4.34 and 4.35, we can see that precision and recall are drawn with all possible thresholds based on those scores. As we know, the default value of the threshold for binary classification is set to 0.5 in most Machine Learning algorithms, including XGBoost and Random Forest. Meanwhile, we also know that different roles have different requirements for precision and recall. For example, in Figure 4.34, regardless of the models XGBoost or RF, we can see that precision is higher than recall when the threshold is 0.5. If a user needs a higher recall, we just set the threshold as less than or equal to 0.3, then the recall will be higher than precision, which corresponds to a slight decline in precision. Therefore, according to different purposes, the threshold needs to be adjusted to different operating points [111]. There are two common methods to set the operating point. First, Equal Error Rate (EER) point: the point where precision equals recall [112]. This feels to some users like a "natural" operating point. In addition, in some problems, users have a natural minimal tolerance of either precision or recall. For example, an HPC user requires that 80% of job run time underestimation prediction is correct, otherwise the users will stop using our models. Hence, it is natural to set precision to 80% (or slightly less) and accept whatever recall results at that point.

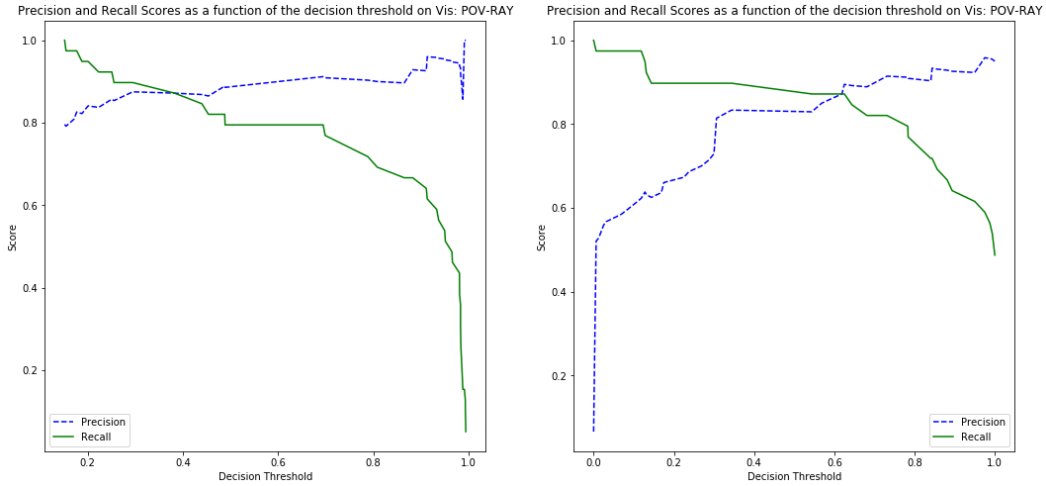


Figure 4.34: Precision and Recall Scores as Function of Threshold on "Vis: POV-RAY" by XGBoost (left) and Random Forest (right)

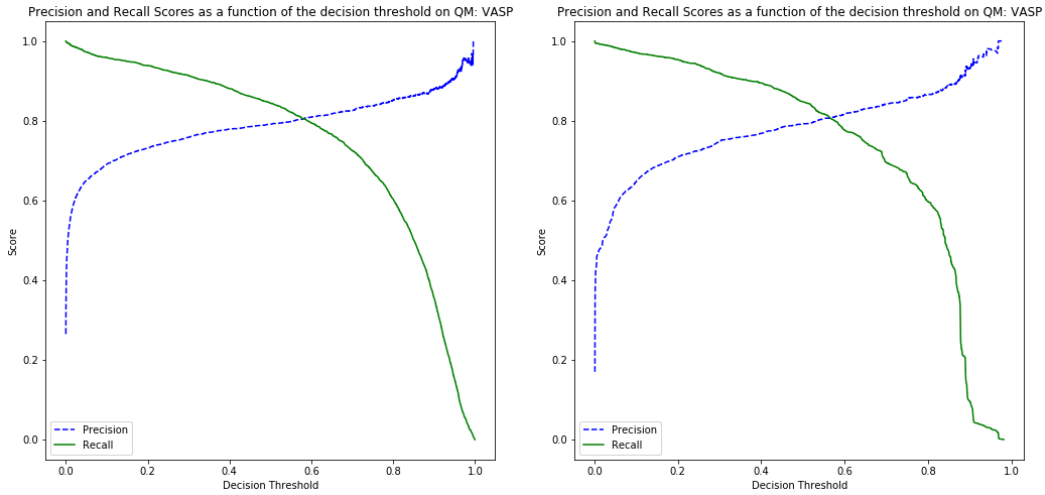
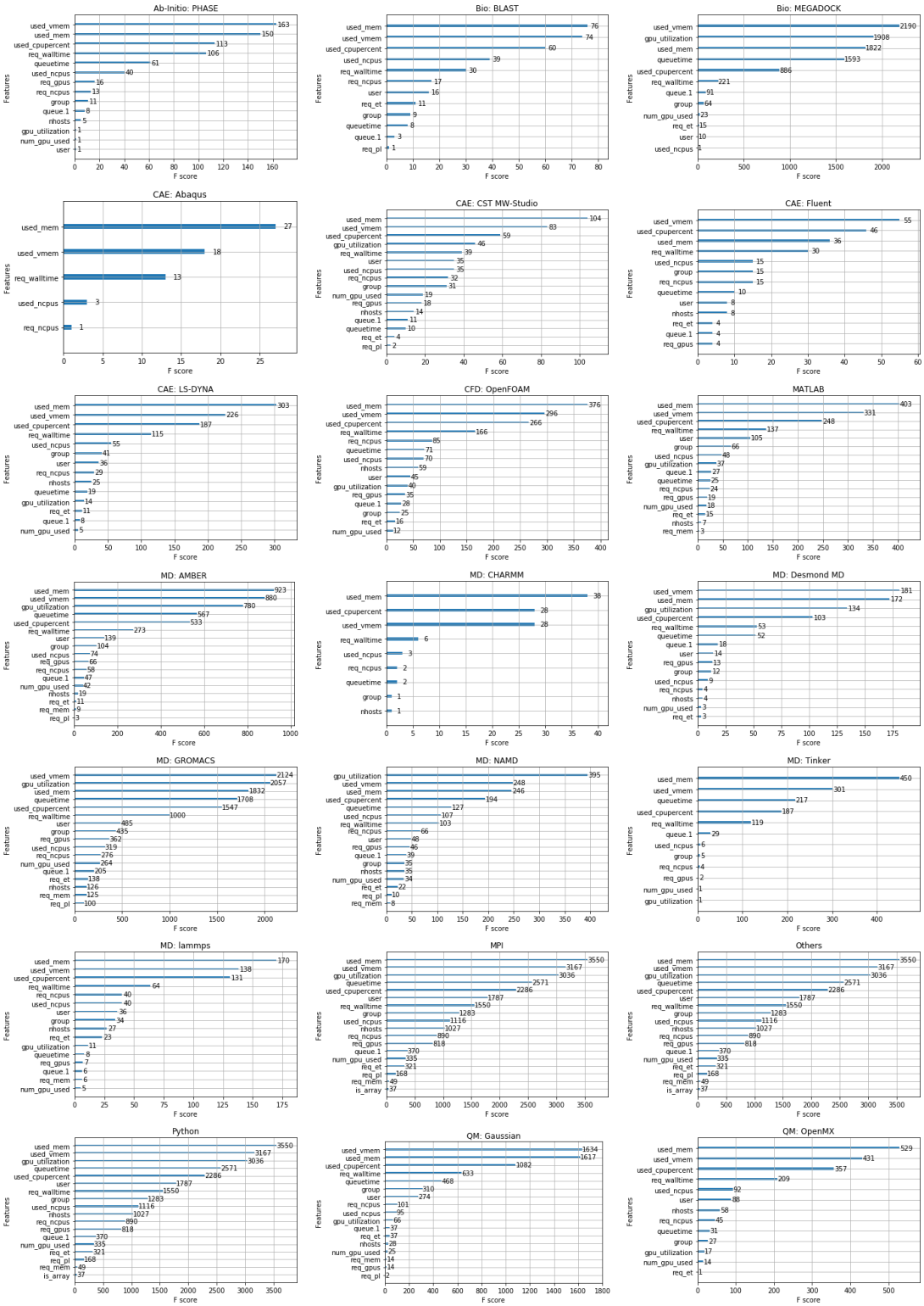


Figure 4.35: Precision and Recall Scores as Function of Threshold on "QM: VASP" by XGBoost (left) and Random Forest (right)

4.7.4 Feature Importance

Feature importance gives a score for indicating how valuable or useful each feature was when building boosted decision trees based models. This score can be calculated by accounting how many times they appear when making its decision trees. In other words, the more times a feature is used for making key decisions during decision trees building, the higher its relative importance is to the model. The score of feature importance will be averaged over all of the decision trees within

the model. The feature importance can be conveniently calculated and ploted by use of built-in function in XGBoost library.



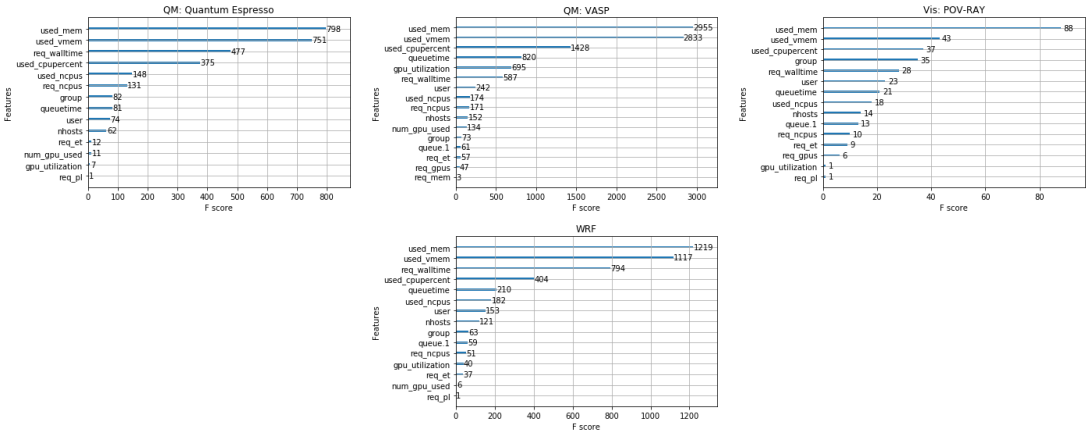


Figure 4.35: Important Features for Different Applications

We plotted feature importance in all subsets with the features ordered according to how high the score was (ie. how important it was) in Figure 4.35. We can see that ***used_mem***, ***used_vmem***, ***used_cpupercent***, and ***gpu_utilization*** are the most important objective features in those applications. In other hands, the req_walltime contains some important information which is related to the running time of applications, since when users change their configuration of applicants, experienced users will modify this option at the same time. However, applications have different weights (namely prediction of run time-underestimation) on different features (namely computing resource usage), which both affected job run time. Our method recognized these patterns and used them to predict job run time-underestimation in HPC systems.

4.7.5 Discussion

We have some speculations as to why a user who has submitted the same job multiple times and already knows the characteristics of the job might still underestimate the job run time, in which case our Machine Learning-based prediction can further improve the job run time estimation accuracy:

The first could be I/O interference. There are lots of users who often submit small jobs that generate individual I/O in HPC systems. A large number of I/O-based small jobs may lead to I/O congestion on a specific node which will significantly affect job running time. Since the user submitting the job is not aware of the I/O congestion, he could potentially underestimate the job run time.

Second, particularly in TSUBAME 2.0, there are some virtual machine (VM) based job co-location. For example, if users submit a lot of single-core VM-based jobs, TSUBAME will co-locate those jobs which may lead to computing resource (e.g. CPU cores and memory bandwidth) congestion, further preventing the user from accurately predicting his job run time.

We can consider above two cases as the noise in an HPC system, which may cause the job run time-underestimation in HPC systems. Even though experienced HPC users might be able to estimate the required wall time of jobs accurately, they cannot predict these system noises which significantly affect job running time. Because of that, objective features (CPU usage, Memory usage etc.) are important, and we cannot fully trust users' estimation. This is why we need to use Machine Learning to build prediction models for helping users to correct their job run time.

We have another speculation that might affect the users prediction in special cases. For example, users may deliberately extend their job's required wall time for ensuring to get their results in case they need to meet a paper deadline. Similarly, when users submit "big" jobs which have long run time (e.g. longer than 12 hours), they might again deliberately overestimate their required wall time to avoid their job running out of time and being killed by the HPC system and avoid the loss of time and money that will result from it. Therefore, users behavior will also affect job run time-underestimation.

Even though it is difficult to concretely show the effect of the above-mentioned points in the users ability in prediction job run time, looking at feature importance

in Figure 4.35, we can see that system characteristics represented by objective features are dominant reasons for job run time-underestimation in HPC systems. Meanwhile, subjective user behavior-based features also affect the prediction results to some extent, but with lower importance compared to the objective ones. Hence, our results show that we cannot solely rely on the users prediction and we also need to take the characteristics of the HPC system into account for accurate prediction of job run time.

4.8 A Preliminary Simulation for Estimating the Benefit and Economy of Prediction Model

Although we have already evaluated the accuracy of our prediction model, we also need to further evaluate its economy and benefits. As we know, predicting ongoing jobs needs models that are trained with data collected when jobs are in-progress. However, that data we use for training our model is generated after a given job finishes, so we need a solution that can provide in-progress-like data. To solve this, we design a preliminary simulation test to simulate in-progress job logs based on existing data. Then, we purpose a metric to approximately evaluate the economy and benefit of our prediction model by use of the simulation test.

4.8.1 A Simulation Test

As we mentioned, we need a simulation test to simulate in-progress job logs based on existing data and to evaluate the economy and benefit of our model. The simulation is based on the assumption that all run time-underestimated jobs produce a negative impact on the overall productivity of the HPC system at hand. In addition, the earlier those run time-underestimated jobs can be predicted and handled, the more time and

system resources can be saved and the overall productivity of the HPC system will also be improved.

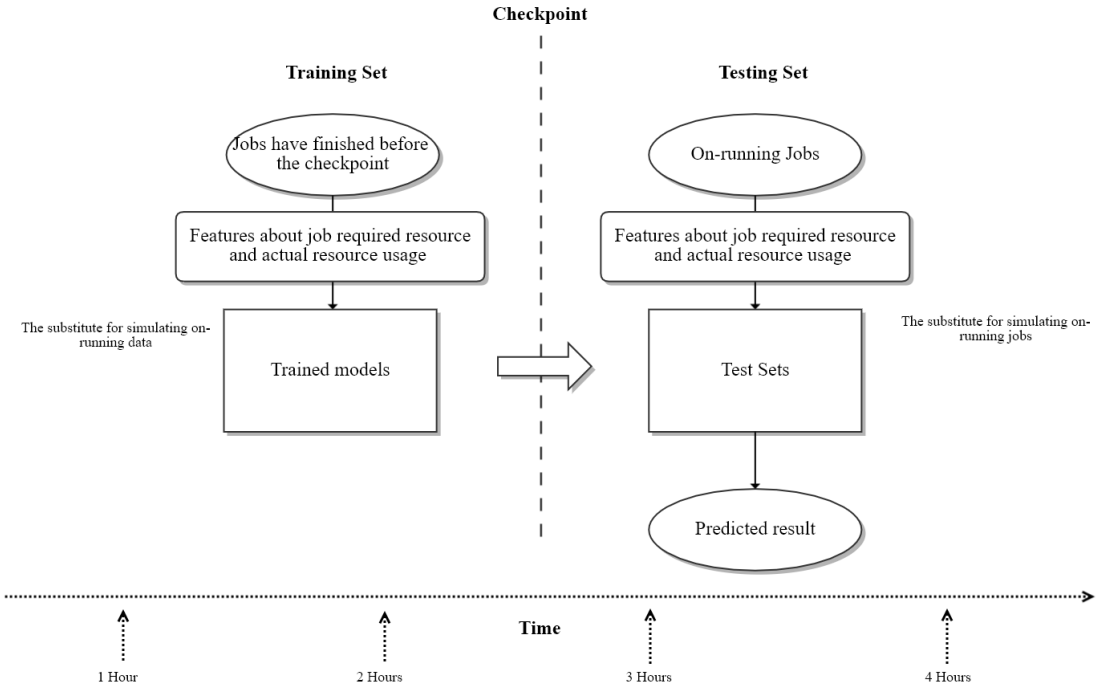


Figure 4.36: The Conception of the Simulation

For the simulation, a checkpoint needs to be set as a time boundary for splitting training test and test set in the simulation. Specifically, since we have no real in-progress data collected at 3 hours after jobs have started, we use the data which finished in 3 hours as the substitute for simulating in-progress job data collected at 3 hours after jobs have started. Then, we use the simulated in-progress data to train our model and test it with test data, which are on-running jobs at 3 hours after jobs have started. The conception of simulation is shown in Figure 4.36.

For example, on application "MD: NAMD", suppose we want to evaluate the number of run time underestimated jobs that can be predicted at 3 hours after jobs have started. We set 3 hours as the checkpoint for those jobs which finish in 3 hours and designate them as training sets, while those jobs which are still running are treated as test sets. Then we build a prediction model with the training set, and test

the model with the test set. This is a simulation for the case that we are doing online prediction at 3 hours after jobs have started.

As is expected, every tested job instance will get one result from TP, TN, FP and FN, which follow the rules defined in chapter 4.6. What is more, the economy and benefit of using the prediction model can be estimated by using the results of the simulation. We propose a metric named "Saved-lost Rate" (SLR) to evaluate benefits of the prediction model with different checkpoints. SLR is defined as the following formula:

$$\boldsymbol{SLR_C = \frac{Saved}{lost + Punishment} = \frac{\sum_{tp=1}^{TP} (j.used\_walltime - C)_{tp}}{\sum_{p=1}^{P} j.used\_walltime_p + \sum_{fp=1}^{FP} C_{fp}}}$$

In this formula, "C" is the checkpoint which theoretically can be any time after jobs started. However, in this simulation, we must fix the checkpoints to a few special time points, such as 1 hour, 3 hours, 6 hours or even 12 hours after the job started. The reason for this is that we should keep enough positive job instances (run time-underestimated job instances) and negative job instances in the training set, essentially for training the imbalanced binary classification model. We treat those job instances which have finished by the checkpoint time as training sets, and those job instances which take more time than the checkpoint time as test sets. Moreover, in most subsets, most of the run time underestimated jobs take more than 1 hour. If we set the checkpoint too early (say, 10 or 30 minutes after a job starts), we cannot get enough positive instances (at least 20 run time-underestimated job instances) for training models.

For those positive (all run time-underestimated jobs) instances, we consider their used_walltime as "lost". The purpose of using the prediction model is to predict those positive instances before they are automatically killed by the HPC system.

For those TN instances, we do not consider them in this formula because they are finished by the users estimated time, and the prediction model also predicts result correctly. Otherwise, we need to consider the three remaining situations (namely TP, FP and FN) and use them to calculate the benefits of the prediction model.

For those TP instances, we calculate the "Saved" of a TP job instance by subtracting the checkpoint time from the job's used_walltime. For example, there is a run time-underestimated job, it will keep running until meeting user req_walltime (assuming it is 12 hours), then it will be killed by the HPC system. We set the checkpoint as 3 hours after the job started, in this case, which means that we want to predict this job is a running time underestimated job or not after it has been running for 3 hours. Our prediction model predicts it as positive instance successfully, and we kill this job manually at this checkpoint, which can be considered that we save 9 hours ("Saved" is 9 hours). Similarly, if the checkpoint is set at 6 hours, the "Saved" will be 6 hours. The more and earlier we predict positive instance, the more "Saved" can be yielded.

For those FP instances, because the prediction model mistakenly treated them as positive instances and terminate them at the checkpoint time, which caused the time (until the checkpoint) spent on these jobs to be wasted. Therefore, we consider those wasted time as the "punishment" for evaluating the benefit of the prediction model. For instances, at checkpoint 3 hours after the job started, the prediction model predicts one job as a positive and terminate it for saving time and resource. However this job is an FP instance, and it would be able to finish by its req_walltime. The 3 hours spent on this job is considered as 3 hours "punishment" for evaluating the benefit of the prediction model.

The earlier the checkpoint is set, the higher the SLR should be if TP job can be predicted correctly. However, in contrast, the earlier the checkpoint is set, the less training data have, which will reduce the AP. (SLR will also be affected by the AP, with a certain degree of decline.) Therefore, we estimate that there will be a trade-off relationship between the checkpoint and the SLR in our simulation.

4.8.2 Result of the Simulation

In the last subsection, we design a simulation test to simulate in-progress job logs based on existing data with. For most of the subsets, most of the run time underestimated jobs cost more than 1 hour and less than 24 hours. We set the checkpoints at 1 hour, 3 hours, 6 hours, 12 hours and 24 hours after the job started. Besides, because of enough positive instances (20 run time underestimated job instances at least) are necessary for training models, the following application subsets cannot be tested in this simulation: "Bio: BLAST", "CAE: Abaqus", "CAE: Fluent", "MD: CHARMM", "MD: Desmond" and "MD: Tinker". For those applications, the distributions of positive instances are too concentrated on one certain period of time, or there are no enough positive instances available in those subsets, which lead to the predicting model cannot be trained for certain periods of time in the simulation.

As Figure 4.36 shown, we given the SLR, How many "Saved/Lost + Punishment" hours and Average Precision (AP) at different checkpoints. We keep blank blocks on this table due to lack of enough positive instances for training or testing at that checkpoint. As we expected, when the checkpoint is set to 1 hour, there are only 11 prediction models of HPC applications that can be trained. In other words, there are more than 20 positive instances in each application (their req_walltime are less than 1 hour, and those jobs did not complete by their req_walltime, which leads to automatic termination by the HPC system after meeting the req_walltime). When we

Table 4.9: Simulation Result

	Checkpoint (Hours)									
	1 Hour		*3 Hours*		*6 Hours*		*12 Hours*		*24 Hours*	
	SLR Saved/lost+P	*AP*	*SLR Saved/lost+P*	*AP*	*SLR Saved/lost+P*	*AP*	*SLR Saved/lost+P*	*AP*	*SLR Saved/lost+P*	*AP*
Ab-Initio: PHASE	0.051	0.418	0.105	0.426	0.02	0.442	0.05	0.649		
	46.29/907.77		92.94/882.93		18.24/886.68		42.99/857.87			
Bio: MEGADOCK	0.105	0.115	0.125	0.305	0.108	0.609				
	2636.48/25038		2233.76/17870.1		559.298/5166.71					
CAE: CST MW-Studio			0.015	0.408	0.005	0.271	0	0.387		
			10.96/687.32		3.23/601.3		0.0/544.16			
CAE: LS-DYNA			0.17	0.578	0.077	0.406	0.063	0.31		
			132.92/778.36		53.98/693.57		36.042/564.35			
CFD: OpenFOAM	0.013	0.435	0.021	0.539	0.069	0.602	0.131	0.662		
	43.77/3229.53		65.43/3108.21		585.68/2949.63		351.64/2683.87			
MATLAB	0.031	0.209	0.067	0.251	0.013	0.227				
	18.23/585.34		44.99/663.92		8.20/604.48					
MD: AMBER	0.0015	0.133	0.101	0.221	0.031	0.311	0.08	0.476	0.0005	0.822
	33.861/21414.5		2218.49/21925.1		656.63/21153.51		1730.49/21621.51		2.72/5308.06	
MD: GROMACS			0.106	0.325	0.011	0.428	0.011	0.449	0.0067	0.6
			11763.93/110002.1		1198.28/108059.8		1189.07/107085.7		718.5575/106270.6	
MD: NAMD			0.121	0.229	0.088	0.244	0.066	0.284	0.028	0.831
			792.79/6535.38		588.31/6682.53		454.45/6786.2		96.18/3336.7	
MD: lammps	0.054	0.228	0.05	0.278	0.144	0.551	0.139	0.684		
	23.03/425.21		18.97/373.32		44.17/305.225		25.55/183.66			
MPI	0.02	0.452	0.036	0.501	0.031	0.7	0.044	0.759		
	1742.88/86879.48		3072.77/83476.16		2468.84/79464.04		3188.04/72394.92			
Python	0.012	0.139	0.01	0.108	0.001	0.308	0.0003	0.394	3.56E-06	0.611
	943.71/76537.33		863.91/81109.89		124.28/62735.1		22.42/57618.27		0.16/47312.18	
QM: Gaussian	0.005	0.203	0.021	0.346	0.065	0.528	0.057	0.76		
	51.16/10156.12		195.63/9292.76		570.61/8739.78		458.46/8022.27			
QM: OpenMX			0.02	0.485	0.088	0.695				
			34.9/1745.79		102.11/1159.82					
QM: Quantum Espresso			0.058	0.311	0.067	0.594	0.052	0.746		
			634.92/10920.55		705.09/10472.34		510.72/9712.11			
QM: VASP			0.066	0.508	0.076	0.518	0.103	0.758		
			1009.86/15144.44		1130.54/14792.89		1412.49/13583.87			
Vis: POV-RAY	0.069	0.332	0.317	0.806	0.337	0.871	0.212	0.892		
	25.108/363.71		104.79/330.5		95.54/283.055		42.055/198.11			
WRF	0.014	0.493	0.05	0.577	0.035	0.708	0.022	0.791		
	73.19/5212.51		258.42/5132.26		180.79/5065.62		102/4494.12			

set the checkpoint to 3 hours after jobs have executed, we can see that all applications have achieved enough positive instances and all SLR are available. Comparing with those applications which have already achieved SLR when the the checkpoint was 1 hour, except for "Python", all SLRs have varying growth, which means that the benefit of performing our prediction 3 hours after jobs have executed is higher than performing it 1 hour after jobs have executed. This is despite taking 2 more hours for job execution. Additionally, APs are also higher than they were at checkpoint 1 hour.

Next, we respectively set the checkpoint to 6 hours, 12 hours and 24 hours after jobs have executed. The SLR in most of those applications are around 0.05-0.12, except for "Vis:POV-RAY", where its SLR reach 0.337. Taking "Vis:POV-RAY" as an example, the prediction model made a set of prediction results in this application after jobs have executed in 6 hour. If we terminate all those TP jobs at that checkpoint time ("Saved" time is calculated by the sum of TP job's req_walltime minus checkpoint time) and sum up all "Saved" time , 95.54 hours will be saved. The sum of all "lost" time (the sum of req_walltime of positive jobs) and all "punishment" time (calculated by the sum of the checkpoint times of FP jobs) at checkpoint 6 hour is 283.05 hours in "Vis:POV-RAY". The SLR (0.337) is calculated by "Saved" dividing the sum of the "lost" and the "Punishment", which can be considered as the approximate benefit of using the prediction model. For most of HPC systems, saving job run time means saving credits (money). Therefore, according to simulation result with the current data, we can say that there are 95.54 hours that can be saved from 283.05 hours, if we use our prediction model after jobs have executed in 6 hour in "Vis:POV-RAY". The higher the SLR, the higher the benefit of using our prediction model (the more credits can be saved).

In general, we find that the SLR of some applications are going up, while some are going down. Some of them achieve their best SLR when the checkpoint is set to 3 hours; some others yield the best SLR when the checkpoint is set to 6 hours or 12 hours (in fact, the best SLRs are in the gray background shown in Table 4.9). For those best SLRs at different checkpoints, I believe it is due to imbalanced data distribution of positive instances and negative instances. Apart from the SLR, the trend of AP is increasing with increasing checkpoint time. This trend is not difficult to explain, since the number of job instances in the training sets are increasing, and number of job instances in the test sets are decreasing. What is more, when the checkpoint is at 24 hours, we can see that some APs can reach up to 0.8 or higher.

However, the SLRs have significantly dropped compared with the same applications at earlier checkpoint times, due to the fact most run time-underestimated jobs have completed by 24 hours in those applications. In other words, even if those on-going jobs can be predicted with high APs 24 hours after jobs have executed, the actual benefit at that time is not large. Obviously, based on the result of simulation, we can also see that the SLR in some applications are very low, less than 0.05 no matter where the checkpoint is (like Python, MPI, CAE:CST MW-Studio), I think the reason is that there is a strong relationship between resource utilization and used_walltime in those applications, since the simulation split job data into training set and test set with a checkpoint, so the prediction model trained with job data which finished in the checkpoint cannot accurately predict those test jobs which are on-running at the checkpoint in this simulation.

According to the results of our preliminary simulation, we conclude that our prediction models are able to predict run time-underestimated jobs with different accuracy at different checkpoints times. If we sum up the "Saved" time of all the applications at checkpoints where they get their best SLRs, we can estimated that our prediction model would save 24962 hours totally based on the result of our preliminary simulation. This would help HPC users for saving time loss (and financial loss) to a certain extent as long as they take appropriate action, such as shutting down those suspicious run time-underestimated jobs in advance. In addition, another benefit of using this prediction model is that it helps HPC system administrators free up computing resources as much as possible when job run time-underestimation occurs. Altogether, our machine learning-based prediction model contributes to improving the overall productivity and efficiency of HPC systems.

We have to state that this simulation with existing data can only approximately reflect the estimated benefits of the prediction models, because we use the job data

which finished in 3 (for example) hours as the substitute for simulating in-progress job data collected at 3 hours after jobs have started, which cannot completely represent the real in-progress job data. In the future work, for real online prediction in the future, we need to collect real in-progress job data at different checkpoint times and add those real in-progress data into corresponding training sets to re-train or transfer learning our existing models. Moreover, the metric–SLR, which we proposed for evaluating the benefit and economy of simulation can be used directly for evaluating the benefit and economy of online prediction in the future.

4.9 Summary

In the previous chapter, we evaluated HPC application subsets with our models by different metrics including AP and PRC. According to the evaluation results, we conclude our work as follows:

Our study starts by observing and collecting basic information and task resource utilization of HPC tasks as job logs on TSUBAME 2.5, using PBS and Ganglia. Then, we process those unstructured time series data into structured data so that it can be imported into traditional machine learning algorithms through features engineering such as feature selection, label encoding and feature scaling. Supervised machine learning algorithms (i.e., XGBoost and Random Forest) are applied to address this binary classification problem (whether or not there is run time-underestimation).

We collect HPC job logs and use it to build prediction models for underestimation of job run time on HPC systems, with state-of-the-art supervised machine learning algorithms. We also introduce some evaluation metrics (AP, PRC) and a trade-off function that show the relationship between Precision and Recall, which are more fair and useful than metrics used in similar previous studies (such as overall accuracy, precision, recall).

We split the entire job dataset into subsets categorized with different HPC application names. According to different applications and their models performance, we analyzed the causes of the performance differences on applications and also provide corresponding improvement measures for the future work.

Additionally, We plot precision and recall scores as a function of the threshold to help select the threshold value that gets the best precision-recall trade-off for different models and different purposes. And we also plot feature importance and decision trees of these prediction models to reveal surprising hidden patterns between different HPC applications, as well as different features on computing resource usage.

Finally, we design a simulation test to simulate in-progress job logs based on existing data. Then, we purpose a metric-"SLR", to approximately evaluate the economy and benefit of our predicting model by use of the simulation test. According to the results of our simulation, we can conclude that our prediction models are able to predict run time-underestimated job with different accuracy at different checkpoints time, which would benefit HPC users with saving time loss (money loss) to a certain extent. Besides, using our prediction model also helps HPC system to free up computing resources as much as possible when the job run time-underestimation occurred. Altogether, our machine learning-based prediction model contributes to improving the overall productivity and efficiency of the HPC system. Dotted box in Figure 4.37 marks the contribution of this section.

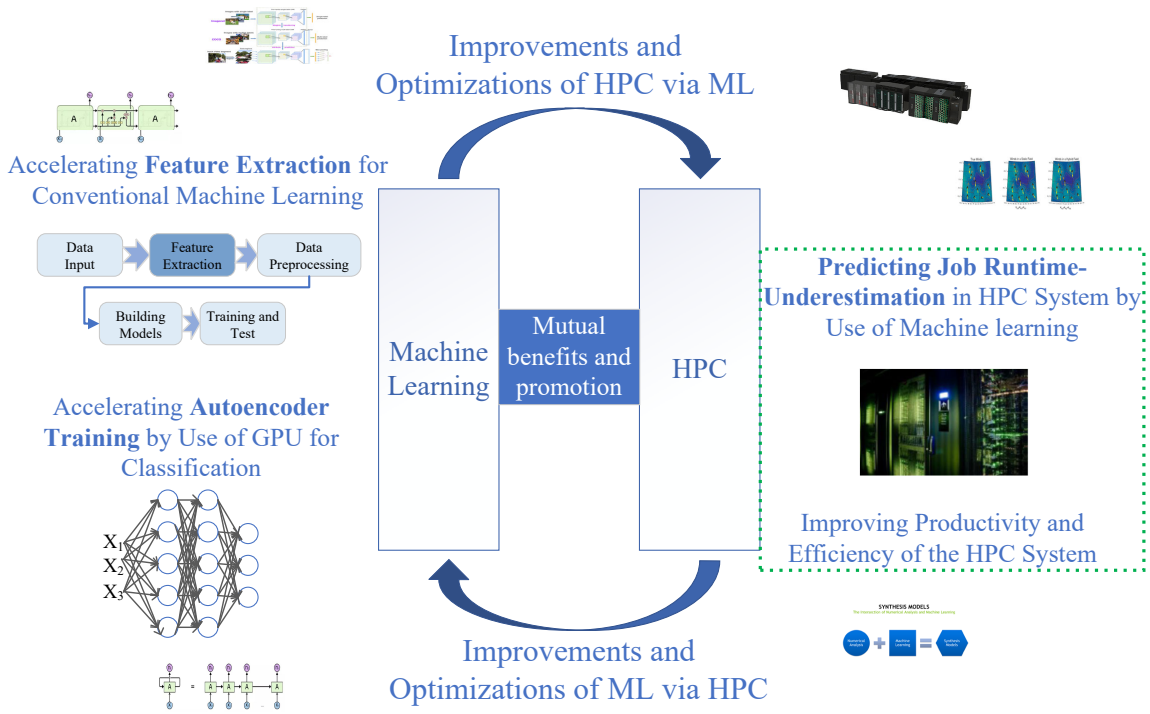


Figure 4.37: Extend and Strengthen the Mutual Relationship between HPC and Machine Learning

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we extend and strengthen the mutual benefit between HPC and Machine Learning by bi-directional research exploration: speeding up Machine Learning with HPC's help, and benefiting the productivity and efficiency of HPC by use of Machine Learning. Through our work, we extend the mutually beneficial bond between HPC and Machine Learning to broader fields.

For improving the overall efficiency of Machine Learning with HPC, we accelerate the feature extraction of bioacoustics data by employing a JIT compiler technique, Numba, which implements automatic parallelisation and performs other optimizations, as well as a GPU-accelerated library going by the principle of modifying Python-based baseline as little as possible. With small modifications in the baseline code of feature extraction, our optimized feature extraction yields maximum 86x speedup over the baseline without losing programming efficiency. In order to reduce the entire time cost of Machine Learning modeling for handcrafted feature-based classification, we speed up the training of SSAE with a soft-max layer for bird sounds classification by use of HPC hardware and software. Our experiment

shows that using GPUs (K20, P100) results in up to 10x speed-up compared to CPU when training with different feature dimensions of bird sounds.

For helping HPC by improving the overall productivity and efficiency of HPC systems with Machine Learning, we propose a Machine Learning based data-driven approach to predict run time-underestimated jobs in HPC systems by learning complex hidden patterns between different HPC applications and different features of computing resource usage, as well as user behavior from job logs. The experiment results show that our prediction models are able to predict run time-underestimated jobs with different accuracy at different checkpoints times in HPC systems, which would benefit HPC users by reducing time and monetary loss. It also helps HPC systems free up computing resources to a certain extent and allows users and administrators take the appropriate action (to terminate those on-going jobs that are predicted to be run time-underestimated). In a word, our Machine Learning-based prediction model would be able to improve the overall productivity and efficiency of the HPC system. In the preliminary simulation, we evaluate benefits of the prediction model with a metric named SLR, most of SLRs are around 0.05-0.12, 0.337 is the best one. If we set the checkpoints for different applications to their best points, we can estimate that our prediction model would save 24962 hours totally based on the result of preliminary simulation from the existing database.

All in all, in this thesis, we perform the bi-directional research exploration between HPC and Machine Learning as Figure 5.1 showing: speeding up Machine Learning with HPCs help; benefiting the productivity and efficiency of HPC by use of Machine Learning. Through our three use cases, we can clearly conclude that HPC hardware and software approaches are able to supply powerful computing resource for acceleration and scaling of Machine Learning. Meanwhile, Machine Learning can benefit HPC in improving the overall productivity, efficiency and etc..

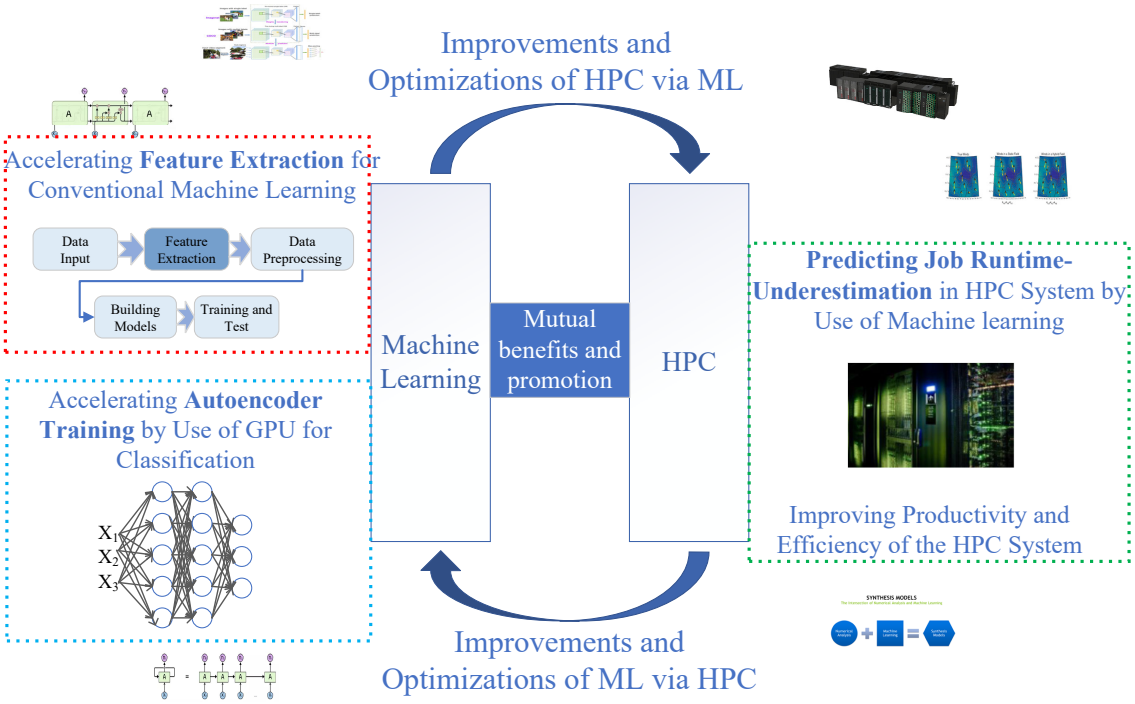


Figure 5.1: We Extend and Strengthen the Mutual Relationship between HPC and Machine Learning

This is the role of interaction between HPC and Machine Learning: mutual benefit and promotion.

5.2 Future Work

In the future, for the feature engineering, we would like to add more simple features such as resource usages of I/O, network communication between nodes, the mean, maximum/minimum values, standard deviation of all resource usages.

Adding some advanced features like functional, wavelet and timefrequency analysis which are some kinds of techniques used to signal processing, so we can apply those techniques to our research in the same way as to signal processing. More features can be extracted from raw computing resource usages data by applying those techniques.

Other very important feature, the Command Lines. As we mentioned at the Chapter 2, command Lines (job scripts) are used for job submitting or job states

checking in HPC system. Job scripts contain important information about packages dependencies, program parameters and etc. which will greatly affect the efficiency and execution time of jobs. Unfortunately, job scripts have not been recording and saving on TSUBAME 2.5. For the future work, there are some preliminary ideas about analysing and utilizing job scripts to build prediction models for abnormal detection, fault tolerance, job execution time prediction, performance prediction and so on, if we start to keep job scripts on TSUBAME 3.0.

The preliminary idea is using NLP techniques like bag-of-words model [113] and Word-to-vector [114] to extract features from hidden information in job scripts.

Bag-of-words model: The bag-of-words method can extract the most common type of characteristics or features such as term frequency, namely, the number of times a term appears from raw data.

Word-to-vector: It is a group of related models that are used to produce word embeddings from raw text. This method convert input raw data into a vector space which can characterize potential patterns of job scripts. Both of these methods can generate new feature sets. When generated features using bag-of-words model and Word-to-vector from job scripts are available, we combine those new features with the previously features we used in this research and to train new prediction models. These models trained with new features can achieve better results theoretically.

End-to-end with RNN: M Wyatt etc. propose to use CNN to process job script [115]. It is a very great idea that using Deep neural networks to train models, however we doubt that how many features or how many important information can be preserved in job scripts after transforming to images? We think that there are suitable choices such as RNN [14] and LSTM [116]. In work [117] [118], feeding raw time series data to RNN and LSTM and training classification models for prediction events has achieves good results. we would like to train end-to-to RNN and LSTM

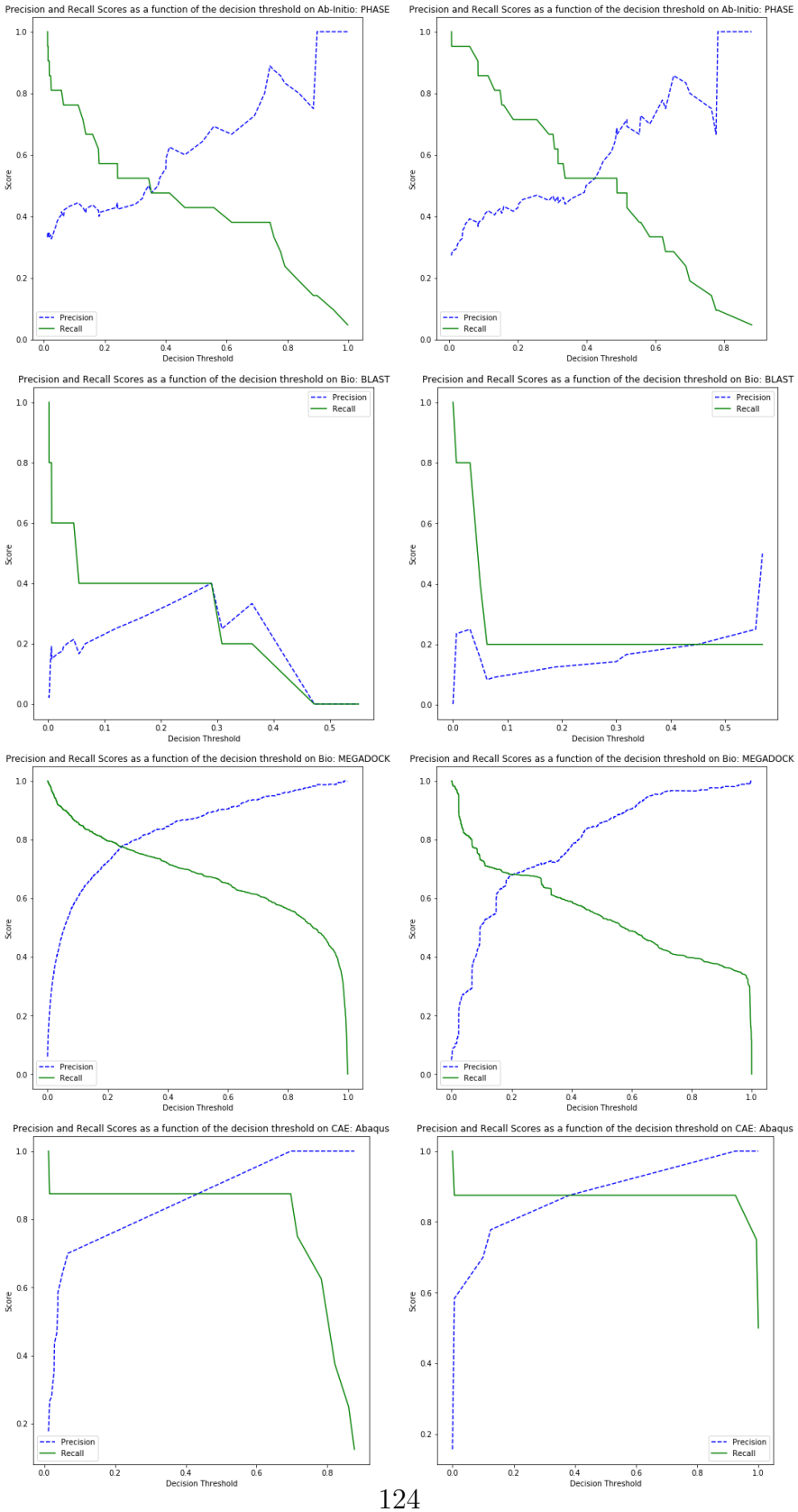
for processing raw time series data of computing resource usages and predict job run time underesimation or abnormal detection in HPC system.

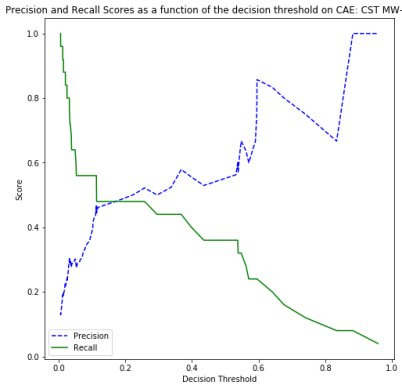
Appendix A

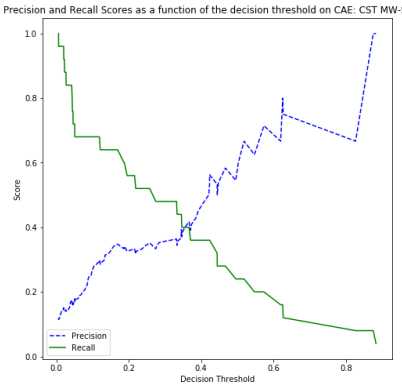
Appendix

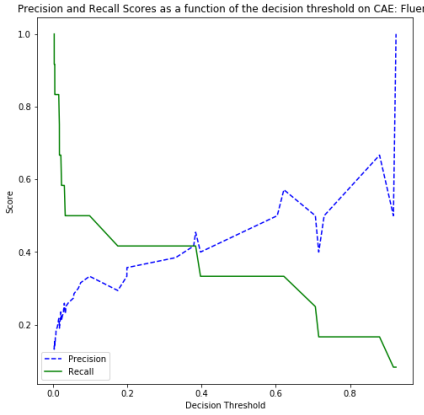
A.1 Figures

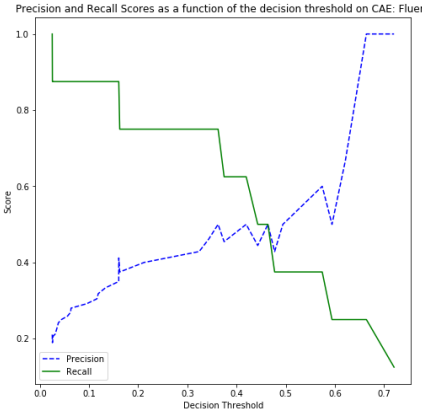
Figure A.1: Precision and Recall as Function of Threshold by XGBoost (Left) and Random Forest (Right) for Different Applications

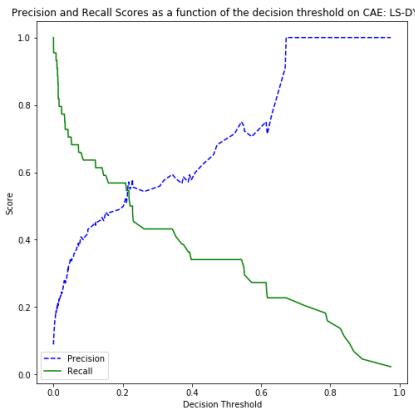


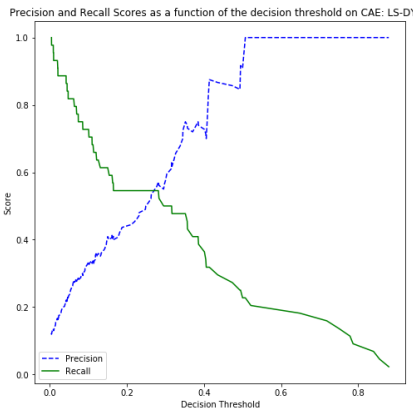

Precision and Recall Scores as a function of the decision threshold on CAE: CST MW-Studio
Score
Decision Threshold
Precision
Recall

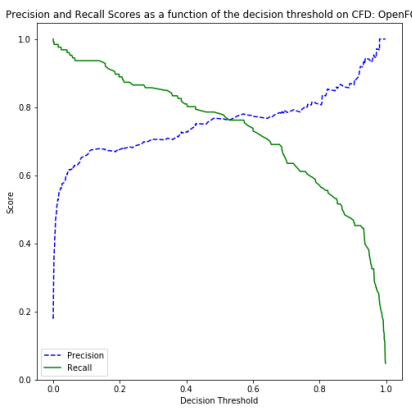

Precision and Recall Scores as a function of the decision threshold on CAE: CST MW-Studio
Score
Decision Threshold
Precision
Recall

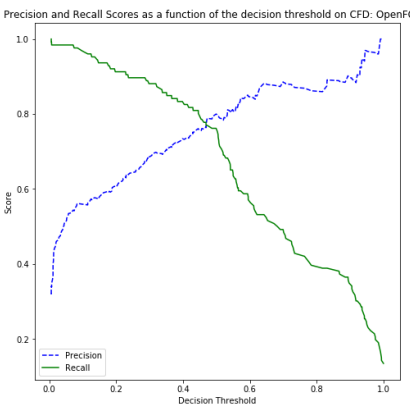

Precision and Recall Scores as a function of the decision threshold on CAE: Fluent
Score
Decision Threshold
Precision
Recall

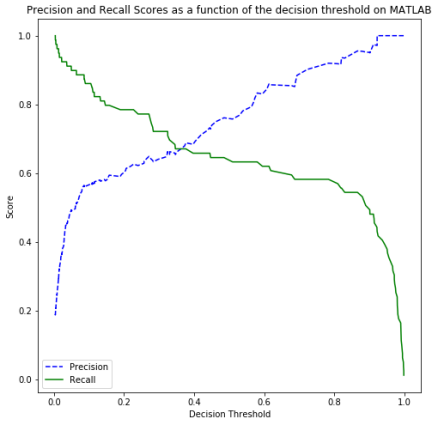

Precision and Recall Scores as a function of the decision threshold on CAE: Fluent
Score
Decision Threshold
Precision
Recall

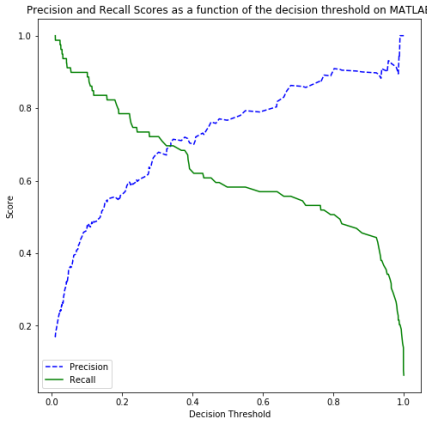

Precision and Recall Scores as a function of the decision threshold on CAE: LS-DYNA
Score
Decision Threshold
Precision
Recall

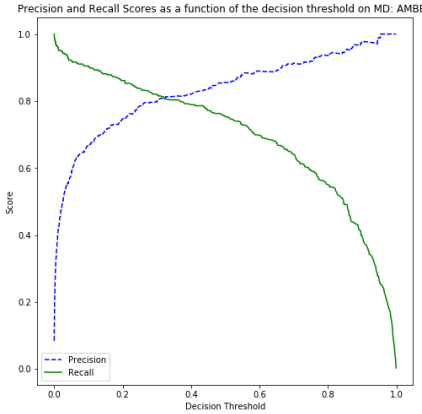

Precision and Recall Scores as a function of the decision threshold on CAE: LS-DYNA
Score
Decision Threshold
Precision
Recall

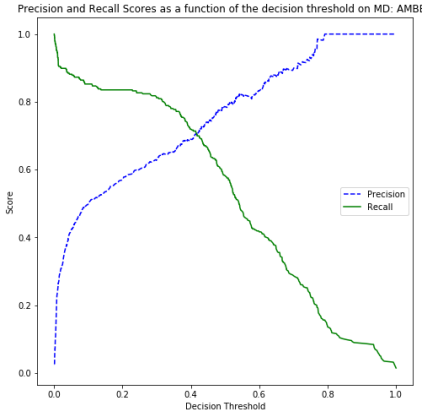

Precision and Recall Scores as a function of the decision threshold on CFD: OpenFOAM
Score
Decision Threshold
Precision
Recall

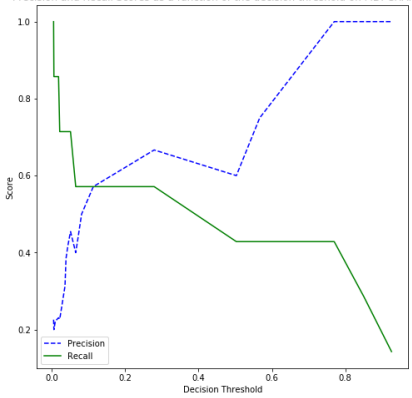

Precision and Recall Scores as a function of the decision threshold on CFD: OpenFOAM
Score
Decision Threshold
Precision
Recall

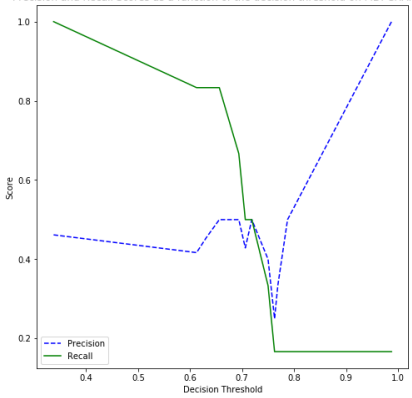

Precision and Recall Scores as a function of the decision threshold on MATLAB
Score
Decision Threshold
Precision
Recall

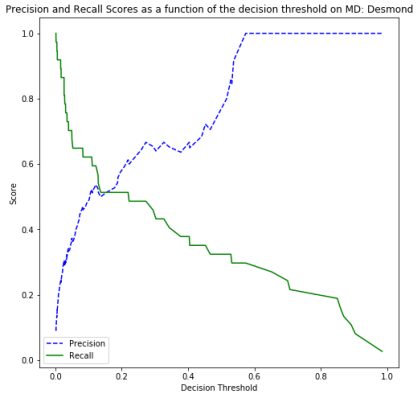

Precision and Recall Scores as a function of the decision threshold on MATLAB
Score
Decision Threshold
Precision
Recall

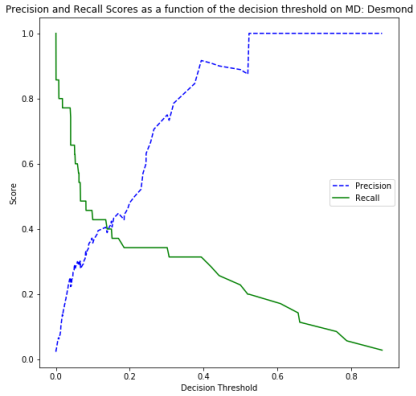

Precision and Recall Scores as a function of the decision threshold on MD: AMBER
Score
Decision Threshold
Precision
Recall

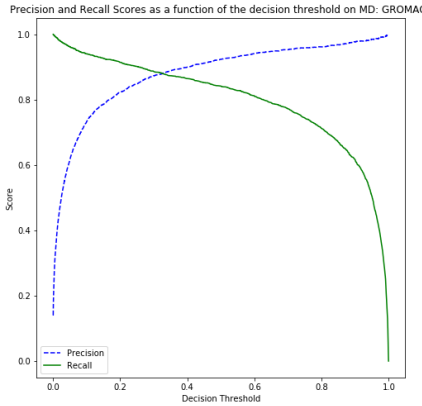

Precision and Recall Scores as a function of the decision threshold on MD: AMBER
Score
Decision Threshold
Precision
Recall

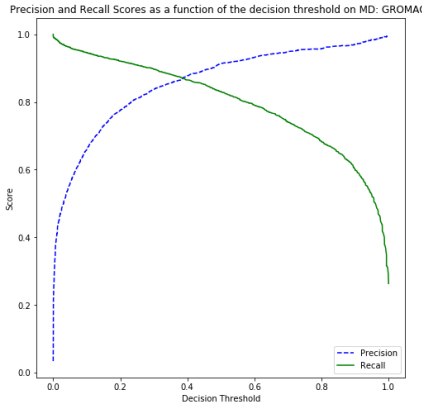

Precision and Recall Scores as a function of the decision threshold on MD: CHARMM
Score
Decision Threshold
Precision
Recall

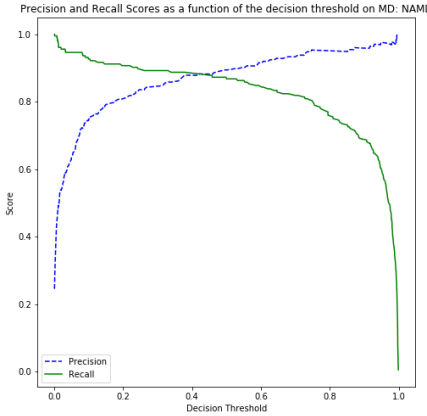

Precision and Recall Scores as a function of the decision threshold on MD: CHARMM
Score
Decision Threshold
Precision
Recall


Precision and Recall Scores as a function of the decision threshold on MD: Desmond MD
Score
Decision Threshold
Precision
Recall

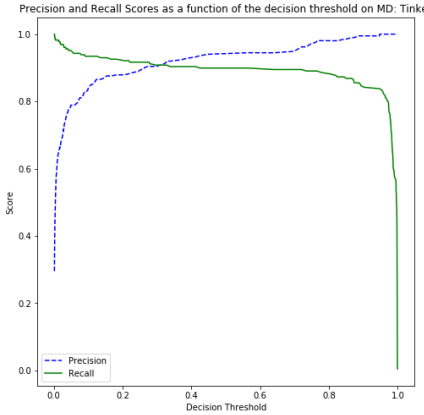

Precision and Recall Scores as a function of the decision threshold on MD: Desmond MD
Score
Decision Threshold
Precision
Recall

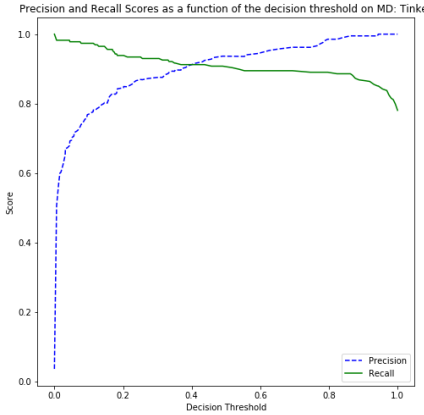

Precision and Recall Scores as a function of the decision threshold on MD: GROMACS
Score
Decision Threshold
Precision
Recall

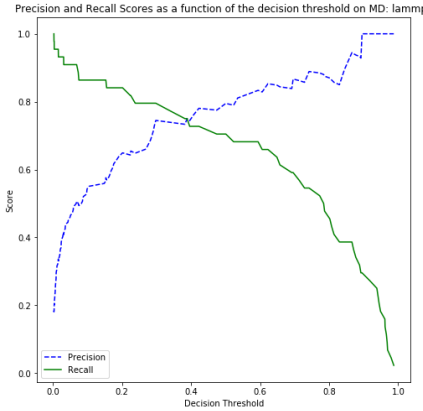

Precision and Recall Scores as a function of the decision threshold on MD: GROMACS
Score
Decision Threshold
Precision
Recall

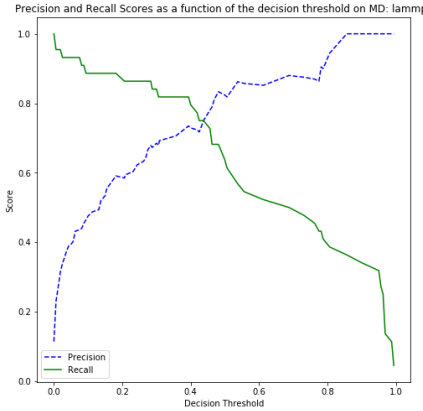

Precision and Recall Scores as a function of the decision threshold on MD: NAMD
Score
Decision Threshold
Precision
Recall

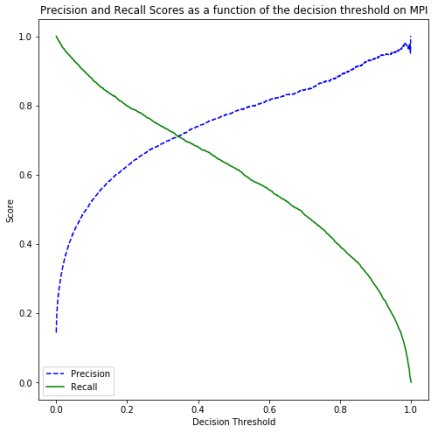

Precision and Recall Scores as a function of the decision threshold on MD: NAMD
Score
Decision Threshold
Precision
Recall

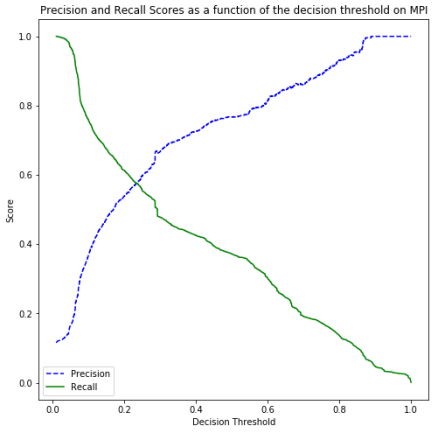

Precision and Recall Scores as a function of the decision threshold on MD: Tinker
Score
Decision Threshold
Precision
Recall

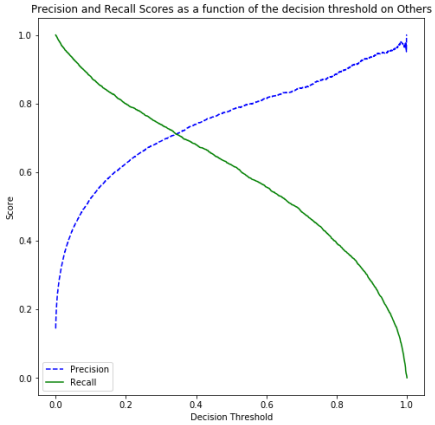

Precision and Recall Scores as a function of the decision threshold on MD: Tinker
Score
Decision Threshold
Precision
Recall

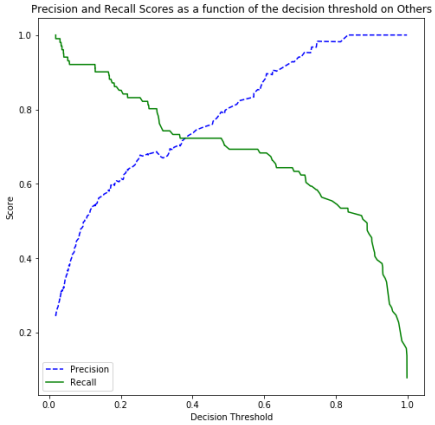

Precision and Recall Scores as a function of the decision threshold on MD: lammps
Score
Decision Threshold
Precision
Recall

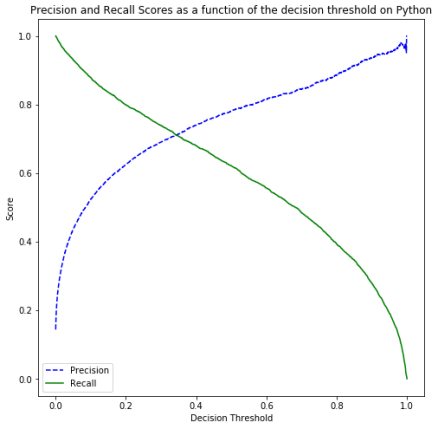

Precision and Recall Scores as a function of the decision threshold on MD: lammps
Score
Decision Threshold
Precision
Recall

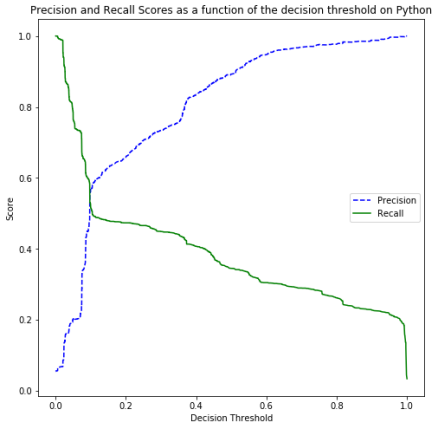

Precision and Recall Scores as a function of the decision threshold on MPI
Score
Decision Threshold
Precision
Recall

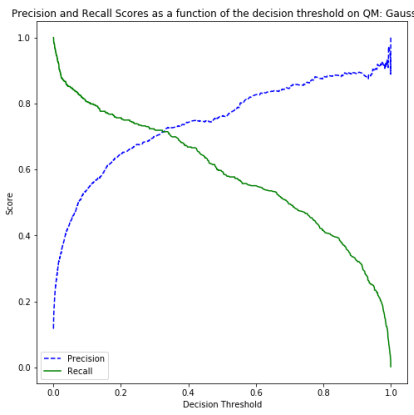

Precision and Recall Scores as a function of the decision threshold on MPI
Score
Decision Threshold
Precision
Recall

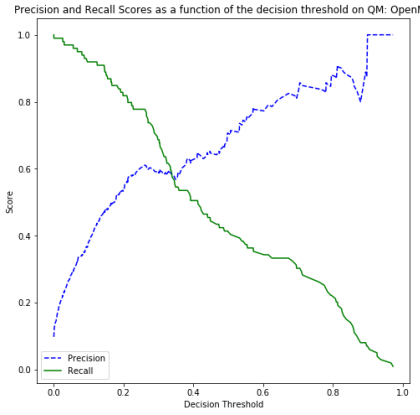

Precision and Recall Scores as a function of the decision threshold on Others
Score
Decision Threshold
Precision
Recall

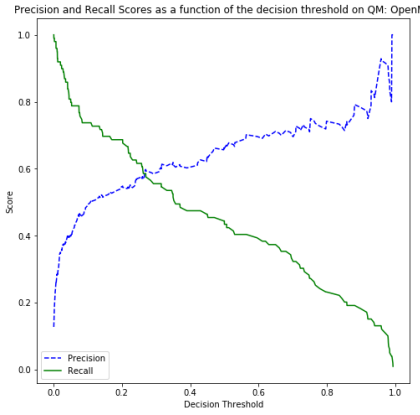

Precision and Recall Scores as a function of the decision threshold on Others
Score
Decision Threshold
Precision
Recall

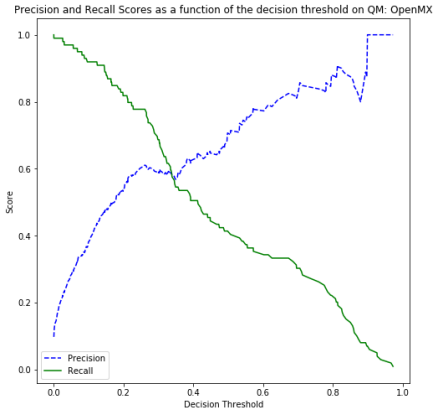

Precision and Recall Scores as a function of the decision threshold on Python
Score
Decision Threshold
Precision
Recall

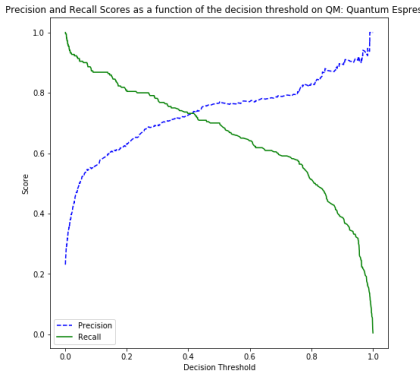

Precision and Recall Scores as a function of the decision threshold on Python
Score
Decision Threshold
Precision
Recall

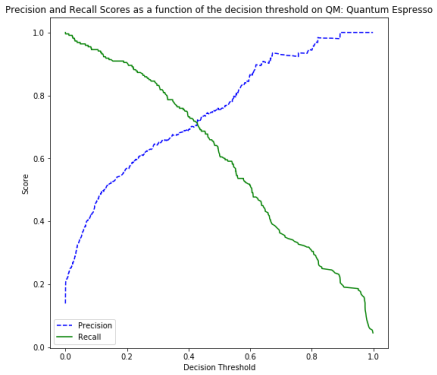

Precision and Recall Scores as a function of the decision threshold on QM: Gaussian
Score
Decision Threshold
Precision
Recall

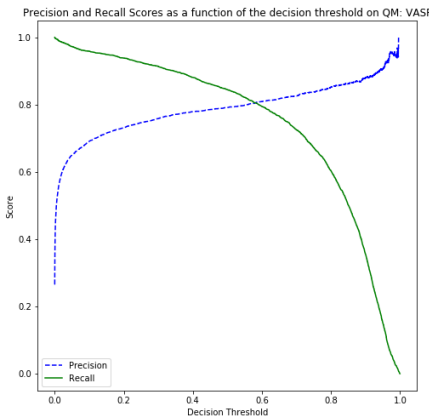

Precision and Recall Scores as a function of the decision threshold on QM: OpenMX
Score
Decision Threshold
Precision
Recall

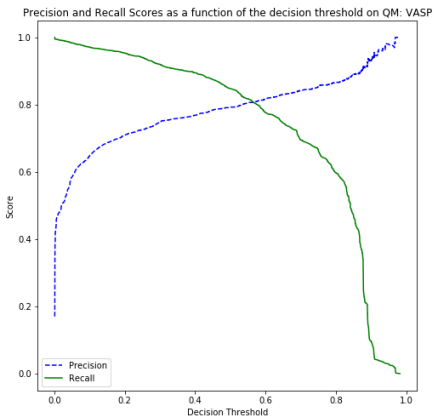

Precision and Recall Scores as a function of the decision threshold on QM: OpenMX
Score
Decision Threshold
Precision
Recall

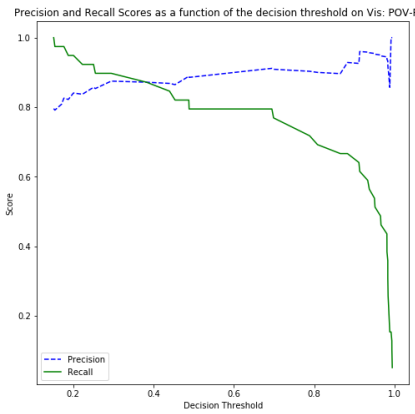

Precision and Recall Scores as a function of the decision threshold on QM: OpenMX
Score
Decision Threshold
Precision
Recall

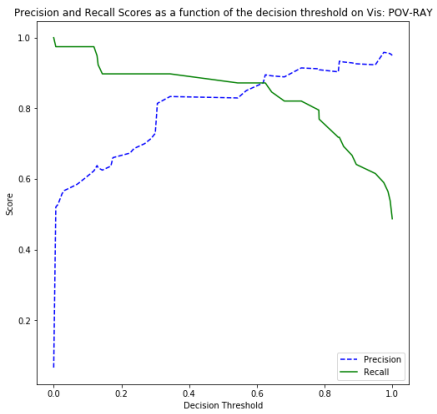

Precision and Recall Scores as a function of the decision threshold on QM: Quantum Espresso
Score
Decision Threshold
Precision
Recall

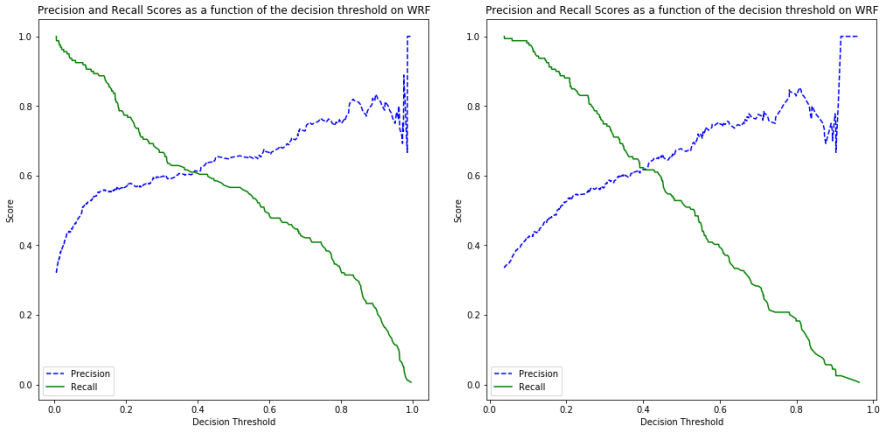

Precision and Recall Scores as a function of the decision threshold on QM: Quantum Espresso
Score
Decision Threshold
Precision
Recall


Precision and Recall Scores as a function of the decision threshold on QM: VASP
Score
Decision Threshold
Precision
Recall


Precision and Recall Scores as a function of the decision threshold on QM: VASP
Score
Decision Threshold
Precision
Recall


Precision and Recall Scores as a function of the decision threshold on Vis: POV-RAY
Score
Decision Threshold
Precision
Recall


Precision and Recall Scores as a function of the decision threshold on Vis: POV-RAY
Score
Decision Threshold
Precision
Recall


Precision and Recall Scores as a function of the decision threshold on WRF
Score
Decision Threshold
Precision
Recall
Precision and Recall Scores as a function of the decision threshold on WRF
Score
Decision Threshold
Precision
Recall

Bibliography

[1] Isha Kapur, Karthik Belgodu, Pankaj Purandare, Iman Sadooghi, and Ioan Raicu. Extending cloudkon to support hpc job scheduling. *Illinois Institute of Technology, Department of Computer Science, Technical Report*, 2013.

[2] Nicole Wolter, Michael O McCracken, Allan Snavely, Lorin Hochstein, Taiga Nakamura, and Victor Basili. Whats working in hpc: Investigating hpc user behavior and productivity. *CTWatch Quarterly*, 2(4A):9–17, 2006.

[3] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[4] Atul. Ai vs machine learning vs deep learning. `https://www.edureka.co/blog/ai-vs-machine-learning-vs-deep-learning/`. Jul 26,2018.

[5] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[6] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[8] Aditya Grover, Ashish Kapoor, and Eric Horvitz. A deep hybrid model for weather forecasting. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 379–386. ACM, 2015.

[9] Yunjie Liu, Evan Racah, Joaquin Correa, Amir Khosrowshahi, David Lavers, Kenneth Kunkel, Michael Wehner, William Collins, et al. Application of deep convolutional neural networks for detecting extreme weather in climate datasets. *arXiv preprint arXiv:1605.01156*, 2016.

[10] Nvidia. Intersection of HPC and Machine Learning, 2018.

[11] Jim Gao and Ratnesh Jamidar. Machine learning applications for data center optimization. *Google White Paper*, 2014.

[12] John Morris. How facebook scales ai. `https://www.zdnet.com/article/how-facebook-scales-ai/`. Accessed June 6, 2018.

[13] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

[14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[15] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, page 1. ACM, 2018.

[16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[17] Sambit Mahapatra. Why deep learning over traditional machine learning? `https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063`. Mar 22,2018.

[18] Florian Eyben, Felix Weninger, Florian Gross, and Björn Schuller. Recent developments in opensmile, the munich open-source multimedia feature extractor. In *Proc. of ACM MM*, pages 835–838, Barcelona, Spain, 2013. ACM.

[19] Philip Guo. Python is now the most popular introductory teaching language at top us universities. *BLOG@ CACM, July*, page 47, 2014.

[20] Kun Qian, Zhiyong Xu, Huijie Xu, Yaqi Wu, and Zhao Zhao. Automatic detection, segmentation and classification of snore related signals from overnight audio recording. *IET Signal Processing*, 9(1):21–29, 2015.

[21] Alan Edelman. High-performance computing programming with ease, 2014.

[22] John K Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[23] Continuum Analytics. Anaconda software distribution. Computer software, nov 2015.

[24] Intel. Intel distribution for python, 2018.

[25] Prechelt and Garret. How long it takes to write a string processing application in various languages.

[26] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 37–49, 2012.

[27] Naresh M Punjabi. The epidemiology of adult obstructive sleep apnea. *Proceedings of the American Thoracic Society*, 5(2):136–143, 2008.

[28] Eva Azagra-Calero, Eduardo Espinar-Escalona, José M Barrera-Mora, José M Llamas-Carreras, and Enrique Solano-Reina. Obstructive sleep apnea syndrome (osas). review of the literature. *Medicina oral, patologia oral y cirugia bucal*, 17(6):e925, 2012.

[29] Clive K Catchpole and Peter JB Slater. *Bird song: biological themes and variations*. Cambridge university press, Cambridge, UK, 2003.

[30] Camille Parmesan and Gary Yohe. A globally coherent fingerprint of climate change impacts across natural systems. *Nature*, 421(6918):37–42, 2003.

[31] Ciira wa Maina. The kenya bioacoustics project.

[32] PJB Slater. Fifty years of bird song research: a case study in animal behaviour.

[33] The Cornell Lab of Ornithology. Birds and climate change.

[34] Joshua Leeds. *The power of sound: How to be healthy and productive using music and sound*. Simon and Schuster, 2010.

[35] Peter Rickwood and Andrew Taylor. Methods for automatically analyzing humpback song units. *The Journal of the Acoustical Society of America*, 123(3):1763–1772, 2008.

[36] RL Grossman and KP White. A vision for a biomedical cloud. *Journal of internal medicine*, 271(2):122–130, 2012.

[37] Hesham Ali. Big data analytics in biomedical informatics. Technical report, BIOSTEC 2015, 2015.

[38] Md Sarwar Kamal and Sonia Farhana Nimmy. Strucbreak: A computational framework for structural break detection in dna sequences. *Interdisciplinary Sciences: Computational Life Sciences*, pages 1–16, 2016.

[39] S Bastrakov, Iosif Meyerov, V Gergel, A Gonoskov, A Gorshkov, E Efimenko, M Ivanchenko, M Kirillin, A Malova, G Osipov, et al. High performance computing in biomedical applications. *Procedia Computer Science*, 18:10–19, 2013.

[40] Jian Guo, Kun Qian, Zhaomeng Zhu, Gongxuan Zhang, and Huijie Xu. A cloud computing system for snore signals processing. In *International Workshop on Advanced Parallel Processing Technologies*, pages 359–366. Springer, 2013.

[41] Jian Guo, Kun Qian, Gongxuan Zhang, Huijie Xu, and Björn Schuller. Accelerating biomedical signal processing using gpu: A case study of snore sound feature extraction. *Interdisciplinary Sciences: Computational Life Sciences*, 9(4):550–555, 2017.

[42] Dirk Pevernagie, Ronald M Aarts, and Micheline De Meyer. The acoustics of snoring. *Sleep medicine reviews*, 14(2):131–144, 2010.

[43] J Mesquita, J Solà-Soler, José Antonio Fiz, José Morera, and Raimon Jané. All night analysis of time interval between snores in subjects with sleep apnea hypopnea syndrome. *Medical & biological engineering & computing*, 50(4):373–381, 2012.

[44] Maximilian Schmitt, Christoph Janott, Vedhas Pandit, Kun Qian, Clemens Heiser, Werner Hemmert, and Björn Schuller. A bag-of-audio-words approach for snore sounds excitation localisation. In *Proceedings of 12th ITG Conference on Speech Communication*, pages 230–234, 2016.

[45] AM Adeshina and R Hashim. Computational approach for securing radiology-diagnostic data in connected health network using high-performance gpu-accelerated aes. *Interdisciplinary Sciences: Computational Life Sciences*, pages 1–13, 2016.

[46] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 7. ACM, 2015.

[47] Anaconda. How does numba work? `https://towardsdatascience.com/speed-up-your-algorithms-part-2-numba-293e554c5cc1`. April 25,2017.

[48] Cheng-Yuan Liou, Jau-Chi Huang, and Wen-Chie Yang. Modeling word perception using the elman network. *Neurocomputing*, 71(16-18):3150–3157, 2008.

[49] Cheng-Yuan Liou, Wei-Chen Cheng, Jiun-Wei Liou, and Daw-Ran Liou. Autoencoder for words. *Neurocomputing*, 139:84–96, 2014.

[50] Pedro Domingos. *The master algorithm: How the quest for the ultimate learning machine will remake our world*. Basic Books, 2015.

[51] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[52] Satoshi Matsuoka. The tsubame 2.5 evolution. *Tsubame e-Science Journal*, 10:2–8, 2013.

[53] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12), 2007.

[54] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[55] Nithin Nakka, Ankit Agrawal, and Alok Choudhary. Predicting node failure in high performance computing systems from failure and usage logs. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1557–1566. IEEE, 2011.

[56] Chao Chen, Andy Liaw, and Leo Breiman. Using random forest to learn imbalanced data. *University of California, Berkeley*, 110, 2004.

[57] Mike Chen, Alice X Zheng, Jim Lloyd, Michael I Jordan, and Eric Brewer. Failure diagnosis using decision trees. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 36–43. IEEE, 2004.

[58] Wucherl Yoo, Alex Sim, and Kesheng Wu. Machine learning based job status prediction in scientific clusters. In *SAI Computing Conference (SAI), 2016*, pages 44–53. IEEE, 2016.

[59] Jian Guo, Kun Qian, Huijie Xu, Christoph Janott, Björn Schuller, and Satoshi Matsuoka. Gpu-based fast signal processing for large amounts of snore sound data. In *Proc. of IEEE GCCE*, pages 523–524, Kyoto, Japan, 2016. IEEE.

[60] Jian Guo, Kun Qian, Björn Schuller, and Satoshi Matsuoka. Gpu-based training of autoencoders for bird sound data processing. In *Consumer Electronics-Taiwan (ICCE-TW), 2017 IEEE International Conference on*, pages 145–146. IEEE, 2017.

[61] Kun Qian, Jian Guo, Huijie Xu, Zhaomeng Zhu, and Gongxuan Zhang. Snore related signals processing in a private cloud computing system. *Interdisciplinary Sciences: Computational Life Sciences*, 6(3):216–221, 2014.

[62] Patrick J Strollo Jr and Robert M Rogers. Obstructive sleep apnea. *New England Journal of Medicine*, 334(2):99–104, 1996.

[63] UR Abeyratne, S De Silva, C Hukins, and B Duce. Obstructive sleep apnea screening by integrating snore feature classes. *Physiological measurement*, 34(2):99, 2013.

[64] Kun Qian, Christoph Janott, Vedhas Pandit, Zixing Zhang, Clemens Heiser, Winfried Hohenhorst, Michael Herzog, Werner Hemmert, and Björn Schuller. Classification of the excitation location of snore sounds in the upper airway by acoustic multi-feature analysis. *IEEE Transactions on Biomedical Engineering*, 64(8):11 pages, in press, 2017.

[65] Xu Huijie, Huang Weining, and Chen Lan Yulisheng. Spectral analysis of snoring sound and site of obstruction in obstructive sleep apnea/hypopnea syndrome. *Journal of Audiology and Speech Pathology*, 1:009, 2011.

[66] Ali Azarbarzin and Zahra MK Moussavi. Automatic and unsupervised snore sound extraction from respiratory sound signals. *IEEE Transactions on Biomedical Engineering*, 58(5):1156–1162, 2011.

[67] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[68] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[69] Beijing Hospital. Beijing hospital. `http://www.bjhmoh.cn/`.

[70] samplebias. numpy float: 10x slower than builtin in arithmetic operations? `https://stackoverflow.com/questions/5956783/numpy-float-10x-slower-than-builtin-in-arithmetic-operations`. May 19,2011.

[71] Florian Eyben. *Real-time speech and music classification by large audio feature space extraction*. Springer, 2015.

[72] Kun Qian, Zixing Zhang, Fabien Ringeval, et al. Bird sounds classification by large scale acoustic features and extreme learning machine. In *Proc. of IEEE 2015 GlobalSIP*, pages 1317–1321. IEEE, 2015.

[73] Kun Qian, Jian Guo, Ken Ishida, and Satoshi Matsuoka. Fast recognition of bird sounds using extreme learning machines. *IEEJ Transactions on Electrical and Electronic Engineering*, 12(2):294–296, 2017.

[74] Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, and Caroline Suen. Ufldl tutorial. `https://http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial`. April 7,2013.

[75] Jian Guo, Akihiro Nomura, Ryan Barton, Haoyu Zhang, and Satoshi Matsuoka. Machine learning predictions for underestimation of job runtime on hpc system. In *Asian Conference on Supercomputing Frontiers*, pages 179–198. Springer, 2018.

[76] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. Failure prediction in ibm bluegene/l event logs. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 583–588. IEEE, 2007.

[77] Hamid Fadishei, Hamid Saadatfar, and Hossein Deldari. Job failure prediction in grid environment based on workload characteristics. In *Computer Conference, 2009. CSICC 2009. 14th International CSI*, pages 329–334. IEEE, 2009.

[78] Fredrik Vraalsen, Ruth A Aydt, Celso L Mendes, and Daniel A Reed. Performance contracts: Predicting and monitoring grid application behavior. In *International Workshop on Grid Computing*, pages 154–165. Springer, 2001.

[79] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times using historical information. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–142. Springer, 1998.

[80] Allen B Downey. Using queue time predictions for processor allocation. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 35–57. Springer, 1997.

[81] Richard Gibbons. A historical application profiler for use by parallel schedulers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 58–77. Springer, 1997.

[82] Allen B Downey and Dror G Feitelson. The elusive goal of workload characterization. *ACM SIGMETRICS Performance Evaluation Review*, 26(4):14–29, 1999.

[83] Robert L Henderson. Job scheduling under the portable batch system. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294. Springer, 1995.

[84] Jyotiprasad Medhi. *Statistical methods: an introductory text*. New Age International, 1992.

[85] Andrew Ng. Machine learning and ai via brain simulations, 2013.

[86] Jason Brownlee. Discover feature engineering, how to engineer features and how to get good at it. *Machine Learning Process*, 2014.

[87] Frank Seide, Gang Li, Xie Chen, and Dong Yu. Feature engineering in context-dependent deep neural networks for conversational speech transcription. In *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pages 24–29. IEEE, 2011.

[88] Mairead L Bermingham, Ricardo Pong-Wong, Athina Spiliopoulou, Caroline Hayward, Igor Rudan, Harry Campbell, Alan F Wright, James F Wilson, Felix Agakov, Pau Navarro, et al. Application of high-dimensional feature selection: evaluation for genomic prediction in man. *Scientific reports*, 5, 2015.

[89] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn:

Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[90] Joel Grus. *Data science from scratch: First principles with Python.* " O'Reilly Media, Inc.", 2015.

[91] Dorian Pyle. *Data preparation for data mining*, volume 1. morgan kaufmann, 1999.

[92] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[93] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[94] Rongwei Song, Siding Chen, Bailong Deng, and Li Li. extreme gradient boosting for identifying individual users across different digital devices. In *International Conference on Web-Age Information Management*, pages 43–54. Springer, 2016.

[95] Didrik Nielsen. Tree boosting with xgboost-why does xgboost win" every" machine learning competition? Master's thesis, NTNU, 2016.

[96] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J Leung, Manuel Egele, and Ayse K Coskun. Diagnosing performance variations in hpc applications using machine learning. In *International Supercomputing Conference*, pages 355–373. Springer, 2017.

[97] Randal S Olson, William La Cava, Zairah Mustahsan, Akshay Varik, and Jason H Moore. Data-driven advice for applying machine learning to bioinformatics problems. *arXiv preprint arXiv:1708.05070*, 2017.

[98] Tin Kam Ho. Random decision forests. In *Document analysis and recognition, 1995., proceedings of the third international conference on*, volume 1, pages 278–282. IEEE, 1995.

[99] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8):832–844, 1998.

[100] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[101] James W Perry, Allen Kent, and Madeline M Berry. Machine literature searching x. machine language; factors underlying its design and development. *Journal of the Association for Information Science and Technology*, 6(4):242–254, 1955.

[102] Stephen Robertson. A new interpretation of average precision. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 689–690. ACM, 2008.

[103] Takaya Saito and Marc Rehmsmeier. The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. *PloS one*, 10(3):e0118432, 2015.

[104] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2017.

[105] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[106] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[107] Jannis Klinkenberg, Christian Terboven, Stefan Lankes, and Matthias S Müller. Data mining-based analysis of hpc center operations. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 766–773. IEEE, 2017.

[108] Hao Zhang, Haihang You, Bilel Hadri, and Mark Fahey. Hpc usage behavior analysis and performance estimation with machine learning techniques. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[109] Robert C Holte, Liane Acker, Bruce W Porter, et al. Concept learning and the problem of small disjuncts. In *IJCAI*, volume 89, pages 813–818. Citeseer, 1989.

[110] Taeho Jo and Nathalie Japkowicz. Class imbalances versus small disjuncts. *ACM Sigkdd Explorations Newsletter*, 6(1):40–49, 2004.

[111] Peter Flach. The many faces of roc analysis in machine learning. *ICML Tutorial*, 2004.

[112] Bradley Efron. Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of the American statistical association*, 78(382):316–331, 1983.

[113] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[114] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[115] M Wyatt, S Herbein, D Ahn, A Moody, T Gamblin, and M Taufer. Unstructured data analytics for next-generation hpc schedulers: Capturing jobs' needs from unstructured job scripts. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.

[116] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[117] Nikolay Laptev, Jason Yosinski, Li Erran Li, and Slawek Smyl. Time-series extreme event forecasting with neural networks at uber. In *International Conference on Machine Learning*, 2017.

[118] Fazle Karim, Somshubra Majumdar, Houshang Darabi, and Shun Chen. Lstm fully convolutional networks for time series classification. *IEEE Access*, 6:1662–1669, 2018.