Algorithmic complexity and practical performance of Top-k compression for communication efficient distributed optimization

1 Introduction and motivation

Over the last several years, deep neural networks [1] have been applied in the solution of many practical problems such as adaptive testing, biological image classification, computer vision, cancer detection, natural language processing, object detection, face recognition, handwriting recognition, speech recognition, stock market analysis, smart city, and many more [2]. Training deep neural networks is computationally expensive and time-consuming. Over the past few years, to improve the practical generalization performance of modern deep learning models, practitioners are tending to use larger and larger datasets in the training process [3], distributed across several devices. These devices can be a desktop computer, laptop, tablet, smartwatch, etc. So, in general, by word "device" or "node" we mean any gadget that can store data, compute functions values and gradients (or stochastic gradients), and communicate with other different devices. For example, this distributed setting appears in federated learning [4], [5] (see Fig. 1 for network topology clarification)

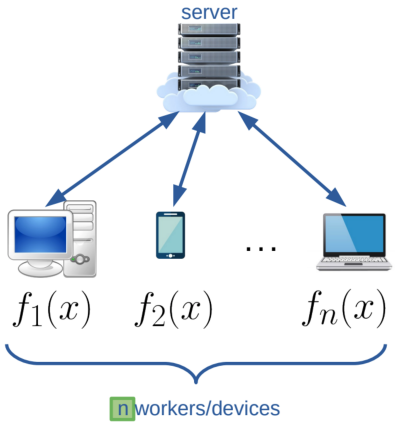


Figure 1: Topology of the data transfers between devices and server. Source [6].

In this setting, distributed methods can be very efficient at decreasing the training time [7], [8]. Therefore, distributed optimization has gained tremendous popularity in recent years.
Dealing with the distributed environment, we consider optimization problem of the form

$$\min_{x\in\mathbb{R}^d}\left\{f(x):=\frac{1}{n}\sum_{i=1}^{n}f_i(x)\right\}$$

where $x\in\mathbb{R}^d$ collects the parameters of a model to be trained, d is the dimensionality of the problem (number of trainable features), n is the number of workers/devices/nodes, and $f_i(x)$ is the loss incurred by model x on data stored on worker i. The loss function $f_i:\mathbb{R}^d\to\mathbb{R}$ often has the form of expectation of some random function

$$f_i(x):=\mathbb{E}_{\xi\sim\mathcal{D}_i}\left[f_\xi(x)\right]$$

with $\mathcal{D}_i$ being the distribution of training data owned by worker i. In federated learning, these distributions are allowed to be different (so-called *heterogeneous* case). This finite sum function form allows us to capture the distributed nature of the problem in a very efficient way.
Quite often, the most successful models are over-parameterized, which means that they involve more parameters (large dimension d) than the number of available training data points [9]. In this situation, distributed methods may suffer from the so-called communication bottleneck: the cost of communication of the information necessary for the workers to jointly solve the problem can be orders of magnitude or even higher than the cost of computation [10]. One of the standard popular (and practical) techniques for resolving this issue is communication compression [11]–[13], which is based on applying a lossy transformation/compression to the models, gradients, or tensors to be sent over the network in order to save on communication. Compression is typically done via the application of a (possibly randomized) mapping $\mathcal{C}:\mathbb{R}^d\to\mathbb{R}^d$, where d is the dimension of the vector/tensor that needs to be communicated, with the property that it is much easier/quicker to transfer $\mathcal{C}(x)$ than it is to transfer the original message x. While compression reduces the number of communicated bits in each communication round, it introduces errors, which generally leads to an increase in the number of rounds needed to find a solution of any desired accuracy. However, compression has been found useful in practice, as the trade-off often seems to prefer compression to no compression.
There are two large classes of compression operators (or compressors) that have been studied in the literature: i) *unbiased* compression operators and ii) *biased* compression operators. Let us explicitly give the definitions of these classes.

Definition 1 (Unbiased compression operator)**.** *A mapping $\mathcal{C}:\mathbb{R}^d\to\mathbb{R}^d$ is called **unbiased compression operator** if there exists $\omega\geq 0$ such that*

$$\mathbb{E}\left[\mathcal{C}(x)\right]=x,\qquad \mathbb{E}\left[\|\mathcal{C}(x)-x\|^2\right]\leq\omega\|x\|^2,\qquad \forall x\in\mathbb{R}^d; \tag{1}$$

Definition 2 (Biased compression operator). *A mapping $\mathcal{C} : \mathbb{R}^d \to \mathbb{R}^d$ is called **biased compression operator** there exists $0 < \alpha \leq 1$ such that*

$$\mathbb{E}\left[\|\mathcal{C}(x) - x\|^2\right] \leq (1-\alpha)\|x\|^2, \qquad \forall x \in \mathbb{R}^d. \tag{2}$$

At this moment, distributed methods using unbiased compressors (1) are very well understood such as [14]–[22]. However, the situation with biased compressors (2) is much more challenging. In practice, they could perform much better than unbiased ones; however, dealing with them in theory can be trickier. The key complication comes from the fact that their direct use within gradient-type methods can lead to divergence. To fix this problem, *Error Feedback* (EF) (or *Error Compensation* (EC)) was proposed by [23],which was a heuristic until recently. Indeed, very recently, [24] and [25] proposed with a new theoretically described EF mechanism called EF21, which enjoys the desirable $\mathcal{O}(1/T)$ convergence rate for the nonconvex case, improving the previous $\mathcal{O}(1/T^{2/3})$ rate of the standard EF mechanism [26]. In particular, the authors paid attention to the Top-k compressor [14] defined as

Definition 3 (Top-k compressor). *Let $(j_1, j_2, \ldots, j_d)$ be the such permutation of $(1, 2, \ldots, d)$ that $|x_{j_1}| \leq |x_{j_2}| \leq \cdots \leq |x_{j_d}|$ (the coordinates of x are ordered by their magnitudes) and e_{j_i} is the j_i-th unit coordinate vector in $\mathbb{R}^d$, then we define the mapping $\mathcal{C} : \mathbb{R}^d \to \mathbb{R}^d$ to be the following*

$$\mathcal{C}(x) \overset{def}{=} \sum_{i=d-k+1}^{d} x_{j_i} e_{j_i}. \tag{3}$$

and for which the the following property holds

Theorem 1 ([27]). *Let $\mathcal{C}$ be the Top-k compressor (Def. 3). Then $\mathcal{C}$ satisfies the inequality (2) with $\alpha = \frac{k}{d}$.*

Thus Top-k is an example of *biased* compressor (in view of the Def. 3). Let us give an example, illustrating the work of Top-k compressor for some fixed k and d.

Example 1 (Application of Top-2 to 5-dimensional vector). *Let $x = (1, -15, 2, -7, 10)^\top$, then $\mathcal{C}(x) = (0, -15, 0, 0, 10)^\top$.*

2 Project description and goals

In this section, we would like to focus our attention on the Top-k compressor.
Looking at its *greedy* nature, we might have a reasonable concern whether its application is practical in terms of real time. Its application in the machine learning algorithms often assumes that the computation of $\mathcal{C}(x)$ is being done by each node on each communication round. How much time do we need in practice to compute $\mathcal{C}(x)$ assuming that dimensionality d can be large (e.g. $10^6 - 10^9$)?
In this project, we would like to investigate these practical aspects from an algorithmic perspective by comparing two algorithms performing Top-k compression based on the Tournament method and Quickselect based approach. We give their rigorous theoretical analysis and provide with the simulation experiments shedding some light to the answers of the aforementioned questions.

3 Top-k compression as a classical selection problem

The compression technique described earlier could be formulated as a selection problem of finding the k-th largest parameters in federated models, gradients, or tensors. The selection problem was initially introduced in 1883 by Lewis Carroll [28] who identified a flaw in the Tennis tournament design in which around 50% of the time a second-best player competes with the best player in earlier rounds of the tournament and thus get unfairly knocked out. Motivated by this problem, the analysis of selection algorithms gained attention. Several algorithms [29] [30] [31] were proposed in the 19th century to find the k-th largest element from d elements.
Notably, A. Hadian and M. Sobel [31] proposed one of the first upper bounds for finding any k-th largest element from d elements using knockout tournament method as:

$$d - k + (k-1)\lceil \log(d + 2 - k) \rceil, \quad d \geq k$$

(Note: all logarithmic functions in this report are binary logarithms)

Although Knuth [28] highlighted that this upper bound might not be accurate for large d and k, a practical implementation of knockout using dynamic programming [32] has shown similar results to Hadian and Sobel's upper bound. Therefore, we will use this practical implementation for our project.
Following Hadian and Sobel's work, M. Blum et al. [33] and Floyd et al. [34] proposed an improved upper bounds for large k and d using algorithms that are similar to Quickselect but have faster performance. In this project, we will mainly consider the simple version of Quickselect and possibly explore these variations as well.

4 Proposed algorithms

Tournament Method

Our first proposed selection algorithm uses a sequence of knockout tournaments as introduced by Hadian and Sobel [31]. The algorithm works as follows [28], we consider each element as a player that gets paired with another player in a competition (i.e., a comparison between two elements). First-round winners are promoted to the second round to compete against each other. Eventually, the best player (largest element) wins the tournament. Running the tournament for d elements requires $d-1$ comparisons.
To find the k-th element, we can use a modified version of the Tournament method as described in [28]. Simply, we construct a tree with a size of $d-k+2$ and keep $k-2$ elements held in reserve. After the algorithm finds the largest element, we replace its external node with a reserved element and recompute the path of the element recently replaced. This takes at most $\lceil log(d+2-k) \rceil$ comparisons. We repeat this procedure for $k-2$ times until we use all elements in reserve. Finally, we replace the external node of the current largest element by $-\infty$ and recompute the path of the element recently replaced to get the k-th element. The overall complexity can be reduced to $\mathcal{O}(d + k\log(d))$. (see Fig. 2 for an example of finding the 4th largest element)
Now that we obtained the k-th element, it is easy to get Top-k elements by looping over all elements and comparing with the k-th element. This process has a complexity of $\mathcal{O}(d)$.

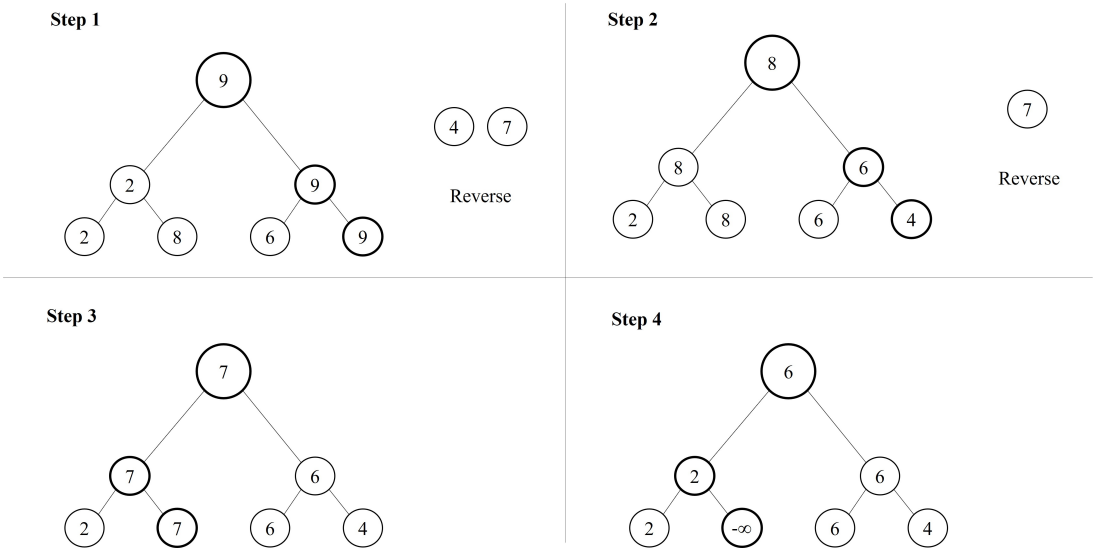


Figure 2: An example of finding the 4th largest element using the Tournament method. Step 1: Find the largest element while keeping two elements in reserve. Step 2: Replace the previous largest element (9 in this case) with a reserved element. Recompute the path of the recently replaced element (shown in bold borders). Step 3: Repeat Step 2. Step 4: Replace the current largest item with $-\infty$ and recompute the corresponding path to get the 4th largest element.

Complexity analysis

This algorithm requires constructing a tournament tree with $d-k+2$ elements and $d-k+1$ winners of all rounds. To represent this tree, we need a space complexity of $\mathcal{O}(2d-2k+3) = \mathcal{O}(d-k)$.
As for time complexity, running the modified version of Tournament method requires $\mathcal{O}(d + k\log(d))$ while performing the single loop to get Top-k has a cost of $\mathcal{O}(d)$. The final time complexity can be simplified to $\mathcal{O}(d + k\log(d))$.

Top k-th selection based on Quickselect

We implement Top-k selection algorithm based on Quickselect in two steps:

1. Find k-th largest element using Quickselect. Average Time complexity - $\mathcal{O}(d)$
2. Identify all elements bigger than the k-th largest element. Time complexity - $\mathcal{O}(d)$.

Quickselect is a recursive algorithm that is based on Quicksort algorithm [35], [36]. Converting Quicksort algorithm to Quickselect is very simple. Instead of searching recursively in both branches after a partition step, Quickselect chooses only the partition in which the k-th element could exist. This decrease of the problem size reduces the time complexity to a linear one.
Quickselect takes a set S of (distinct) d numbers where S = $[a_1, a_2, \ldots, a_d]$, and calculates the k-th largest number in that set [37]. Figure 3 describes the proposed algorithm. For each call of the recursive algorithm select(S, k), a pivot element P is chosen. Then, S is split into S_- (has elements smaller than P) and S_+ (has elements greater than P).

- If the size of $\mathcal{S}_+ = k-1$, then k-th largest number is P.

- If the size of $\mathcal{S}_+ > k-1$, then k-th largest number exists in S_+. Call select(S_+, k) recursively.
- If the size of $\mathcal{S}_+ < k-1$, then k-th largest number exists in S_-. Call select(S-, k-1-sizeof(S_+)) recursively.

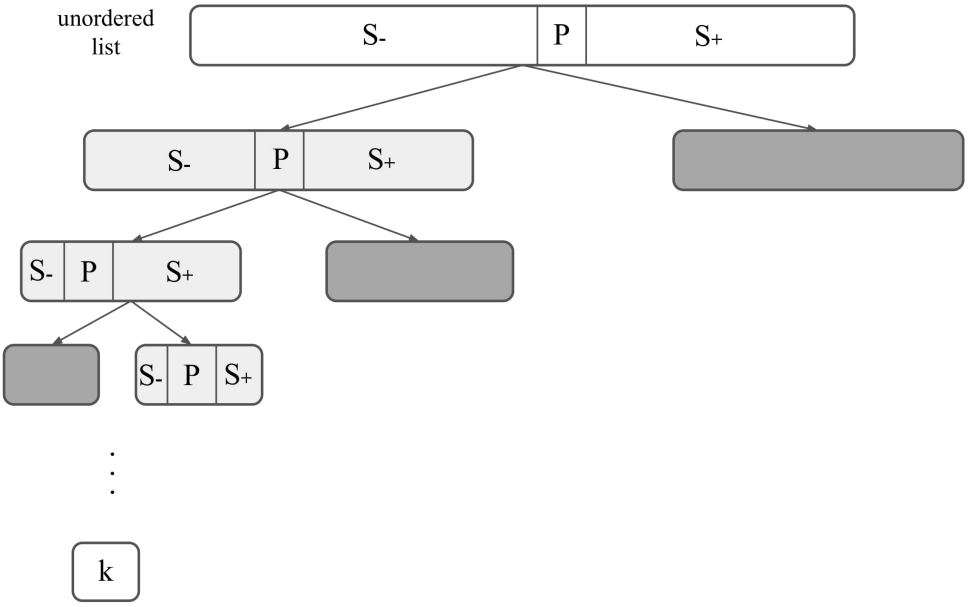


Figure 3: A recursive graph that shows how Quickselect decreases the search space in each iteration.[38]

Below is a pseudocode implementation of Quickselect algorithm based on Lomuto partition scheme [39]:

Algorithm 1 Lomuto Partition Scheme

```
function partition(list, left, right, pivot)
    pivotValue := list[pivot]
    swap list[pivot] and list[right]                    ▷ Move pivot to end
    storeIndex := left
    for i from left to right − 1 do
        if list[i] < pivotValue then
            swap list[storeIndex] and list[i]
            increment storeIndex
        end if
        swap list[right] and list[storeIndex]           ▷ Move pivot to its final place
        return storeIndex
    end for
end function
```

Algorithm 2 Quickselect

```
function quickselect(list, left, right, k)
    if left = right then                                ▷ If the list contains only one element,
        return list[left]                               ▷ return that element
    end if
    pivot := left + floor(rand() % (right − left + 1))  ▷ select a pivot between left and right
    pivot := partition(list, left, right, pivot)        ▷ The pivot is in its final sorted position
    if k = pivot then
        return ist[k]
    else if k < pivot then
        return quickselect(list, left, pivot − 1, k)
    else
        return quickselect(list, pivot + 1, right, k)
    end if
end function
```

After we find k-th element in an array, on the next step we compare the rest the elements with the k-th element to find top k elements. This operation requires d number of comparisons.

Pivot element

The method of choosing the pivot element for each iteration significantly affects both the theoretical time complexity and the practical average time complexity. There are three main ways to choose the pivot element:

1. Using a deterministic method (such as the last element or the middle element). While this method is simple and adds zero overhead, its worst case time complexity is $\mathcal{O}(d^2)$. This method is also sensitive to the input. If, for example, the input is sorted in non-decreasing order and the pivot is chosen as the first element, $\mathcal{O}(d^2)$ time is guaranteed to occur.
2. Choosing a random pivot on each iteration. This method is similar to the first one in that it still has a time complexity of $\mathcal{O}(d^2)$. However, this is extraordinarily unlikely and, on average, linear time complexity is almost always guaranteed.[33]
3. Median of medians. This method divides the input into chunks, finds the median of each chunk, then finds the median of those medians. The theoretical worst case time complexity of this algorithm is $O(d)$. However, the added overhead and increased average time can make this method worse than random pivot in practical terms[34].

Complexity analysis

This algorithm could be implemented in-place and should have $\mathcal{O}(1)$ space complexity.
Worst case time complexity for this algorithm depends on how the pivot is chosen (if it is random then $\mathcal{O}(d^2)$). Average time-complexity should be linear $\mathcal{O}(d)$. More strict theoretical estimation shows that an expected time-complexity will be upper-bounded by $9d$ [37].

5 Experiments

We performed an extensive amount of numerical experiments comparing the aforementioned Top-k algorithms on different settings and setups. In this section we provide explanations of the obtained results. experiments in this section can be divided into two subsections defined as follows:

1. Synthetic experiments
2. Real experiments

5.1 Synthetic experiments

In this part, we test and compare the performance of Top-k algorithms on a set of inputs coming from some priorly known distributions. Plots in this part are shown for values of d from 10^2 to 10^5 where each increment is an order of magnitude higher. For each value of d, we choose these four values of $k = \{0.1d, 0.2d, 0.5d, 0.7d\}$.
Before analyzing the complexity of Top-k algorithms, it is important to identify whether the input data drawn from a known prior distribution has an impact on the complexity of the results. The details of the prior analysis are discussed in the next section 5.1.1 while the number of comparisons results are detailed in sections 5.1.2-5.1.4.

5.1.1 Impact of the prior distribution

To understand the relationship between the choice of the prior distribution and the number of comparisons, we measure the number of comparisons for each algorithm type when input data are drawn from two distributions: Standard Normal distribution and Uniform distribution in the range of $(-100, 100)$. The experiment is performed on both Tournament method and Quickselect (3 pivots options).
The results shown in Figures 4-7 demonstrates that the choice of prior distribution has little impact on the number of comparisons. Furthermore, we observe differences in the variances of the results from each algorithm type which we will comment on in the next part.

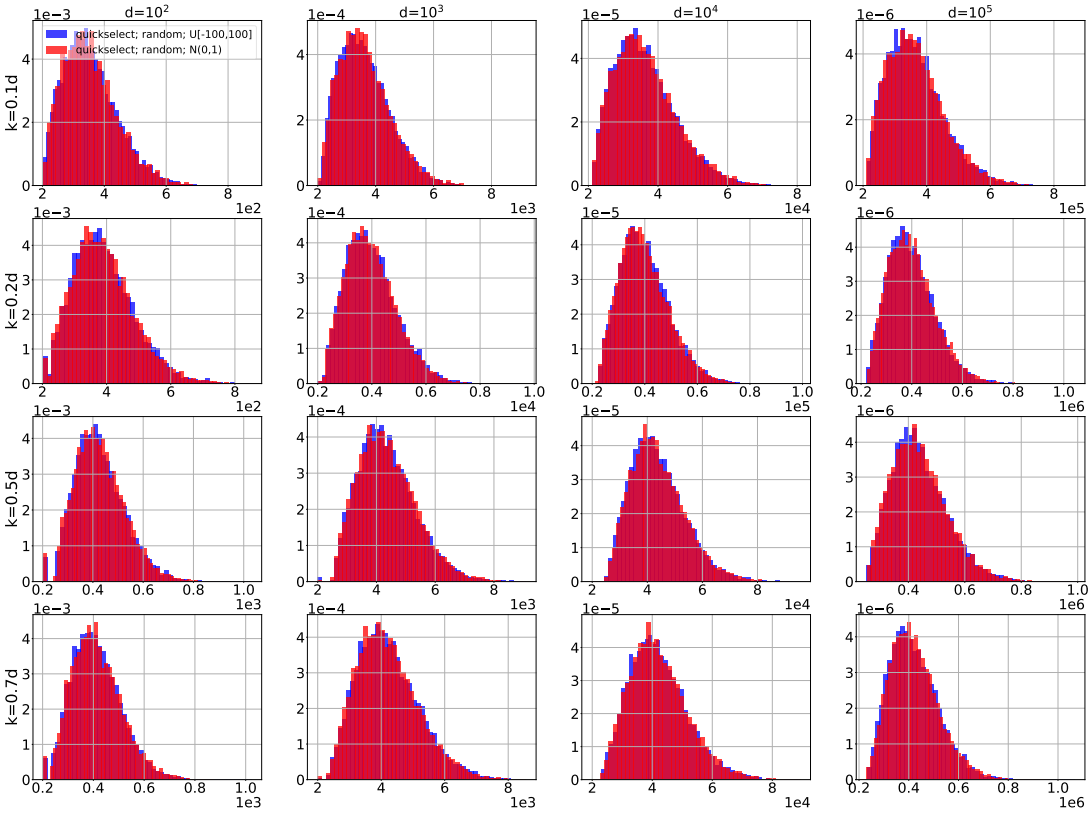


Figure 4: Histograms of number of comparisons on two distributions using Quickselect with random pivot

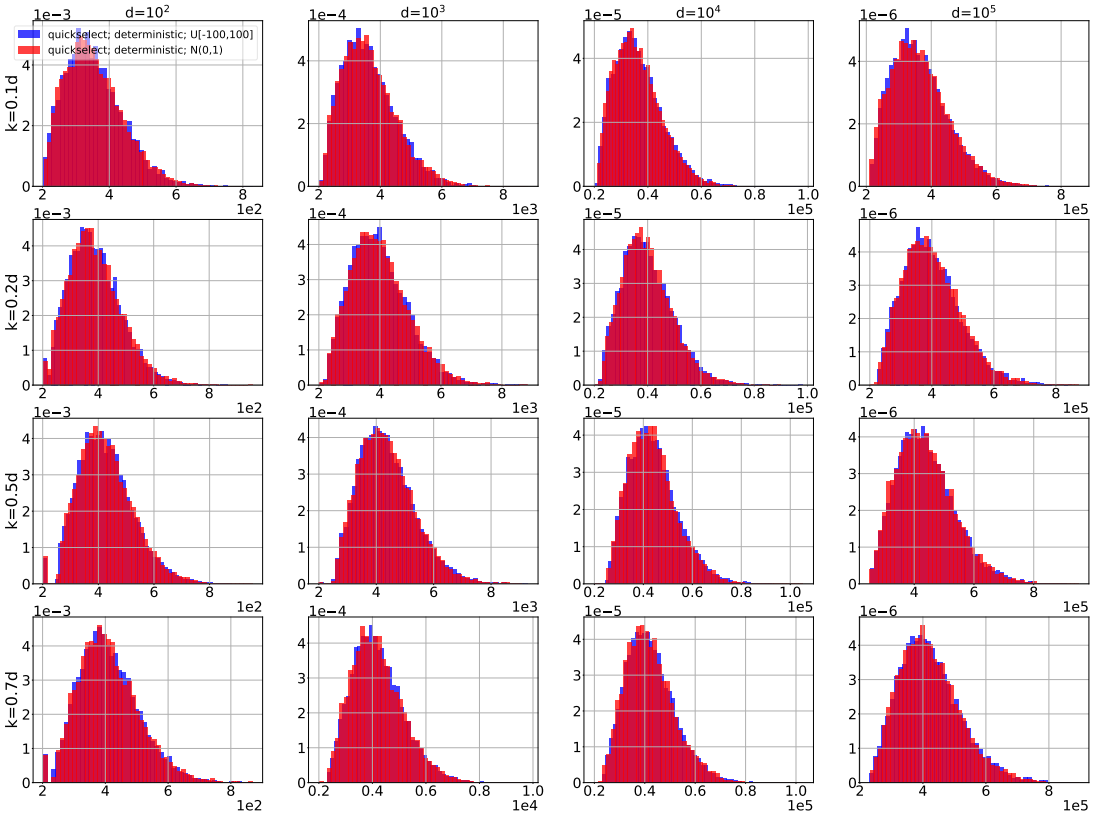


Figure 5: Histograms of number of comparisons on two distributions using Quickselect with deterministic pivot

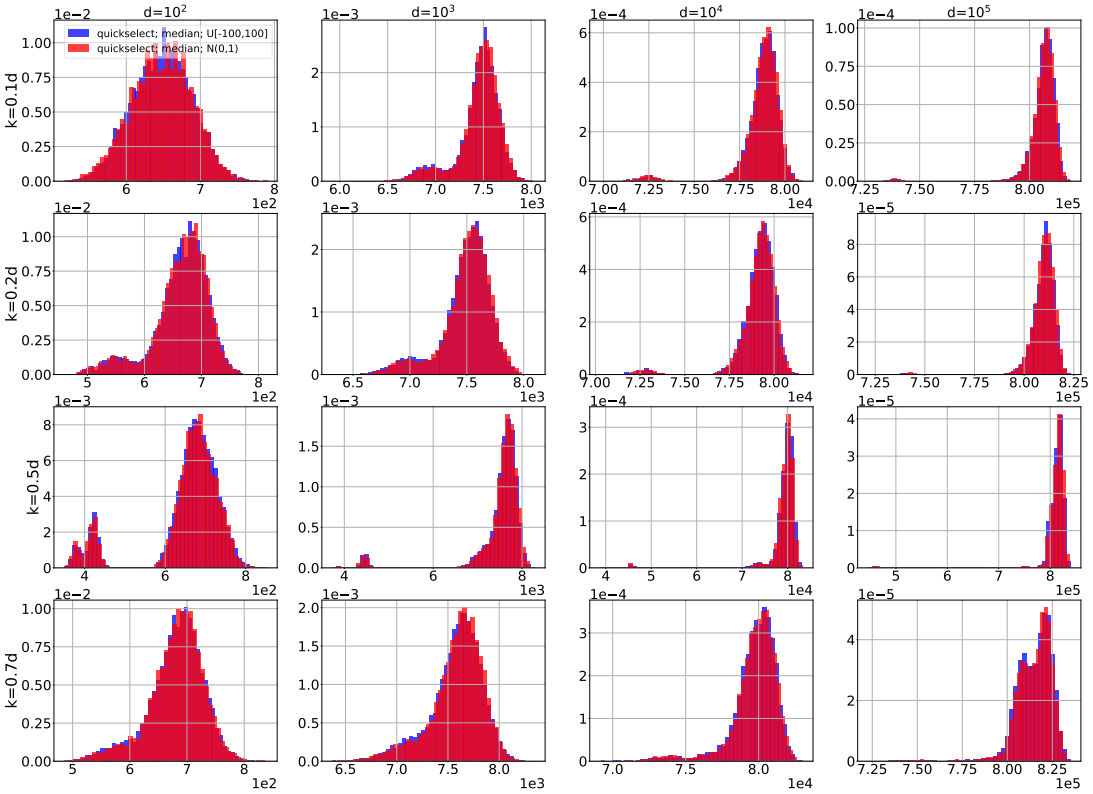


Figure 6: Histograms of number of comparisons on two distributions using Quickselect with median of medians pivot

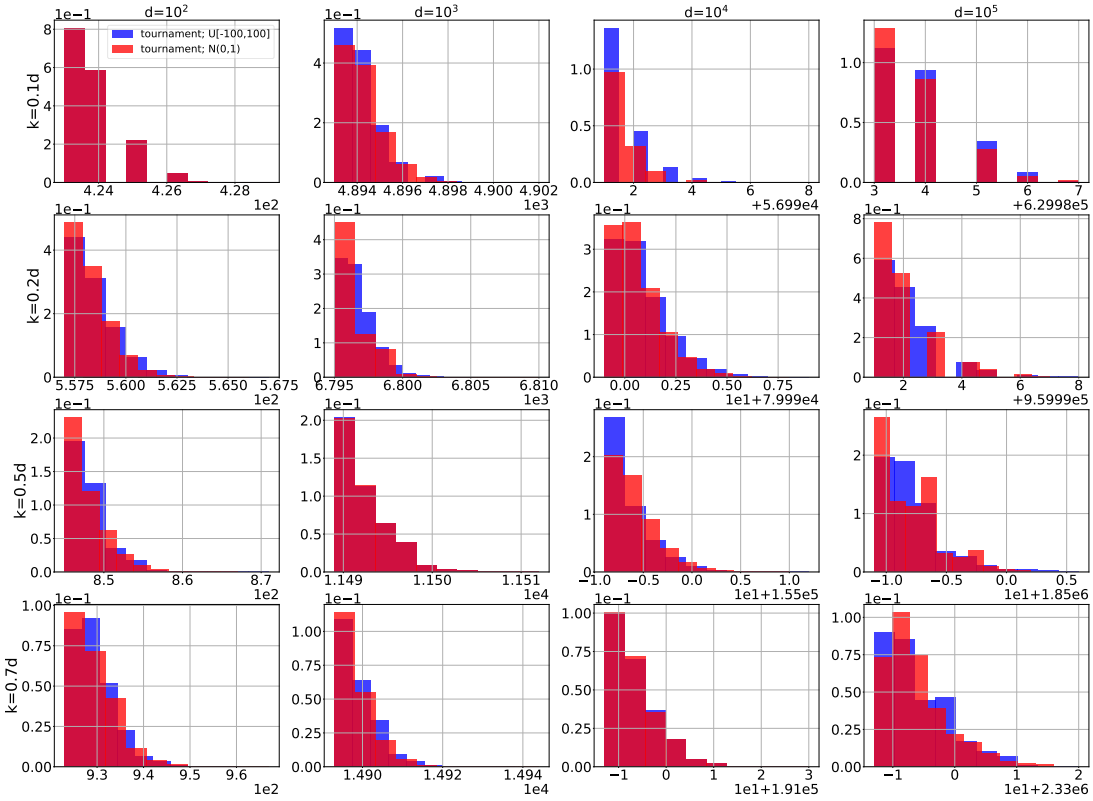


Figure 7: Histograms of number of comparisons on two distributions using Tournament method

5.1.2 What Quickselect is the best and how are they compared to Tournament

In this part, we compare the performance of the different Top-k algorithms in terms of number of comparisons. The input data is the set of Standard Gaussian vectors of size d.
For better visualization, we show the results in two plots. Figure 8 is a number of comparison histogram for Quickselect methods only while Figure 9 is a number of comparison histogram with y-axis in log scale for Quickselect methods and Tournament

method. Note that for large d, the Tournament method gets slow with high number of comparisons and its histogram becomes very thin and hard to see.
We observe that the results for Quickselect algorithm with random and deterministic pivots have larger variance compared to Quickselect with median of medians pivot. On the other hand, the Tournament algorithm has the lowest variance.
In general, Quickselect with random and deterministic pivots show similar results, outperforming median pivot and Tournament method. For $k < 0.1d$, Tournament method performs similarly to Quickselect but since it has a small variance it might be preferred over Quickselect.

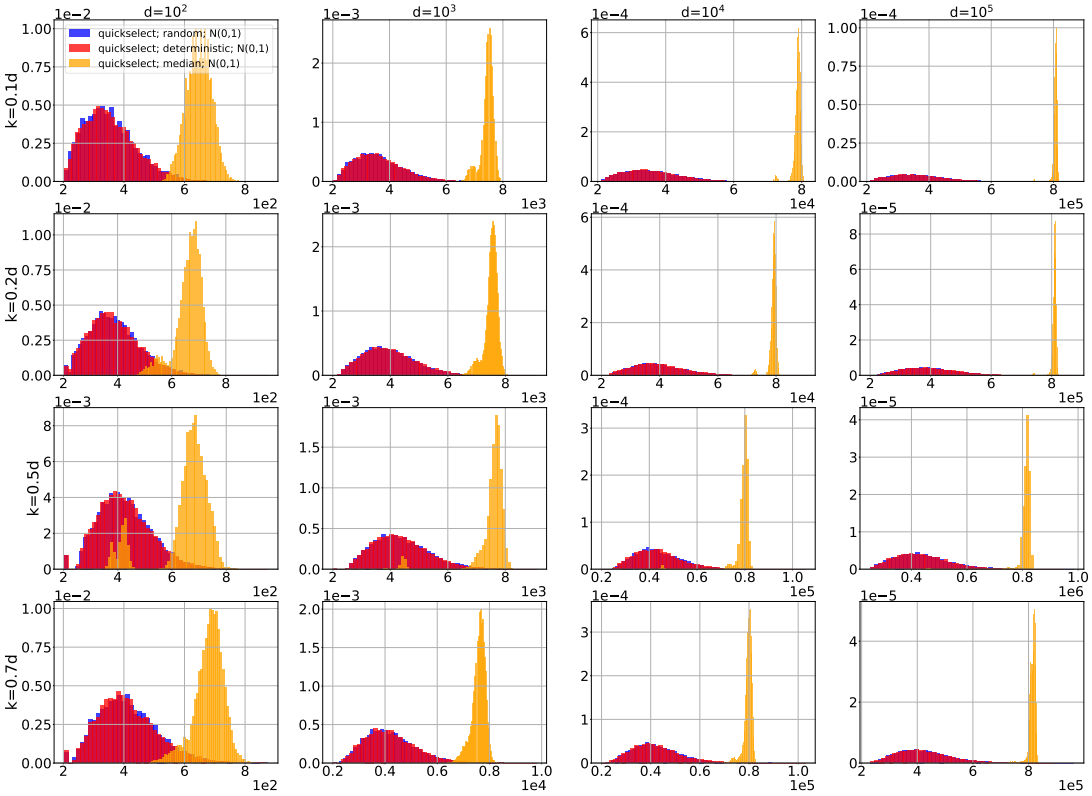


Figure 8: Performance of Quickselect methods in terms of number of comparison

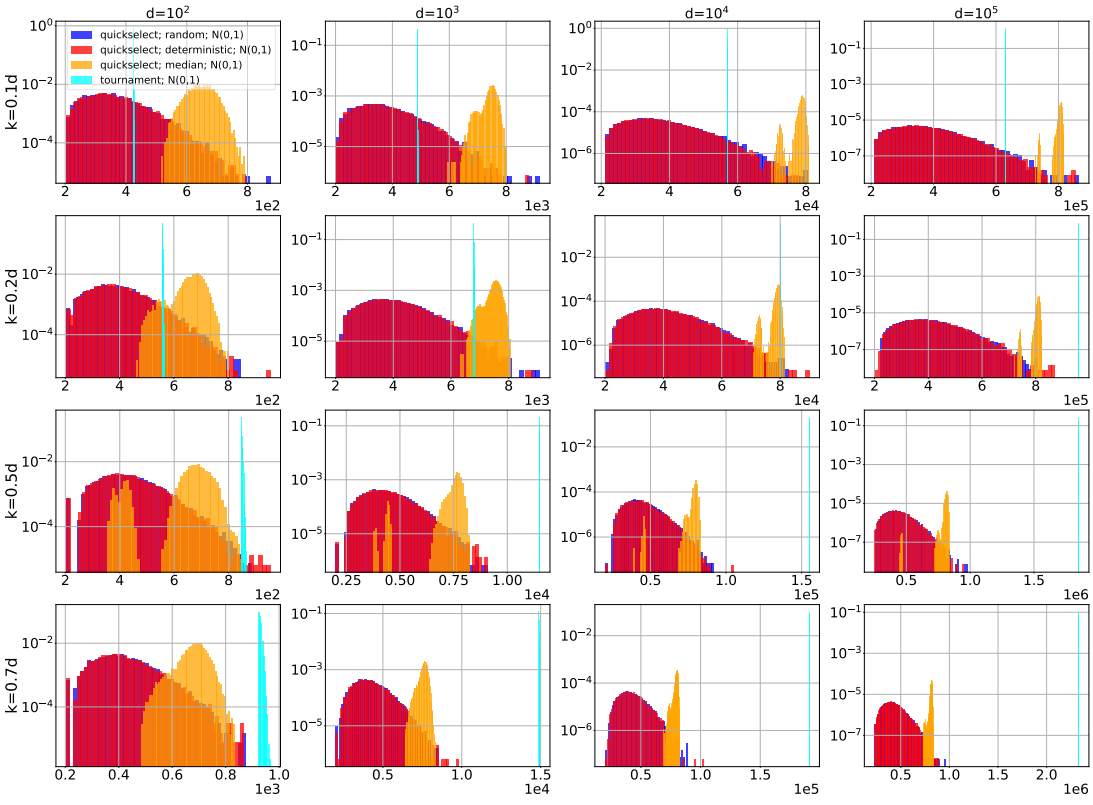


Figure 9: Performance of Quickselect methods and Tournament method in terms of number of comparison

5.1.3 Dependence on k for fixed d

Now, we consider the relationship between k and number of comparisons for a fixed size d. This analysis helps in deciding how to choose among algorithms given a value of k. As shown in Figure 10, the Tournament method has the lowest number of comparisons for small $k < 0.1d$ and small $d < 10^3$. For large k and d, Quickselect with Random and Deterministic pivots have the best performance.

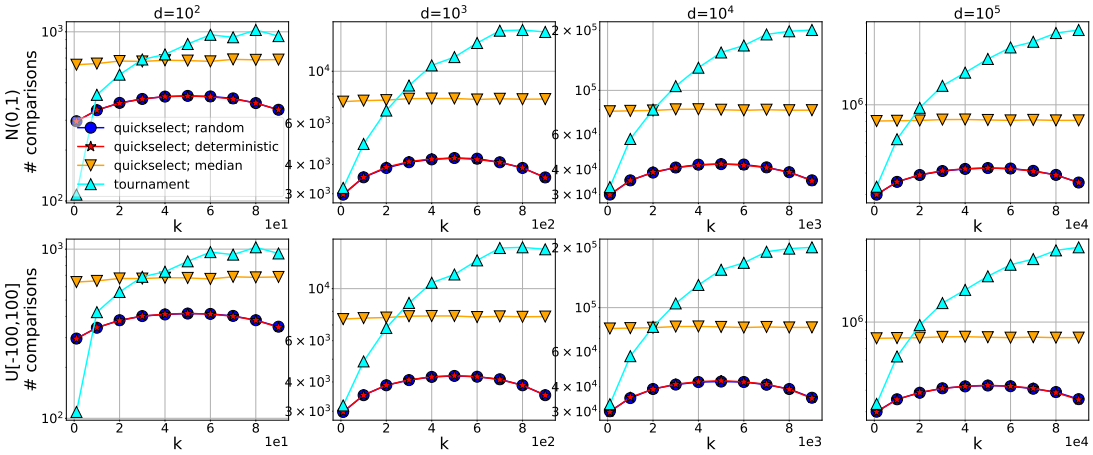


Figure 10: Relationship between number of comparisons and k for a fixed size d

5.1.4 Theoretical vs Experimental time complexity

In this part, we validate that the experimental time complexity matches with the theoretical one referenced in section 4. For each Top-k algorithm, we empirically found a constant number that when multiplied with the theoretical complexity would provide an upper bound for the experimental complexity as shown in Figure 11.

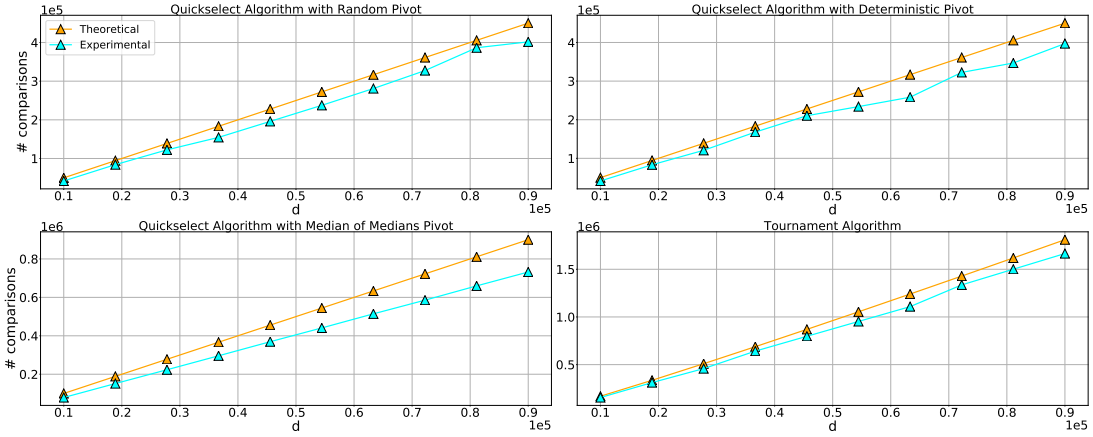


Figure 11: Theoretical and Experimental complexity for each of the Top-k algorithms

5.2 Real experiments

Machine learning model. In this part, we test and compare the performance of Top-k algorithms on a set of inputs coming from the training algorithm of the machine learning model. Let us now describe this setting in detail. As it was mentioned in Section 1, training of machine learning can be mathematically formulated in terms of the minimization of the function having a finite sum form, i.e. $\min_{x\in\mathbb{R}^d}\left[f(x) = \frac{1}{n}\sum_{i=1}^n f_i(x)\right]$. In our experiments we choose the problem of training *logistic regression* model with a non-convex regularizer which can be written in the form:

$$\min_{x\in\mathbb{R}^d}\left[f(x) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{N_i}\sum_{j=1}^{N_i}\log\left(1+\exp\left(-y_j a_j^\top x\right)\right) + \lambda\sum_{j=1}^{d}\frac{x_j^2}{1+x_j^2}\right], \tag{4}$$

where $a_i \in \mathbb{R}^d, y_i \in \{-1, 1\}$ are the training data (rows of the dataset matrix), $\lambda \geq 0$ is the regularization parameter, N_i is the number of datapoints of client i. Later, we will also use N as the total number of datapoints, i.e. $N \overset{\text{def}}{=} \sum_{i=1}^n N_i$ (see more details below). We choose this kind of function since it is frequently used as a comparatively simple benchmark for testing optimization algorithms on non-convex objectives [24], [25], [40]. Also, it is worth noting that optimizing non-convex objectives frequently appear in practice (eg. in neural networks), and these are more challenging functions to minimize than convex and strongly-convex ones.

Algorithm 3 EF21 (with Top-k compressor)

1: **Input:** starting point $x^0 \in \mathbb{R}^d$; parameter $1 \leq k \leq d$ of the Top-k compressor $\mathcal{C}$, $g_i^0 = \mathcal{C}(\nabla f_i(x^0))$ for $i = 1, \ldots, n$ (known by nodes and the master); learning rate $\gamma > 0$; $g^0 = \frac{1}{n}\sum_{i=1}^n g_i^0$ (known by master)
2: **for** $t = 0, 1, 2, \ldots, T-1$ **do**
3: Master computes $x^{t+1} = x^t - \gamma g^t$ and broadcasts x^{t+1} to all nodes
4: **for all nodes** $i = 1, \ldots, n$ **in parallel do**
5: Compress $c_i^t = \mathcal{C}(\nabla f_i(x^{t+1}) - g_i^t)$ and send c_i^t to the master
6: Update local state $g_i^{t+1} = g_i^t + \mathcal{C}(\nabla f_i(x^{t+1}) - g_i^t)$
7: **end for**
8: Master computes $g^{t+1} = \frac{1}{n}\sum_{i=1}^n g_i^{t+1}$ via $g^{t+1} = g^t + \frac{1}{n}\sum_{i=1}^n c_i^t$
9: **end for**

Optimization algorithm. We now describe how we apply Top-k to solve the problem (4). In view of the mentioned communication bottleneck issues appeared in the distributed settings, we will use mentioned in the Section 1 EF21 algorithm [1] [2] (see Alg. 3) with the contractive compressor $\mathcal{C}$ to be exactly the Top-k one. The EF21 algorithm is initialized with some model estimation[3] $x^0 \in \mathbb{R}^d$, parameter k and initial states $g_i^0 = \mathcal{C}(\nabla f_i(x^0))$ for $i = 1, \ldots, n$ storing on each node i independently. On the iteration $t+1$ (for $t = 0, \ldots, T-1$)[4], Top-k algorithms apply the compression individually on each node to the difference $\nabla f_i(x^{t+1}) - g_i^t$, and communicate the compressed gradients to the master. Note that we only need to communicate the compressed differences $\mathcal{C}(\nabla f_i(x^{t+1}) - g_i^t)$ since the additive terms g_i^t appearing in the Line (6) were communicated in the previous round. Master then averages all gradient estimators and obtain $g^{t+1} = \frac{1}{n}\sum_{i=1}^n g_i^{t+1}$. This can be done by computing $g^{t+1} = g^t + \frac{1}{n}\sum_{i=1}^n c_i^t$, where g^t is the averaged estimator from the previous round maintained by the master, and $c_i^t = \mathcal{C}(\nabla f_i(x^{t+1}) - g_i^t)$ are the compressed messages that workers sent by he workers. After this, the master does a gradient-like step, and then broadcasts the new model x^{t+2} to all nodes. By choosing this kind of optimization algorithm, we assume that in our setting, master-workers communication is cheap [5], and the only bottleneck of the method is worker-master communication. Later, by the term *experiment* we will assume a single run of the EF21 method at each iteration of which Top-k compression algorithms are applied (and their time complexities are stored).

Datasets, hardware and implementation. Regarding the training data, we use standard LibSVM datasets [45]. To simulate the distributed training environment, we split the dataset into the $n = 100$ parts to assign each worker exactly one piece from this partition (see Figure 1). The first $n-1$ clients own equal parts, and the remaining part, of size $N - n \cdot \lfloor N/n \rfloor$, is assigned to the last client. We consider the heterogeneous data distribution regime (i.e., we do not make additional assumptions on workers' data similarity). We test different values of k in the Top-k compressor being equal approximately to $0.01d, 0.1d, 0.25d, 0.5d$. A summary of datasets, details of splitting data among workers, and exact k values for each dataset can be found in Table 1. In all experiments, we keep the constant k in the Top-k compressor to be constant during the iteration procedure. The whole experimental environment (as well as the Top-k algorithms themselves) is implemented in Python 3.8 [6]; we use three different CPU cluster node types in all experiments:

1. AMD EPYC 7702 64-Core;
2. Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz;
3. Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz.

For all algorithms, at each iteration, we compute the squared norm of the exact/full gradient for measuring the convergence of the Algorithm to the stationary point[7]. Moreover, we track the average time complexity (number of comparisons) over the workers that are needed to compress the gradient difference $\nabla f(x^{t+1}) - g_i^t$ (see Line (5) of the Algorithm 3) by using all kind of the compressors considered in the project. We terminate our optimization procedure after certain amount of iteration or if the following stopping criterion is satisfied: $\|\nabla f(x^t)\|^2 \leq 10^{-7}$.

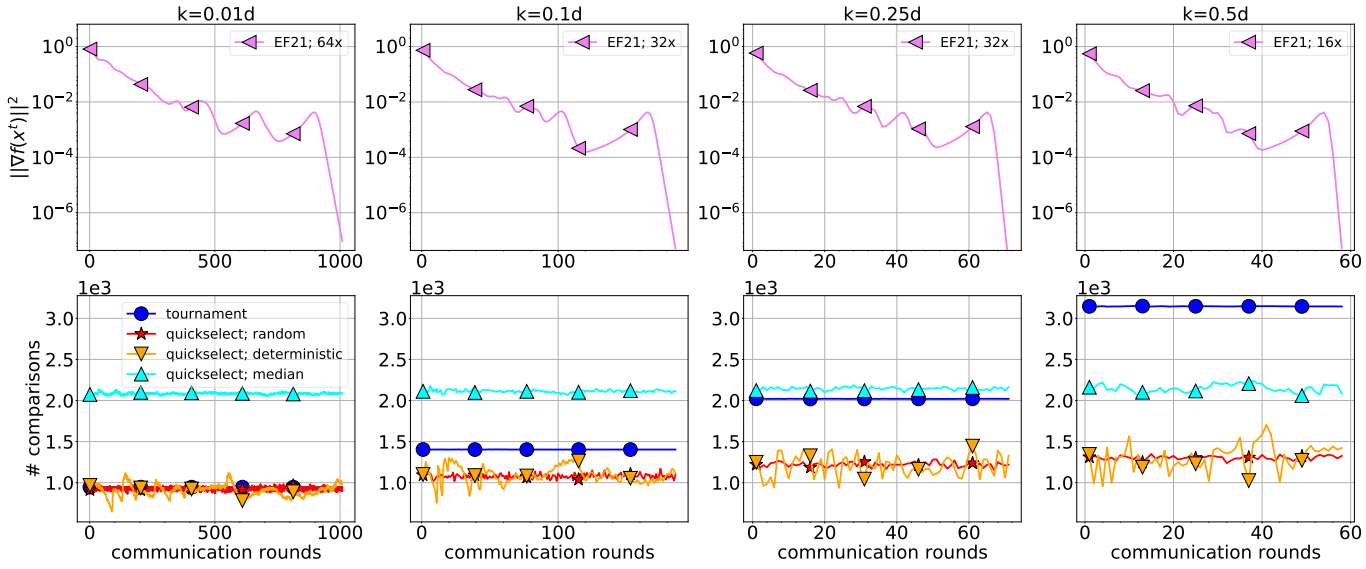
Table 1: Summary of the datasets and splitting of the data among clients. Here N_i denotes the number of datapoints per client.

Dataset	n	N (total # of datapoints)	d (# of features)	$k \approx 0.01d$	$k \approx 0.1d$	$k \approx 0.25d$	$k \approx 0.5d$	N_i
w8a	100	49,749	300	3	30	75	150	497
a9a	100	32,560	123	1	12	30	60	325

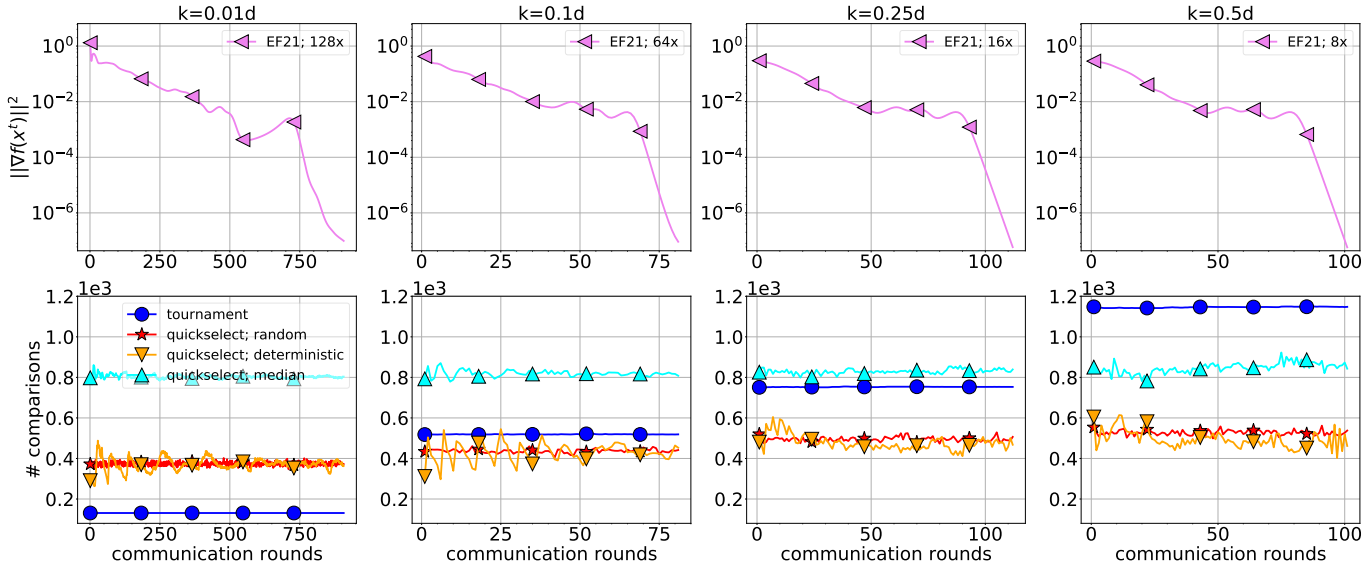
[1] For more details, we refer reader to the original paper [24] proposing this method.
[2] Implementation of this algorithm is freely available by its authors [41].
[3] In practise we take x^0 as a random vector sampled from standard Gaussian distribution.
[4] By T we mean some upperbound for the total number of iteration. Theoretical algorithms in the optimization are usually written using similar notation. One can also consider the stopping criterion to be a certain condition on the $\|x\|^2$.
[5] However, in some cases, this assumption does not hold. See [42]–[44] for references.
[6] Source code of the project is freely available on GitHub [46].
[7] This kind of measure is usually being tracked during the iteration procedure in the case of non-convex objectives

In all experiments, the stepsize γ of the EF21 algorithm me is set to the largest stepsize predicted by its theory (Theorem 1 from [24]) multiplied by some constant multiplier which was individually tuned in all cases within the set

$$\{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096\}.$$



(a) Experiments on w8a dataset



(b) Experiments on a9a dataset

Figure 12: Comparison of time complexities for Tournament and Quickselect algorithms and their dependendence on the iteration counter. By $1\times, 2\times, 4\times$ (and so on) we indicate that the stepsize was set to a multiple of the largest stepsize predicted by theory for EF21.

Experimental results. In the Figures 12a and 12b, top rows shows the convergence of EF21 method in terms of communication rounds[8] from the same starting point x^0 [9] to some stationary point. Each column relates to a single k value. Looking at the bottom rows of the Figures 12a and 12b, one can make the following observations:

1. For all Top-k algorithms, their time complexity does not change with the increase of the number of communication rounds; According to the EF21 theory, during the optimization procedure, the value $\sum_{i=1}^{n} \left\|\nabla f_i(x^{t+1}) - g_i^t\right\|^2$ converges to 0. And in fact, by the design of the Top-k algorithms, their time complexity depends on the relation between entries but not on the norm on the input. Therefore this kind of observation is reasonable.

2. Time complexity for median pivot choice does not change with the increase of parameter k and remain the slowest version of Quickselect in all experiments;

[8] Note that, in EF21 method, workers communicate on each iteration

[9] Starting point x^0 is shared for different k values in experiments on the same dataset.

3. Time complexity for deterministic and random pivot choices slightly grows as k increases, whereas for the Tournament method, it's rise is more crucial noticeable;
4. However for $k \approx 0.01d$ Tournament method shows either compatible (for `w8a`) or significantly better time complexity results (for `a9a`) than for Quickselect;
5. For `w8a` dataset (for which the parameter d is about twice as bigger than for `a9a`), each Top-k algorithm has about twice as bigger time complexity than for `a9a` dataset.

Conclusion. The behavior of Top-k algorithms (as well as dependence on k and d) observed in the real experiments is similar to the results on synthetic data and does not contradict the theoretical bounds.

6 Conclusion

In this project, we investigated the time complexity of Tournament and Quickselect algorithms, with three pivot choices, to perform Top-k compression. We have conducted a considerable number of experiments with significant sample sizes to capture the variability of each algorithm. The experiments results demonstrate that the prior distribution used to generate input data has little impact on the performance of our considered selection algorithms. Furthermore, we provide several analyses on the dependency between algorithms' number of comparisons, the values of input size d, and the parameter k. We hope that our results provide a basis to guide the choice of a selection algorithm among considered ones given values of d and k. Finally, the experimental time complexity we obtained for each algorithm has closely matched the corresponding theoretical complexity.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[2] S. Dargan, M. Kumar, M. R. Ayyagari, and G. Kumar, "A survey of deep learning and its applications: A new paradigm to machine learning," *Archives of Computational Methods in Engineering*, vol. 27, no. 4, pp. 1071–1092, 2020.

[3] S. Vaswani, F. Bach, and M. Schmidt, "Fast and faster convergence of sgd for over-parameterized models and an accelerated perceptron," in *The 22nd International Conference on Artificial Intelligence and Statistics*, PMLR, 2019, pp. 1195–1204.

[4] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[5] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*, PMLR, 2017, pp. 1273–1282.

[6] E. Gorbunov, "Linearly converging error compensated sgd," in *Federated Learning One World Seminar*, 2020.

[7] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[8] Y. You, J. Li, S. Reddi, *et al.*, "Large batch optimization for deep learning: Training bert in 76 minutes," *arXiv preprint arXiv:1904.00962*, 2019.

[9] S. Arora, N. Cohen, and E. Hazan, "On the optimization of deep networks: Implicit acceleration by overparameterization," in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

[10] A. Dutta, E. H. Bergou, A. M. Abdelmoniem, C.-Y. Ho, A. N. Sahu, M. Canini, and P. Kalnis, "On the discrepancy between the theoretical analysis and practical implementations of compressed communication for distributed deep learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 3817–3824.

[11] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[12] A. T. Suresh, X. Y. Felix, S. Kumar, and H. B. McMahan, "Distributed mean estimation with limited communication," in *International Conference on Machine Learning*, PMLR, 2017, pp. 3329–3337.

[13] J. Konečnỳ and P. Richtárik, "Randomized distributed mean estimation: Accuracy vs. communication," *Frontiers in Applied Mathematics and Statistics*, vol. 4, p. 62, 2018.

[14] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 1709–1720.

[15] S. Khirirat, H. R. Feyzmahdavian, and M. Johansson, "Distributed learning with compressed gradients," *arXiv preprint arXiv:1806.06573*, 2018.

[16] K. Mishchenko, E. Gorbunov, M. Takáč, and P. Richtárik, "Distributed learning with compressed gradient differences," *arXiv preprint arXiv:1901.09269*, 2019.

[17] S. Horváth, D. Kovalev, K. Mishchenko, S. Stich, and P. Richtárik, "Stochastic distributed learning with gradient quantization and variance reduction," *arXiv preprint arXiv:1904.05115*, 2019.

[18] Z. Li, D. Kovalev, X. Qian, and P. Richtárik, "Acceleration for compressed gradient descent in distributed and federated optimization," in *International Conference on Machine Learning (ICML)*, 2020.

[19] Z. Li and P. Richtárik, "CANITA: Faster rates for distributed convex optimization with communication compression," *arXiv preprint arXiv:2107.09461*, 2021.

[20] ——, "A unified analysis of stochastic gradient methods for nonconvex federated optimization," *arXiv preprint arXiv:2006.07013*, 2020.

[21] R. Islamov, X. Qian, and P. Richtárik, "Distributed second order methods with fast rates and compressed communication," *arXiv preprint arXiv:2102.07158*, 2021.

[22] E. Gorbunov, K. Burlachenko, Z. Li, and P. Richtárik, "MARINA: Faster non-convex distributed learning with compression," *arXiv preprint arXiv:2102.07845*, 2021.

[23] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs," in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[24] P. Richtárik, I. Sokolov, and I. Fatkhullin, "Ef21: A new, simpler, theoretically better, and practically faster error feedback," *arXiv preprint arXiv:2106.05203*, 2021.

[25] I. Fatkhullin, I. Sokolov, E. Gorbunov, Z. Li, and P. Richtárik, "Ef21 with bells & whistles: Practical algorithmic extensions of modern error feedback," *arXiv preprint arXiv:2110.03294*, 2021.

[26] A. Koloskova, T. Lin, S. Stich, and M. Jaggi, "Decentralized deep learning with arbitrary communication compression," in *International Conference on Learning Representations (ICLR)*, 2020.

[27] A. Beznosikov, S. Horváth, P. Richtárik, and M. Safaryan, "On biased compression for distributed learning," *arXiv preprint arXiv:2002.12410*, 2020.

[28] D. E. Knuth, *The art of computer programming*. Pearson Education, 1997, vol. 3.

[29] J. Schreier, "On tournament elimination systems," *Mathesis Polska*, vol. 7, pp. 154–160, 1932.

[30] S. Kislitsyn, "On the selection of the k th element of an ordered set by pairwise comparisons," *Sibirskii Matematicheskii Zhurnal*, vol. 5, no. 3, pp. 557–564, 1964.

[31] A. Hadian and M. Sobel, "Selecting the t-th largest using binary errorless comparisons," University of Minnesota, Tech. Rep., 1969.

[32] M. Bhasin. (2012). "Finding kth minimum (partial ordering) – using tournament algorithm," [Online]. Available: http://malkit.blogspot.com/2012/07/finding-[]kth-[]minimum-[]partial-[]ordering.html (visited on 09/29/2021).

[33] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, R. E. Tarjan, *et al.*, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, no. 4, pp. 448–461, 1973.

[34] R. W. Floyd and R. L. Rivest, "Expected time bounds for selection," *Communications of the ACM*, vol. 18, no. 3, pp. 165–172, 1975.

[35] H. M. Mahmoud, R. Modarres, and R. T. Smythe, "Analysis of quickselect : An algorithm for order statistics," en, *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, vol. 29, no. 4, pp. 255–276, 1995.

[36] B. Vallée, J. Clément, j. A. fill james, and P. Flajolet, "The Number of Symbol Comparisons in QuickSort and QuickSelect," in *36th International Colloquium on Automata, Languages and Programming*, S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. Nikoletseas, and W. Thomas, Eds., ser. LNCS - Lecture Notes in Computer Science, vol. 5555, Rhodes, Greece: Springer, Jul. 2009, pp. 750–763. doi: 10.1007/978-[]3-[]642-[]02927-[]1_62.

[37] M. Moshkov, *Lecture notes in design and analysis of algorithms*, Sep. 2021.

[38] A. Rodríguez-Hoyos, J. Estrada-Jiménez, D. Rebollo-Monedero, A. Mohamad-Mezher, J. Parra-Arnau, and J. Forné, "The fast maximum distance to average vector (f-mdav): An algorithm for k-anonymous microaggregation in big data," *Engineering Applications of Artificial Intelligence*, vol. 90, Feb. 2020. doi: 10.1016/j.engappai.2020.103531.

[39] Wikipedia, *Quicksort — Wikipedia, the free encyclopedia*, http://en.wikipedia.org/w/index.php?title=Quicksort&oldid=1046926664, [Online; accessed 29-September-2021], 2021.

[40] A. Khaled and P. Richtárik, "Better theory for SGD in the nonconvex world," *arXiv preprint arXiv:2002.03329*, 2020.

[41] I. Sokolov, *Implementation of ef21 and ef mentods*, https://github.com/IgorSokoloff/ef21_experiments_source_code, 2021.

[42] S. Horváth, C.-Y. Ho, Ľ. Horváth, A. N. Sahu, M. Canini, and P. Richtárik, "Natural compression for distributed deep learning," *arXiv preprint arXiv:1905.10988*, 2019.

[43] H. Tang, X. Lian, C. Yu, T. Zhang, and J. Liu, "DoubleSqueeze: Parallel stochastic gradient descent with double-pass error-compensated compression," in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2020.

[44] C. Philippenko and A. Dieuleveut, "Bidirectional compression in heterogeneous settings for distributed or federated learning with partial participation: Tight convergence guarantees," *arXiv preprint arXiv:2006.14591*, 2020.

[45] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, pp. 1–27, 2011.

[46] I. Sokolov, Y. Ghunaim, and M. Makarenko, *Python implementation of top-k compressor based on quickselect and tournament methods*, https://github.com/IgorSokoloff/top-[]k-[]algorithms, 2021.