SketchML in Apache Flink[*]

Big Data Analytics Project

Ali Reza
TU-Berlin & DFKI
alireza.rm@dfki.de

Behrouz Derakhshan
TU-Berlin & DFKI
behrouz.derakhshan@dfki.de

Batuhan Tuter
TU-Berlin
batuhan.tuter@campus.tu-berlin.de

Kashif Rabbani
TU-Berlin
kashif.rabbani@campus.tu-berlin.de

Marc Garnica
TU-Berlin
marc.garnicacaparros@campus.tu-berlin.de

ABSTRACT

Gradient descent is one of the most used optimization method in supervised distributed machine learning (ML) algorithms. Recently, several researchers have proposed strategies and approaches to parallelize the computation of stochastic gradient descent (SGD) and to distribute the workload across a number of nodes in the cluster. These methods usually involve communication of the gradients through the network. The computational power is becoming more and more efficient and an active research is going on to reduce the communication cost between nodes in a distributed network by means of gradient compression.

A recently published research paper known as SketchML [5] presents a gradient compression method. This method introduces a probabilistic data structure known as Sketch to approximate and reduce the communication cost of gradients. Extensive experiments are performed to analyze the rate of compression, model correctness, and error bound in Sketch implementation on top of Apache Spark. This study aims to present the details of the compression technique implemented in SketchML and mimic its implementation on top of Apache Flink. Our SketchML implementation remains the same as the one proposed in the research paper. However, its properties are tested on top of a Flink SGD optimization algorithm. The outcomes of this study include an effective comparison between SketchML implementation in Spark and Flink, and an analysis of the impact of SketchML gradient compression technique in Flink ML.

1 INTRODUCTION

ML techniques are used widely in numerous fields. Most common applications of ML are recommendation engines, text mining and image recognition. Decentralized ML is no more considered as an enough solution to cope up with the growing demands of high-volume data. In this project, we focus on a subclass of ML models called Multiple Linear Regression (MLR) [6]. This class of ML models is trained with first-order gradient optimization methods known as Stochastic Gradient Descent (SGD) [3]. To train our MLR model in a distributed way, we need to set up an infrastructure to distribute the calculation of gradients by partitioning the training dataset across different workers. Under such setting, each worker computes gradients independently and the gradients are exchanged via a network for aggregation and the model is updated at each step. State-of-the-art ML research acknowledges that a large model gives a better representation and higher prediction. In order to train a large model in a distributed manner, an expensive cluster infrastructure is required. Reducing the communication between nodes in the distributed settings has become one of the most active and productive lines of research in distributed ML. Proposed methods should be able to reduce the time required by nodes to send gradients through the network while retaining the algorithmic correctness of the model.

Our study is organized as follows. The main contribution of SketchML with underlying data structures, compression framework, and its theoretical proof are discussed in Section 2. Objectives and initial assumptions are mentioned in Section 3. The main study and implementation plan is presented in Section 4. Finally, the experimental results are analyzed in Section 5.

Implementation of SketchML in Apache Flink is available in the GitHub repository[1].

2 OVERVIEW OF SKETCHML FRAMEWORK

This section gives an overview of the technical details of SketchML framework. The framework consists of three major components, i.e. quantile bucket quantification, MinMaxSketch, and dynamic delta binary encoding. First two components are responsible for compressing the gradient values (encode phase), and the third component is solely responsible for compressing the gradient keys (decode phase). We will explain both phases in the details below.

2.1 Encode Phase

In the encode phase, candidate splits are generated using quantile sketches. Quantile sketch uses a small data structure to approximate the exact distribution of the data items in a single pass by building quantile summaries [4]. Bucket sort is used to summarize the values of the sketches. Each bucket has indexes related to the corresponding values. Different hash functions are applied to the keys to insert bucket indexes into a MinMaxSketch. MinMaxSketch works on a simple principle, i.e. while inserting it prefers to keep the minimum value and while querying it returns the maximum of value. Keys in the MinMaxSketch are incremented (delta keys) and finally, binary encoding is used to encode the keys to save space. Figure 1 shows the encoding phase in five simple steps.

2.2 Decode Phase

This phase is responsible for recovering the compressed gradients by transforming the delta keys to original keys, use the retrieved original keys to query the MinMaxSketch, retrieve the bucket index of each value from the sketch, and recover the value by querying the bucket with the corresponding bucket index, as shown in figure 1.

[*] Accelerating Distributed Machine Learning with Data Sketches

[1] https://github.com/Kashif-Rabbani/SketchMLFlink

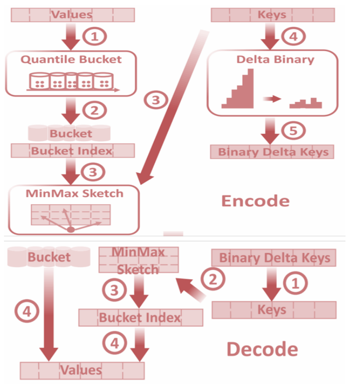


Figure 1: SketchML Encode Decode Phase

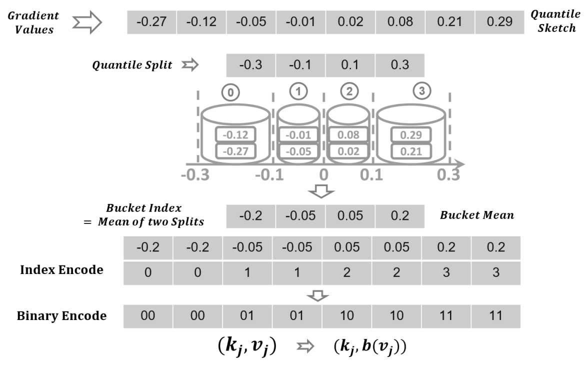
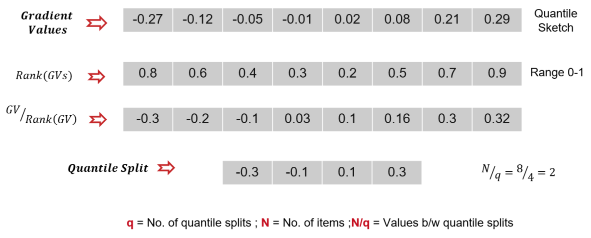


Figure 3: Bucket Quantification & Encoding

2.3 Example

In order to understand the core steps of the encode and decode phase, we have chosen a few gradient values to pass through both phases step by step.

Quantile Splits: In the decode phase, we will first generate quantile splits. To generate the quantile splits, gradient values are scanned and added into a quantile sketch, absolute rank is calculated for all the values in the quantile sketch, each gradient value is divided by its absolute rank, and finally, quantile splits are generated from the divided values by a formula shown in figure 2.



Gradient Values	-0.27	-0.12	-0.05	-0.01	0.02	0.08	0.21	0.29	Quantile Sketch
$Rank(GVs)$	0.8	0.6	0.4	0.3	0.2	0.5	0.7	0.9	Range 0-1
$^{GV}/_{Rank(GV)}$	-0.3	-0.2	-0.1	0.03	0.1	0.16	0.3	0.32	
Quantile Split		-0.3	-0.1	0.1	0.3				$^{N}/_{q} = {^{8}/_{4}} = 2$

q = No. of quantile splits ; N = No. of items ;N/q = Values b/w quantile splits

Figure 2: Quantile Splits Generation

Bucket Sort: Next step is to quantify the gradient values using bucket sort according to the generated quantile splits. Each interval between two splits is considered as a bucket, having the smallest value as a lower threshold and the largest value as the upper threshold of the bucket. Each gradient value belongs to a specific bucket based on its threshold values. Each bucket is represented by the average of two splits known as a mean of the bucket. The final step of bucket sort is to transform or map each gradient value in the bucket to its mean value. This step also introduces some quantification errors. Figure 3 shows the transformation of our chosen gradient values after applying a bucket sort, index and binary encoding.

Index Encode: Gradient values are quantified with bucket means which are floating point number and are still consuming space. Therefore, we encode the mean value of the bucket as a bucket's index to save space. E.g. quantified 0.21 gradient value is quantified to the mean of the 4th bucket. It is further encoded by its bucket's index i.e. 3.

Binary Encode: Encoded index are integer numbers consuming 4 bytes at least, we compress these indexes by binary encoding.

MinMaxSketch: Encode phase has converted key value gradient pairs into key and binary encoded value. Next step is to insert these pairs into MinMaxSketch. Note that during decompression, we also need to query the MinMaxSketch. Therefore, we will discuss both aspects here. MinMaxSketch uses a predefined number of hash functions to choose a hash bin to insert the $(k_J, b(V_J))$. Each hash function returns a bin number, hash bin having a current value greater than the value to be inserted is chosen and its value is replaced with the minimum value. Similarly, in the query phase, input to the MinMaxSketch is a gradient key, hash functions are applied on the key and the hash bin having the maximum value is returned. This is the reason it is called MinMaxSketch.

Keys Compression: The final step of encoding phase is to compress the gradient keys. It is mandatory to choose a lossless compression method because losing a gradient key will lead to losing the gradient value. Therefore, gradient keys are transformed into delta keys by calculating the difference between consecutive gradient values starting from the end of the list. Delta keys are passed to a binary encoding module to convert it into binary format via threshold encoding.

Decode Phase: In order to decode the compressed gradient key value pairs, reverse methodology is used. Gradient binary delta keys are transformed into original keys using reverse binary encoding. This gradient key is used to query the MinMaxSketch to retrieve the corresponding bucket index, which is used to query the bucket to find the gradient value i.e. the mean of the bucket. We recommend reading the original paper for more details and formal proofs of the chosen approaches.

3 OBJECTIVES

As presented in the previous section, SketchML contribution relies on the accelerations of distributed machine learning jobs by reducing the communication among workers. In the original implementation of SketchML in Spark [5], Jiang et al. analyzed the error bounds of the proposed compression technique and the model convergence.They proved that the SketchML technique can be 10x faster than the related current state-of-the-art techniques. This demonstration is based on the results obtained by extensive experiments of generalized linear ML models on KDDB10 [1] and KDDB12 [2] datasets.

The main goal of this study is to understand the compression technique presented in the SketchML research paper and to implement a proof of this concept in Apache Flink. Therefore, SketchML core (including data structures and compressing/decompressing algorithms) is conceived as a third party library because its implementation in Apache Spark is also considering it as a library for the compression and decompression of Generalized Linear Models. Consequently with this hypothesis, our objective is to implement a linear model in Apache Flink that can handle the addition of SketchML data structures and analyze its impact on overall performance.

3.1 Plan of implementation

The abstraction of the required SketchML functionalities into a self-sufficient software library and the following injection into a baseline Flink optimization algorithm has not been trivial and several approaches have been tested iteratively. The implementation options were as following.

1. **Generalized Linear Model of SketchML**: Using the algorithms provided by SketchML library while mapping Spark operators to Flink operators one-to-one.
2. **Custom optimization algorithm for Apache Flink**: Implementing an optimization algorithm from scratch, using Flink DataSet API and injecting compression algorithms provided by SketchML library.
3. **FlinkML Multiple Linear Regression**: Adopting the Multiple Linear Regression implementation provided by FlinkML library and injecting the SketchML compression algorithms in it.

3.2 Experiments overview

In order to provide a fair comparison between SketchML implementation in Apache Spark and Apache Flink, this study performs extensive experiments to measure and analyze the relative impact achieved by both of the data flow processing engines using SketchML compression technique. In other words, we aim to assess the performance of our proposed implementation with and without the compression and map the relative speedups with the result provided in [5] for SketchML in Apache Spark.

4 IMPLEMENTATION

4.1 Generalized Linear Model of SketchML

The first implementation approach derived directly from the inspection of Spark implementation [5]. SketchML library implements a Generalized Linear Model with three core arguments: weights vector with the model state, optimizer object responsible for the gradient computation, model update and calculation of the loss function. The three values overwritten by the different implementations provided are Linear Regression, Logistic Regression and SVM Model.

The generalized linear model partitions the data to as many workers as defined. Each worker node interacts with global variables to receive the assigned portion of the data locally. After loading the data, the model loops for a number of epochs defined and performs a mini-batch gradient descent optimization at each epoch. On each batch, each worker in the distributed set is computing a local gradient. These partial gradients are gathered all together to compute the final gradient, which is used to update the model, and the algorithm proceeds with the next iteration.

First approach is to remain consistent with the original code structure of SketchML while substituting Spark operators with similar Flink operators. In some cases, this task was trivial but in some others, technical mismatch of Spark and Flink operators made the code transition harder.

Two main examples can give an overview for the approach and the problems encountered. First, we tried to mimic the rebalance operation of Spark with custom map partitioning on Flink. In parallel, we also tried to implement the distributed task of computing the gradients on each worker. Spark is allowing the developers to operate through a high-level API with the Aggregate Operator. This operators expects three arguments, the initialization of the result, the operation to be distributed across the cluster, namely seqOp, and the operation in charge of gathering and reducing all the results, the combOp.

These operators do not have directly mapped operators in Flink, which made the implementation in-feasible without having to change few characteristics of the initial implementation. This approach turned out to be very cumbersome since using Spark's and Flink's dependencies in the same project leads to many problems, such as conflicts in class names and function calls. Those are the reasons why we proceeded with option 2 and worked in the implementation of baseline optimization problem on top of Flink DataSet API from scratch.

4.2 Custom optimization algorithm for Apache Flink

In this implementation step, our main goal was to develop an end-to-end optimization algorithm using gradient descent as its core. To achieve this, training data needed to be partitioned across the workers, and each worker run as much batch gradient descent execution as defined by the batch ratio. After the batch computation, the gradients are sent to a centralized node where all the partial gradients are aggregated and the complete gradient is used to update the weights model. The last step of updating the model can be done either in a centralized manner in the driver node or by distributing the total gradient through the network and letting each worker update its local instance of the model. In order to achieve this, we need to use the native iteration operators provided by Flink DataSet API i.e. *Iterate* and *Delta-Iterate* functions.

The main focus of this study was to identify and measure the impact of how SketchML contribution and compression technique could be applied to the current state-of-the-art Flink implementation of gradient descent. Therefore, instead of investing our efforts to implement a new optimal SGD implementation from the scratch, we decided to make use of already in place implementation of gradient descent optimization methods using Flink DataSet API.

4.3 FlinkML Multiple Linear Regression

FlinkML is the Machine Learning library for Flink. Its main objective is to gather and optimize scalable ML algorithms using Flink Dataset API as well as provide a high-level API and tools to minimize the coding of end-to-end ML systems and prioritizing the analysis of data and results.

After a brief research we identified that FlinkML provides a Multiple Linear Regression module. Multiple linear regression algorithm aims to find a linear representation which best fits the input data. This model basically finds a weight vector to be applied to the data so that the sum of the squared residuals is minimized.

The important thing in this model is that FlinkML applies SGD to compute an approximation of the optimal weight vector. The resulting gradient is applied to the current weight vector and the updated vectors is processed to the next iteration.

FlinkML allows the algorithm to finish with a fixed number of iterations or by a custom convergence criteria. For the study purposes we have only focused on the first option.

4.3.1 *Injection of SketchML.*

As mentioned before, after investigating the FlinkML source code, we have found that Flink has implemented a Stochastic Gradient Descent class[2] with some example ML algorithms such as Linear Regression[3]. We have decided to use these classes as our optimization function and the preferred ML algorithm. It is important to note that current version of the Flink (v 1.7) does not provide a sampling operator thus, this SGD implementation is effectively used as a Batch Gradient Descent algorithm.

In the SGD implementation of Flink uses the *iterate* function to compute on each iteration a SGD step. There are 3 main operations that are applied in each iteration (epoch):

$$MAP => REDUCE => MAP$$

In the first Map operation, initial weights are broadcasted to each node so that they can individually calculate the gradients on their training data. After gradients are calculated on each worker node, the reduce operation is applied to these gradients at each node. A node receives two gradients (called left and right), adds these two gradients into one and sends this sum to another node for the next accumulation. Next node receives the previous sum as the left gradient and another gradient as the right gradient to be added to the left gradient.

This accumulation continues until all gradients are summed up and rests in one of the nodes. The last Map operation is receiving again the current weights as a broadcast variable and it is computing the new weight vector for next iteration taking into account: the resulting gradient, the iteration, and the effective learning rate which computed dynamically.

The injection of SketchML libraries into this algorithm has been implemented with two different approaches, focusing on the reduce part of the data flow. After the first map operator computes the gradient, a new map function has been introduced to convert the resulting partial gradients into SketchML format, i.e. gradients are compressed.

The first approach was to use initial Reduce operation provided within the SGD class of FlinkML for the pair-wise aggregation of gradients, while being aware of the new data structures injected. The second approach substitute the pair-wise reduce operator with a group reduce operator. With a single group reduce operation, the algorithm can decompress all the partial gradients and aggregate them at once, rather than having to compress/decompress the gradients several times during the pair-wise reduce.

4.3.2 *Pair-wise Reduce Operation.*

After the first Map operation, where each node calculates their gradients individually, we applied a second Map operation on these gradients to compress them into sketches. After the gradients are compressed, the reduce operation is applied to these compressed sketch gradients and they are transferred over the network for the accumulation. When two sketch gradients are received by a node, compressed gradients are decompressed for the summation operation. After the accumulation of two gradients, accumulated gradient needs to be compressed again to be sent over the network for the next reduce operation. After all the gradients are accumulated in one of the nodes, weight vectors are updated by that node and the epoch finishes.

We have identified that the main bottleneck of this approach is the re-compression of the summed gradients in each Reduce operation.

4.3.3 *Reduce Group Operation.*

In the original SketchML gradient descent implementation, gradients are collected into one machine and accumulated at once. This operation of Spark corresponds to the ReduceGroup operation in Flink. As we have identified that re-compressing the accumulated gradients in Reduce function increases the computation in each operation, we have decided to use a ReduceGroup operation instead of the Reduce operation to eliminate the re-compression step.

Once the second map operation is finished in each node to compress the calculated gradients, we call the GroupReduce operation without applying any grouping to the compressed gradients so that one of the nodes receive all the compressed gradients at once, in contrast to the Reduce operation where each node receives two gradients each time. After receiving all the compressed gradients, they are decompressed for accumulation. When all the gradients are accumulated, the last Map operation is applied by this node to update the weight vectors and the epoch is finished as the previous approach.

It is important to note that neither ReduceGroup nor Reduce operation is better. It is intuitive that ReduceGroup operation has less computation overhead thus, at each epoch, run time is expected to be lower compared to the Reduce operation. On the contrary, GroupReduce operation does not perform as efficient as Reduce operation when the parallelism is high since all the gradients are accumulated in only one worker machine. If the parallelism is low, then ReduceGroup operation improves the run time per epoch but if scaling in terms of parallelism is desired, then Reduce operation should be preferred. For the rest of the report, we will use the ReduceGroup operation for our experiments.

5 EXPERIMENTS

In order to prove this case study, multiple experiments are performed to analyze the performance of the proposed algorithm "SketchML in Flink" in comparison with the SGD implementation in Flink and its relative impact in contrast with the SketchML in Spark.

5.1 Dataset and Cluster

The KDDB10 [1] dataset is used in *LIBSVM* format for all the experiments. This data set contains 5GB of data with 29 million features. As a test environment, IBM Cluster is used, which is provided by TU-Berlin having 7 IBM machines with 62 GB memory and 42 cores each with 1TB total disk space. Operating system that is used is Fedora 27 with Linux version 4.18.12-100.fc27.ppc64. As the Flink server, version 1.7.0 is used.

5.2 Metrics of evaluation

This study aims to evaluate the impact of SketchML compression technique in the FlinkML optimization algorithms. The list of evaluation metrics are total runtime, runtime per epoch, and average sum squared error per record to measure the model error bound.

[2]https://github.com/apache/flink/blob/master/flink-libraries/flink-ml/src/main/scala/org/apache/flink/ml/optimization/GradientDescent.scala
[3]https://github.com/apache/flink/blob/master/flink-libraries/flink-ml/src/main/scala/org/apache/flink/ml/regression/MultipleLinearRegression.scala

5.3 Test execution

Experiments have been executed automatically with a bash script provided in the project repository. This bash script iterates over different command line parameters. The variations in the executions are the following:

- **Execution: SketchML or FlinkML**: The experiment is executing FlinkML baseline code or SketchML-aware new functionality.
- **Compression: Sketch or None**: The experiments can run with compression "None" or "Sketch". Sketch compression means algorithm is compressing the gradients using SketchML compression technique.
- **Parallelism**: Number of parallelism applied to the operators to check the behaviour of the algorithm to smaller or bigger distribution environments.
- **Iterations**: Number of iterations to run.
- **Dimensionality**: Maximum number of features introduced in the algorithm.
- **Data size**: Experiments have been run with different data sizes: 10MB, 100MB and 1GB.

5.4 Experiments

5.4.1 Defining the number of iterations. The experiments performed did not include any convergence criteria to stop the computation of gradients, but a maximum number of iterations. To simplify the number of tests to be performed, we first analyzed the impact of the number of iterations executed in the averaged sum squared error and the total run time of the execution.

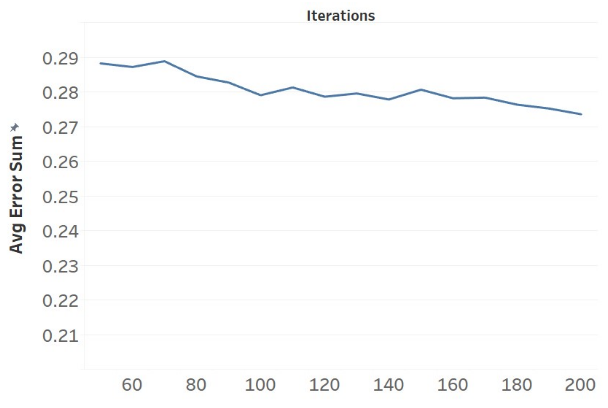


Figure 4: Average Sum Square error evolution as iterations grow with system parallelism 10

In figure 4, we are showing the evolution of the squared sum error average per record depending on the number of iterations. As it can be seen, after a certain amount of iterations the average error is below 0.29 and it starts to decrease its sharpness with a limit approximately 0.27. Our goal was to try to find a proper value which keeps the error small but does not take a lot of time to finish all the iterations. In the following figure 5 it can be seen that the total time increases accordingly to the number of iterations performed. 120 iterations matched our objectives for this study.

5.4.2 Base comparison between algorithms. FlinkML original implementation and the proposed adaption using SketchML (FlinkSketch) were tested with 10MB sized dataset and different values for parallelism and dimensions.

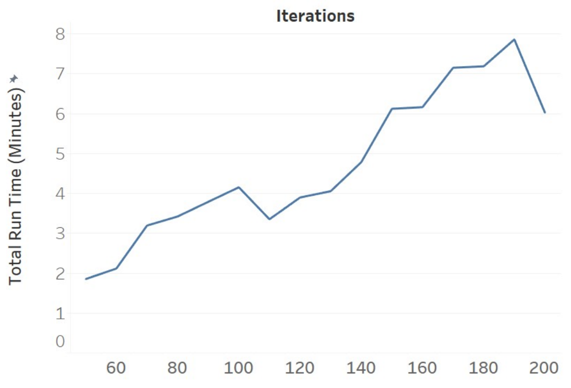


Figure 5: Total run time execution depending in the number of iterations.

In figures 6 and 7 we can see the impact of parallelism with a low number of features. The model error remains stable independently on the parallelism. Figure 6 shows the average run time per epoch with 1000 number of dimensions. Figure 7 shows that FlinkML outperforms FlinkSketch execution and that FlinkSketch does not use parallelism to get a proper speedup. More tests with several dimensions can be found in Appendix A.

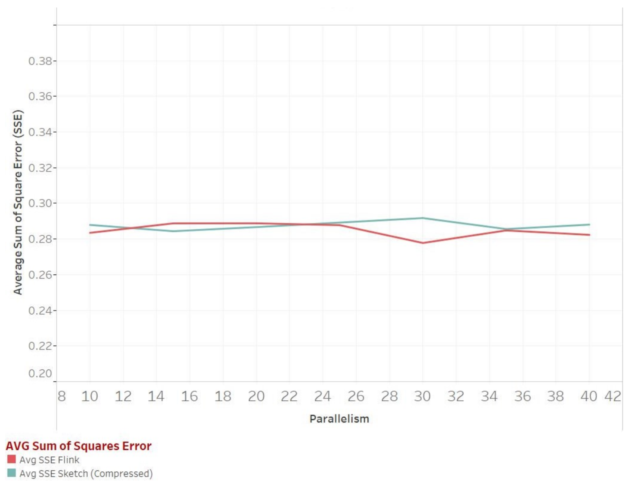


Figure 6: Average Sum Square error evolution tests with 1000 dimensions and varying parallelism values for FlinkML and FlinkSketch artifacts

FlinkML performs better with low dimensionality as it uses a pair-wise reduce which enables the parallelism of the gradients summation. On the other hand, FlinkSketch reduce group implementations forces the parallelism to one, every time the algorithms needs to aggregate the gradients. With lower levels of parallelism this approach can be effective since the algorithm avoids the overhead of re-compression of the gradients. However, reduce group implementation is not able to scale up with the number of nodes of the cluster as the reduce option of FlinkML is able to.

It is also important to analyze how dimensionality growth is effecting the two implementations under test. As expressed before, Figure 8 shows that the error of the model remains stable as dimensionality grows.

Interestingly, FlinkML starts to produce some very bad results when the data dimensionality increases while FlinkSketch implementation achieves better results, as it can be seen in Figure 9.

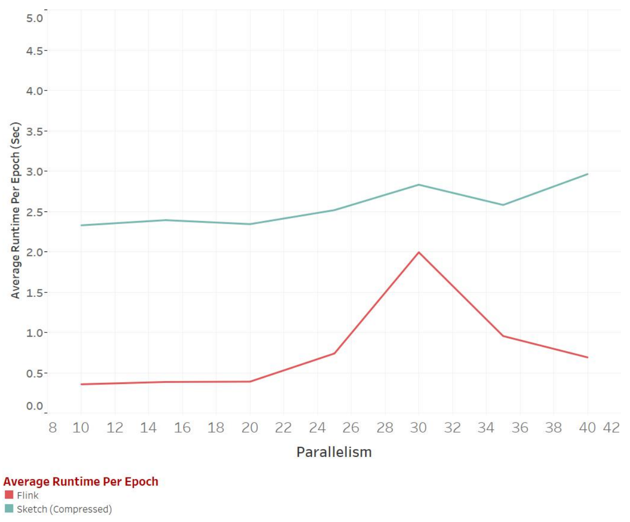


Figure 7: Average run time per epoch evolution tests with 1000 dimensions and varying parallelism values for FlinkML and FlinkSketch artifacts

The scope of our tests did not manage to identify the reason why this behaviour is observed but we identified two possibilities:

- Apache Flink's pair-wise reduce is not efficient for high dimensional data, which makes the usage of a reduce group operator able to handle this more efficiently.
- FlinkSketch compression technique allows the system to reduce communication cost while dealing with high dimensionality and improves the performance.

In both cases, it is clear that FlinkML suffers from bad performance when dealing with a relatively high number of features, while adoption of FlinkSketch solves this problem.

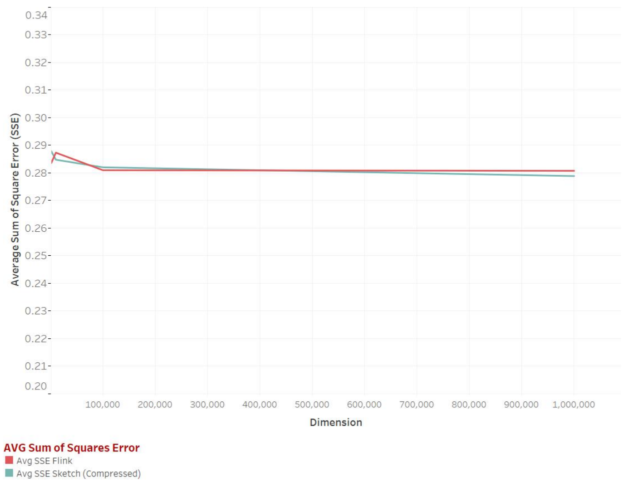


Figure 8: Average Sum Square error evolution tests with 1000 dimensions and varying parallelism values for FlinkML and FlinkSketch artifacts

5.4.3 Big data experiments. This study aims to analyze how much the implementations under test can deal with bigger sizes of data. Intuitively, as FlinkSketch implementation suffers from lack of efficient parallelism utilization, it was quite clear at prior that FlinkML original implementation would be able to cope with bigger data size more efficiently.

For this experiment, a 1GB data file from the original data set was used. To be able to get some results in a proper time only 1000

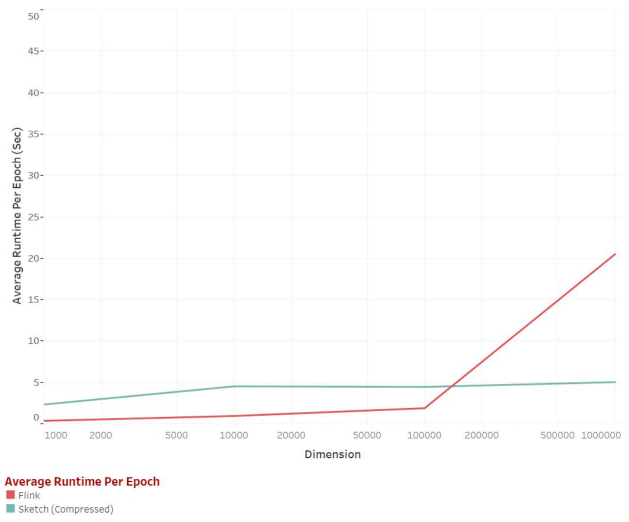


Figure 9: Average run time per epoch evolution tests with 1000 dimensions and varying parallelism values for FlinkML and FlinkSketch artifacts

features were accepting in the execution. For these experiments, a third option was tested, namely Sketch None, which consist of the Sketch implementation but with no compression technique applied to the gradients.

Figure 10 shows that FlinkML is performing better with 1GB data than both implementation for FlinkSketch, with Sketch compression or without compression. As expected, the group reduce implementation is not able to efficiently deal with that amount of data as it forces all the gradients to be aggregated in a single node.

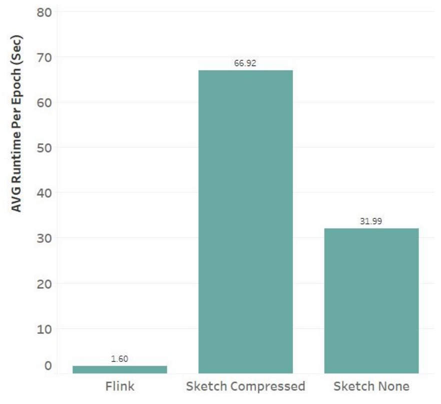


Figure 10: Average run time per epoch for the tested implementations with 1GB data file

What is also interesting to extract from figure 10 is that the None compression option is performing better than the Sketch compression option. Run time per epoch is clearly lower but the None option as well as the FlinkML original implementation require much more data transfer that should also be studied in detail in further work.

6 FUTURE WORK AND CONCLUSION

With the trend of increasing data size and model size, reducing the communication between nodes in a distributed settings has become one of the main bottleneck of the system. To address this problem, SketchML proposed a novel way to save the communication cost between nodes hence making the distributed machine learning more efficient. We have implemented the SketchML gradient compression technique in Apache Flink and analyzed its overall performance by extensive experiments.

The adaptation of SketchML compression technique using Flink's dataset API (FlinkSketch) has been quite challenging. Our main assumption was that SketchML is ready and capable to be imported as a third party library into any other project. After the closure of this project, this assumption has not been certainly verified as this study identified several dependencies and breaking points in SketchML implementation which makes it difficult to be used outside the box. The result of this study has proved that a proper adaptation of SketchML in Apache Flink should be done from scratch i.e. understanding the data structures of FlinkML optimization algorithms and implementing the SketchML compression technique at low-level.

Nevertheless, the presented study was satisfactory and it proved the initially proposed hypothesis. Sketch implementation of the Multiple linear regression model in Flink DataSet API could not improve the performance of original FlinkML code in a low dimensional space. However, it showed that it can handle large number of features more efficiently.

The proposition regarding the efficiency of the pair-wise reduce and the reduce group remained unattainable throughout the project, as this study could not come up with a functional implementation of the pair-wise reduce using SketchML library. Implementing the optimization problem with a pair-wise reduce allows the algorithm to scale and be efficiently distributed. On the other hand, in each pair-wise reduce operation, gradients need to be decompressed, aggregated and compressed again, adding a relatively larger overhead as compared to reduce group operation. The reduce group operation avoids this overhead by gathering all the partial gradients at once and operate without the need to decompress them again. Although, the group reduce appears to be a better option in terms of computational cost, the fact that this operation can not distribute resulted in a very bad performance during experiments. As a future work, it is recommended to quantify and isolate the runtime of group reduce operation and the compression overhead introduced by pair-wise reduce. This report has documented all the steps of implementation and analyzed the experimented results in detail, which will be useful for further studies of distributed machine learning optimization methods in Apache Flink.

REFERENCES

[1] 2010. KDDB. (2010). https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#kdd2010(bridgetoalgebra)

[2] 2012. KDDB12. (2012). https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#kdd2012

[3] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. (2010), 177–186.

[4] Michael Greenwald, Sanjeev Khanna, et al. 2001. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record* 30, 2 (2001), 58–66.

[5] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. 2018. SketchML: Accelerating Distributed Machine Learning with Data Sketches. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1269–1284.

[6] George AF Seber and Alan J Lee. 2012. *Linear regression analysis*. Vol. 329. John Wiley & Sons.

A APPENDIX

Figure 11 shows the evolution of average of sum of squared errors for FlinkML (Red) and FlinkSketch (Green) with increasing number of dimensions from 1,000 till 1,000,000. For each of the mentioned dimension, parallelism is increased from 10 to 50. Both algorithms show almost the same accuracy for the model being trained.

Missing or unconnected lines indicate that experiments couldn't complete due to memory issues with the cluster.

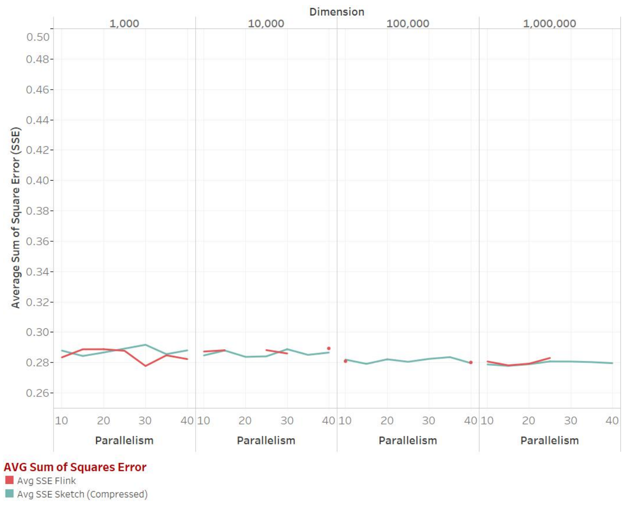


Figure 11: Average sum square error evolution with varying dimensions and parallelism.

Figure 12 shows the evolution of average runtime per epoch for FlinkML (Red) and FlinkSketch (Green) with increasing number of dimensions from 1,000 till 1,000,000. For each of the mentioned dimension, parallelism is increased from 10 to 50. It is observed that Flink SGD can not afford large number of dimensions.

Missing or unconnected lines indicate that experiments couldn't complete due to memory issues with the cluster.

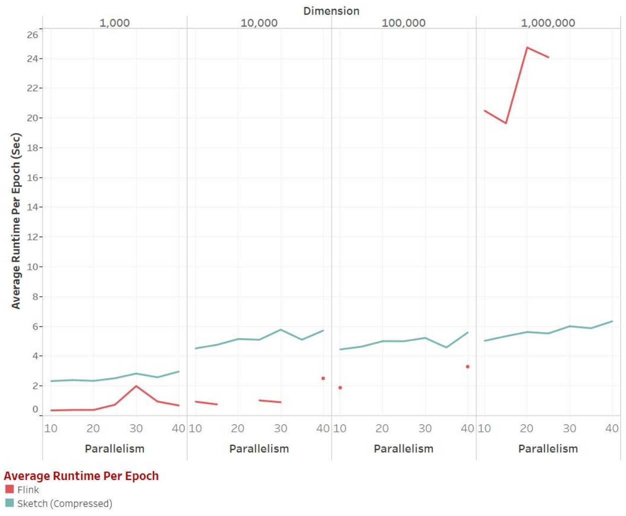


Figure 12: Average run time per epoch evolution with varying dimensions and parallelism.