Jubatus: Scalable Distributed Processing Framework for Realtime Analysis of Big Data

Satoshi Oda, Kota Uenishi, and Shingo Kinoshita†

Abstract

This article describes a distributed machine learning framework called Jubatus for deep realtime analysis of big data that we are jointly developing with Preferred Infrastructure Corporation. The main challenge for Jubatus is to achieve scalable distributed computing for profound analytical processing such as online machine learning algorithms and provide a common framework for different supported algorithms. After giving an overview of Jubatus, this article focuses on a key mechanism called *mix*, which is a new synchronization method among servers to scale out online machine learning algorithms, which are potentially difficult to distribute and parallelize. It synchronizes servers loosely and has a relaxed consistency to the extent allowed by the performance and learning accuracy requirements. This article also evaluates performances such as throughput and scalability and verifies the degree to which the consistency requirement is relaxed.

1. Introduction

With large quantities of data of various types being produced, distributed, and shared around the world, it is becoming increasingly important for businesses worldwide to acquire new knowledge from data such as people's behaviors, system operational logs, and environmental information obtained through their business activities. This huge amount of data is known as *big data*. Much of it is unstructured data that is not stored in databases, and in the past it was too large to analyze and was therefore discarded. However, the analysis of big data at a low cost has recently been facilitated by open-source software systems such as Hadoop [1], which allow distributed computing across clusters of inexpensive commodity computers. This has made it possible to perform analysis within realistic time limits and obtain useful insight from the analysis.

This kind of analytical processing is based on batch processing in which data has been temporarily stored and processed as a batch. As analytical methods, it utilizes statistical analysis, such as summation, and machine learning. In addition, a machine learning library called Mahout running on Hadoop has been developed; it enables batch-type sophisticated analysis of big data in a scalable manner. There is increasing need for realtime capabilities, prompted by demonstrations of the validity of such large-scale data analysis [2].

One of the technologies with realtime capabilities is online stream-processing. It supports push-type analysis methods in which analysis results are calculated incrementally as soon as data arrives without the data being stored in a database, instead of the conventional pull-type analysis performed on a database after data has been stored. Representative examples of such online stream-processing systems are IBM InfoSphere Streams, Oracle CEP, StreamBase, Sybase Event Stream Processor, and Truviso. Each of these is a commercial product with a full lineup of capabilities, functions, development environments, and operating tools. They are already in use, including use in mission-critical areas such as finance, communications, the military, and medicine. However,

† NTT Software Innovation Center
Musashino-shi, 180-8585 Japan

these deployments are based on the assumption that scaling up requires expensive hardware and they do not support sophisticated analysis functions such as machine learning but only simple statistical functions.

Therefore, for further business success, it is becoming more important to enable more sophisticated and timely analysis of big data at a low cost. For this, the key challenge is to create a scalable distributed computing framework across clusters of inexpensive commodity computers for realtime and profound analytical processing using online machine learning algorithms.

2. Jubatus

To address this challenge, NTT Software Innovation Center and Preferred Infrastructure Corporation have cooperated in the development of Jubatus [3], [4] since 2011. Jubatus is a distributed computing framework for realtime and profound analytical processing using online machine learning algorithms, rather than the batch processing provided by Mahout/Hadoop and the simple statistics processing provided by online stream-processing systems. It has scale-up capability whereby the performance of the system increases linearly with the addition of inexpensive commodity computers.

To achieve this scalability, Jubatus must distribute online machine learning processing among many computers and synchronize their learning results. An iterative parameter mixture [5], [6] has been found to be effective as an algorithm for the synchronization of this distributed processing [7]. Jubatus has been modified to use this algorithm to achieve realtime processing.

Among various types of online stream-processing, two types of machine learning—linear classification and linear regression—were initially implemented for Jubatus because they are basic analytical functions and have a much broader range of applications. To support these types of machine learning, Jubatus is aimed at stateful stream-processing, where the status of a stream-processing node can be updated in accordance with the content of arriving data. To enable application programs to be developed more efficiently, Jubatus supports a full range of feature conversion functions that facilitate the conversion of unstructured data into a format that can be used in machine learning [8].

2.1 Current position and roadmap

Jubatus is intended to be a framework for realtime analytical processing. However, the initial version applies only a few online machine learning algorithms, such as linear classification and regression algorithms, to the distributed computing environment by using an iterative parameter mixture, so it is not yet ready to be provided as a full-range framework. This is because, although machine learning is generally taken to be a problem of numerical optimization, it is not obvious how to design generic forms that will enable distributed processing, in other words, frameworks.

In the future, the challenge of designing a suitable framework will be to extract generic calculation models in a similar way to MapReduce by applying more machine learning algorithms and prototyping applications aimed at real-life tasks.

2.2 Distributed processing

2.2.1 Overview

Jubatus basically achieves realtime analysis of a huge amount of data that cannot be processed by a single server by distributing the data over multiple servers. In machine learning, there are two types of processing: training and prediction. In the training process, the model data is updated by a machine learning algorithm such as a passive-aggressive (PA) algorithm using supervised data. In the prediction process, data that a server has received is processed using the model data and this enables the prediction to be performed. For example, in the case of classification algorithms, the input data is predictively classified into several specified groups on the basis of the model data.

In the training process, all the servers involved in the distributed processing initially have the same model data, but the model data varies as the individual training proceeds. With an iterative parameter mixture, if training is done with a given amount of data at each server, the model data of all servers is integrated into a new set of model data. This new model data is shared among servers again and each server then performs training individually.

As Jubatus is based on online processing, the servers collaborate in synchronizing the model data when either of the following conditions is satisfied: any one server has trained a given number of data items or a specific given time has elapsed. This synchronization *mix* enables training results to be shared while all the servers are performing the training in parallel. Since the training and prediction traffic is distributed to

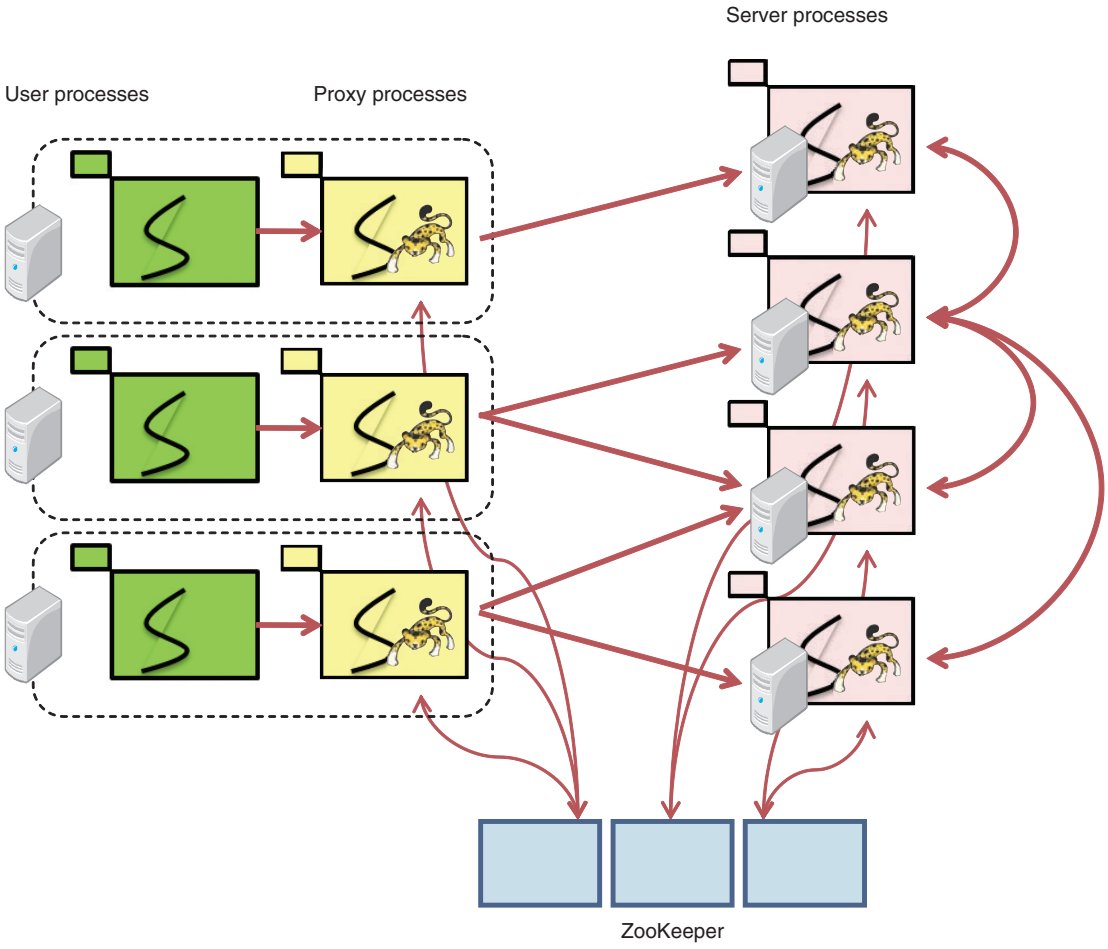


Fig. 1. System architecture of Jubatus.

individual servers, it is possible to scale the throughput in accordance with the number of servers and the amount of calculation resources, while maintaining the response time. The distributed processing architecture of Jubatus is discussed below.

2.2.2 Basic architecture and membership management

The Jubatus system consists of server processes that perform feature conversion and machine learning processing (training and prediction), proxy processes that allocate requests from clients to servers, ZooKeeper* [9] processes, and user processes that clients have assembled, as shown in **Fig. 1**. Note that if there is only one server process, a user process can access it directly without going through a proxy process.

A server process, with the name jubaclassifier or jubaregression in the Jubatus system, maintains model data and performs the training and prediction processing. The system implements horizontal scalability in response to the load by deploying a number of these server processes. Each server process registers its own identity (ID), e.g., its Internet protocol (IP) address and listening port number, in ZooKeeper. ZooKeeper exchanges KeepAlive messages with the server processes at regular intervals. If a server process stops for some reason, such as a hardware failure, ZooKeeper detects that and automatically deletes its registered ID.

A user process is a process executed by a user program of Jubatus. It may collect data from other systems, request server processes to train and predict the data, and receive the prediction results. When the user process is used by a web application, it is assumed to be a web server process such as Apache.

A proxy process, called jubakeeper in the Jubatus system, relays communications transparently from a

* ZooKeeper is a trademark of The Apache Software Foundation.

user process to a server process. It receives a request from a client, selects a suitable server from the list of IDs registered in ZooKeeper, and transfers the request to that server. The client can therefore execute the request without being aware of either the server process that is operating or any increase or decrease in the number of server processes.

Since the remote procedure calls of Jubatus are performed by synchronous communications, an environment is created in which the best performance occurs when the total number of user process threads is greater than the total number of proxy process threads, which is greater than the total number of server process threads.

While enabling scale-out, this distributed configuration also ensures that even if part of an individual process is stopped, the performance (and precision, in some cases) will deteriorate temporarily, but the overall system will not halt; thus, this configuration corresponds to a distributed system with no single point of failure.

2.3 Synchronization method: *mix*

Jubatus has an extension of the iterative parameter mixture, which is called *mix*. This concept is unique to Jubatus. An iterative parameter mixture is a method of machine learning in which all of the model data trained in the servers is collected together and averaged and then shared again for further training. This is regarded as a problem of replication when data updated by an individual server is synchronized by all of the servers. In other words, from the data management viewpoint, the training, average calculations, and predictions in machine learning are equivalent to the updating, synchronization, and reading of data, respectively.

In an ordinary database system, the traditional requirement is to satisfy *atomicity, consistency, isolation, and durability* (ACID) [10]. In other words, updated data is always synchronized, even in a distributed environment, and reading must be enabled from the instant that the update was successful. However, it is difficult to implement a distributed data management system that satisfies strict ACID properties because of the constraint called the CAP theorem (C: consistency, A: availability, P: partition tolerance) [11]. Recent distributed storage techniques have been able to implement practicable performance in a range in which this constraint is satisfied. They achieve this by defining a behavior called *basically available, soft state, eventually consistent* (BASE), in which mainly the consistency (C) part of the constraint is relaxed. In BASE, it is sufficient to have matching databases in which all of the data updates are eventually implemented. This consistency model is called Eventual Consistency [12].

In the field of numerical optimization problems such as statistical machine learning, deterioration in the consistency of data synchronization is considered to be a loss of data to be trained, and it results in a decrease in accuracy. Conversely, by applying the scale-out approach, which enables a large amount of data to be processed in parallel, it is possible to use enough training data to overcome that loss and preserve realistic accuracy while increasing performance. Focusing on this point, we devised a data synchronous algorithm called *mix* which is of a form that has an even more relaxed consistency requirement. The *mix* in linear classifiers currently implemented in Jubatus is described below.

With an iterative parameter mixture, once training with a given quantity of data is complete, the system calculates the average of the model data in all of the servers and uses it as an initial value for model data in the next phase of the training. In other words, each phase is partitioned in accordance with the data size.

With Jubatus, servers do not necessarily all process equal amounts of training data, so each phase is partitioned by either data quantity or time, as mentioned in section 2.2.1. Each phase ends when any of the conditions has been satisfied, and the *mix* starts. More specifically, the sequence (shown in **Fig. 2**) is as follows.

(1) The server process that started the *mix* acquires a lock on ZooKeeper and becomes the master.
(2) It acquires the server process list from ZooKeeper and receives model data that will become the subject of the *mix* from all the processes.
(3) It performs synchronization, e.g., calculates the average, with respect to the received data.
(4) It distributes the synchronized data to all the processes.
(5) Each server process updates its own model data with the synchronized data.
(6) The master that has confirmed the update releases the lock on ZooKeeper.

In the case of a batch-type iterative parameter mixture, there would be no loss of training data because no training is done during steps (2) to (5), but with the Jubatus *mix*, training is performed during this time. This is because Jubatus is designed to increase accuracy during this time because it supports online-type realtime processing in which training and prediction requests arrive continuously during the time and must

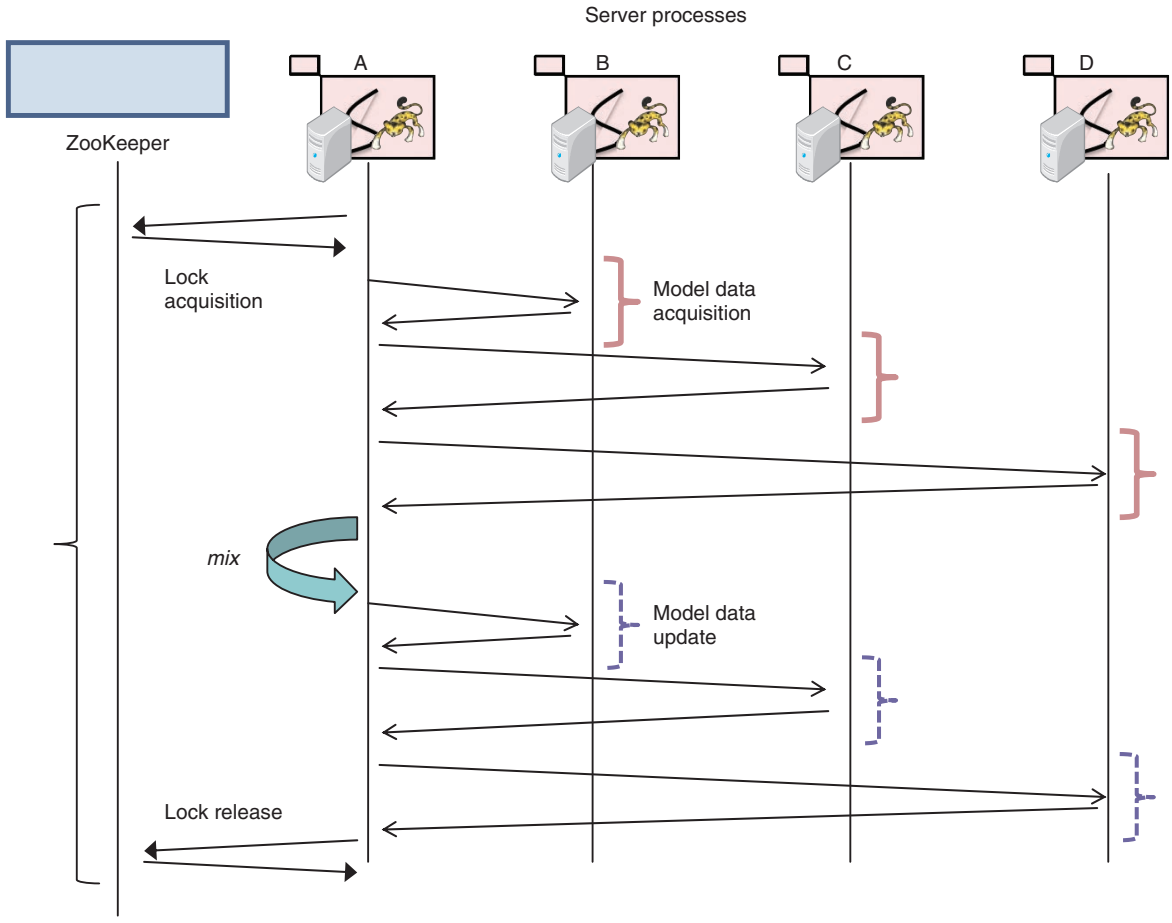


Fig. 2. Sequence of *mix*.

be responded to in real time.

Jubatus also provides a further advantage: it is simple to create a configuration in which, if the master process should fail during this time, so that the *mix* processing is interrupted, the status is handed over to another master that starts a new *mix*. The individual training results of each server obtained during steps (2) to (5) are updated with the synchronized model data received in step (5), so they are not reflected in subsequent model data. Note that it might be possible to reflect the training results during steps (2) to (5) by buffering them and retraining or remixing.

In previous research [8], we demonstrated that it is possible to implement linear scalability and a timely response with high training accuracy by means of a simple *mix* operation that permits such training data losses. We have also demonstrated that the effect on accuracy was limited when certain real data was used and indeed there have been no problems in actual use.

The present article, on the basis of experiments, further generalizes this situation and clarifies the data training performance and its behavior in distributed processing by Jubatus using *mix*.

3. Applicable algorithms

Jubatus implements several online machine learning algorithms, such as a linear classification algorithm (classifier), linear regression algorithm (regression), and nearest neighbor search algorithm, in addition to basic statistics functions. Among them, this article describes a linear classification algorithm (classifier) and linear regression algorithm (regression) as machine learning algorithms, and an iterative parameter mixture in the algorithms to make both algorithms correspond to distributed processing.

3.1 Classifier

The linear classification problem is the problem of predicting $y \in \{+1, -1\}$ according to whether a feature vector $\phi(x) \in R^m$ corresponding to an input x belongs to a certain class C. Jubatus implements five perceptrons: PA [13], PA2, PA3, CW, and ARROW.

3.2 Regression

The regression problem is the problem of assigning a real-valued output $y \in R$ for a feature vector $\phi(x) \in R^m$ corresponding to an input x. Jubatus implements a linear regression model using PA. With a linear regression model, we use the parameter $w \in R^m$ and forecast by means of $\hat{y} = w^T \phi(x) \in R$ with respect to input x.

3.3 Iterative parameter mixture

As mentioned in section 2.3, the iterative parameter mixture is used in the synchronization of training results, i.e., model data. The iterative parameter mixture for the PA algorithm, which is used by classifiers and regressions, is presented in this section, and a generic case of the *mix* that we assume to be installed in Jubatus is also demonstrated. In a certain phase τ, assume that the model data retained by a server $_i$ (where $_i = 1, \ldots, N$) is $w_i(\tau)$ and that the model data trained using supervised data obtained after the previous *mix* is $w'_i(\tau)$. The iterative parameter mixture in PA is expressed by the cumulative average:

$$w(\tau+1) = \frac{1}{N} \sum_{i=1....N} w'_i(\tau) .$$

With a linear classifier implementation in Jubatus, the model data w is a vector with few enough dimensions for storage in physical memory; sending all of the w_i data over the network and calculating averages would be unrealistic in practice. Therefore, only differences that accumulate during each *mix* are transferred over the network to synchronize w, as shown in the following equations.

$$w_i(\tau+1) = w_i(\tau) + \Delta w(\tau) \quad (1)$$

$$\Delta w(\tau) = \frac{1}{N} \sum_{i=1....N} \Delta w'_i(\tau) \quad (2)$$

The only data transferred over the network is $\Delta w'_i(\tau)$ and $\Delta w(\tau)$. Each of these is limited to at most a quantity equivalent to the number of feature quantities obtained by the training after the previous *mix*. Note that the model data used during actual training and classification is given by

$$w = w_i(\tau) + \Delta w'_i(\tau)$$

and this calculation is done as required. This format is similar to linear regression.

4. Experiments

The precision and performance of machine learning are strongly dependent on the dataset. For that reason, these experiments evaluated the number of pieces of supervised data that were trained per unit time (throughput), with the data dimensions being fixed. Machine learning has a computing part in which the central processing unit (CPU) is a bottleneck and a data update part in which memory access is a bottleneck. In addition to these, Jubatus also has bottlenecks on the network side because it is based on the client/server model. We provided sufficient numbers of client and proxy processes, 16 threads × 4 machines each, to ensure that network-related bottlenecks could be ignored, and we evaluated the server-side performance.

Random data with specified data dimensions was generated and used as a training dataset. More specifically, if we assume that the supervised data is (label, datum) and that the number of dimensions of a datum is N, we obtain data for which label $\leftarrow \{0, 1\}R$ and datum = $[0, 1]_i$, where $_i = 1, 2, \ldots, N$. Since there is no correlation between label and datum, the throughput level is lower than for a dataset of real data of the same magnitude. Since this dataset is random data generated from a uniformly random distribution, the data could be very dense in contrast to the dataset of sparse vectors assumed by Jubatus. Therefore, with this dataset, memory bottlenecks can readily occur, so the time required by the *mix* could be high.

4.1 Experiments in a single-server environment

A Jubatus server can initiate a number of threads with a single process. First, the throughput of a single thread was evaluated. The CPU of the server used in the experiments was a Xeon X3430 2.4 GHz (4 CPUs, 4 cores). Results for $N = 32, 64, 128, 256, 512$, and 1024 are shown in **Fig. 3**.

This figure shows that the numbers of dimensions and the numbers of queries per second (qps) are substantially inversely proportional to each other. (Memory or CPU creates a bottleneck.) We can also see that there is no great difference between classifier and regression.

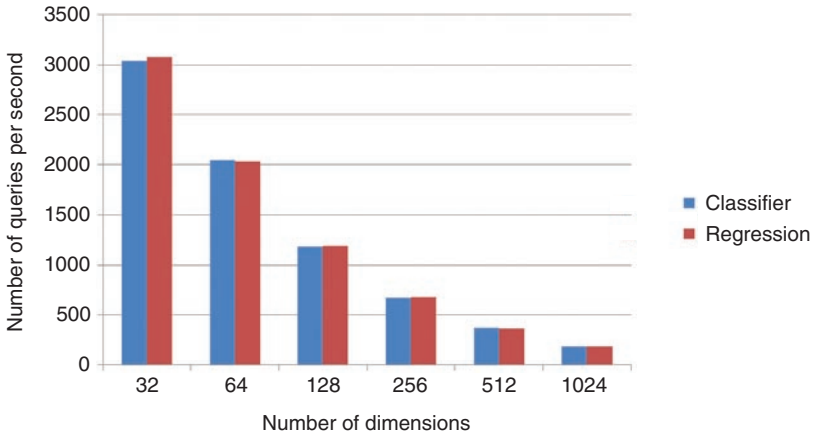


Fig. 3. Relationships between the number of dimensions and the number of queries per second with a single thread.

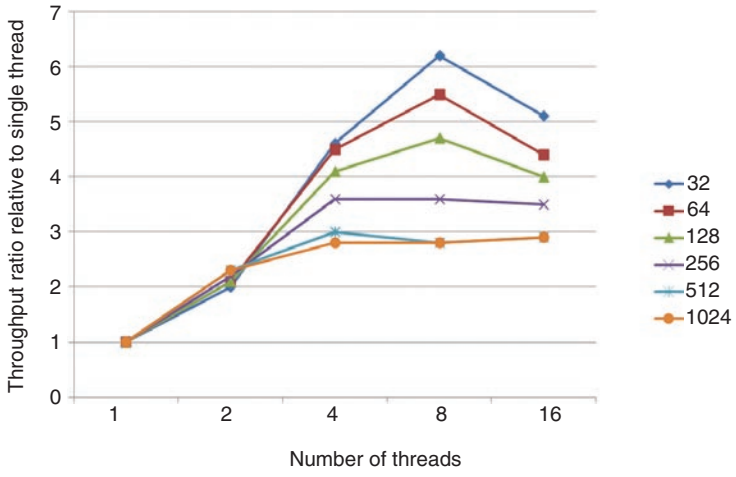


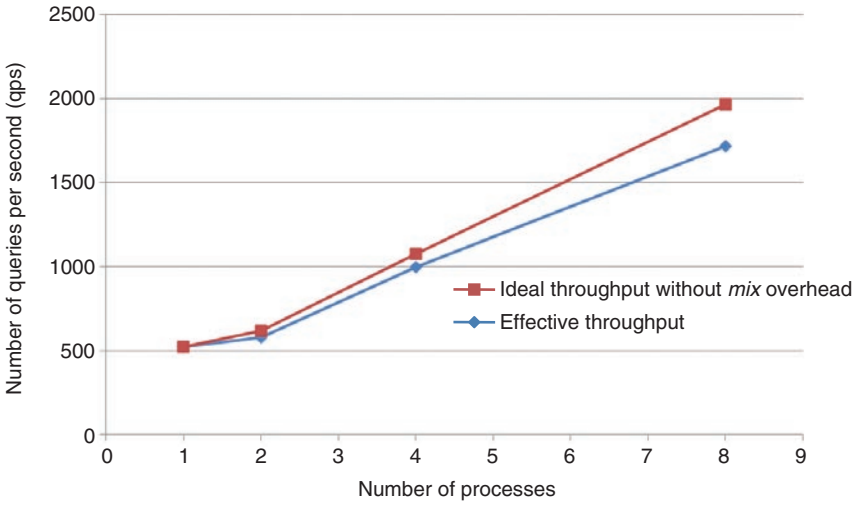
Fig. 4. Training throughput with single server.

Second, changes in throughput when the number of threads was increased were evaluated. The number of server threads, assuming qps to be 1 when there is one thread, is plotted on the horizontal axis of the graph in **Fig. 4** and the relative qps is plotted on the vertical axis. Since the computer has 16 cores installed, which is at least the maximum number of threads, a graph of $x = y$ should be drawn when the bottleneck is caused by the CPU rather than by memory.

It is clear that performance reaches a maximum when the number of threads is between 4 and 8, depending on the number of dimensions, and performance is lower for 16 threads. The bottleneck at 8 or 16 threads may be caused by memory: too many threads could result in slow throughput because of memory lock and unlock overheads in general. This means that increasing the number of CPUs and the number of cores in the single-server environment will cause a scale-up limit in this vicinity.

4.2 Experiments in a distributed environment

We evaluated the throughput of Jubatus in a distributed server environment using the *mix* process. In the light of the evaluation described in section 2.3, we performed evaluations with a configuration in which the number of threads was 8 and $N = 1024$. The *mix* timing was when 16 s had elapsed or when 512 instances of training had been performed in the initial



Number of processes, threads	Ideal throughput without mix overhead (qps)	Drop rate by *mix* process (%)	Effective throughput (qps)
1, 8	521	0	521
2, 16	620	6.6	579
4, 32	1076	7.2	999
8, 64	1967	12.7	1717

Fig. 5. Numbers of processes and training throughput.

setting of Jubatus. Since the quantity of arriving data was sufficiently large in this experimental environment, the number of instances of training data was taken as the trigger for *mix*. As introduced in section 4.1, training data is lost during the *mix* processing between the sending of differences and the receiving of synchronized data. The number of pieces of data (data items) being trained by each server during the *mix* process was evaluated (the number of data items that dropped out). Effective throughput, i.e., the actual number of training data items, was calculated by subtracting the number of lost data items from the total number of trained data items.

The experimental results are shown in **Fig. 5**. The throughput increased linearly with the number of servers because more than enough proxy processes were provided. Since the time required for the *mix* process also increased as the number of servers increased, the proportion of supervised data that was not reflected in the training results also increased as a consequence. However, the number of pieces of trained data per unit time in the entire system increased substantially as the number of servers increased. Note that the throughput for two nodes does not increase proportionally, in contrast to that for one node, owing to the overhead of a proxy process, which is unnecessary for the single-node configuration.

5. Conclusions and future challenges

This article has introduced Jubatus, a distributed computing framework for realtime analysis of big data. In particular, it described *mix*, which is a key method of Jubatus. *Mix* performs asynchronously with respect to the training of the entire system by an iterative parameter mixture in online classification and online regression problems in a distributed environment. We also evaluated the performance of the training process in both a single-server environment and a distributed server environment in a version of Jubatus implementing the *mix* method and confirmed its scalability.

Experiments in a distributed environment with eight nodes utilizing the *mix* demonstrated that there was an overall loss of 12.7% of the training data, but the number of items of training data per unit time increased in a substantially linear manner. This characteristic will be useful when statistical machine learning is applied to large quantities of data.

Future work includes supporting more machine learning algorithms, such as graph mining and

clustering, and confirming that the calculation framework based on a *mix* that is a generalized iterative parameter mixture will be valid for other machine learning algorithms and analysis tasks.

References

[1] Apache Hadoop. http://hadoop.apache.org/

[2] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer, "Apache Hadoop Goes Realtime at Facebook," Proc. of the 2011 International Conference on Management of Data (SIGMOD'11) Athens, Greece.

[3] Jubatus. http://jubat.us

[4] NTT press release. http://www.ntt.co.jp/news2011/1110e/111026a.html

[5] R. McDonald, K. Hall, and G. Mann, "Distributed Training Strategies for the Structured Perceptron," Proc. of the 2010 Annual Conference of the North American Chapter of the ACL, pp. 456–464, Los Angeles, California, USA.

[6] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker, "Efficient Large-scale Distributed Training of Conditional Maximum Entropy Models," Neural Information Processing Systems (NIPS), 2009.

[7] D. Okanohara, "Machine Learning Utilizing Large-scale Data by MapReduce," Hadoop Conference Japan, 2011 (in Japanese).

[8] D. Okanohara, Y. Unno, K. Uenishi, and S Oda, "Future Prospects for Techniques Supporting Large-scale Distributed Real-time Machine Learning," WebDB Forum, Tokyo, Japan, 2011 (in Japanese).

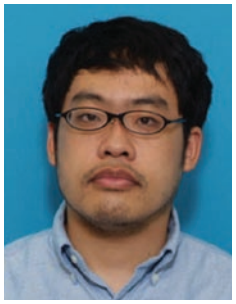
[9] Apache ZooKeeper. http://zookeeper.apache.org

[10] T. Haerder and A. Reuter, "Principles of Transaction-oriented Database Recovery," ACM Comput. Surv., Vol. 15, No. 4, pp. 287–317, 1983.

[11] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services," SIGACT News, Vol. 33, No. 2, pp. 51–59, 2002.

[12] D. Pritchett, "BASE: An Acid Alternative," Queue, Vol. 6, No. 3, pp. 48–55, 2008.

[13] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer, "Online Passive-aggressive Algorithms," Journal of Machine Learning Research, 2006.



Satoshi Oda

Researcher, Cloud System SE Project, NTT Software Innovation Center.

He received the B.E. and M.E. degrees in engineering from Keio University, Kanagawa, in 2003 and 2005, respectively. Since joining NTT Information Sharing Platform Laboratories in 2005, he has been engaged in R&D of information security, fast implementation of cryptography, and security protocols. As a result of organizational changes in April 2012, he is now in NTT Software Innovation Center. He received the 2007 Outstanding Presentation Award from the Japan Society for Industrial and Applied Mathematics (JSAIM) and the 2009 Life Intelligence and Office Information System (LOIS) Research Award. He is a member of JSAIM.

Kota Uenishi

Engineer, Distributed Data Processing Platform Project, NTT Software Innovation Center.

He received the B.E. degree in engineering and the M.S. degree from the Department of Frontier Informatics at the University of Tokyo in 2005 and 2007, respectively. He joined NTT Information Sharing Platform Laboratories in 2007. Since 2008, he has been engaged in R&D of fault-tolerant distributed computing systems for a search engine backend. As a result of organizational changes in April 2012, he is now in NTT Software Innovation Center.

Shingo Kinoshita

Senior Research Engineer, Supervisor, Group Leader, Distributed Computing Project, NTT Software Innovation Center.

He received the B.E. degree in solid state physics engineering from Osaka University in 1991. Since joining NTT Information and Communication Systems Laboratories in 1991, he has been engaged in R&D of fault-tolerant distributed computing systems, Internet protocols, especially a reliable multicast protocol, information security, especially RFID privacy protection technology, and big data computing. During 2006–2007, he studied the management of technology at University College London and received the M.Sc. degree in 2007. During 2008–2011, he worked in human resources in the planning section of the laboratories. He is currently managing distributed computing projects including Hadoop, Jubatus, and the mobile cloud computing technology Virtual Smartphone. As a result of organizational changes in April 2012, he is now in NTT Software Innovation Center. He received the 2005 Information Processing Society of Japan (IPSJ) Research and Development Award, the 2003 IPSJ Symposium CSS Best Paper Award, the 1998 IPSJ Symposium DiCoMo Best Presentation Award. He is a member of IPSJ and the steering committee of IPSJ SIGDPS.