Distributed Summarization of Dynamic Data

Emma Alexander and Eric Balkanski

May 9, 2016

We affirm our awareness of the standards of the Harvard College Honor Code.

1 Introduction

In many large datasets, it is useful to have a summary of its content. One way to build this summary is to select a set of representative samples. For example, given a large collection of images, it might be useful to have a small collection of images that represent the entire dataset. This summary can be used to communicate an at-a-glance version of the dataset. The problem of finding a good representative set can be formulated as a clustering problem such that the greedy algorithm finds a near-optimal summary of the input data. However, clustering algorithms run in quadratic time, which makes the computation aspect of this problem unfeasible for large datasets. To address this issue, Mirzasoleiman et al. [7] show that near-optimal representatives of datasets can be found efficiently with a distributed greedy algorithm.

Many large real-world datasets are dynamic (Twitter feeds, criminal records, image collections, . . .) where elements are constantly being inserted and deleted. How should algorithms for data summarization react to such a dynamic setting? In this paper, we explore approaches for this setting with insertions and deletions and their tradeoffs. A (very) naive approach is to run the entire distributed data summarization algorithm for every insertion and deletion. Of course, the communication and computation costs associated with this approach are not desirable, even though it always guarantees a good summarization. Our main interest is in the tradeoff between communication complexity and quality of a summarization.

Can the communication complexity incurred by entirely recomputing a summary at every insertion and deletion be drastically diminished while still maintaining a good summary?

1.1 Our approach

In this work, we consider the distributed algorithm for data summarization from Mirzasoleiman et al. [7] and extend it to account for settings with insertions and deletions. Their exact algorithm will be described precisely in Section 2 and we start by only describing its high level ideas that will be important to discuss our approach. This algorithm first randomly partition the dataset consisting of elements in between multiple local machines. Each local machine computes a set of representatives for the elements it received and sends this set to a central machine. The central machine then aggregates all the local representatives and compute a central solution that is a subset of all these local representatives. This central solution is then the summary of the entire dataset.

Rerunning the entire algorithm at every insertion or deletion would cause every local machine to have to recompute its local solution, send it to the central machine, and the central machine to have to recompute the central solution. The main idea behind our approach is the following simple observation: a local solution does not necessarily need to send its updated local representatives to the central machine at every update.

More precisely, our extension for insertions and deletions uses a threshold approach. A local machine only sends to the central machines its updated local representatives when the difference between these representatives and the last set of representatives sent to the central machine exceeds a certain threshold. This approach is quite general, the measure of "difference" and the value of the threshold can be adapted as desired.

To test our approach, we use the dataset of criminal records for the city of Chicago from 2001 to present. Each crime record consists of multiple features and the goal is to find the most representative crimes from this dataset. Insertions and deletions are natural since new crimes happen every minute and some records might be erroneous or outdated. Applying the distributed data summarization algorithm to this dataset without updates is of interest in itself since, to the best of our knowledge, this algorithm has never been used for criminal records, but mostly for summarization of collections of images. Our first experiments therefore focus on finding good summaries without updates. The results we obtain make semantic sense (drug related arrests happen on streets and sidewalks, certain neighborhoods have much more crimes than others...) and we believe that they offer a simple and concise representation of the most common crimes in Chicago. Our other experiments focus on understanding the tradeoff between quality of summary and communication complexity. In these experiments, we demonstrate that very good summaries can be maintained with very little communication compared to the communication needed to recompute a summary at each insertion and deletion.

1.2 Previous work

The distributed algorithm for data summarization that we extend for insertions and deletions is from Mirzasoleiman et al. [7] and is for the general problem of distributed submodular maximization. In subsequent work, instead of picking at most k elements to maximize a submodular function, the question of minimizing the number of elements picked to achieve at least a certain value of a submodular function was studied in a distributed setting [8].

An alternate approaches for distributed submodular maximization and distributed clustering uses core-sets (Mirrokni and Zadimoghaddam [6], Indyk et al. [4], Balcan et al. [2], Bateni et al. [3]). A core-set is a subset of points such that a solution for this core-set is guaranteed to be approximately a good solution for the original set of points.

A completely different approach to data summarization for very large datasets is via streaming (Badanidiyuru et al. [1], Kumar et al. [5]). In the streaming model, elements arrive one by one in a stream and the algorithm maintains a small subset of the elements as the solution while never having access to the entire data set at the same time. A main difference between streaming algorithms and our approach to insertions is that the goal of streaming algorithms is to obtain a good solution when the stream ends and that it is assume to have finite length. We are interested in having a good solution at anytime and we allow for an unbounded number of insertions.

1.3 Paper organization

Preliminaries are in Section 2. We then describe and discuss our approach in Section 3. We demonstrate the effectiveness of our approach by implementing our system (Section 4) and running experiments on a real world criminal dataset (Section 5). Finally, we briefly discuss how to achieve fault tolerance in Section 6.

2 Preliminaries

2.1 Setup

There is a ground set $N = \{e_1, \dots, e_n\}$ of elements. The goal is to pick a small set $S \subseteq N$ of size at most some integer k that is a good summary of N. To measure the quality of a summary, we consider a clustering function measuring the loss $l(S)$ associated with a metric distance $d(e_i, e_j)$. The distance $d(e_i, e_j)$ between two elements e_i and e_j measures how similar these two elements are, this distance function is general and defined differently for different applications. An example of a distance function in the case where elements consists of multiple feature is the hamming distance between these two elements, i.e., how many features they differ on. Once we have a distance function, the clustering function loss $l(S)$ is then the sum over all elements in N of their distance to their best representative, i.e.,

$$l(S) = \sum_{e_i \in N} \min_{e_j \in S} d(e_i, e_j).$$

2.2 Submodularity

The objective function $f(S)$ is then defined to be

$$f(S) := l(e_0) - l(S \cup e_0)$$

for some auxiliary element e_0. The motivation for this definition is that $f(S)$ is monotone submodular. Submodularity is the desirable property of diminishing marginal return. Submodularity is desirable because it allows for theoretical guarantees. More precisely, the simple greedy algorithm is a $e/(e-1)$ approximation algorithm for maximizing a submodular function $f(S)$ under a cardinality constraint, i.e.,

$$\max_{\substack{|S| \leq k \\ S \subseteq N}} f(S).$$

Our goal is therefore a distributed algorithm for this optimization problem with the above definition of $f(S)$ and where insertions and deletions on the ground set N of elements occur.

2.3 Approximation ratio

The performance of an algorithm, i.e., the quality of the summary it computes, is measured by the approximation ratio α obtained by this algorithm, which is the ratio of the value of the optimal solution to the value of the solution it computes. More precisely, let S be the set returned by an algorithm, this solution is an α-approximation if

$$\alpha \cdot f(S) \geq f(S^\star)$$

where $S^\star$ is the optimal solution to the problem. An algorithm is then an α-approximation algorithm if it is guarantees to always return a solution that is an α-approximation, for all inputs.

2.4 Non-dynamic algorithm

We now formally describe the algorithm from Mirzasoleiman et al. [7] for distributed submodular maximization that does not deal with insertions and deletions. The ground set N is partitioned into m parts $V_1, \ldots, V_m$. There are m *local machines* each containing a part V_i. Each local machine runs the greedy algorithm on its part V_i and obtains a *local solution* $S_i \subseteq V_i$ of size k. Each local machine then sends its local solution to the *central machine*. The central machine then aggregates all local solutions and computes a *central solution* by running the greedy algorithm to pick a subset $S \subseteq \cup_i S_i$ that best represents a subset V of the ground set N that the central machine has access to.

By exploiting the submodularity of the problem, the above approach is a $e/((e-1) \cdot \max(m, k))$ approximation algorithm for the general case of submodular function. This approximation ratio is improved for datasets with certain geometric structures and/or very large datasets. In addition to theoretical guarantees, the effectiveness of this distributed algorithm is also demonstrated with experiments on large collections of images.

3 Our Approach

In this section, we describe and discuss our approach to dealing with dynamic data, i.e., insertions and deletions of elements to the ground set N in the context of the distributed algorithm from Mirzasoleiman et al. [7] described previously. Our approach is based on the simple observation that a local machine does not need to send its local solution to the central machine at every update. The goal is to minimize the communication complexity, which is the number of elements being sent between machines. As we will see, this communication complexity is minimized at the expense of the approximation ratio of the solution of the algorithm.

3.1 Warm-up: a naive algorithm

A naive approach to insertions and deletions is to recompute a local solution S_i and send this updated local solution to the central machine every time an element e is inserted or deleted on the machine with part V_i. Of course, the issue is that insertions and deletions may happen frequently with large datasets, in which case the communication complexity of this algorithm is very poor since local solutions are constantly being sent to the central machine. Note that the approximation ratio of the non-dynamic algorithm is always maintain here since this algorithm is equivalent to rerunning the entire non-dynamic algorithm at every insertion and deletion. Can we cut down the communication complexity of this naive algorithm while maintaining a good approximation ratio?

3.2 A thresholding algorithm

The main idea of our algorithm is that an updated local solution S_i is sent to the central machine if it has significantly changed compared to the last local solution S_i' sent to the central machine from this local machine. More precisely, let $d(S, T)$ be a distance metric between sets of elements measuring how close S and T are. Then, when an element e is inserted or deleted from V_i on a local machine, a local solution S_i is recomputed and if

$$d(S_i, S_i') \geq t$$

for some threshold t, then S_i is sent to the central machine.

This algorithm is general, the distance function $d(S, T)$ and the threshold t can be designed as desired for different applications. Note that as the threshold t increases, the communication complexity improves since local solutions are less often sent to the central machine. However, as t increase, the approximation ratio worsens since the central machine has access to increasingly outdated local solutions. The main tradeoff with this algorithm is therefore between the communication complexity and the approximation ratio. Our algorithm is designed so that this tradeoff can be tuned as desired with the threshold t. As we will show in Section 5, it is possible to drastically decrease the communication complexity while maintaining a good approximation ratio for the application we consider.

3.3 The distance function d(S,T)

As for the threshold t, the distance function $d(S, T)$ can be adapted for different applications and different needs. A simple and effective distance function is the hamming distance, i.e., how many elements S and T disagree on. More sophisticated distance functions may be dependent on the function $f(\cdot)$ that we wish to optimize. For example, consider inserting an element e to a local solution S_i which has high marginal contribution, meaning that e increases the value $f(S_i)$ significantly. Then even if the hamming distance between S_i and S_i' is only one, we might want to send this updated solution to the central machine since this one element has a large impact.

4 Implementation

The implementation of the system can be found at github.com/ericbalkanski/cs262project, and contains 5 python files that interact according to the diagram in Figure 1:

- `datamanager.py` contains functions:
  - `features`, used internally to preprocess data.
  - `initialinsert`, which inserts data from the large crimes set to the local files read by local and central processes, selecting which file uniformly at random.
  - `update`, which appends additional entries to these files at a set rate.
  - `simulate`, which calls `initialinsert`, starts local and central processes, then calls `update`. It also performs oracle scoring (see Section 5 for definition).
- `local.py` contains the single function `local`, which monitors a data file for updates, recalculates representatives each time that file is updated, and sends its representatives (and process ID number) to the central process using sockets when enough changes have occurred since its last communication. On each update, the system time, number of local data points, and local score are recorded.

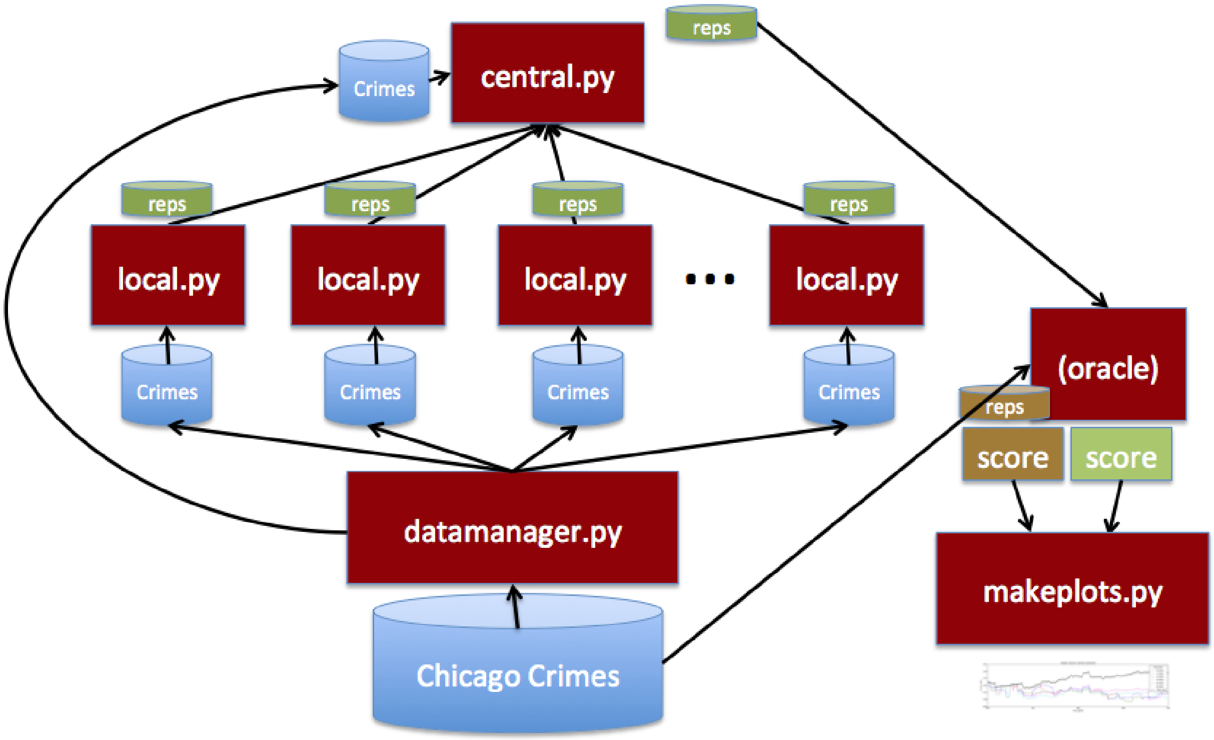

Figure 1: System diagram

Module or Function	Purpose in our System
`socket`	sends messages between local and central machines
`json`	encodes data to be sent over wire
`itertools.islice`	allows access to parts of files
`random.randint`	generates random integers for machine selection
`random.random`	generates random floats between 0 and 1, for deletion
`time.sleep`	allows pause before and between updates
`time.time`	allows system time in logs
`os.rename`	allows swap file for deleting entries
`csv.reader`	separates data elements without splitting on commas within quoted strings, as are found in our data
`multiprocessing`	starts local, central, and oracle simulations
`copy`	for manipulating list values

Table 1: Imported code

- `central.py` contains a single function `central`, which receives data over a socket from several local processes. When this happens, it selects from its received representatives those that best represent the contents of its data file and writes them to logs, along with system time, number of data points, and local score.

- `makeplots.py` can be called from the command line and reads central and oracle logs to produce plots of their raw scores and the ratio of the oracle score to each central process' score, per update.

- `greedy.py` contains several functions implementing the greedy algorithm of Mirzasoleiman et al. [7]. The two that are called by other components are:
  - `greedy`, which returns the representatives and their score for a dataset.
  - `score`, which only returns the $f(\cdot)$ score defined in Section 2. This is used to score the central representatives on the complete dataset.

Experiments are run by calling `datamanager.simulate` with the desired parameters, and the plots shown in Section 5 are generated from the resulting logs by calling `makeplots`. The system is built on top of a number of pre-existing components. The details are can be found in the documentation of each function, but a brief summary of the functions and modules imported can be found in table 1.

5 Experiments

Semantically, we found that our system's results were reasonable. The crimes were well-distributed, repeated crimes were generally quite different (a common feature was that battery would be repeated, with domestic events in the home showing up separately from nondomestic outdoor battery leading to arrest), but also showed that some times of day and districts were more or less likely to contain crime (e.g. few afternoon crimes, multiple representatives from "bad neighborhoods"). A sample set of 10 representatives is shown, in no particular order, in Figure 2.

Time of Day	Crime	Location	Arrest	Domestic	District
EVENING	BATTERY	RESIDENCE	false	false	006
NIGHT	NARCOTICS	STREET	true	false	011
NIGHT	THEFT	STREET	false	false	004
EARLY MORNING	CRIMINAL DAMAGE	APARTMENT	false	false	007
EVENING	NARCOTICS	SIDEWALK	true	false	011
NIGHT	BATTERY	APARTMENT	false	true	006
AFTERNOON	THEFT	STREET	false	false	025
MORNING	BURGLARY	RESIDENCE	false	false	009
NIGHT	BATTERY	SIDEWALK	false	false	012
EVENING	BATTERY	RESIDENCE	false	true	004

Figure 2: A summary of crimes in Chicago of size 10.

Quantitatively, we tested the performance of our algorithm for several thresholds t (see Section 3 for definition), finding 10 representatives initialized on 300 entries from the Chicago crime dataset with 2 local processes, as 200 insertions were performed. We only used 2 local processes to maximize the impact of a small number of insertions or deletions. On each insertion, the central solution corresponding to each threshold was scored, as was the oracle solution (defined below). The raw scores (defined below) are shown in Figure 3 for each insertion and threshold.

- **Raw score.** The raw score, also called f score, of a set S obtained is simply the value $f(S)$ where $f(\cdot)$ is defined in Section 2 and that we aim to maximize.
- **Oracle score.** The raw score obtained by running the greedy algorithm on the entire dataset.
- **Approximation ratio.** Note that it is computationally unfeasible to find the optimal solution maximizing $f(\cdot)$, we therefore use the oracle score as a proxy for the optimal solution when measuring approximation ratios, defined in Section 2. Recall that we wish to minimize the approximation ratio.

In black, the oracle's solution improves steadily, while in black stars a naive never-update algorithm is shown to perform better than one might expect, given the near-doubling of the dataset. The colored lines in between show that by using a threshold, performance can be increased even to level of the global greedy solution with far less frequent communication. More revealing is the approximation ratio between the oracle and central scores, shown in Figure 4.

This measurement, for which 1 is optimal and lower numbers indicate better performance, shows that thresholds improve performance over the naive baseline, though their ordering is fairly variable. We even see the thresholded performance underperforming the naive solution near 330 entries. This plot tells us that a threshold can be expected to usually but not strictly improve performance, and that in the case of this dataset the threshold can exceed half of the representatives (6 changes out of 10) without a serious loss in performance. The threshold of 8 cannot be seen in this plot because it is identical to the naive never-update performance, indicating a major change in behavior between these settings. Finding this tradeoff point for other datasets will determine the lowest communication cost for sensitivity to data changes.

For details on the generation of these plots and the underlying data, see the README file on github.

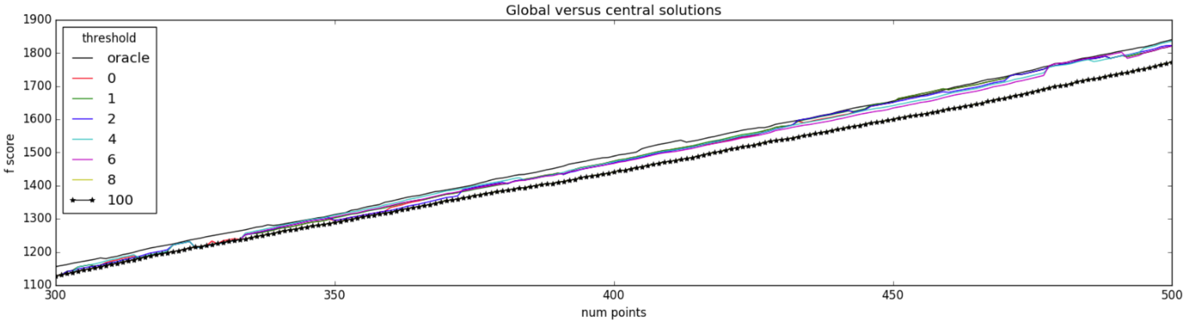

Figure 3: Raw scores for solutions found under varying thresholds t (colors), effectively infinite solution (black with stars), and with access to all data (black line).

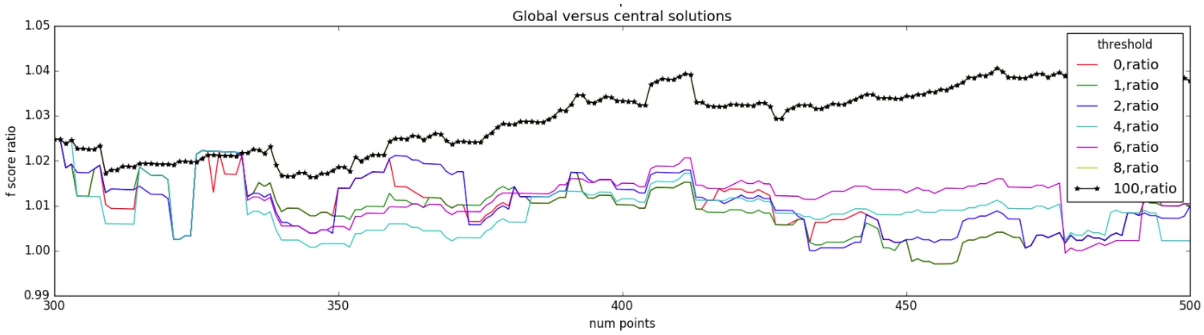

Figure 4: Approximation ratios obtained from figure 3.

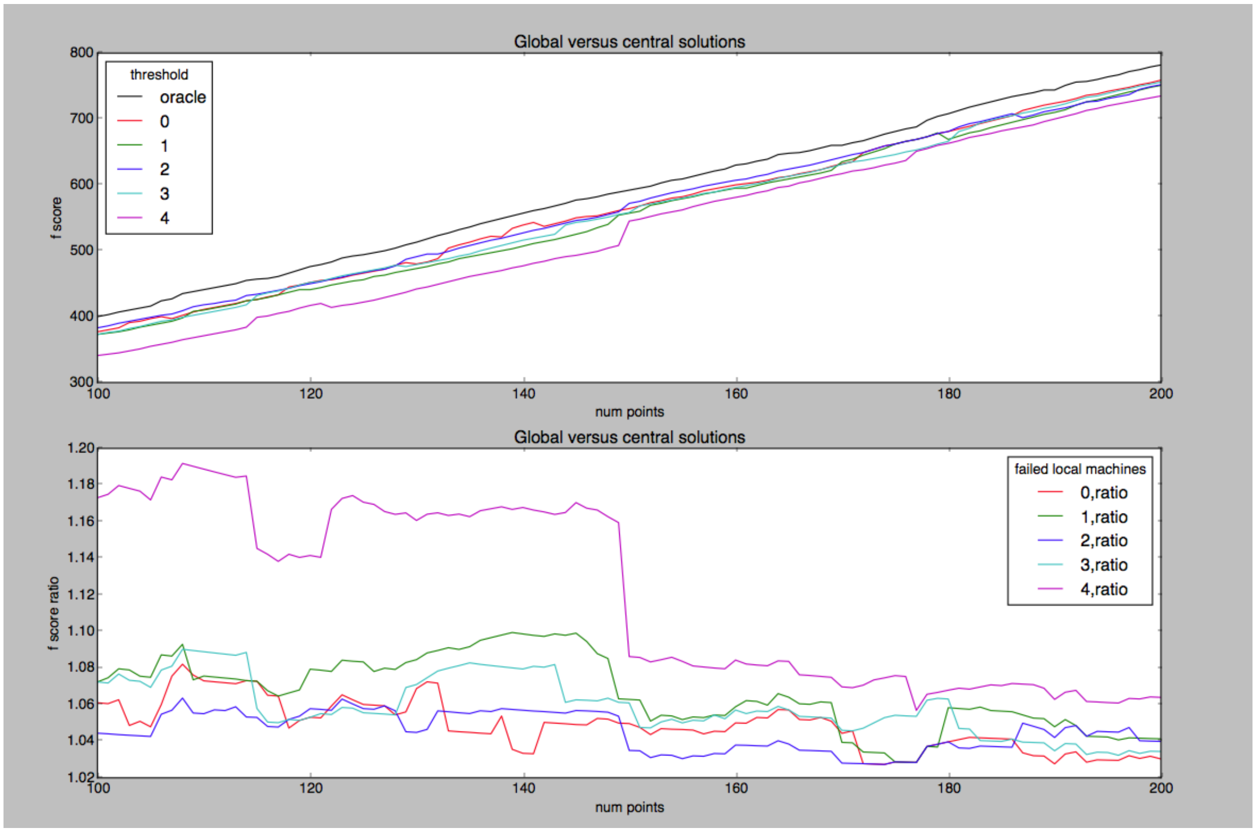

Figure 5: Robustness to local machine failure

6 Failures

We propose several natural approaches to achieve T-fault tolerance, both for local and central machines, where the different approaches differ in which aspect of the system is sacrificed to achieve such a fault tolerant system. For simplicity, we only discuss failures in the non-dynamic case where we are trying to build a summary from a fixed ground set of elements. Note that the issue of failure is not addressed in [7] where the non-dynamic algorithm is presented.

6.1 Failures on local machines

The first technique to achieve T-fault tolerance on local machines is very simple. The central machine waits until it receives $m - T$ local solutions from $m - T$ local machines before computing a central solution. It is easy to see that the central solution will be built unless there are more than T local machines that fail. The tradeoff here is between achieving a higher fault tolerance system and the *approximation ratio* since the central machine has access to less elements as if it waited for all local solutions.

Experimentally, we can demonstrate this fault tolerance. The central process in our system recalculates every time it receives an update and does not in fact wait to receive a full set of representatives before it selects from them. Having local processes fail, the central process is just left with fewer or less up-to-date candidates to choose from. Figure 5 shows the effect on the central process score and approximation ration of local machine failure. This plot shows the results of a simulation with 100 initial data points, 100 insertions, and 5 local files. Different colors correspond to different numbers of local machines that never communicate with the central server, so e.g. the pink line corresponds to a run in which the solution is only affected by 1/3 of the data (1/6 in the central process and 1/6 in the lone functioning local process). We see that even for a very small dataset, local failure degrades performance fairly gracefully.

The second approach maintains $T+1$ replications of each element from the ground set N on $T+1$ distinct local machines and, similarly as with the previous technique, the central machine waits to receive $m-T$ local solutions. Again, it is easy that this system is T-fault tolerant. The main difference with this approach is that instead of sacrificing the approximation ratio, the *memory per local machine* needed increases. The approximation ratio is not sacrificed anymore since for each element $e \in N$, the central machine will receive a solution from at least one local machine containing e.

6.2 Failures on central machine

Regarding failures on central machine, a natural approach is to run a leader election algorithm among the local machines and to rerun the non-dynamic algorithm if the central machine fails. The drawback of this approach is the *runtime* since the algorithm needs to rerun every time the central machine fails. To avoid sacrificing the runtime, an alternative approach is to have $T+1$ central machines instead of one. The sacrifice here is the *communication complexity* since the local machines must send their local solutions to $T+1$ different machines instead of one.

References

[1] Ashwinkumar Badanidiyuru, Baharan Mirzasoleiman, Amin Karbasi, and Andreas Krause. Streaming submodular maximization: Massive data summarization on the fly. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 671–680. ACM, 2014.

[2] Maria-Florina Balcan, Steven Ehrlich, and Yingyu Liang. Distributed clustering on graphs. *NIPS, page to appear*, 2013.

[3] MohammadHossein Bateni, Aditya Bhaskara, Silvio Lattanzi, and Vahab Mirrokni. Distributed balanced clustering via mapping coresets. In *Advances in Neural Information Processing Systems*, pages 2591–2599, 2014.

[4] Piotr Indyk, Sepideh Mahabadi, Mohammad Mahdian, and Vahab S Mirrokni. Composable core-sets for diversity and coverage maximization. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 100–108. ACM, 2014.

[5] Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. Fast greedy algorithms in mapreduce and streaming. *ACM Transactions on Parallel Computing*, 2(3):14, 2015.

[6] Vahab Mirrokni and Morteza Zadimoghaddam. Randomized composable core-sets for distributed submodular maximization. *arXiv preprint arXiv:1506.06715*, 2015.

[7] Baharan Mirzasoleiman, Amin Karbasi, Rik Sarkar, and Andreas Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *Advances in Neural Information Processing Systems*, pages 2049–2057, 2013.

[8] Baharan Mirzasoleiman, Amin Karbasi, Ashwinkumar Badanidiyuru, and Andreas Krause. Distributed submodular cover: Succinctly summarizing massive data. In *Advances in Neural Information Processing Systems*, pages 2863–2871, 2015.