Efficient Sample Generation for Scalable Meta Learning

Sebastian Schelter*, Juan Soto, Volker Markl
Technische Universität Berlin
firstname.lastname@tu-berlin.de

Douglas Burdick, Berthold Reinwald, Alexandre Evfimievski
IBM Almaden Research Center
{drburdic,reinwald,evfimi}@us.ibm.com

***Abstract*—Meta learning techniques such as cross-validation and ensemble learning are crucial for applying machine learning to real-world use cases. These techniques first generate samples from input data, and then train and evaluate machine learning models on these samples. For meta learning on large datasets, the efficient generation of samples becomes problematic, especially when the data is stored distributed in a block-partitioned representation, and processed on a shared-nothing cluster. We present a novel, parallel algorithm for efficient sample generation from large, block-partitioned datasets in a shared-nothing architecture. This algorithm executes in a single pass over the data, and minimizes inter-machine communication. The algorithm supports a wide variety of sample generation techniques through an embedded user-defined sampling function. We illustrate how to implement distributed sample generation for popular meta learning techniques such as hold-out tests, k-fold cross-validation, and bagging, using our algorithm and present an experimental evaluation on datasets with billions of datapoints.**

I. Introduction

Numerous applications benefit from statistical analysis over massive, complex datasets. Examples include the automatic detection of spam email, search engines learning to rank results, and media platforms which recommend music and videos to their users. For large datasets, these machine learning (ML) applications often require parallel processing platforms on shared-nothing clusters [6]–[10], and efficient training of ML models has received recent attention [4], [11], [12]. However, properly deploying ML in real-world settings requires more than efficient, scalable model training alone. Other 'meta' issues must be dealt with as well, such as model selection, parameter tuning and estimating model generalizability to new data. Another potential issue is that some complex ML applications may require more than one model to achieve satisfactory accuracy.

Data scientists typically address the first set of issues using *cross-validation* for estimating prediction quality of a model for a given problem [1]–[3], while the latter is addressed using *ensemble learning* to create a stronger model by combining weaker models [4], [5]. The wide variety of meta learning techniques for cross-validation and ensemble learning have two common steps. First, samples are generated from the input data. Second, ML models are trained and built on these samples. Although efficiently conducting cross-validation and ensemble learning at scale has been recognized as an important challenge [13], existing systems for distributed, scalable ML [6]–[9] lack a comprehensive meta learning layer supporting both steps of meta learning. These systems typically provide data-parallel techniques to efficiently perform model training and evaluation of the latter step, but do not readily support efficient, scalable sample generation required in the first step.

The focus of our work is in developing a scalable sample generation algorithm to support meta learning which meets three key desiderata. First, the common range of sampling methods used by different meta learning techniques must be supported. Adaption to a particular sample generation technique must only require minimal modification of the algorithm. Second, for reasons of efficiency, the algorithm must only conduct a single pass over the data, even for a large number of samples to generate. Third, the algorithm must be general enough to be used in a range of distributed, shared-nothing platforms.

Obtaining these desiderata for sample generation from large datasets in a ML setting is challenging for three reasons. First, many scalable ML systems represent data as *block-partitioned matrices* [6], [9], [10], which leads to the rows and columns of the input matrix being spread across multiple blocks, each stored separately, potentially on different machines in a distributed file system. Thus, sample generation must also consume distributed, blocked data as input and produce distributed, block-partitioned sample matrices as output. If we choose to sample a row or column from the given data, we must ensure that it is correctly restored in the newly formed blocks of the generated sample matrix.

The second challenge is that meta learning techniques may require to read a composition of generated samples (e.g., during training on $k-1$ folds in a round of k-fold cross-validation, see Section IV-B for details). This composition is not easily possible with a blocked representation. Naïve solutions for this challenge pose a scalability bottleneck as they generate many copies of the input data.

The third challenge arises from the difficulty of mapping particular sampling techniques to a distributed, shared-nothing setting. For example, sampling with replacement is difficult to conduct in a distributed setting, as the number of occurrences of individual observations in a sample is correlated [11], [14]. Because of this correlation, we cannot easily generate the components of a sample with replacement in parallel, for reasons described in Section IV-C.

The contributions of this paper are the following:

- We present a novel, general algorithm for distributed sample generation from block-partitioned matrices in a single pass over the data, which leverages *skip-ahead* random number generators to sample partitioned rows or columns in parallel. (Section III)

preprint

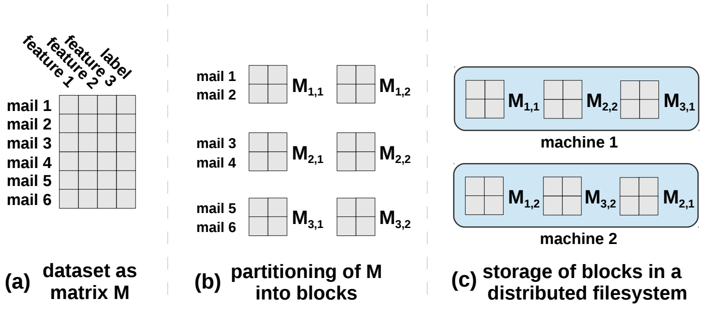
* work performed while visiting IBM Almaden Research Center

- We discuss how to efficiently generate samples for a variety of meta learning techniques via a user-defined sampling function embeddable in our algorithm. This approach allows running different sampling techniques by modifying only a few lines of code while still guaranteeing efficient sample generation. (Section IV)
- We extend our algorithm to support generating blocked training sets for k-fold cross-validation, which requires aggregate size almost equal to the given data. (Section IV-B)
- We show how our method supports distributed sampling with replacement by leveraging an approach to compute a multinomial random variate in a distributed, parallel fashion. (Section IV-C)
- We perform a thorough experimental evaluation of our algorithm on datasets with billions of datapoints. Our proposed algorithm scales linearly with input data size and number of samples, and is up to an order of magnitude faster than matrix-based sample generation techniques. (Section V)

II. Background

A. Meta Learning Example

Our running example for the role of meta learning in ML involves the use case of automatic spam detection. Given a dataset of labeled emails, both spam and non-spam, obtain a predictive model which can accurately classify newly arriving emails as spam or non-spam. One example of a predictive model would be a k-nearest neighbor (k-NN) classifier, which for a given unseen email determines its label using labels of the k most similar emails in the training data. The value of k serves as a *hyperparameter* for k-NN, and must be tuned to gain sufficient quality of the learned classifier. An example for automating the tuning of the hyperparameter in k-NN using *hold-out test* cross-validation would work as follows. First, we draw two disjoint samples from the mails. Next, we train nearest-neighbor classifiers with different values for k on the first sample. Then, we evaluate the accuracy of the resulting classifiers for predicting the labels of unseen mails in the second sample. Finally, we select the value for k with the best performance. Other ML classifiers have similar hyperparameters which can be tuned using cross-validation in a comparable way. A complementary meta learning technique is ensemble learning. Ensemble learning builds a stronger model by combining a set of weaker models [4], [5]. In our case, we draw many independent subsamples of the email corpus, train a classifier on each of them and combine the predictions of these classifiers afterwards. Datasets for such ML use cases have a matrix-representation, where rows correspond to training instances and columns to instance features. For the spam detection example, the matrix has a row for each email, and a column for each feature. Although our running example is small for explanatory purposes, we note a real-world dataset would have millions of rows (emails) and thousands of columns (features).



(a) dataset as matrix M (b) partitioning of M into blocks (c) storage of blocks in a distributed filesystem

Fig. 1: Sample dataset representation: matrix partitioned into blocks, stored on different machines in distributed filesystem.

B. Block-Partitioned Matrices

Although meta learning techniques are straightforward to implement on a single machine with random access to the data, efficient implementations are difficult when the input data is block-partitioned and stored on different machines in a distributed filesystem in a shared-nothing architecture. However, block-partitioning provides significant performance benefits, including a reduction in the number of tuples required to represent and process a matrix, block-level compression, and the optimization of operations like matrix multiplication on a block-level basis. Such benefits have led to the widespread adoption of block-partitioned matrices in several parallel data platforms [6], [9], [10].

Figure 1 illustrates our example dataset as a block-partitioned matrix in a distributed filesystem. As shown in Figure 1(a), the data is represented as a 6×4 matrix M where each row represents an email and each column a feature. The matrix is divided into square blocks according to a chosen block size. In our example, we choose a block size of 2×2, resulting in six different blocks, as shown in Figure 1(b). We refer to individual blocks using horizontal and vertical block indices, e.g., $M_{2,1}$ denotes the block in the second row and first column of blocks. Blocks are stored on different machines across the distributed file system, where the assignment of blocks to machines is usually random, as in Figure 1(c). As we discuss in subsequent sections, this distributed block-partitioned data representation creates multiple challenges for efficient sample generation for scalable meta learning.

III. Distributed Sample Generation Algorithm

In this section, we explain our algorithm for generating sample matrices from a large, block-partitioned matrix stored in a shared-nothing cluster. We discuss the case of generating sample matrices from rows of the input matrix. However, the same techniques apply to sampling columns of the input matrix as well. We define the sample generation task as follows. Given an $m \times n$ input matrix M, stored in blocks of size $d \times d$, we generate N sample matrices $S^{(1)}, \ldots, S^{(N)}$ such that the rows of a particular sample matrix $S^{(i)}$ are randomly drawn from M. We store the sample matrices in blocks of size $d \times d$.

A. Handling Partitioned Rows

Let r be a row of M partitioned in blocks with the horizontal block index b. Let v be the number of vertical blocks of M. Then r is distributed over the v blocks, $M_{b,1}, \ldots, M_{b,v}$. All machines processing one of these blocks must come to the same conclusion regarding the occurrences of r in a sample matrix $S^{(i)}$. Figure 2 illustrates the scenario for our mail example. We must determine in which sample matrices

preprint

`mail 3` will occur. Due to the blocked representation, the row corresponding to `mail 3` is partitioned and its partitions are contained in blocks $M_{2,1}$ and $M_{2,2}$ which are stored on different machines. The sample generation algorithm invokes sampling in parallel on both machines holding the blocks, and must give a deterministic and therefore reproducible result. For two invocations with the same row on different machines, it must return exactly the same destination in the sample matrices. Otherwise, we could not guarantee that the row is correctly restored in the sample matrix as row partitions might be missing or placed in the wrong order. Furthermore, this consistency must be guaranteed without requiring inter-machine communication in order to achieve a high degree of parallelism.

We solve this challenge by carefully choosing the pseudo random number generator (PRNG) which our algorithm deploys. A PRNG generates a sequence of values $p_0, p_1, p_2, \dots$ that are approximately uniformly distributed. The generation is initialized with a seed s and typically uses a recursive transition function, φ, such that $p_i = \varphi(p_{i-1})$. A first step towards guaranteeing consistent results in our algorithm is to use the same seed s on all machines. This ensures all PRNGs produce the exact same random sequence. Second, we use the unique index of a row as the position to jump to in the random sequence. For a row with index j, sampling must leverage the j-th element, p_j, from the produced random sequence. This method will provide consistent results when partitions of row j are processed in parallel on different machines. However, this approach leaves open a performance question, as all random elements up to p_{j-1} would have to be generated in order to determine the value of p_j with the recursive function φ. To circumvent this performance issue, we use a special class of PRNGs, which enables us to directly skip to an arbitrary position in the random sequence (a so-called *skip-ahead*). Mathematically speaking, these generators allow us to compute the random element p_j directly with a function $\hat{\varphi}(s, j)$ via the seed s and the position j. PRNGs with this property are also used in parallel data generation frameworks [15], [16].

B. Sampling-UDF

We embed into the algorithm a user-defined sampling function, refered to as *sampling-UDF*, to specify the sampling technique to apply. This function allows adaption to a wide variety of sampling techniques with only a few lines of code. We assume during integration of our algorithm into a large-scale ML system that necessary sampling-UDFs are implemented by experts and available to end users.

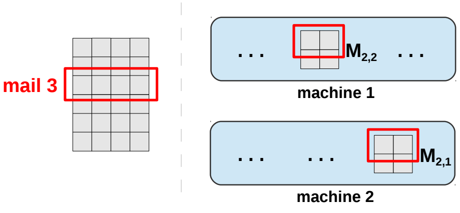


Fig. 2: Parallel invocation of sampling on different machines that process partitions of the same row.

Our algorithm calls the sampling-UDF to determine occurrences of a row in the sample matrices. The sampling-UDF has the following signature:

$$row\_id, rng, params \rightarrow (samplematrix\_id, relative\_pos)^*$$

The first input parameter is the identifier of a particular row to process: row_id. The second input rng refers to a PRNG instantiated by the algorithm, which provides the 'skip-ahead' functionality discussed in Section III-A. The third input $params$ denotes a set of user-defined parameters (e.g., the number of samples to draw). The implementation of the sampling-UDF must output a potentially empty set of $(samplematrix\_id, relative\_pos)$ tuples. Each tuple dictates an occurrence of the row in a sample matrix denoted by $samplematrix\_id$. This allows to generate occurrences in different sample matrices with a single invocation. The value $relative\_pos$ is used by the algorithm to determine the order of the rows in the sample matrices by sorting them according to $relative\_pos$. For instance, we can randomize a sample matrix by setting $relative\_pos$ to a random number drawn from rng.

C. Phased Execution

Our algorithm conducts the distributed sample generation in three phases: (i) the *'local-sampling'*-phase decides which rows of the input matrix M will occur in which sample matrices, (ii) the *'block-preparation'*-phase groups and sorts the selected rows, and (iii) the *'streaming-re-block'*-phase finally materializes the blocks of the sample matrices. Figure 3 shows an example of the distributed execution of the three phases. It details the individual steps in the algorithm for generating two sample matrices, $S^{(1)}$ and $S^{(2)}$. Both sample matrices contain three mails out of our toy dataset of six emails M. We assume execution in a distributed data processing system that runs MapReduce-like programs, such as Hadoop [17], Flink [18], Spark [19], and Hyracks [20], or in a distributed database that supports UDTFs and UDAFs. Furthermore, we assume that the overall size of the dataset (i.e., the dimensions of the input matrix) is known. The three phases execute in a single pass over the data. A traditional MapReduce system runs the *'local-sampling'* within the map-stage, the *'block-preparation'* within the shuffle and the *'streaming-re-block'* during the reduce-stage.

1) 'Local-Sampling'-Phase: This phase decides which rows of the input matrix occur in which sample matrices. Every machine in the cluster processes locally stored blocks of the input matrix in this phase and uses the sampling-UDF to sample partitions of rows from the blocks. Algorithm 1 illustrates the individual steps of this phase. Each machine processes a block of the input matrix at a time and iterates over the row partitions contained in the block (cf. line 4). For each such row partition, it invokes the sampling-UDF (cf. line 6) to determine the occurrences of this row in the sample matrices. For every $(samplematrix\_id, relative\_pos)$ tuple returned by the sampling-UDF, the processing machine emits a record with the following structure: $samplematrix\_id$ | $vertical\_block\_index$ | $relative\_pos$ | row_id | $row\_partition$ as shown in line 7 & 8. The field $samplematrix\_id$ denotes the sample matrix in which

preprint

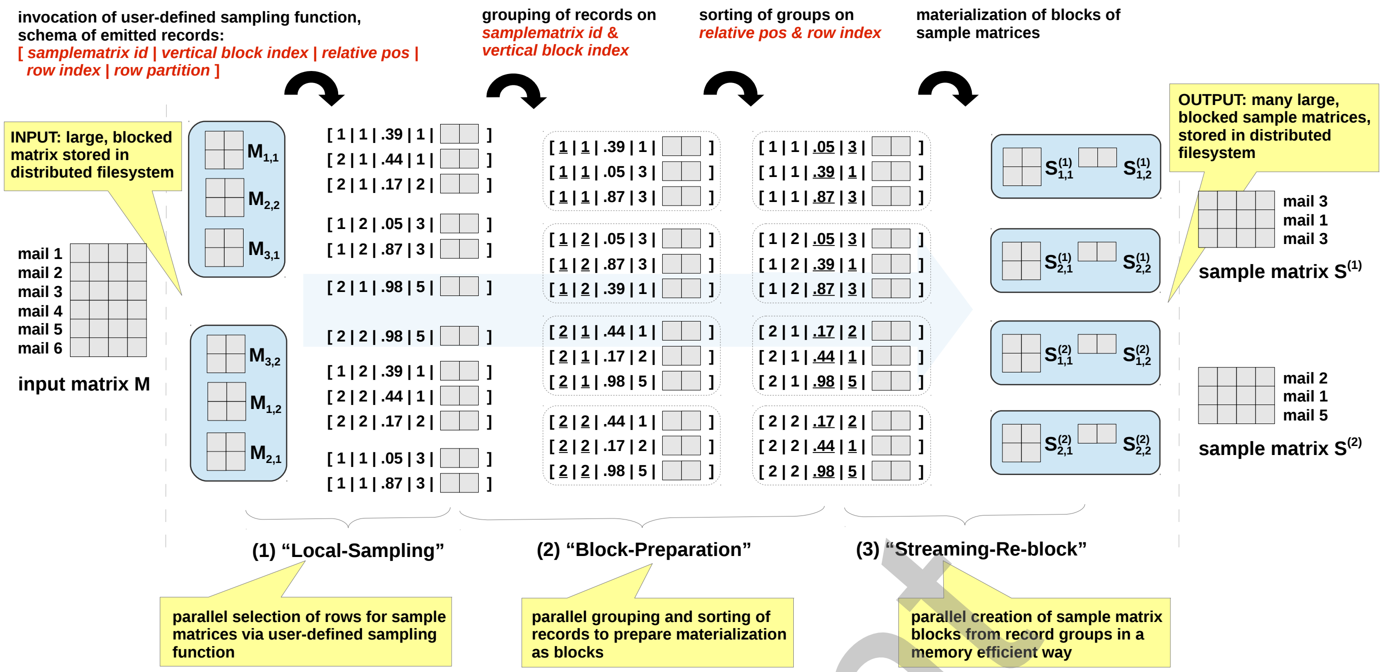


Fig. 3: Example: Distributed generation of two blocked sample matrices, $S^{(1)}$ and $S^{(2)}$ from a blocked input matrix M.

Preprint

this row will occur, the field *vertical_block_index* is the vertical index of the block of the input matrix from which we sampled the row partition. The field *row_id* has the row index (e.g., the i-th row has index i) and the field *relative_pos* holds the *relative_pos* value from the tuple returned by the sampling-UDF. These two fields will later be used to determine the position of the row in the sample. The field *row_partition* holds the data from the row partition that we sampled. The 'local-sampling' phase is embarassingly parallel as every block of the input matrix is processed independently. Its maximum degree of parallelism is equal to the number of blocks of the input matrix. In Figure 3, we see that the 'local-sampling-phase' results in six records being emitted by the first machine that stores and processes the blocks, $M_{1,1}$, $M_{2,2}$, $M_{3,1}$, and another six records emitted by the second machine that processes the blocks, $M_{3,2}$, $M_{1,2}$, $M_{2,1}$.

2) 'Block-Preparation'-Phase: The purpose of the next phase is to prepare the materialization of the records emitted from the 'local-sampling'-phase as blocked matrices. For that, we must group and sort the records. We use the data from the fields, *samplematrix_id* and *vertical_block_index* of the record as a composite key for grouping. After grouping, every resulting group contains the data for the column of blocks denoted by *vertical_block_index* of the sample matrix denoted by *samplematrix_id*. As different column blocks will be processed by different machines, we need to ensure that all of the groups contain their records in the same order. Failing to guarantee this would result in inconsistent sample matrices. Therefore, we sort the groups on the field *relative_pos* to ensure collection of partial rows in the same order on all machines. We additionally sort on the *row_id* to obtain a unique ordering even when *relative_pos* is the same random number for multiple rows. We omit pseudocode for this phase, since distributed data processing systems provide sorting and grouping functionality.

Figure 3 illustrates the two steps in this phase. First, we form four groups of three records from the twelve records emitted in the previous phase. Every group contains the row partitions for a column of blocks of a sample matrix. The topmost group, for example, contains the data for the first column of blocks of sample matrix, $S^{(1)}$, which consists of the blocks, $S^{(1)}_{1,1}$ and $S^{(1)}_{1,2}$. Analogously, the bottommost group contains the data for the second column of blocks of sample matrix $S^{(2)}$, namely, the blocks, $S^{(2)}_{2,1}$ and $S^{(2)}_{2,2}$. In a second step, we sort these groups on the *relative_pos* and *row_id* fields from the record, so that the records in each group have the same order in which the corresponding rows will later appear in their intended sample matrix.

3) 'Streaming-Re-block'-Phase: The final phase materializes the distributed blocked sample matrices. Every participating machine processes a sorted group from the 'block-preparation'-phase and forms the blocks for a column of blocks of the corresponding sample matrix. Algorithm 2 shows the required steps. We allocate a single block as buffer (c.f., line 3) which is filled with data from the field *row_partition* of incoming records in order (c.f., line 5). Whenever the buffer

Algorithm 1: 'Local-Sampling': Sampling from locally stored matrix blocks.

```
Input: matrix block $blk$, random number generator $rng$,
  sampling parameters $params$
Output: records containing the sampled rows
$v \leftarrow$ read vertical index of $blk$
for $row\_partition$ in $blk$
  $j \leftarrow$ index of row
  for $occ \leftarrow$ sampling_UDF( $j$, $rng$, $params$ )
    emit_record( $occ$.samplematrix_id, $v$, $occ$.relative_pos,
                 $j$, $row\_partition$ )
```

Algorithm 2: 'Streaming-Re-block': Materialization of sample matrix blocks.

```
1  Input: sorted record group $rec\_group$, sample matrix
   identifier $s$, vertical block index $v$
2  Output: sample matrix blocks written to DFS
3  allocate buffer block $blk$ with indexes $1, v$
4  for $rec \leftarrow rec\_group$
5    add row partition from $rec$ to $blk$
6    if $blk$ is full then
7      write $blk$ to DFS location for $s$
8      clear contents of $blk$
9      increment horizontal index of $blk$
10 if $blk$ is not empty then
11   write $blk$ to DFS location for $s$
```

block is completely filled, we write it back to the DFS and clear it (cf. lines 7 & 8). We increment the horizontal index of the buffer block afterwards. Finally, we write back potentially remaining buffered data, to handle the case when the final buffer block is not filled (c.f., lines 10 & 11).

Reblocking in this manner achieves a high degree of parallelism by allowing every column of blocks of every sample matrix to be processed independently. The maximum degree of parallelism of this phase is the number of sample matrices N to be generated multiplied by the number of column blocks (which equals the number of columns n of the input matrix divided by the block size d). We also achieve high memory efficiency, as only a single buffer block is necessary to materialize a column block of a sample matrix. Figure 3 illustrates an example for this phase. We transform the four record groups from the previous phase into blocked matrices. Every group results in a column of blocks of a sample matrix. We transform the topmost group into the blocks, $S^{(1)}_{1,1}$ and $S^{(1)}_{1,2}$ of sample matrix $S^{(1)}$ and the next group into the blocks, $S^{(1)}_{2,1}$ and $S^{(1)}_{2,2}$. The remaining two groups analogously form the blocks, $S^{(2)}_{1,1}$, $S^{(2)}_{1,2}$, $S^{(2)}_{2,1}$, and $S^{(2)}_{2,2}$ of sample matrix $S^{(2)}$. We store the blocks in the distributed filesystem.

This section focuses on generating sample matrices by sampling rows of an input matrix, which is common for most supervised learning use cases where each row corresponds to a labeled training example. However, other ML use cases require sampling at matrix cell level. One example is recommendation mining, which typically analyzes a matrix of interactions between users and items to predict the strength of unseen interactions [21]. Here, cross-validation holds out individual interactions, which involves sampling matrix cells. Our algorithm can extend to such cases, by processing blocks of the input matrix cell-wise instead of row-wise, applying the sampling-UDF at the cell level, and slightly modifying the re-blocking to aggregate cells.

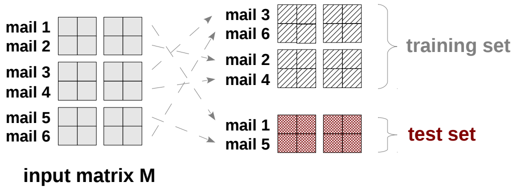


Fig. 4: Sample generation for a hold-out test: Creating a training set from two thirds of the input rows and a test set from the remaining rows.

IV. Sample generation for different meta learning techniques with the sampling-UDF

In this section we show how to implement the sample generation for common meta learning techniques based on the previously described sample generation algorithm with the sampling-UDF. Users aiming to implement generation techniques beyond the ones described here can leverage the presented UDFs as a template.

A. Hold-out Tests

Section II-A described a cross-validation technique known as the hold-out test. The idea behind hold-out tests is to randomly generate two sample matrices according to a user-defined train/test ratio using sampling without replacement. We use the data from the first sample matrix (the training set) to train a model and evaluate its prediction quality using the second sample matrix (the test set). Figure 4 illustrates the sample generation for a hold-out test using our example dataset of six emails and a train/test ratio of two thirds. We randomly choose four emails to form the training set, and the remaining two comprise the test set. Algorithm 3 shows an implementation of the sampling-UDF that generates the sample matrices for a hold-out test. As described in Section III-C1, the algorithm invokes the sampling-UDF for every row partition of a processed block, and the result describes in which sample matrices the row occurs. In the case of a hold-out test, every row either occurs in the training set or in the test set. In order to make this decision for a row with index j, we first skip the random number generator rng to the position j in its random sequence (c.f., line 3) to ensure reproducible results. Next, we generate a random number between zero and one from the rng and compare it to a user-defined train/test ratio to decide whether the row occurs in the training set or the test set (c.f. lines 4 to 8).

In order to improve the estimate of the prediction quality, we might want to conduct more than a single hold-out test. It is easy to adapt the implementation to generate several pairs (say w) of training sets and test sets for a series of hold-out tests. Algorithm 4 shows the necessary code. We simply repeat the approach of generating a random number and using this number to determine the occurrence of a row in the training set or test set w times (c.f., lines 5 to 10). There is one difficulty here: we need w random numbers for each invocation and we have to ensure that every position in the random sequence

Algorithm 3: Sampling-UDF for a single hold-out test.

```
1  Input: row index $j$, random number generator $rng$, $params$:
   user-defined train/test ratio
2  Output: tuple denoting occurrence in training set or test set
3  skip ahead to position $j$ in $rng$
4  $r \leftarrow$ random number generated by $rng$
5  if $r \leq$ user-defined train/test ratio then
6    return occurrence in training set at position $r$
7  else
8    return occurrence in test set at position $r$
```

preprint

is only used for a single row to ensure that the results are uncorrelated. Therefore, for each row index $j \in \{1, \ldots, m\}$ (where m is the number of rows of the input matrix), we skip the rng to the position $(j-1) * w$ in its random sequence and use the subsequent w random numbers for the current invocation to determine the occurrences of row j (c.f. line 4). A further feature could be to only use a subset of the data for the training set and test set. This functionality can be easily added by scaling down the train/test ratios and ignoring the non-matching rows.

Algorithm 4: Sampling-UDF for a series of hold-out tests.

```
1  Input: row index $j$, random number generator $rng$, $params$:
   number of hold-out tests $w$, user-defined train/test ratios
2  Output: tuples $occs$: occurrences in training set or test set
3  $occs \leftarrow \emptyset$
4  skip ahead to position $(j-1) * w$ in $rng$
5  repeat $w$ times
6    $r \leftarrow$ random number generated by $rng$
7    if $r \leq$ user-defined $w$-th train/test ratio then
8      add occurrence in $w$-th training set at position $r$ to $occs$
9    else
10     add occurrence in $w$-th test set at position $r$ to $occs$
11 return $occs$
```

B. K-fold Cross-validation

The most popular cross-validation approach is k-fold cross-validation [1], [22]. Here, we randomly divide the data into k disjoint logical subsets (called folds) using sampling without replacement, and execute k experiments afterwards. In every experiment, we train a parameterized model with all but the k-th fold as training data and test its prediction quality on the held-out data from the k-th fold. In the end, the overall estimate of the prediction quality of the model is computed from the outcomes of the individual experiments.

Figure 5 illustrates 3-fold cross-validation on our mail classification example. On a logical level, we assign the six emails randomly to the three folds: `mail 2` and `mail 4` form the first fold,`mail 3` and `mail 6` the second, and finally the third fold is comprised of `mail 1` and `mail 5`. We have to create the training sets and tests from the logical folds for the three rounds of experiments. In the first round, the test set consists of the mails from the first fold and the training set is comprised of the rest of the data. In the second round, the second fold becomes the test set and the remaining data is used for training. In the third and final round, we train on folds one and two, and test on the third fold. Algorithm 5 shows a straightforward implementation of the sampling-UDF, which directly creates the k training sets and tests for the k rounds of k-fold cross-validation. Analogous to previous cases, we skip the random number generator rng to the position in the random sequence denoted by the row index j (c.f., line 3). Next, we generate a random number between zero and one and compute the target fold f for the row from that (c.f., lines 5 & 6). Then, we make the function emit k occurrences of the row: it occurs in the test set of the f-th round and in the $k-1$ training sets of all other rounds (c.f., lines 7 to 11).

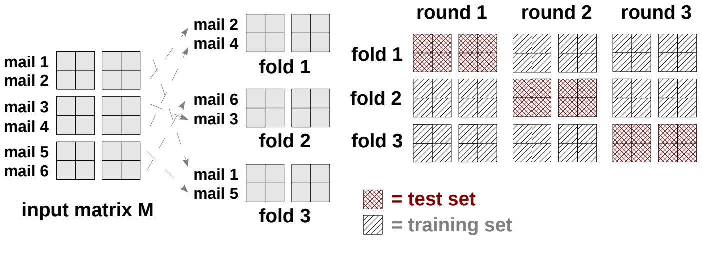


Fig. 5: Sample generation for 3-fold cross-validation

Algorithm 5: Naïve sampling-UDF for k-fold cross-validation.

```
1  Input: row index $j$, random number generator $rng$, $params$:
   number of folds $k$
2  Output: tuples $occs$: occurrences in training set or test set
3  $occs \leftarrow \emptyset$
4  skip ahead to position $j$ in $rng$
5  $r \leftarrow$ random number generated by $rng$
6  $f \leftarrow$ compute fold to assign to as $\lceil r * k \rceil$
7  for $w \leftarrow 1 \ldots k$
8    if $w = f$ then
9      add occurrence in $w$-th test set at position $r$ to $occs$
10   else
11     add occurrence in $w$-th training set at position $r$ to $occs$
12 return $occs$
```

This naïve approach is highly problematic as it creates k copies of the input data, which poses a serious performance and scalability bottleneck. Ideally, we would only want to create the folds once (resulting in only a single copy of the input data) and read $k-1$ of them as training set in every round. Unfortunately, treating a set of generated sample matrices as if they were a single big matrix does not work out-of-the-box. The block-partitioned representation causes difficulties when the dimensions of the sample matrix for a fold are not an exact multiple of the chosen block size. In such cases, there will be partially filled blocks at the bottom of the sample matrices, which prevent us from treating a collection of sample matrices as a single big matrix. Figure 6 shows an example for such a case. We divide a matrix with ten rows into three folds, using a block size of 2×2. The resulting sample matrix for the first fold consists of four rows (with indices 1,2,3,4), while the sample matrices for the second and third fold only consist of three rows (with indices 5,6,7 and 8,9,10). This inevitably leads to non-fully filled blocks at the bottom of the sample matrices for the second and third fold, as we are representing three rows with a block size of 2×2. In the first round of 3-fold cross-validation we need to treat the second and third fold as a single large matrix to be used as training set. Unfortunately, the rows of the matrix are not correctly aligned, as we represent six rows with four blocks of size 2×2. In a single big matrix, the rows with indices 7 and 8 should share a block, but are spread over two different blocks in our case, analogous to the rows with indices 9 and 10. Having operators of an ML system consume such an incorrectly blocked matrix would lead to inconsistent results.

In order to circumvent the above issue in our algorithm, we introduce a special mode called ‘dynamic sample matrix composition’ for k-fold cross-validation. With this mode activated, the algorithm copies the non-fully filled bottom blocks

preprint

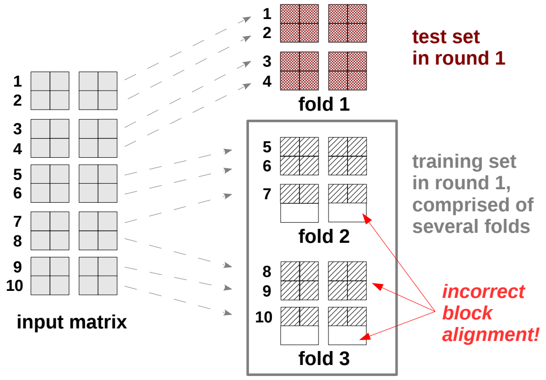


Fig. 6: Difficulties of treating a set of sample matrices as a single large matrix during a round of k-fold cross validation when matrix dimensions and block size do not align.

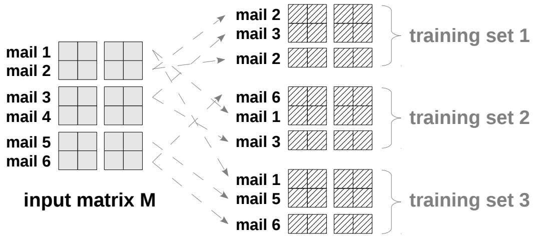


Fig. 7: Generating bootstrap sample matrices from an input matrix. Note that individual rows can occur more than once in the sample matrices.

of each generated sample matrix to a specific location in the distributed filesystem and runs an additional job that creates an additional matrix per training set from those blocks[1]. Finally the algorithm only reads the full blocks of the sample matrices (neglecting the non-fully filled blocks) plus the additionally created matrix as one large composed matrix. 'Dynamic sample matrix composition' requires only a single copy of the input matrix plus k additional small matrices consisting of at most $k-1$ blocks. With 'dynamic sample matrix composition' activated, we implement k-fold cross-validation with a very simple UDF that only triggers a single occurrence of each row, as shown in Algorithm 6. After computing the fold f for the row with index n analogous to Algorithm 5, we simply emit a single occurrence in the sample matrix of fold f (c.f., line 6). We refer to this as fold-based approach.

Algorithm 6: Fold-based sampling-UDF for k-fold cross-validation.

```
1 Input: row index $j$, random number generator $rng$, $params$:
  number of folds $k$
2 Output: tuple denoting occurrence in fold
3 skip ahead to position $j$ in $rng$
4 $r \leftarrow$ random number generated by $rng$
5 $f \leftarrow$ compute fold to assign to as $\lceil r * k \rceil$
6 return occurrence in $f$-th fold at position $r$
```

preprint

C. Sampling with Replacement (Bootstrap and Bagging)

Sampling with replacement is required by multiple meta learning techniques, including approaches for both cross-validation and ensemble learning. Bootstrap is a powerful statistical method for evaluating the quality of statistical estimators, and is applied to many cross-validation techniques [30]. One popular ensemble method leveraging sampling with replacement is bagging [5], which trains a set of models on bootstrap samples of the input data and combines their predictions afterwards. In our example, we would draw bootstrap samples of mails from our mail corpus (c.f., Figure 7), and train a classification model on each sample matrix in isolation. In order to decide whether to consider a newly arriving mail as spam, we would combine the predictions of the individual classifiers, i.e., by majority vote [22]. Efficiently drawing bootstrap samples in a parallel, distributed way is a hard problem [11]. In the following, we will explain why this is the case and present our solution to the problem. So far, we viewed the sample generation techniques discussed as distributed generation of a categorial random variate. Take k-fold cross validation on an input matrix with m rows as an example: for each row we have to decide to which of the k folds it must be assigned. This is equivalent to generating a random variate $r \in \{1,\dots,k\}^m$ from a categorial distribution of dimensionality m with possible outcomes $\{1,\dots,k\}$, each having probability $\frac{1}{k}$. The j-th component r_j of r holds an outcome that describes to which of the k folds the row with index j is assigned. In our algorithm, every invocation of the sampling-UDF generates such a component. The generation of a categorial random variate is easy to conduct in parallel in a distributed system, as all components of such a random variate are independent of each other, allowing us to generate them in isolation.

In contrast to the sample generation that we have seen so far, bootstrap samples are drawn using *sampling with replacement*, which means every row occurs at each position of every sample matrix with equal chance. A single row can occur zero or many times in a generated sample matrix. Given an input matrix with m rows and a desired bootstrap sample size s, we have to generate a random variate r from the multinomial distribution $M_m(s; \frac{1}{m}, \dots, \frac{1}{m})$ for the creation of a sample matrix with bootstrap sampling. Here, a component r_j of r denotes the number of times that the row with index j occurs in the sample matrix. Unfortunately, the components of a multinomial random variate are not independent, but negatively correlated: Say $(r_1, \dots, r_m) \sim M_m(s; \frac{1}{m}, \dots, \frac{1}{m})$, then if we fix the first component r_1 to have value z, the variate composed of the remaining components of r follows a multinomial distribution with different probabilities [23]: $(r_2, \dots, r_m) \sim M_{m-1}(s-z; \frac{1}{m-1}, \dots, \frac{1}{m-1})$. Because of this correlation, we cannot easily generate the components of r in parallel in isolation (existing approaches usually generate multinomial random variates sequentially or with random access to the variate as a whole [14]). Another constraint in our case is that we do not know the exact number of rows that a particular machine of the cluster is going to process, as distributed data processing systems typically schedule processing tasks adaptively. A simple solution to integrate sampling with replacement in our algorithm is to generate the whole random variate for every desired sample matrix once on every machine (our algorithm already exposes a random number

[1] Although this changes the order in the generated training set, it still provides a uniformly random order as result.

generator with a fixed seed on every machine), and then to access the components corresponding to the rows to process. This approach however has the constraint that there must be enough memory available for all these variates.

To circumvent this constraint, we present a method that enables us to compute a single compontent of a multinomial random variate in isolation with constant memory requirements. We use the property that a multinomial is partitionable by conditioning on the totals of subsets of its components. Let $r = (r_1, \dots, r_m)$ be a random variate from the multinomial distribution $M_m(s; p_1, \dots, p_n)$ and let $z = \sum_{j=1}^{i} r_j$ be the sum of the first i components of r. By fixing z we also fix the sum of the remaining components $\sum_{j=i+1}^{n} r_j$ as $s - z$. The left partition $(r_1, \dots, r_i)$ then follows the multinomial distribution $M_i(z; \frac{p_1}{p_l}, \dots, \frac{p_i}{p_l})$ with $p_l = \sum_{j=1}^{i} p_j$. Analogously, the right partition follows the multinomial distribution $M_{m-i}(s - z; \frac{p_{i+1}}{p_r}, \dots, \frac{p_n}{p_r})$ with $p_r = \sum_{j=i+1}^{n} p_j$. The generation of the cumulative sum z necessary for the partitioning can be conducted by sampling a binomial distribution: $z = \sum_{j=1}^{i} r_j \sim B(s, \sum_{j=1}^{i} p_j)$.

We use this property to define a recursive algorithm that generates the components of a multinomial random variate in parallel in a distributed setting (c.f., Algorithm 7). In order to determine the component r_n of a multinomial random variate r of size t with equally probable outcomes $(p_1, \dots, p_m)$, we proceed as follows. First, we determine the middle index of the current partition of the random variate that we look at (c.f., lines 4 to 6). Analogously to previous implementations, we skip the random number generator to this index to get repeatable results when processing this partition on different machines (c.f., line 7). Next, we sample from a binomial distribution to determine how many occurrences to assign to the left and right side of the current partition (c.f., line 8). Finally, we recursively invoke the function for the left or right side of the current partition, depending on the index j of the component we aim to compute. The algorithm terminates when we reach a single element partition (the component we look for) or when the current partition is assigned zero occurrences (c.f. lines 2 & 3).

This approach enables us to compute a multinomial random variate in a distributed, parallel fashion with constant memory requirements and without any intermachine communication. However, it has the drawback of requiring a logarithmic number of samples from binomials per component in contrast to sequential techniques, which only require a single such sample per component [14]. We can improve the runtime however with additional memory by caching the sampling results from early invocations for large partitions that have to be computed for many components. The sampling-UDF for generating bootstrap samples (c.f., Algorithm 8) directly invokes the recursive function *sample_multinomial* to determine the number of occurrences of a row with index j in the bootstrap sample (c.f., line 4). We skip the PRNG to the position $j * s$ (c.f., line 5), where $j \in \{1, \dots, m\}$ is a row index and s is the desired sample size. This leaves the first s positions in the random sequence for the computation of the number of occurrences (via *sample_multinomial*) and another s positions for generating random numbers for the *relative_pos* values of the occurrences of row j.

Algorithm 7: Recursive generation of a component of a multinomial random variate.

```
1  function sample_multinomial( $j$, $start$, $end$, $t$, $rng$ )
2    if $start = end$ or $t = 0$ then
3      return $t$
4    $size \leftarrow end - start + 1$
5    $size_{left} \leftarrow \lfloor size/2 \rfloor$
6    $mid \leftarrow start + size_{left} - 1$
7    skip ahead to position $mid$ in $rng$
8    $t_{left} \leftarrow$ binomial_rand( $rng$, $t$, $size_{left}/size$ )
9    if $j \leq mid$ then
10     return sample_multinomial( $j$, $start$, $mid$, $t_{left}$ )
11   else
12     return sample_multinomial( $j$, $mid + 1$, $end$, $t - t_{left}$ )
```

The special case of block-partitioning of the input data gives rise to even further performance improvements. Every machine processes the data one block at a time, where a block contains a large number of rows (e.g., 1000 by default in SystemML [6]). We leverage this for an approach that uses a single partitioning step only: We first create a random variate $\hat{r}$ from the multinomial distribution $M_b(s; p_b)$, where b equals the number of blocks to consider (the number of rows divided by the block size) and the probability vector p_b assigns probabilities proportional to the number of rows per block[2]. A component $\hat{r}_h$ of $\hat{r}$ holds the totals of the components of r corresponding to the rows included in the block with horizontal index h. We use a fast sequential algorithm [14] to generate $\hat{r}$ on every machine in the cluster. When we encounter the first row of a block, we generate a temporary multinomial random variate of dimensionality l and cardinality $\hat{r}_h$, where l is the number of rows contained in the current block and h is the horizontal index of that block. This random variate contains the number of occurrences for every row of the current block and only is kept in memory until the current block is processed.

Algorithm 8: Sampling-UDF for bootstrap sampling.

```
1  Input: row index $j$, random number generator $rng$, $params$:
   number of rows $m$, sample matrix size $s$
2  Output: tuples $occs$: occurrences in bootstrap sample
3  $occs \leftarrow \emptyset$
4  $t \leftarrow$ sample_multinomial( $j$, 0, $m$, $s$, $rng$ )
5  skip ahead to position $j * s$ in $rng$
6  for $1 \dots t$
7    $r \leftarrow$ random number generated by $rng$
8    add occurrence at position $r$ to $occs$
9  return $occs$
```

V. Experimental Evaluation

We conduct experiments with Apache Hadoop 1.0.4, Apache Spark 1.0 and Apache Hive 0.12 [25]. In every experiment, we spend effort to tune system parameters to improve performance. We run the evaluation on a 192-core cluster consisting of 24 machines. Each machine has two 4-core Opteron CPUs, 32 GB memory and four 1 TB disk drives. The datasets in use are synthetic. We generate random matrices with uniformly distributed non-zero cells for a specified dimensionality and sparsity. Skew is not an issue for our proposed approach, since the sampling is based on the row

[2] Non-fully filled bottom blocks have a lower probability of being selected.

preprint

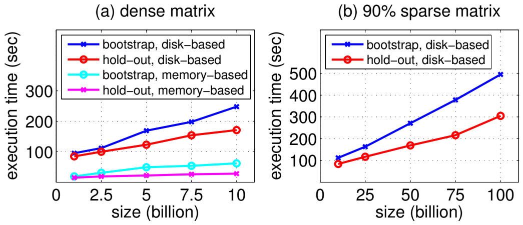


Fig. 8: Linear speedup for the generation of 20 sample matrices when increasing the input data size on a fixed cluster size.

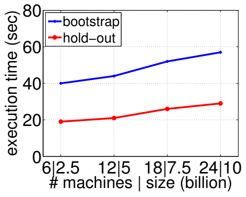


Fig. 9: Effect of increasing the cluster and input data size.

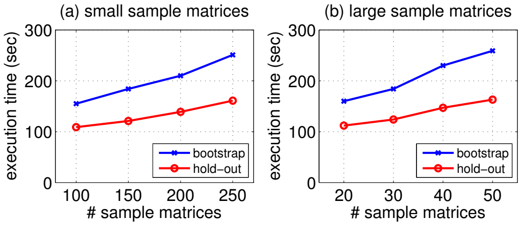


Fig. 10: Linear speedup when increasing the number of sample matrices for different sampling techniques.

index and not the data value. Analogous to [6], we employ a fixed block size of 1000 × 1000 and optimize the physical block representation for dense and sparse data.

The implementations of our algorithm in Hadoop and Spark are straight-forward. The *'local-sampling'*-phase is implemented within a map operator, the *'block-preparation'*-phase is automatically conducted in the shuffle-phase of Hadoop and implemented with a group operator in Spark. Finally, the *'streaming-re-block'*-phase is conducted by a reduce operator.

A. Scalability

The goal of the first set of experiments is to evaluate the scalability of our algorithm with regard to the size of the input data, the number of machines in the cluster and the number of samples drawn.

Scalability with increasing data size: First, we look at the effects when generating samples from increasing input sizes on a fixed number of machines (the whole 192-core cluster). We start with a dense $100,000 \times 10,000$ matrix with 1 billion entries. We repeat this experiment on larger matrices by increasing the number of rows of the input, so that we obtain matrices with 2.5 billion, 5 billion, 7.5 billion and finally 10 billion entries. We test two sampling techniques on every input matrix. First, we generate 10 training set/test set pairs for hold-out-testing (c.f., Section IV-A). Second, we draw 20 bootstrap samples of the data (c.f., Section IV-C). In order to make the results comparable, we configure the sample sizes so that the aggregate size of the sample matrices is equal to the size of the input matrix. The disk-based Hadoop implementation reads the input matrix from the DFS and writes back the sample matrices to the DFS. For our memory-based Spark implementation, we test the sample generation from in-memory matrices: here the input matrices are already cached in-memory and we only materialize the resulting sample matrices in-memory. Figure 8(a) shows the results of this series of experiments. We observe a linear speedup for both implementations and sampling techniques. Bootstrap sampling incurs a higher overhead to the runtime, due to the higher complexity of sampling from a multinomial (c.f., Section IV-C). As expected, the Spark implementation is up to ten times faster than Hadoop, because it reads from and writes to memory and applies hash-based grouping. We repeat this set of experiments for matrices with 90% sparsity. We start with a $500,000 \times 20,000$ input matrix comprised of 10 billion entries (and 1 billion non-zeros). Again, we increase the number of rows to obtain matrices of size 25 billion, 50 billion, 75 billion and 100 billion, and execute both sampling techniques on those. Analogous to the previous experiment, the Hadoop implementation shows a linear speedup for both, bootstrap and hold-out sampling (c.f., Figure 8 (b)). Unfortunately, Spark was not able to efficiently handle the large matrices used in this set of experiments (we terminated the execution when Spark's shuffle did not complete after 15 minutes).

Scalability with increasing data and cluster size: Next, we investigate the effects of increasing the number of machines proportionally to a growing amount of input data. Again we generate 20 bootstrap and hold-out samples. We start with a dense $250,000 \times 10,000$ matrix (having 2.5 billion non-zeros) as input on a cluster of six machines. In the following experiments, we double the number of machines as well as the number of rows of the input matrix, up to 24 machines and 10 billion non-zeros. We measure the sample generation from memory-resident input with our Spark implementation. Figure 9 depicts the resulting execution times. Ideally, we would see a constant runtime. However, it is not possible to reach this ideal scale-out for many reasons (e.g., network overheads). The same effect has been observed in other large-scale ML systems [6]. Nonetheless, our algorithm achieves a steady increase in execution time with growing data and cluster size.

Scalability with increasing number of sample matrices: Finally, we focus on the effects of increasing the number of sample matrices to generate. We make our Hadoop implementation generate samples from a $500,000 \times 10,000$ matrix with 5 billion non-zeros. First, we generate up to 250 small sample matrices, where every sample matrix contains one percent of the rows of the input matrix (c.f., Figure10(a)). Next, we increase the size of the sample matrices to five percent of the input data and generate up to 50 sample matrices per run (c.f., Figure10(b)). We test both cases with bootstrap and hold-out sampling. Our results show that the runtime increases linearly with the number of samples to generate and that our algorithm is able to generate a large number of samples efficiently with a single pass over the data. Again, the runtime is higher for bootstrap sampling due to the higher complexity of the underlying sampling process.

B. Benefits of 'Dynamic Sample Matrix Composition' for K-fold Cross-validation

Next, we evaluate the benefits of our 'dynamic sample matrix composition' technique for k-fold cross-validation (c.f., Section IV-B). We make our Hadoop implementation generate the training sets and test sets for 5-, 10- and 20-fold cross-validation on a dense $100,000 \times 10,000$ matrix with 1 billion entries as well as on a sparse $500,000 \times 20,000$ matrix with 1 billion non-zeros. For every case, we first

Draft

generate the training sets and test sets directly with the naïve approach. We compare this to the generation of the training sets and test sets via k-folds with 'dynamic sample matrix composition' activated in our algorithm. Figures 11(a) and 11(b) depict the results of this experiment for the dense and sparse matrices. The plots on the left show the execution times of both approaches. We see that the time of the naïve method grows linearly with increasing k, while the execution time of the fold-based approach stays constant. This is expected because the naïve approach has to create k copies of the data, while the fold-based method only has to create the folds (summing to a single copy of the input data) plus some extra blocks to allow for the dynamic composition of sample matrices. In the case of 5-fold cross-validation on the sparse matrix, the naïve approach is faster as it only runs a single pass over the input data, while the fold-based approach has to additionally process the partially filled blocks of the sample matrices after generation, for creating the extra blocks for dynamic composition. For $k > 5$, the fold-based approach outperforms the naïve approach in all cases by a factor between 1.6 and 3.3. The plots on the right side of Figure 11 show the ratio of output size (the sample matrices forming the training sets and test sets) to input size. As expected, the naïve approach produces an output that is k-times as large as the input size. On the contrary, the fold-based approach produces a much smaller output: it creates k folds, which, in combination, have the same size as the input. Additionally, the fold-based approach has to create extra blocks per training set from the non-fully filled blocks of the folds (c.f., Section IV-B). In the case of the dense matrix, the additional output size for the extra blocks amounts to a factor between 0.1 and 2.2 of the input size (which is still small compared to the sizes produced by the naïve approach). In the case of sparse matrices, the additional output size for the extra blocks is tiny (as we represent only the non-zeros of blocks) and ranges from a factor of 0.003 to 0.008. In summary, the fold-based approach provides a runtime constant in k (instead of linear in k for the naïve approach) and produces output sizes that are up to an order of magnitude smaller.

C. Comparison against Matrix-based Sample Generation

This experiment demonstrates that our proposed algorithm provides a huge performance benefit over implementing sampling with matrix operations. Many large-scale ML systems (e.g., [6], [8]–[10]) offer operators to conduct linear algebraic operations on large matrices, including ones required for sample generation, which is performed using linear algebra as follows. Assume we want to generate sample matrices with a total of p rows from an $m \times n$ input matrix M. For that, we have to create a binary, sparse 'selection' matrix S of size $p \times m$. This selection matrix contains only a single non-zero entry per row and is multiplied to the left with the input data matrix: $SM = A$. The resulting matrix A contains all p rows selected for the sample matrices. Setting the value (i, j) in the selection matrix S makes the j-th row of the input matrix M appear at the i-th position of A. We obtain individual sample matrices by slicing A.

We compare our algorithm to sample generation with existing distributed matrix operations. The latest version of Apache Mahout [8] provides a Scala-based domain specific language (DSL) for distributed linear algebraic operations on row-partitioned matrices. Programs written in this DSL are automatically parallelized and executed on Apache Spark. We implement matrix-based sample generation as follows in Mahout's DSL. First, we load the input matrix M from the DFS into memory on our cluster. Next, we programatically create the sparse selection matrix S in-memory. We multiply the distributed data matrix M on the left by the sparse selection matrix S to obtain A. Since A is a concatenation of all requested sample matrices, we finally read the individual sample matrices in parallel as slices of A.

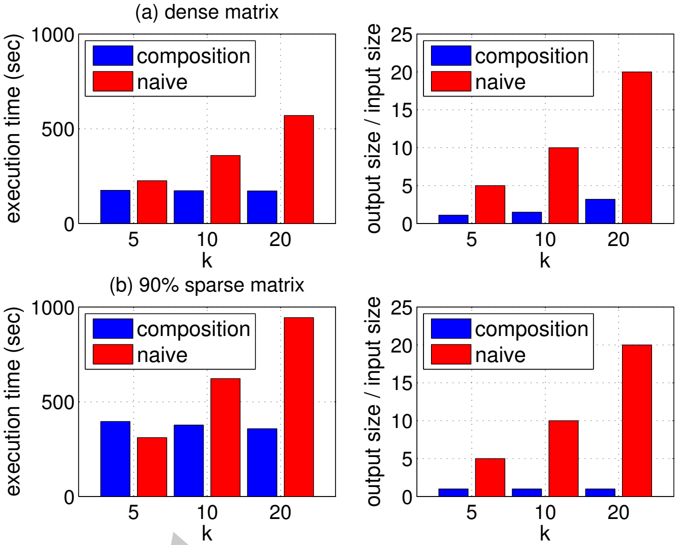


Fig. 11: Runtime and output sizes when generating training sets and test sets for k-fold cross-validation with and without 'dynamic sample matrix composition'.

```
// load distributed input data matrix
val M = loadMatrixFromHDFS(...)
// create in-memory selection matrix
val S = createSelectionMatrix(input, numSamples, ...)
// distributed multiplication of selection matrix
// and input matrix
val A = S %*% M
// read individual sample matrices as slices
// of A in parallel
(0 until numSamples).par.foreach { ... }
```

We compare the performance of the matrix-based sample generation with Mahout's DSL on Apache Spark against a Spark implementation of our algorithm. Since Mahout is optimized for handling sparse data, we use matrices with 90% sparsity in this evaluation. We generate 20 hold-out samples from several large matrices with up to $500,000,000$ non-zeros and measure the execution time. Both approaches read the input matrix from the DFS and materialize the individual sample matrices in-memory. The results shown in Figure 12 show our algorithm handles the sample generation in very short time (only 87 seconds for the largest matrix) with its runtime only moderately affected by the input size. On the contrary, the matrix operations in Mahout need 5 to 20 times longer for the sample generation. Furthermore, the matrix-based approach does not scale well, as a linear increase of the data size leads to super-linear increases in the runtime. These results indicate that large-scale ML systems benefit from a dedicated sample generation machinery.

preprint

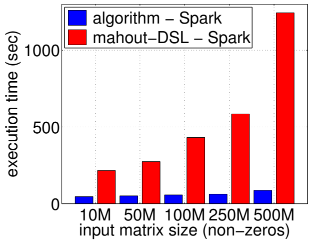


Fig. 12: Comparison of our Spark implementation to a matrix-based approach with Mahout's Spark DSL.

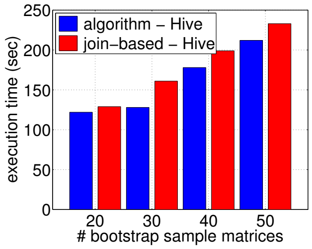


Fig. 13: Comparison of our Hive implementation to join-based sample generation in Apache Hive.

D. Comparison against Join-based Sample Generation

Our final experiment compares a SQL-based bootstrap sampling technique [24] to our approach. This technique assumes the input matrix is stored in a relational table and executes in two steps. First, a temporary table describing assignments of rows to sample matrices is created. Second, this temporary assignments table is joined with the input matrix table to form the sample matrices. This approach has several drawbacks for our ML use case. First, the non-blocked matrix representation, where every cell is individually represented, has huge performance penalties for many operations [6]. Secondly, while many ML toolkits and systems offer primitives for linear algebraic and statistical operations, many lack join primitives [6], [8]–[10]. Nevertheless, we perform this experiment to illustrate the benefits of our approach which eliminates both the need for the temporary assignment table and the join.

We adapt the join-based approach to use a blocked matrix representation for dense matrices. We implement the join-based approach in the distributed query processor Apache Hive using HiveQL[3]. The input is held in the table `input` with the schema `(h_index INT, v_index INT, block ARRAY<ARRAY<DOUBLE>>)`, where `h_index` and `v_index` refer to the horizontal and vertical index of a block and `block` represents its contents as a two-dimensional array. Again, we employ a block size of 1000×1000. The temporary table is called `assignments` with the schema `(sm_id INT, row_id INT, sm_pos INT)` where `sm_id` denotes the sample matrix, `row_id` the row to select and `sm_pos` the position for the row in the sample matrix to create. We implement the user-defined aggregation function `reblock`, which forms blocks from a collection of rows (c.f., 'streaming re-block' in Section III-C3). Unfortunately, we do not find an elegant way to create and fill the assignment table during query time in Hive, so we decide to create and fill it upfront and omit the time required for that in the evaluation. After creation of that table, we execute the sample generation in a single query: we extract row partitions from the blocks of the input matrix, join these row partitions with the `assignments` table, group the result and materialize the blocks with `reblock`:

```
-- Creation and filling of assigments table omitted
SELECT a.sm_id, floor(a.sm_pos / 1000),
  r.v_index, reblock(r.row, a.sm_pos) FROM (
  SELECT h_index * 1000 + offset AS row_id,
    v_index, row AS r FROM input
  LATERAL VIEW explode(block) AS offset, row)
JOIN assignments a
ON r.row_id = a.row_id
GROUP BY a.sm_id, r.v_index, floor(a.sm_pos / 1000);
```

Next, we implement our algorithm in HiveQL, with the sampling-UDF implemented as a UDTF. Analogous to the join-based approach, the sample generation requires only a single query: we extract row partitions from the blocks, apply the sampling function, group the result and form blocks with `reblock`.

```
SELECT sm_id, sm_h_index, v_index, sm_block FROM (
 SELECT sm_id, v_index, collect(row, rel_pos) AS col
 FROM (
  SELECT sm_id, v_index, rel_pos, row FROM input
   LATERAL VIEW explode(block) AS offset, row
   LATERAL VIEW sampling_UDF(h_index *
    1000 + offset, params) AS sm_id, rel_pos)
 GROUP BY sm_id, v_index)
LATERAL VIEW reblock(col) AS sm_h_index, sm_block
```

Figure 13 shows the execution times for generating 20, 30, 40, and 50 bootstrap sample matrices with $10,000$ rows each from a dense $500,000 \times 10,000$ matrix. We omit the time for creating and filling the assignments table in the join-based approach, as we had to conduct this manually before running the experiment. Both approaches only require a single MapReduce job. Hive selects a broadcast hash-join for executing the join-based approach. The results illustrate that our proposed approach outperforms the join-based approach in all experiments, although the latter one has competitive performance. However, our approach has the additional benefit of eliminating both materialization of a temporary assignments table and the required join of this table with the input data.

VI. Related Work

Non-distributed languages and libraries for machine learning offer extensive meta learning functionality. R [26] has support for cross-validation in the 'Design', 'DAAG' and 'Boot' packages, scikit-learn [27] offers the 'sklearn.cross_validation' package and furthermore includes models with built-in cross-validation. MATLAB provides a 'model building and assessment' module in its statistics toolbox.

In recent years, many systems and libraries have been proposed for scalable, distibuted ML; however, these lack support for a comprehensive meta learning layer. A comprehensive vision for meta learning on large ML datasets is presented as part of MLBase [13]; however, the proposed optimizer has not been implemented yet. SystemML [6] executes programs written in an R-like language called DML in parallel on MapReduce. SystemML requires manual expression of sample generation and model evaluation in terms of matrix operations in DML [12]. Cumulon [9] executes matrix-based data analysis programs in the cloud, but presents no meta learning functionality. Apache Mahout [8] offers implementations of ML algorithms in MapReduce and recently switched to building upon an R-like DSL that executes in parallel on Apache Spark. Mahout provides limited meta learning functionality for individual algorithms (e.g. hold-out testing for recommendation models), but lacks a principled meta learning layer. GraphLab [28] trains ML models using an asynchronous, graph-parallel abstraction, and presents a Python API supporting evaluation for recommenders and regression models. However, we are unaware of any description for mapping meta learning onto GraphLab's

preprint

[3] LATERAL VIEW is HiveQL's statement for applying table-generating functions. We omit aliases for lateral view statements to improve readability.

vertex centric data model. Apache Spark [19], which forms the runtime for MLBase, provides rudimentary sample generation functionality. Similar to our proposed sampling-UDF, a customizable random sampler is applied to large datasets. However, this sampling functionality cannot handle block-partitioned matrices with partitioned training instances and does not offer the generation of a large number of samples in a single pass over the data. Grover et al. [29] propose extending the MapReduce execution model to efficiently implement predicate-based sampling so that required cluster resources for sample computation are a function of sample size instead of input data.

Cohen et al. [24] present a SQL-based approach for distributed generation of bootstrap samples. In contrast to our work, this approach is not designed for block-partitioned data and requires the generation of a temporary assignment table as well as join primitives (c.f., Section V-D). Spark produces bootstrap samples from distributed datasets via a so-called 'takeSample' action. This action uses Poisson sampling, which will not guarantee a fixed sample size and therefore requires to draw a much larger sample than requested. Furthermore, the sampling must potentially be repeated if the Poisson sample is too small. Kleiner et al. [30] present an alternative to the classical bootstrap called 'bag of little bootstraps' which combines the results of bootstrapping multiple small subsets of a larger original dataset.

VII. Conclusions and Outlook

In this paper, we propose a general, single-pass algorithm for generating sample matrices from large, block-partitioned matrices stored in a distributed filesystem. Embedded in the algorithm is a user-defined sampling function allowing adaption to a variety of meta learning techniques with minimal code modification. We discuss implementing common meta learning techniques using the sampling UDF, particularly distributed sampling with replacement and generation of training sets for k-fold cross-validation. We implemented our algorithm in three different systems: Apache Hadoop, Apache Spark, and Apache Hive. Through extensive empirical evaluation, we found our algorithm scales linearly for both input data size and number of samples. Our algorithm outperforms matrix-based sample generation techniques by more than an order of magnitude and is faster than an approach using a distributed hash-join.

In future work, we plan to extend the scope of meta learning and sampling techniques implemented (e.g., adding support for stratified sampling). Tighter integration with existing large scale ML systems would further leverage their capabilities for optimizing meta learning comprehensively (e.g., have ML-system optimizers provide algebraic rewrites to exploit specific data partitioning such as the folds generated for k-fold cross-validation). Finally, we plan to explore application of our algorithm to use cases beyond meta learning.

References

[1] S. Geisser, "The predictive sample reuse method with applications," *Journal of the American Statistical Association*, vol. 70, no. 350, pp. 320–328, 1975.

[2] B. Efron and R. Tibshirani, "Improvements on cross-validation: the 632+ bootstrap method," *Journal of the American Statistical Association*, vol. 92, no. 438, pp. 548–560, 1997.

[3] P. Burman, "A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods," *Biometrika*, vol. 76, no. 3, pp. 503–514, 1989.

[4] J. Lin and A. Kolcz, "Large-scale machine learning at twitter," *Sigmod'12*, pp. 793–804.

[5] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[6] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative machine learning on MapReduce," *ICDE'11*, pp. 231–242.

[7] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "Mli: An api for distributed machine learning," *ICDM'13*.

[8] "Apache Mahout." [Online]. Available: http://mahout.apache.org

[9] B. Huang, S. Babu, and J. Yang, "Cumulon: optimizing statistical data analysis in the cloud," *Sigmod'13*, pp. 1–12.

[10] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," *ICDM'09*, pp. 229–238.

[11] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "Planet: massively parallel learning of tree ensembles with mapreduce," *PVLDB'09*, pp. 1426–1437.

[12] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, D. Burdick, and S. Vaithyanathan, "Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML," *PVLDB'14*.

[13] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. Franklin, and M. Jordan, "Mlbase: A distributed machine-learning system." *CIDR'13*.

[14] C. S. Davis, "The computer generation of multinomial random variates," *Computational statistics & data analysis*, vol. 16(2), pp. 205–217, 1993.

[15] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen, "Bigbench: Towards an industry standard benchmark for big data analytics," *Sigmod'13*, pp. 1197–1208.

[16] A. Alexandrov, K. Tzoumas, and V. Markl, "Myriad: scalable and expressive data generation," *PVLDB'12*, pp. 1890–1893.

[17] "Apache Hadoop." [Online]. Available: http://hadoop.apache.org

[18] A. Alexandrov, R. Bergmann, S. Ewen, C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, and others, "The Stratosphere platform for big data analytics," *VLDB Journal'14*

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *NSDI'12*.

[20] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," *ICDE'11*, pp. 1151–1162.

[21] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, pp. 30–37, 2009.

[22] C. M. Bishop *et al.*, *Pattern recognition and machine learning*, 2006.

[23] J. L. Schafer, *Analysis of incomplete multivariate data*. CRC, 1997.

[24] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton, "Mad skills: new analysis practices for big data," *PVLDB'09*, pp. 1481–1492.

[25] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major Technical Advancements in Apache Hive," *Sigmod'14*, pp.1235–1246

[26] RC Team, "R: A language and environment for statistical computing," *R foundation for Statistical Computing*, 2005.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *JMLR'11*, vol. 12., pp. 2825–2830.

[28] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed Graphlab: a framework for machine learning and data mining in the cloud," *PVLDB'12*, vol. 5, no. 8, pp. 716–727.

[29] R. Grover, and M. Carey, "Extending map-reduce for efficient predicate-based sampling," *ICDE'12*, pp. 486–497.

[30] A. Kleiner, A. Talwalkar, P. Sarkar, and M. I. Jordan, "A scalable bootstrap for massive data," *arXiv preprint arXiv:1112.5016*, 2011.

preprint