# Machine Learning Predictions for Underestimation of Job Runtime on HPC System

Jian Guo[1(✉)], Akihiro Nomura[2(✉)], Ryan Barton[1(✉)], Haoyu Zhang[1(✉)], and Satoshi Matsuoka[1,2,3(✉)]

[1] Department of Mathematical and Computing Science, Tokyo Institute of Technology, Tokyo, Japan
{guo.j.ae,barton.r.aa}@m.titech.ac.jp, lynkzhang@gmail.com, matsu@is.titech.ac.jp

[2] Global Scientific Information and Computing Center, Tokyo Institute of Technology, Tokyo, Japan
nomura.a.ac@m.titech.ac.jp

[3] Real World Big-Data Computing Open Innovation Laboratory, National Institute of Advanced Industrial Science and Technology, Tokyo, Japan



**Abstract.** In modern high-performance computing (HPC) systems, users are usually requested to estimate the job runtime for system scheduling when they submit a job. In general, an underestimation of job runtime will cause the HPC system to terminate the job before its completion. If users could be notified that their jobs may not finish before its allocated time expires, users can take actions, such as killing the job and resubmitting it after parameter adjustment, to save time and cost. Meanwhile, the productivity of HPC systems could also be vastly improved. In this paper, we propose a data-driven approach – that is, one that actively observes, analyzes, and logs jobs – for predicting underestimation of job runtime on HPC systems. Using data produced by TSUBAME 2.5, a supercomputer deployed at the Tokyo Institute of Technology, we apply machine learning algorithms to recognize patterns about whether the underestimation of job runtime occurs. Our experimental results show that our approach on runtime-underestimation prediction with 80% precision, 70% recall and 74% F1-score on the entirety of a given dataset. Finally, we split the entire job data set into subsets categorized by scientific application name. The best precision, recall and F1-score of subsets on runtime-underestimation prediction achieved 90%, 95% and 92% respectively.



**Keywords:** HPC · Job log analysis
Underestimation on job runtime · Machine learning


---

The original version of this chapter was revised: The affiliation of the second author has been corrected. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-69953-0_17


© The Author(s) 2018


R. Yokota and W. Wu (Eds.): SCFA 2018, LNCS 10776, pp. 179–198, 2018.
https://doi.org/10.1007/978-3-319-69953-0_11

## 1 Introduction

Modern high-performance computing (HPC) systems are built with an increasing number of CPU/GPU cores, memory, and storage space. Meanwhile, scientific applications have been growing in complexity. However, not all users have enough experience working reasonably with supercomputing resources. Writing and executing programs on an HPC system requires more experience and techniques than on a PC. First, HPC users need to have relevant knowledge about system-specific information, such as parallel programming on multi cores or multi nodes in HPC environments, and how many compute nodes or cores are appropriate for a specific application job. Furthermore, when submitting a job to an HPC system, users are usually requested to estimate the runtime of said job for system scheduling. In general, an underestimated runtime will lead to the HPC system terminating the job before its completion. On the other hand, an overestimated runtime of the job usually results in a longer queuing time. In both cases, the productivity of HPC users is hindered [1]. Especially in the case of underestimation, the system will directly terminate the undergoing job when its estimated runtime expires. Users will lose their processing data, and furthermore, 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.

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 those jobs. For instance, if an ongoing task execution of a job is predicted to be *runtime-underestimated* based on the characteristic patterns, system administrators or an automated agent can explicitly send a notification to the user who submitted the job. The user can fix the problem by killing the job and resubmitting it after parameter adjustment.

In this study, we propose a data-driven approach for predicting job statuses on HPC systems. Here, "data-driven" means that our approach actively observes, analyzes, and logs jobs collected on TSUBAME, a large-scale supercomputer deployed at the Tokyo Institute of Technology. Supervised machine learning algorithms (i.e., XGBoost and Random Forest) are applied to address this binary classification problem (having runtime-underestimation or not).

Our experimental results show that our approach predicts the underestimated job with 80% precision, 70% recall and 74% F1-score on the entirety of a given dataset. Then, we split the entire job data set into subsets categorized by scientific application name. The best precision, recall and F1-score of subsets on job runtime-underestimated prediction achieve 90%, 95% and 92% respectively. This achievement means that, for some scientific applications on HPC systems, our model can be used to accurately predict whether a job can be completed before its estimated runtime expires.

Our specific contributions are:

- We introduced some evaluation metrics (precision, recall, and F1-score on minority classes) which are more fair than metrics used in similar previous studies (overall precision, recall, and F1-score).

– Additionally, we plotted feature importance and revealed surprising hidden patterns between different HPC applications and different features on computing resource usage.

The rest of the paper is organized as follows: Related work about gathering and analyzing job logs in HPC systems are introduced in Sect. 2, followed by an overview description of the dataset and feature engineering used for preprocessing the dataset in Sect. 3. Design and implementation of our machine learning-based prediction methods and the evaluation of our approach are described in Sect. 4. In Sect. 5, we presented detailed analysis based on experiment results and discussion. Finally, we gave our future work and conclude the paper in Sect. 6.

## 2 Related Work

Gathering and analyzing job logs in HPC systems is a widely studied topic in computer science literature. In recent years, there have been many studies on analyzing job logs focusing on anomaly detection, failure prediction, runtime prediction, and so on.

Klinkenberg et al. [2] proposed and evaluated a method for predicting failures with framed cluster monitoring data and extracted features describing the characteristic of the signals. Authors in [3] presented a machine learning based Random forests (RF) 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. Tuncer et al. [4] presented a method to detect anomalies and performance variations in HPC and Cloud environments. However, they run kernels representing common HPC workloads and infuse synthetic anomalies to mimic anomalies observed in HPC systems, which may deviate from anomaly situations in reality.

There exists research focusing on predicting other job features, such as I/O, CPU, GPU, memory usage and runtime in clusters. McKenna et al. [5] utilized several machine learning methods (kNN, Decision Tree, and RF) for predicting runtime and I/O usage for HPC jobs with training data from job scripts. Rodrigues et al. [6] predicted job execution, wait time, and memory usage with job logs and batch schedulers by an ensemble of machine learning algorithms such as RF and kNN. Fan et al. [7] proposed an online runtime adjustment framework for trade-off between prediction accuracy and underestimation rate in job runtime estimates.

Additionally, others have worked on log file analysis with machine learning, anomaly detection and so on. In this work, we use system log data collected by ganglia to predict whether a job runtime was underestimated. We found that our method has particularly good results at predicting underestimated runtime for some applications after splitting the entire job data set into subset categorized by scientific application name.

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 runtime-underestimation, ended up affecting HPC applications' runtime.

## 3 Data Collection and Feature Engineering

Our purpose is to model and predict users' runtime-underestimation on job runtime that wastes time and system resources. To address this, we propose a machine learning-based technique which takes advantage of utilization data of different computing resources as well as users' resource usage requirements at job submission. The technique builds classifiers that can recognize hidden patterns in the collected data, which are necessary to understand what jobs are running in the system and the number of resources allocated at each node.

The overall system architecture from data gathering to result prediction is depicted in Fig. 1.

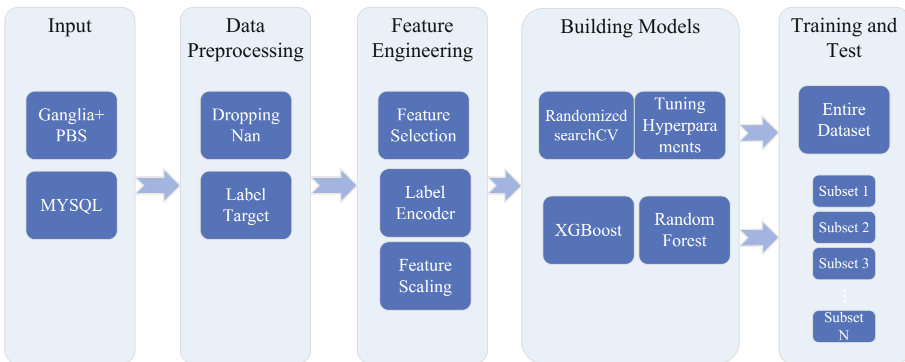


**Fig. 1.** Overall workflow diagram in this study

### 3.1 Gathering TSUBAME DATA

TSUBAME 2.5 [8] is GPU supercomputer located at Tokyo Institute of Technology, operated from November 2010 to July 2017, including its ancestor TSUBAME 2.0. TSUBAME 2.5 is well known as "the greenest supercomputer in the world" in the Green500 List [9] on 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 SSD as local scratch storage. The nodes are interconnected with dual-rail InfiniBand QDR full fat-tree network. All nodes run SUSE Enterprise Linux 11 and compute jobs are managed by PBS Professional 11. System load (and power) information, including GPU usage, is monitored and recorded using Ganglia [10]. All nodes process information is also recorded via process accounting interface in Linux.

We created MySQL database containing anonymized job history data from PBS's log, associated CPU and GPU usage information as features from Ganglia, and application information from accounting logs of each job. Prediction of

online runtime underestimation needs to be based on the progress information, rather than the post-processing of measured information after jobs are finished. All features regarding computing resource usage were normalized by dividing by used wall clock timings. This provides progress information in the form of a ratio – resource usage related measurements by wall clock units. As they are normalized by time-based value, those normalized performance related measurements serve as appropriate data sources for machine learning based job status prediction. Indeed, this can also be extended to online prediction [3]. In total, 14.3 million jobs were recorded, with a total database size of 8.5 GiB.

Table 1 shows the really world data we collected by Ganglia and PBS from TSUBAME 2.5.

**Table 1.** List of computing resource usage features based on normalized time series data and job requests information

| Features | Description | Features | Description |
|---|---|---|---|
| used_cpupercent | Recorded CPU usage | req_pl | Requested priority |
| used_mem | Recorded memory usage | req_et | Requested option to extend maximum runtime |
| used_ncpus | Recorded number of CPU used | nhosts | Calculated number of host involved |
| used_vmem | Recorded memory address space usage | is_array | 1 if the job is a part of array (parameter survey) job |
| req_mem | Requested memory amount | gpu_utilization | Recorded average GPU utilization per node (0.0–3.0) |
| req_ncpus | Recorded number of CPU used | num_gpu_used | Number of GPU per node which is actually used |
| req_walltime | Requested runtime (wallclock time) | app | Recorded application which run inside of the job |
| req_gpus | Requested number of GPUs per node | grouphash | Anonymized project name |
| used_walltime | Recorded runtime (wallclock time) | userhash | Anonymized user name |
| 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 |
| used_nodesec | Used runtime on per node | used_cputime | Used runtime per CPU |
| queue | Job class name | | |

## 3.2 Feature Engineering

The purpose of a feature, other than being an attribute, would be much easier to understand in the context of a problem. A feature is a characteristic that might

help when solving the problem [11]. Features describe the structures inherent in data, and furthermore, they are very important to the predictive models and will influence the result. The quality and quantity of features have direct impact on whether the 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.

**Feature Selection.** From the previous sections, we know that the raw data about compute resources usage was time series data of extreme size. Directly using raw time series data will produce unacceptable compute overhead, which may lead to serious time gaps between data collection and analysis as well as wasted computational resource. Instead of using raw time series data, we selected a set of relevant features from the raw job logs data for use in model construction by normalizing and converting them to MySQL database. In machine learning tasks, this is an essential step to make results easier to interpret by researchers. Additionally one can enjoy shorter training times, avoid the curse of dimensionality and enhanced generalization by reducing overfitting [12].

In this research, our purpose is building a machine learning technique-based model that can predict whether a job is underestimated on its runtime. Therefore we selected features as training set **X** by removing redundant or irrelevant features such as *used_cputime*, *used_nodesec*, *used_walltime*, *queue_time*, *start_time* and *end_time* without incurring much loss of information. This is a preliminary study in which we try to reveal complex patterns hidden in utilization of computing resources, user behaviors, and different applications on an HPC system. Those features are redundant, which have a large impact on the prediction of job runtime.

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

$$y' = j.used_walltime - j.req_walltime$$

where j.used_walltime is actual runtime of a job, and j.req_walltime is user estimated time of a job. If $y' < 0$, we label this job as 0 in the test set y, which means that the actual runtime of this job does not exceed the user's estimated time when its user submitted it. Relatively, if $y' >= 0$, this job will be labeled as 1 in the test set, which indicates runtime-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 runtime-underestimated after job submission.

**Feature Preprocessing.** 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 Labelencoding. For most traditional machine learning algorithms, the data fed to them must be in numerical type. Based on Table 1, however, we can see that there are some feature columns that are non-numerical type. For instance,

**Table 2.** 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 |

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. Labelencoder can also be used to transform non-numerical variables (as long as they are hashable and comparable) to numerical variables. For example, LabelEncodeing can turn [G, S, G, H, S] 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.

Second is feature standardization. Based on Table 1, we can see that the range of values of 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). Given this wide variation of training set values in some machine learning algorithms, objective functions will not work properly without normalization. For example, most of classifiers calculate the distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. Therefore, the range of all features should be feature scaled so that each feature contributes approximately proportionately to the final distance.

Feature standardization can make the values of each feature in the data have zero-mean (when subtracting the mean in the numerator) and unit-variance. This method is widely used for normalization in many machine learning algorithms (e.g., SVM, logistic regression, and neural networks). The general method of calculation is to determine the distribution mean and standard deviation for each feature. Next we subtract the mean from each feature. Then we divide the values of each feature by its standard deviation (since mean is already subtracted) [13], which can be

presented in the following formula:

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

Where $x$ is the original feature vector, $\bar{x}$ is the mean of that feature vector, and $\sigma$ is its standard deviation. We give 5 job instances - with 18 selected features which as Table 2 showed.

One important thing to note, researchers usually split the data into training, validation, and testing sets in the training phase. If we perform feature scaling to take the mean and variance (or standard deviation) over the whole set of predictor variables, future information will be introduced into the training predictor variables; namely, the future information contained in the mean and variance. Therefore, we perform feature scaling over the training data and save the mean and variance. Then we apply feature scaling to the predictor variables of the validation and test data sets, using the training mean and variances. A model can be applied on unseen data which, in general, is not available at the time the model is built. The validation process (including data splitting) simulates this. In order to get a good estimate of the model quality (and generalization power), one needs to restrict the calculation of the normalization parameters (mean and variance) to the training set.

Finally, for data cleaning 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 one million job instances, which are enough for training and testing our models. Table 3 shows the summary of the entire job data set and subsets categorized by scientific application name. In Table 3, 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 runtime-underestimation) are less than 1%, 1.45% and 2.2% respectively on those subsets. The majority class (labeled 0, not having runtime-underestimation) occupies overall 94.82% over the entire data set.

## 4 Performance Metrics and Algorithm Coverage for Binary Classification Problem on Imbalanced Dataset

So far, we have realized very clearly that we are dealing with a binary classification problem on an extremely imbalanced dataset. Almost all classifications that will predict every sample as the majority class can still achieve very high performance [14]. 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 data, the overall classification accuracy is often not an appropriate metric of performance. There are 2 ways that are given by data scientists and researchers to deal with imbalanced

**Table 3.** Imbalanced subsets categorized by scientific application name

| Name of dataset | Number of instance | Number of instance with label 0 (majority class percentage) | Number of instance with label 1 (minority class percentage) |
|---|---|---|---|
| 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%) |

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.

### 4.1 Metrics for Evaluating Imbalanced Data

In machine learning tasks with extremely imbalanced datasets, we use a set of alternative metrics such as false positive rate (FPR), true positive rate (TPR), receiver operating characteristic (ROC), Area under the Curve of ROC (AUC), precision, recall, and F1-score to evaluate the performance of our model on imbalanced data:

True Positives (TP): the true positive are the cases when the actual class of the target label was 1 (True) and the predicted is also 1 (True). In this research, the case where a job is actually runtime-underestimated (1) and the model classifies the case as runtime-underestimated (1) falls under True Positives.
True Negatives (TN): the true negative are the cases when the actual class of the target label was 0 (False) and the predicted is also 0 (False). In this research, the case where a job is NOT runtime-underestimated and the model classifies the case as NOT runtime-underestimated falls under True Negatives.
False Positives (FP): the false positive are the cases when the actual class of the target label was 0 (False) and the predicted is 1 (True). In this research, the case where a job is NOT runtime-underestimated and the model classifies the case as runtime-underestimated comes under False Positives.
False Negatives (FN): the false negative are the cases when the actual class of the target label was 1 (True) and the predicted is 0 (False). In this research, the case where there is a runtime-underestimated job and the model classifies the case as NOT runtime-underestimated comes under False Negatives.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$\text{F1-score} = \frac{2 \times Precision \times \text{Rec}all}{Precision + \text{Rec}all} \tag{4}$$

$$FPR = \frac{FP}{FP + TN} \tag{5}$$

$$TPR = \frac{TP}{TP + FN} \tag{6}$$

$$SPC = \frac{TN}{TN + FP} \tag{7}$$

The ROC is a kind of curve graph that represents the diagnostic ability for a binary classification problem with all possible threshold values. ROC can be drawn with coordinates ranging between FPR and TPR along the x and y axes. Adjusting the threshold will change the FPR and TPR. In a binary classification problem, the prediction result for each sample is usually made based on a continuous random variable $X$, which is a "score" computed for this sample. Setting a threshold $T$, the sample will be classified as "positive" if $X > T$, and "negative" otherwise.

The AUC it indicates the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one (assuming 'positive' ranks higher than 'negative') [15]. The AUC is a single metric which can be used for an overall performance summary of a classifier, calculated by following formula:

$$\begin{aligned} AUC &= \int_{\infty}^{-\infty} TPR(T)(-FPR'(T))dT \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} I(T' > T) f_1(T') f_0(T) dT' dT = P(X_1 > X_0) \end{aligned} \tag{8}$$

where $X_1$ is the score for a positive instance and $X_0$ is the score for a negative instance, and $f_0$ is the probability density when the sample actually belongs to class "positive", and $f_1$ otherwise [16].

Due to space limitations, we will not describe it in detail here. What we need to know about AUC are as follows: The range of the value of AUC is between 0 and 1, the higher the better; When AUC is 1, this means that it is a perfect classifier, and with this prediction classifier, there is at least one threshold that leads to a perfect prediction (no FP and FN). However, there is no perfect classifier in most real world cases. $0.5 < \text{AUC} < 1$ means that the performance of this model is better in cases of a random guess. If the AUC is around 0.5, that means the performance of this model is generally the same as the result of a random guess.

The AUC was the first metric used to evaluate the overall accuracy performance of a classifier in the evaluation stage. After the best classifiers were chosen with the AUC, we used ROC to trade off precision vs recall in the minority class, because the majority class always has very high scores on all metrics in extremely imbalanced datasets. F1-score was a useful metric as we desired harmonic average of the precision and recall.

In all classifiers, a trade off will always occur between true negative rate (SPC, specificity) and true positive rate (TPR). The same occurs with precision and recall. In our study, we hope to train a classifier that gives high precision over the minority class (label 1, a job having runtime-underestimation), while maintaining reasonable precision and recall for the majority class. In the case of modeling on 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.

### 4.2 Machine Learning Algorithms for Imbalanced Data

In this study, we compare two popular supervised machine learning algorithms: Random Forest (RF) [17] and XGBoost [18]. XGBoost is a scalable tree boosting system that implements the gradient boosting decision tree algorithm, which is widely used by data scientists and provides state-of-the-art results on many problems. The reason we chose to compare these two algorithms is that there are 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, kNN in data science [4,14,18–21].

## 5 Experiment Results and Analysis

Since we split the entire job data set into subsets, there are some subsets in which the absolute number of minority class samples is too small. Therefore, we use the leave-one-out cross-validation (LOOCV) in our work [22]. The LOOCV method keeps a certain percentage of the full data set as a test set, then the rest of the data is used to perform k-fold cross-validation (k-fold CV). Next, it records $k$ scores and calculates the standard deviation (std) of $k$ scores as reference for choosing the best classifier from them. At the same time, it evaluates the robustness of the model. The final performance score of this model can be obtained from using the best-chosen classifier to predict the test set.

Meanwhile, most machine learning algorithms have several hyperparameters that will affect a model's performance. Tuning hyperparameters 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% [21]. In this study, we train our model and tune hyperparameters via LOOCV with the RandomizedSearchCV function from scikit-learn [23]. The RandomizedSearchCV is an estimator used to optimizing hyperparameters from parameter settings. In contrast to GridSearchCV, not all parameter values are attempted, but rather a fixed number of parameter settings is sampled from the specified distributions. We set 30% of the each dataset as the test set with a random state, n_iter to 50, and we also set AUC as the scoring metric in RandomizedSearchCV. Parameter settings and optimized parameters are presented in Table 4.

### 5.1 Classification with Entire Dataset

We trained and tuned classifiers with the XGBoost and the RF on the entire dataset. 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). Tables 5 and 6 shows that the XGBoost and the RF have an extremely similar overall performance result. The result consist of similar values of runtime-underestimation prediction

**Table 4.** Hyperparameters settings of Random Forest, XGBoost and the best parameters after tuning for our study

| Algorithm | Hyperparameters | Best parameters after tuning |
|---|---|---|
| Random Forest | *n_estimators:* Number of decision trees in the ensemble | n_estimators = 500 |
| | *min_weight_fraction_leaf:* The minimum number of (weighted) samples for a node to be considered a leaf. Controls the depth and complexity of the decision trees | max_features = 2 |
| | *maxfeatures:* Number of features to consider when computing the best node split | criterion = "entropy" |
| | *criterion:* Function used to measure the quality of a split | min_weight_fraction_leaf = 8 |
| XGBoost | *n_estimators:* Number of decision trees in the ensemble | n_estimators = 500 |
| | *learningrate:* Shrinks the contribution of each successive decision tree in the ensemble | learning rate = 0.8 |
| | *maxdepth:* Maximum depth of the decision trees. Controls the complexity of the decision trees | max depth = 3 |
| | *max_delta_step:* Set it to a finite number (say 1) will help convergence | max_delta_step = 2 |
| | *maxfeatures:* Number of features to consider when computing the best node split | max features = "log2" |

(in terms of overall precision, recall, and F1-score) in the entire dataset. As we estimated, the precision, recall, and F1-score 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 (e.g. F1-score: 0.74 vs 0.99). However, the overall average of precision, recall, and F1-score 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, and F1-score on the minority class are fairer metrics than those of the majority class when evaluating model performance.

**Table 5.** 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 6.** 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) |

### 5.2 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 [1–3].

In this research, our main purpose is predicting whether a job may or may not finish before its runtime estimated by its user. The runtime 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 runtime of HPC applications. According to Table 3, 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). We used XGBoost and RF to build prediction models with the optimized hyperparameters presented in Table 4 and run them through on each subset by 5-fold LOOCV respectively. The performance evaluation results including AUC, precision and recall on the minority class were plotted in Figs. 2 and 3.

Figure 2 shows the AUC and the standard deviation (std) of the AUC by 5-fold LOOCV for 26 subsets after taking "others" as a subset and removing "RISM", "CAE: MSC", from all training dataset. This was because there is no instance of runtime-underestimation (labeled 1, minority class) in their subsets. The AUC (XGBoost) was chosen as an indicator to sort the results in descending order for observation and analysis purposes. We can see that the XGBoost outperforms or tie

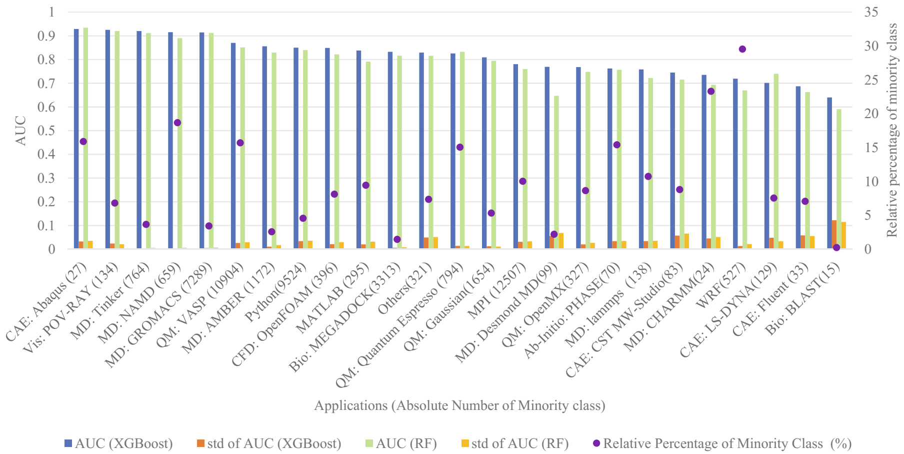


**Fig. 2.** The AUC and its STD after running through subsets with 5-fold LOOCV

with RF slightly in most application subsets with the AUC as the indicator, except application named "CAE: LS-DYNA". The std of AUC show the model stability; the smaller the std is, the more stable the model's performance is. The percentage of minority class of each application was also plotted in Fig. 2. We can see that, for most of cases in this study, the percentage of minority class almost has no impact on the AUC and the std of the AUC. However, we found that, the higher the absolute number of minority class is, the more stable the model is relatively. We believe that the high std of AUC in some subsets is due to the low absolute number of minority class. The AUC shows the overall performance of models. We can see that both algorithms achieved very good AUC on 5 subsets including "CAE: Abaqus", "Vis: POV-RAY", "MD: Tinker", "MD: NAMD" and "MD: GROMACS". Except for "MD: NAMD" by RF, the AUC in the rest of 4 subsets are greater than or equal to 0.9, which means that both algorithms provide very good prediction about runtime-underestimation for those 5 applications in the HPC environment. In contrast to "CAE:Abagus" and "Vis: POV-RAY", the results of "Bio:BLAST" by both 2 algorithms are the worst in all subsets. Since in "Bio:BLAST" subset, the absolute number (15) and the relative percentage (0.45%) on minority class are much lower than those on other subsets, our models cannot handle with this kind of problem. The "CAE:FLUENT" has similar result with "Bio:BLAST", because of its absolute number (33) on minority class is also very low. But its std of AUC is better than "Bio:BLAST", due to its relative percentage on minority class is higher than "Bio:BLAST".

In Fig. 3, we used best-chosen classifier from 5-fold LOOCV to plot precision and recall in minority class on all subsets, which follows the sorting in Fig. 2. Taking stable, precision, recall and F1-score into consideration together, we think that "Vis: POV-RAY" achieves the best result on minority class by XGBoost (90% precision, 95% recall, 92% F1-score). This figure helps to find out which algorithms

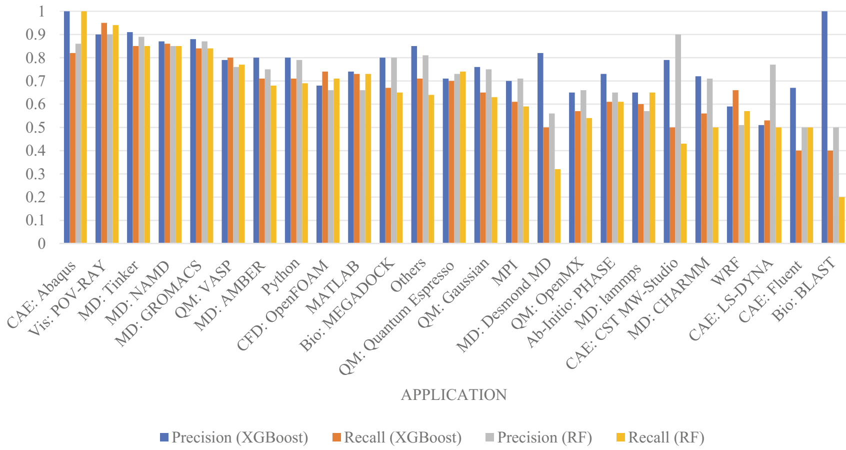


**Fig. 3.** Precision and recall on minority class after running through XGBoost and Random Forest

is good at which metric. For example, if we need the best recall on subset "Vis: POV-RAY", XGBoost will be the best selection to build model.

If we want our model to provide the best precision for "CAE: LS-DYNA", RF should be chosen to build model. In this research, from the user's point of view, the precision is more important than the recall, due to the FP is more important than the FN in the job runtime-underestimated prediction. Since the FP can be much more costly than FN. On the contrary, if looking at the angle of HPC system administrators for saving system resources as much as possible, the recall will be more critical than FP.

Figure 4 represent ROC, AUC and std of AUC after 5-fold CV on "CAE:Abagus" and "Bio: BLAST" by XGBoost. Adjusting the threshold will change the FPR. For instance, increasing the threshold will decrease FP (and increase FN), which corresponds to moving in the left direction on the curve. The curve is more

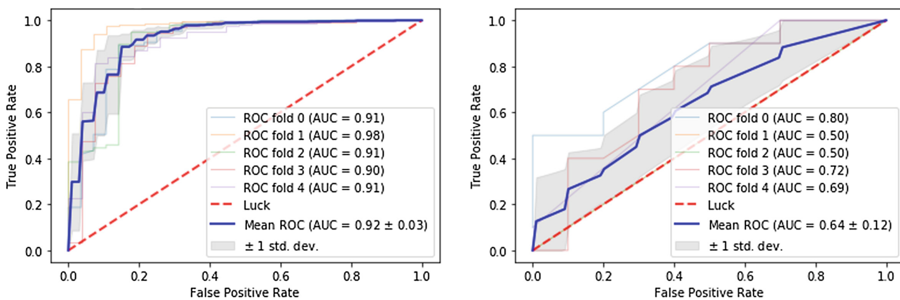


**Fig. 4.** ROC, AUC and standard deviation after 5-fold CV on subsets "CAE:Abagus" (left) and "Bio: BLAST" (right) by XGBoost

inclined to the upper left corner $(0, 1)$, where the performance of the model is better at distinguishing positive and negative classes. Adjusting the threshold on ROC will be the last step to improve the performance of a model.

## 5.3 Feature Importance

Feature importance gives a score (F score) for indicating how valuable or useful each feature was when building boosted decision trees based models. With the features

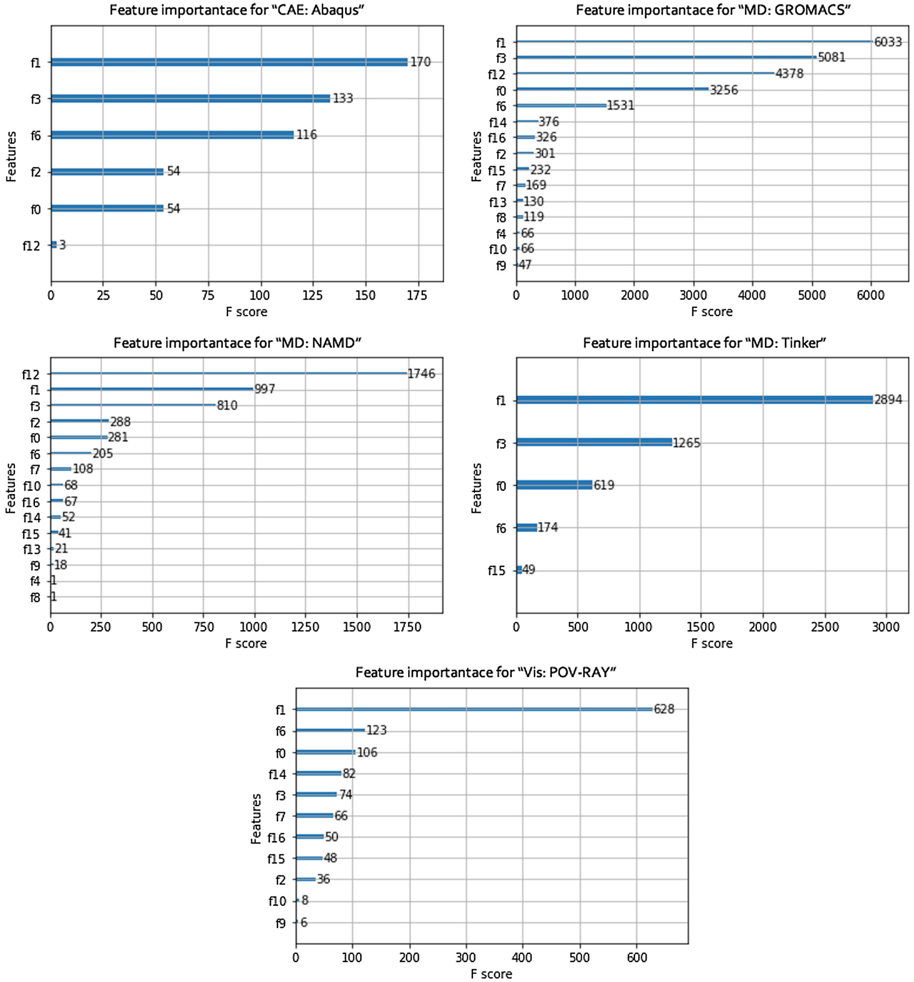


**Fig. 5.** Important features for different applications; features are automatically named according to their index, f0: *used_cpupercent*, f1: *used_mem*, f2: *used_ncpus*, f3: *used_vmem*, f4: *req_mem*, f5: *req_ncpus*, f6: *req_walltime*, f7: *req_gpus*, f8: *req_pl*, f9: *req_et*, f10:*nhosts*, f11: *is_array*, f12: *gpu_utilization*, f13: *num_gpu_used*, f14: *group*, f15: *queue* and f16: *user*, from f0 to f16 respectively

sorted according to how many times they appear, the more a feature was used to make key decisions within the decision trees, the higher its relative importance was to the model.

In our study, we plotted feature importance in the top 5 AUC indicated subsets with the features ordered according to how many score they have (how important it was) in Fig. 5. We can see that *used_mem*, *used_vmem*, *used_cpupercent*, *req_walltime* and *gpu_utilization* are the most important features in those applications. However, applications have different weights (namely prediction of runtime-underestimation) on different features (namely computing resource usage), which both affected job runtime. Our method recognized these patterns and used them to predict job runtime-underestimation in HPC systems.

### 5.4 Discussion

Papers [2–5] demonstrate related research, such as job status prediction, failure prediction and anomaly detection, based on log file analysis with machine learning with good results. Whether abnormal detection or job status prediction, 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 dataset presented here. However, in those works, authors used the overall accuracy, precision, recall, and F1-score to evaluate the model performance without considering those of on the minority class. As we explained in this paper, because of the imbalanced absolute number and relative percentage of the majority classes and the minority classes (the minority class will be more than 1 in multi-classification problems), the overall metrics cannot accurately reflect the predictions of minority class. 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, and F1-score on minority classes, rather than overall, is a promising metric for future work.

## 6 Conclusions and Future Work

Predicting whether a job is runtime-underestimated after job submission can greatly benefit HPC users and system administrators. In this study, we built a machine learning based model to mine patterns hidden in HPC job logs for predicting runtime-underestimation. Additionally, we introduced some evaluation metrics (precision, recall, and F1-score on minority classes) which are more fair than metrics used in similar previous studies (overall precision, recall, and F1-score). We split our dataset into subsets, and found that the best precision, recall, and F1-score of subsets on job runtime-underestimated prediction (minority class) achieved 90%, 95% and 92% respectively. These results outperform some recent related studies to date. Finally, we plotted feature importance and revealed surprising hidden patterns between different HPC applications and different features on computing resource usage.

As future work, we would like to improve prediction by extracting more features such as the network traffic I/O, the standard deviation of computing resource usage etc. which may affect the prediction performance. Also, we would like to do more test with data collected from different time periods to prove and improve the robustness of our model.

**Acknowledgment.** This work was supported by JST CREST Grant Number JPMJCR1303 and JPMJCR1687, Japan. This work was partially conducted as research activities of AIST - Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory (RWBC-OIL).

## References

1. Zhang, H., You, H., Hadri, B., Fahey, M.: 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), the Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), p. 1 (2012)
2. Klinkenberg, J., Terboven, C., Lankes, S., Müller, M.S.: Data mining-based analysis of HPC center operations. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 766–773. IEEE (2017)
3. Yoo, W., Sim, A., Wu, K.: Machine learning based job status prediction in scientific clusters. In: SAI Computing Conference (SAI), pp. 44–53. IEEE (2016)
4. Tuncer, O., Ates, E., Zhang, Y., Turk, A., Brandt, J., Leung, V.J., Egele, M., Coskun, A.K.: Diagnosing performance variations in HPC applications using machine learning. In: Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D. (eds.) ISC 2017. LNCS, vol. 10266, pp. 355–373. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58667-0_19
5. McKenna, R., Herbein, S., Moody, A., Gamblin, T., Taufer, M.: Machine learning predictions of runtime and IO traffic on high-end clusters. In: 2016 IEEE International Conference on Cluster Computing (CLUSTER), pp. 255–258. IEEE (2016)
6. Rodrigues, E.R., Cunha, R.L., Netto, M.A., Spriggs, M.: Helping HPC users specify job memory requirements via machine learning. In: Proceedings of the Third International Workshop on HPC User Support Tools, pp. 6–13. IEEE Press (2016)
7. Fan, Y., Rich, P., Allcock, W.E., Papka, M.E., Lan, Z.: Trade-off between prediction accuracy and underestimation rate in job runtime estimates. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 530–540. IEEE (2017)
8. Matsuoka, S.: The TSUBAME 2.5 evolution. TSUBAME e-Sci. J. **10**, 2–8 (2013)
9. Feng, W., Cameron, K.: The Green500 list: encouraging sustainable supercomputing. Computer **40**(12), 50–55 (2007)
10. Massie, M.L., Chun, B.N., Culler, D.E.: The ganglia distributed monitoring system: design, implementation, and experience. Parallel Comput. **30**(7), 817–840 (2004)
11. Brownlee, J.: Discover feature engineering, how to engineer features and how to get good at it. Machine Learning Process (2014)
12. Bermingham, M.L., Pong-Wong, R., Spiliopoulou, A., Hayward, C., Rudan, I., Campbell, H., Wright, A.F., Wilson, J.F., Agakov, F., Navarro, P., et al.: Application of high-dimensional feature selection: evaluation for genomic prediction in man. Sci. Rep. **5**, 1–12 (2015)

13. Grus, J.: Data Science from Scratch: First Principles with Python. O'Reilly Media, Inc., Sebastopol (2015)
14. Chen, C., Liaw, A., Breiman, L.: Using random forest to learn imbalanced data, vol. 110. University of California, Berkeley (2004)
15. Fawcett, T.: An introduction to ROC analysis. Pattern Recogn. Lett. **27**(8), 861–874 (2006)
16. Powers, D.M.: Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation (2011)
17. Liaw, A., Wiener, M., et al.: Classification and regression by randomforest. R News **2**(3), 18–22 (2002)
18. Chen, T., Guestrin, C.: XGBoost: a scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785–794. ACM (2016)
19. Song, R., Chen, S., Deng, B., Li, L.: eXtreme gradient boosting for identifying individual users across different digital devices. In: Cui, B., Zhang, N., Xu, J., Lian, X., Liu, D. (eds.) WAIM 2016. LNCS, vol. 9658, pp. 43–54. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39937-9_4
20. Nielsen, D.: Tree boosting with XGBoost-why does XGBoost win "every" machine learning competition? Master's thesis, NTNU (2016)
21. Olson, R.S., La Cava, W., Mustahsan, Z., Varik, A., Moore, J.H.: Data-driven advice for applying machine learning to bioinformatics problems. arXiv preprint arXiv:1708.05070 (2017)
22. Cawley, G.C., Talbot, N.L.: Efficient leave-one-out cross-validation of kernel fisher discriminant classifiers. Pattern Recogn. **36**(11), 2585–2592 (2003)
23. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.