Declarative Meta Learning using MapReduce

Authors

ABSTRACT

With enormous amounts of data generated at very large scales, the problem of efficiently learning patterns and features in the data has become increasingly challenging. To this end, there have been significant efforts toward building systems for solving such large-scale machine learning problems, many of them using MapReduce. While existing systems support efficiently building machine learning models from data, they currently do not consider three key issues in large-scale machine learning: 1) estimating the *generalizability* of learned models to future unseen data sets, 2) *feature and model selection* (i.e., determining the best feature set and parameter values to use in model construction), and 3) using *collections of models* to improve classification accuracy for difficult learning problems. As with current approaches for addressing these "meta" questions in machine learning, we propose to solve the first two problems using cross-validation based algorithms, and ensemble methods to solve the third.

In this paper, we develop a *declarative system* for supporting the meta learning tasks of cross-validation (CV) and ensemble learning (EL) on large datasets using MapReduce clusters. We abstract the common operations from CV and EL, e.g., generating splits, and formulate an algebra of operators to support their execution. These operators can not only be parameterized to support arbitrary machine learning models, but also provide opportunities for plan-based optimization. We introduce a cost-based optimization framework for choosing the best plans to evaluate, based on the nature of the input and available resources. Our preliminary experiments demonstrate the necessity of a declarative system for meta learning, and the scalability of our proposed approaches.

1. INTRODUCTION

Large-scale machine learning is a significant research problem due to the enormous amount of data generated at increasing scale from a variety of data sources. Examples of such data include system, click, search, and ad-impression logs, messages on social networks, product reviews, financial trading data, and so on. Efficiently learning models from such data at large scale is critical for several applications such as spam detection, personalized search and advertising, recommender systems, customer retention, churn analysis, and fraud detection. Further interest in developing supervised and unsupervised machine learning techniques on large amounts of data arises from recent observations that accuracy of learned models can improve significantly by simply using larger amounts of training data rather than increasingly complex models [14]. However, simply being able to build models on large amounts of data is not sufficient for real-world applications. Assessing the quality of the model, determining the subset of features and model parameters, or even building an ensemble of multiple models may be required to improve accuracy.

1. *Quality.* Overfitting [21] is a commonly observed problem in machine learning that occurs when the built model also learns noise. When a model is overfit to the training data, its generalization is poor and it does not work well with unseen data. Hence, it is very important to validate the quality of learned models.

2. *Feature Selection.* Identifying the subset of features for building models is critical in handling high dimensional datasets [22] as it significantly impacts model quality [13].

3. *Model Selection.* Model selection involves estimating the best values for model parameters [33]. For example, consider a document clustering application using k-means clustering. If the selected number of clusters k is very small then multiple documents that are not similar would end up in the same cluster, while a very large k places similar documents in different clusters.

4. *Ensemble methods.* Ensemble methods train a collection of models and combine their outputs to provide a single answer to the learning task. The motivation for ensemble methods is that the combined answer from the ensemble will, on average, have higher accuracy than any individual model in the ensemble. Ensemble methods have been used successfully in many applications, including the 2009 KDD Cup [26] and the Netflix prize [2].

Cross-validation (CV) and ensemble learning (EL) are well-established *meta-learning* concepts to address "meta"

Proceedings of the VLDB Endowment, Vol. X, No. Y

questions. The first three problems listed above are typically solved using CV, while the last is addressed by EL.

CV involves creating multiple copies of the dataset called *folds*, each of which has a training set and a test set. For each fold, the model is learned using the training set and evaluated against the test set by measuring the error, e.g., the number of misclassifications or error in regression. The average error across all the folds measures the *generalizability* of the learned model, i.e., how it performs on unseen data sets. A large CV error indicates overfitting and the need for simpler models. For solving the model selection problem, the CV process is repeated for different choices of the model parameters, and the parameters with the smallest error are chosen. A similar process is used for feature selection.

Ensemble learning involves learning a diverse collection of models to improve generalizability, and combining the output of each individual model in the collection into a single answer. The answer from the ensemble should have greater accuracy than any individual model provided the individual predictions are combined appropriately so that correct predictions have stronger influence than incorrect ones. Examples of ensemble methods include learning individual models with different subsets of input data records (bagging [5]), different features (random subspace [15]), or learning each model using different parameters. Example techniques for combining answers from individual models include weighted voting and averaging.

Existing systems provide little or no support for general CV or EL. R and Matlab provide specialized support for CV and EL, but it is tightly integrated with specific algorithms such as linear regression (e.g., glm package in R) and classification (e.g., random forests package in R for decision trees). Consequently, users have to develop their own CV and EL methods for other machine learning algorithms, leading to limited reuse and constant re-invention of the wheel. Furthermore, as shown in recent publications [11], the above systems are not scalable to very large datasets. There are existing tools for machine learning on very large datasets, such as Revolution R [32], Parallel-Matlab [29], Apache Mahout [1], and GraphLab [7, 25]. However, they do not support CV or EL at all, or, such as Mahout, provide support only to a few algorithms (e.g., decision trees, stochastic gradient descent).

Supporting generic CV and EL at scale is challenging for the following reasons:

- CV and EL are data and computationally intensive since multiple copies of a potentially very large dataset must be generated to learn multiple models.
- CV and EL require sophisticated algorithms for generating folds and training sets for models. For instance, both bootstrap CV and bagging in EL require *sampling with replacement*, which has been shown to be non-trivial to parallelize in MapReduce architectures [28].
- CV fold construction for unsupervised learning algorithms such as PCA, SVD and NMF [24] require novel matrix partitioning techniques (holding out cells & sub-matrices) that are challenging to implement for very large matrices with millions of rows and columns.
- A generic abstraction that covers all widely used methods of CV and EL needs to be developed. The abstraction should not compromise efficiency.

Our Contributions

In this paper, we augment SystemML [11] to support scalable meta-learning. SystemML is a declarative system for large scale machine learning. It provides a *declarative* language, called DML, which allows users to write complex machine learning algorithms. We augment DML with a CV and EL language construct. Each construct abstracts the core tasks such as generating splits, and specifies different CV and EL methods. The main research contributions of our work are:

1. We develop a generic abstraction to apply CV and EL to a large class of machine learning models – both supervised and unsupervised.
2. We design techniques for efficiently executing complex sampling algorithms over MapReduce using database-style joins in MapReduce.
3. We develop three methods for generating data splits, and a cost-model for selecting the most efficient method for a given dataset.

The rest of the paper is organized as follows. Section 2 introduces SystemML provides background, while Section 3 introduces the meta-learning abstractions and CV and EL operators. Section 4 details the system design and implementation, with Section 7 providing the results of an experimental evaluation. Section 8 presents related work, and we conclude in Section 9.

2. PRELIMINARIES

In this section, we first provide a brief overview of SystemML and necessary technical details in relationship to meta-learning. Following, we provide a short description of commonly used CV methods for both supervised and unsupervised learning tasks. Finally, we describe commonly used EL methods for supervised algorithms.

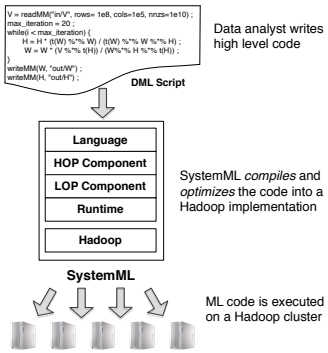
2.1 SystemML

SystemML exposes a high level language called DML [11]. The system translates the DML code into a workflow of MapReduce jobs, that are executed on a Hadoop cluster. As an example, the implementation of *non-negative matrix factorization*, a commonly used method for topic modeling and recommender systems, is shown in Figure 1(a). As shown in the figure, constructs in DML include linear algebra operations such as matrix multiplications, looping constructs (while), and conditioning (if).

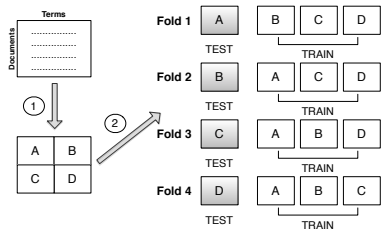
SystemML is designed as a five layer system as shown in Figure 1(b). The DML code is transformed and optimized at each layer, finally into a MapReduce dag which is scheduled/orchestrated by the Control system. The *parser* performs static analysis over the code and computes a *dag* of high level operations (HOP dag). The *HOP* layer optimizes the dag (e.g., computes a good order for matrix multiplications) and constructs a new dag over low-level operations. While high level operations (HOPs) represent operations such as matrix multiplications over the data matrices, low-level operations (LOPs) correspond to operations over key-value pairs (which constitute the data matrices). Finally, the *LOP* layer topologically sorts the lops dag and groups together the low-level operations which can be executed in a single MapReduce job. The *control system* then

```
V = read("V", rows=1000000, cols=1000)
W = Rand(rows=1000000, cols=10, min=0, max=1);
H = Rand(rows=10, cols=1000, min=0, max=1);
max_iterations = 20;
i = 0;
while (i < max_iterations){
H = H * (t(W) %*% V) / (t(W) %*% W %*% H);
W = W * (V %*% t(H)) / (W %*% H %*% t(H));
i = i+1;
}
write(W, "w.output");
write(H, "w.output");
```

(a) DML code for NMF (non-negative matrix factorization)



(b) SystemML architecture



(c) 2×2 Gabriel holdouts

Figure 1: Figure 1: (a) High-level DML code for NMF is shown here. (b) The five layer design for SystemML (c) Illustration of 2×2 Gabriel holdouts. In the first step, the rows and columns are split into two groups. Subsequently, 4 folds are constructed in which a given row and column group is held out and the remaining data is held in.

schedules the jobs over the Hadoop cluster. We refer the reader to [11] for a more comprehensive discussion of SystemML.

2.2 Crossvalidation (CV)

Cross-validation is a technique for estimating the accuracy of a machine learned model over a future (unseen) dataset. Widely used in supervised classification problems, CV techniques compute an error for a given machine learned model, which is indicative of its generalization performance. Cross-validation error is computed by splitting the given data set into a training (held in) and testing (held out) components. Such a split is called a *fold*. A model is learned over the training component and the error is evaluated over the test component. Typically, the error is averaged over multiple such folds to reduce bias. CV has been used for a number of problems such as preventing overfitting, model selection, and feature selection. We now illustrate the most common methods used for cross-validating both supervised and unsupervised models.

Supervised learning:
The input dataset to a supervised learning problem is a set of data points with known class labels. We visualize the data as a matrix with rows corresponding to the data points and the columns corresponding to the features. We assume that one of the columns corresponds to the class label. There are a number of different ways of constructing folds, three of which are illustrated next.

- *k-fold:* In k-fold cross-validation, the input matrix is initially split (row-wise) into k groups. Following this, k folds are constructed: the i^{th} fold is constructed by using the i^{th} group as the test component and the remaining $k-1$ groups as the train component. For each fold, a model is learned over the training data and its error is evaluated over the test data set. Finally, errors are averaged across the folds.
- *μ-holdout:* In holdout cross-validation, a fraction (μ) of the input rows is held out and forms the test component. The remaining set of rows form the train component and are used to train the model. Typically, the above operation is repeated for a number of user-specified iterations.
- *Bootstrap:* For Bootstrap cross-validation, we create the training component by sampling the original dataset with replacement. The entire dataset acts as the test component. As before, we construct the user specified number of folds.

K-fold cross-validation ensures that every data point is part of exactly one test component. Hence, we use every data point as a test component in some fold. Holdout cross-validation allows us to construct small holdouts without replicating the dataset too many times. For instance, if we want to holdout out 10% of the data for testing, then using k-fold cross-validation requires us to use k=10, which involves replicating the data set 10 times. Using holdout cross-validation allows us to use 10% of the data for testing using arbitrary number of iterations. *Leave-one-out* cross-validation is essentially k-fold cross-validation where k is equal to the number of data points; however, this method is expensive and unsuitable for large-scale data. Another commonly used parameter in CV for supervised models is *stratified sampling*. Consider a spam detection task where we need to classify a given email as spam or not. Suppose that in our training dataset, we have 80% genuine emails and 20% spam emails. In that case, stratification allows each fold to have the same percentage of spam and genuine emails in both the train and test components. We describe how to support stratification in Section 6.

Unsupervised learning:
It is not immediately obvious how to cross-validate unsupervised learning models since the training data is not annotated with class labels. Indeed, there has been very little prior work [18, 37] on cross-validating unsupervised models, much of it focusing on PCA (principal component analysis). With increasing usage of unsupervised techniques such as NMF (for topic modeling and recommender systems) and SVD (for dimensionality reductions) there have been proposals [19, 30] for cross-validating unsupervised models. Recently, [19] demonstrated the connection between matrix completion [6] and cross-validating unsupervised models, resulting in many novel methods for cross-validation. These techniques make use of the observation that unsupervised learning algorithms essentially learn the *distribution* of the input data for developing cross-validation algorithms

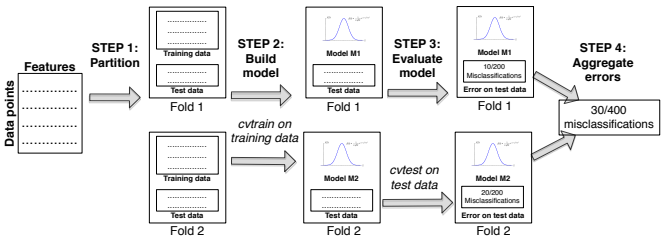


Figure 2: The four steps in cross-validation: generate splits, train, test, and aggregate errors.

for them. For example, a clustering algorithm essentially learns the underlying distributions that generates the given dataset. A topic modeling algorithm over documents builds a generative model from which the document collection can be generated. Hence, most of the proposed techniques hold-out a portion of the data, build an unsupervised model over the held-in data and either reconstruct the held-out data points using the learned model or measure the likelihood that the held-out data points actually come from the learned model. By measuring such errors (reconstruction or perplexity [3]), CV is used to estimate the parameters of the model, i.e., number of useful principal components in SVD or the number of topics in a topic model. Next, we describe two commonly used fold construction methodologies for unsupervised models.

- *$h \times l$ Gabriel holdout:* In this case, the rows of the matrix are partitioned into h row-groups and the columns of the matrix are partitioned into l column-groups. Following this, hl folds are defined in which a given row-group and a column-group is held-out and the remaining components of the matrix are held-in. An illustration of Gabriel holdout for 2×2 is shown in Figure 1(c). As shown in the figure, (for Fold 1), a model is learned using matrices B, C and D. Similarly, for Fold 2, B is held out and the model is learned over matrices A, C and D. Owen & Perry [30] describe a methodology for reconstructing the held-out matrix (A in Fold 1) based on the entries in the held-in matrix (B, C and D in Fold 1).

- *Wold holdout:* In this case, a fraction of cells in the matrix are heldout, similar to μ-holdouts. In recent work, [19] develops techniques for reconstructing the values of these cells based on weighted NMF [34].

As illustrated above, supervised and unsupervised models use different strategies for fold construction. Supervised models typically holdout rows from the matrix since each row has a class label associated with it and unsupervised models holdout cells or submatrices.

2.3 Ensemble Learning (EL)

Ensemble Learning (EL) is a meta-learning technique which involves learning a collection of models, and combining the output from each model into a single answer. Ensembles are a powerful technique for addressing the issues of underfitting and overfitting which may arise for complex learning tasks. *Underfitting* arises when a simple model may not be contain an appropriate hypothesis for learning the complex decision space of the task. If more complex models are used, then *overfitting* may become an issue, with the learned model not generalizing well to unseen data (i.e., the more flexible complex model "learns" noise in the training data). Ensembles address these issues by allowing a "committee" of simpler models to be used to learn a complex decision space, with the ensemble having higher accuracy than any individual model in the collection. For the ensemble to have high accuracy on unseen data, two objectives must be meant. First, the set of models composing the ensemble must have *diversity*, so the ensemble will generalize. Second, the outputs from the individual models must be combined in such a way that correct answers are amplified in determining the final result while incorrect answers are dampened.

Techniques to ensure diversity in the learned ensemble fall into four broad categories. The first category involves learning individual models with different samples of input data records (e.g., bagging [5] samples with replacement, holdout samples without replacement) or data features (e.g., random subspace [15] samples features without replacement). The second category involves learning each model using different parameters, and the third using different learning methods for each ensemble model.

The fourth category are *boosting methods*, which combines a collection of weak classifiers to create a strong classifier. For boosting methods, a distribution of weights is assigned to the input data set. For each boosting iteration, a new classifier is learned using the weighted data set. The weight distribution is updated so incorrectly classified tuples have their weights increased, while correctly classified tuples have their weights decreased. The intuition is that emphasis in future iterations will be placed on training tuples incorrectly handled by the current ensemble. One of the earliest boosting methods is AdaBoost [10], which can be proven to increase in accuracy as more boosting iterations are performed (provided each base classifier in the ensemble has correctness $> \frac{1}{2}$). Since AdaBoost is iterative and each iteration requires complete access to the data, it is not readily parallelizable in a MapReduce setting. [27] presents an approach to parallelize AdaBoost in a MapReduce setting which partitions the input data, performs AdaBoost over each partition, and combines the ensembles from the partitions into a single ensemble. Although the provable correctness guarantee of the original AdaBoost does not apply to the technique in [10], they prove a weaker correctness bound and demonstrate their approach performs well empirically.

Predictions from the individual models need to be combined into a single prediction so that correct predictions have stronger influence than incorrect ones. The appropriate method depends on both the problem setting and what is being predicted. Real-valued predictions (e.g., regression value) can be combined using algebraic methods (e.g., sum, product, or mean), while class labels require vote-based methods. Probabilistic methods which return a probability distribution over possible answers, such as stacked generalization (stacking) or mixture of experts are possible as well [31].

3. META-LEARNING ABSTRACTIONS

In this section, we provide two abstractions that unify the various cross-validation and ensemble learning strategies discussed in Section 2 by language constructs for CV and EL. This supports generic cross-validation and ensemble learn-

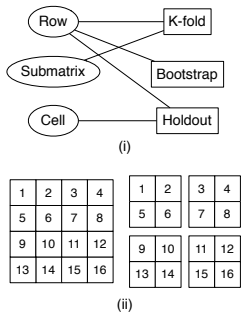
```
mydata = readMM("mydata", rows=1000, cols = 1000,
                nnzs = 100000, format="text");
numtopics = 1;
while(numtopics < 20){
  crossval mydata
    partition (type ='kfold', element='submatrix',
          numRowGroups=2, numColGroups=2)
          as (A,B,C,D)
    train NMFCVTrain(B,C,D) as (W,H)
    test test(A,W,H) as (error)
    aggregate avg(error) as (output);
  numtopics = numtopics + 1;
}
```

(a) CV construct for model-selection in NMF

```
function NMFCVTrain(X, Y, Z) returns
(W, H) {
  //Assume function (W,H) = NMF(X);
  (W_z, H_z) = NMF(Z);
  (W_z, H) = NMF(Y);
  (W, H_z) = NMF(X);
}
```

```
function test(W, H) returns (error) {
  H = H[0...(length(H)-1)];
  A = H[length(H)];
  diff = A - W * H;
  error = norm(diff * diff);
}
```

(b) Test & train routines called in part (a)



(c)

```
V = readMM("v.input", rows=1000, cols=1000, format="text") ;
y = readMM("y.input", rows=1000, cols=1, format="text") ;
crossval (V,y)
  partition (method='holdout', element='row', frac=0.30)
        as (Vtrain, Vtest);
  train linearRegressionTrain (Vtrain) as (w);
  test test (t(w),Vtest) as (error);
  aggregate avg(error) as output;
writeMM(w, "w.output", format="test");
```

(d) CV for computing generalization error in regression

```
function linearRegressionTrain (V) returns (w) {
  numIterations = 0 ; eps = 0.001 ;
  V = V[1...(length(V)-1)]; y = V[length(V)];
  p = V %*% t(V); q = V %*% y ;
  w = Rand(rows = 1000, cols = 1, min = 0, max = 0);
  while (numIterations < 20){
    deriv = p %*% w - q ;
    w = w + eps * deriv ;
    numIterations = numIterations + 1;
  }
}
```

(e) Linear regression training function, used in part (d).

Figure 3: (a) CV language construct in use for model-selection. Notice CV error is computed for different values of numtopics inside the while loop. (b) NMFCVTrain and test routines called in part (a) for cross-validating NMF. (c) (i) Currently supported CV options in SystemML (ii) Illustration of blocked structures in matrices. (d) CV for linear regression. (e) Linear regression training function used in (d). Note (b) and (d) use the same test routine.

ing for a large class of machine learning models.

3.1 Cross-validation

Cross-validation can be schematically described using four modules as illustrated in Figure 2, which illustrates CV using two folds. First, we *generate splits* of the input data for folds, where each fold has a training and a testing component. Note the training component may either be a single matrix or a set of matrices. For each fold, we *train* a model using the specified algorithm over the training component of the fold, then *test* the learned model using the testing component and return the error. Finally, we *aggregate* the errors computed in each fold, e.g., using either `SUM` or `AVG`.

Based on the above abstraction, we introduce the following high-level construct for generic cross-validation.

```
CROSSVAL <dataset>
GENSPLITS (type='kfold'|'holdout'|'bootstrap',
          element ='row'|'submatrix'|'cell',
          <type-specific parameters>) as <folds>
TRAIN trainFunction (<inputs>) as <models>
TEST testFunction (<inputs>) as <error>
AGGREGATE aggrFunction (<error>) as <output>;
```

The explanation for the construct is as follows. The dataset over which we are learning the model is provided in the variable `dataset`, typically as a path to the location of the dataset(matrix) on HDFS. Following the crossval statement, there are four main clauses. `GENSPLITS` specifies the split generation methodology, i.e., either using *k-fold*, *holdout* or *bootstrap* methods of cross-validation (Section 2.2). Next, we designate the elements on which we split the given matrix to generate the folds. For instance, in supervised classification problems, we typically partition the matrix into rows since each row has a class label. Other elements that the given dataset matrix can be partitioned on are cells ('cell' option, *Wold holdout*, Section 2.2) or submatrices ('submatrix' option, *Gabriel holdout*, Section 2.2). Note that we can partition the matrix into columns by calling row-partition on the transpose of the matrix.

We can also define method-specific parameters. For example, for k-fold cross-validation we must specify the number of folds k, while for holdout cross-validation the fraction of elements to be held out is specified. For supervised classification tasks, we need to specify whether to *stratify* the samples, based on the class label. The output of `GENSPLITS` is specified in <`folds`>, and these variables refer to the generated splits *per fold*. For example, in the case of supervised classification, we specify two variables {`test,train`}, while for unsupervised models as in Figure 1(c), we need to specify four variables {`A,B,C,D`}. These variables can be passed to user-defined functions (UDFs) specified in the `TRAIN` clause for training a model and the `TEST` clause for testing the learned model. We observe the train and test UDFs can either be functions specified in DML or calls to external Java packages to support machine learning methods not readily expressible in DML currently (e.g., decision trees). Finally, the `AGGREGATE` clause specifies how to aggregate the errors output from the TEST clauses, one for each fold. The most commonly used aggregation functions are sum and average.

Examples of the crossval construct are shown in Figure 3(a,d). In Figure 3(a), the CV construct is used for model-selection,

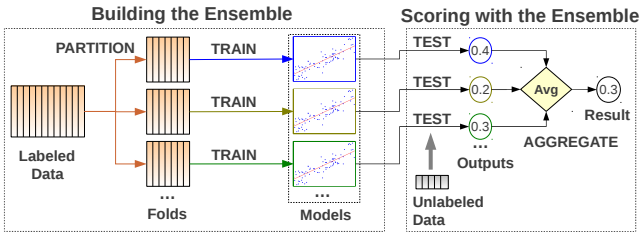


Figure 4: EL has four distinct operations: partition, train, test and aggregate

i.e., computing the best number of topics in the topic modeling application (using NMF). In this code, we compute the cross-validation error for different values of the `numtopics` variable and finally pick the number of topics (not shown here) with the smallest cross-validation error. Note that we are using the TRAIN routine `NMFCVtrain` and the `TEST` routine `test` which is shown in Figure 3(b). In Figure 3(d), we compute the cross-validation error for a linear regression problem. Note that we reuse the `test` function from before. In this example, we use two inputs (V, y) in the crossval construct – y is the column matrix of class labels. As described in Section 4, we ensure that MapReduce jobs appropriately partition both the matrices. In the next section, we describe how we implement the above language constructs in SystemML.

3.2 Ensemble Learning

As with cross-validation, we develop a similar abstraction for ensemble learning. The phases of EL bear a close resemblance to those for CV, as shown in Figure 4; however, there is one key difference. In ensemble learning, the split generation and model training phase (together *ensemble build*) occur once and the ensemble is persisted for future usage. The testing/scoring and prediction aggregation phases (together *use ensemble*) can be run multiple times over the ensemble on future datasets. Thus, the DML language abstraction for ensemble learning has both `BUILD ENSEMBLE` and `USE ENSEMBLE` constructs, as shown below.

BUILD ENSEMBLE <myEnsemble> **ON** <dataset>
GENSPLITS (**method**='bagging'|'holdout'|'rsm'
| 'partition', <type-specific parameters>) as <folds>
TRAIN trainFunction (<inputs>) [as <models>];

In the **BUILD ENSEMBLE** construct, the variable `myEnsemble` specifies the metadata file for persisting the ensemble, and contains a list of files encoding the ensemble models as either matrices in HDFS or PMML (Persistent Model Markup Language) files. The variable **dataset** specifies the input dataset, usually as a path to a matrix file on HDFS. `GENSPLITS` for EL specifies the method for split generation, which is either bagging, holdout, or random subspaces (RSM). Since the split method for EL implicitly defines both the type and element, the split element does not need to be separately specified as in CV. E.g., RSM method does holdout sampling on columns, while bagging does bootstrapping on rows. The **TRAIN** clause builds the model by invoking the trainFunction UDF. The output models are persisted and their locations are stored in the metadata file as described above.

USE ENSEMBLE <myEnsemble>
PREDICT predFunct(<inputs>) as <predictions>
AGGREGATE aggrFunc(<predictions>) as <result>;

The `USE ENSEMBLE` construct evaluates the ensemble referenced in `myEnsemble` on the given test dataset (specified as a variable in <`testInputs`>) as follows. Using the ensemble metadata from myEnsemble, the predFunct UDF specified in the `PREDICT` clause performs the computation of applying each ensemble model to the given dataset. The aggrFunct UDF specified in the `AGGREGATE` clause combines the predictions from each model together to compute the answer from the ensemble in `result`.

We make the following observations about the DML language constructs for ensemble learning. First, similar to CV, the UDFs in `TRAIN`, `PREDICT`, and `AGGREGATE` clauses may refer to DML functions or calls to external Java machine learning packages to support learning methods not readily expressible in DML (e.g., decision trees for random forests). `AGGREGATE` may refer to a learning method to support probabilistic combination functions such as stacking or mixture of experts, described in Section 2.3. Second, the `GENSPLITS` operation is common to both CV and EL, and requires handling large-scale data with multiple folds being constructed, which are subsequently used for training and testing. Figure 5 summarizes the intersection of CV and EL methods with respect to the partitioning types and elements we consider. Finally, our ensemble language construct can support parallel boosting approaches as proposed in [27] for AdaBoost. We use 'partition' for `GENSPLITS`, specify the `TRAIN` method to run AdaBoost for each partition to create a partition ensemble, and the `PREDICT` and `AGGREGATE` methods combine the ensembles from each partition in the manner specified in [27].

Type	Row (/Column)	Submatrix	Cell
k-fold	k-fold CV	Gabriel CV	N/A
Bootstrap	Bootstrap CV, Bagging EL	N/A	N/A
Holdout	Holdout CV, Holdout EL, RSM EL	N/A	Wold CV

Figure 5: Partitioning methods common to CV and EL

4. SYSTEM DESIGN

To implement the above constructs for CV and EL, we need to support two main tasks. First, we must partition and replicate the datasets appropriately to construct folds. Note that bootstrap CV and bagging EL require with replacement samples of the input. Second, we need to execute user defined functions to train models, measure errors (for cross-validation), and combine model predictions (for ensemble learning). The first item is is the main technical challenge for scalable meta-learning, and this section presents scalable algorithms to partition and sample large-scale data in a MapReduce setting. The scheduling and execution of UDFs to handle training and testing of models

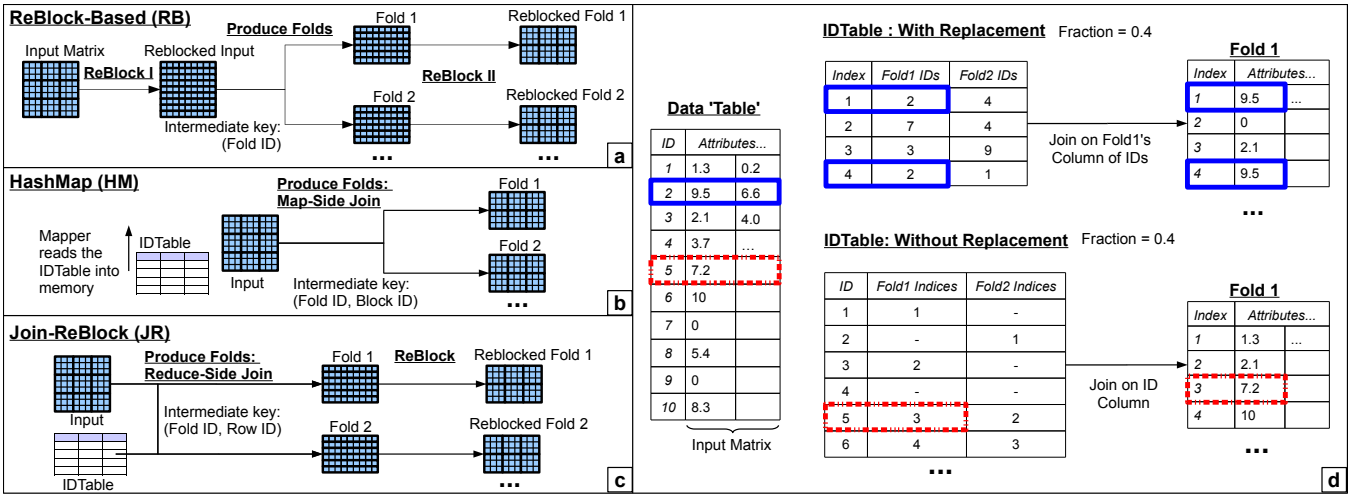


Figure 6: Illustrating row partitioning approaches: (a) Reblock-Based approach (RB) (b) HashMap approach (HM) (c) Join-Reblock appraoch (JR). (d) Using an IDTable for with replacement (bootstrap) and without-replacement (holdout) sampling.

leverages existing SystemML infrastructure, including available performance optimizations [11].

Preliminaries

The input data to the partitioning operation is a large, sparse matrix of dimensions m by n, where m and n are usually in the millions. The input matrix can either be in *cell* or *block* representation. In cell representation, the key-value pairs are of the form $((i,j),v)$, where i and j represent the matrix indexes and v represent the value of the cell (double). For block representation, the key-value pairs are of the form $((b_i,b_j),block)$ where (b_i,b_j) represents the block index and each block has $rsize$ rows and $csize$ columns. The index of cell (b_i,b_j) inside the block is given by $(i{*}rsize{+}b_i, j{*}csize{+}b_j)$. The generated outputs are also matrices, i.e., we need to assign valid row and column identifiers for the output entries. Block-representations are much more efficient that cell-representations [11], mainly due to the reduction in the number of key-value pairs required in the shuffle stage. Thus, for the partitioning, we assume both the inputs and outputs are in blocked format.

4.1 Row-based Fold Construction

As explained in Section 3, the `GENSPLITS` operation for CV and EL requires support for three approaches for row-based fold constructions. First, a *partition-based* approach is required for k-fold CV, i.e., each record in the input data is placed either in the training or test set for each fold. Second, *sampling without replacement* from the given data to generate the train and (for CV) test set for each fold / ensemble is used by holdout CV, holdout EL, and RSM EL (which samples columns from the given dataset, or rows from its transpose). Finally, *sampling with replacement* is required for bootstrap CV and bagging EL, where the training and test set for each fold is an independent with replacement from the given dataset. Row-based partitioning is challenging for a blocked matrix representation since each matrix block contains only part of a row of the input matrix, and matrix blocks (key-value pairs) are randomly assigned to different mappers.

Reblock-based (RB) Approach. A straightforward solution is to leverage the SystemML Reblock operation to convert the given $rsize \times csize$ matrix blocks to $1 \times n$ blocks, where each block contain the entire row and is referred to as a *row-block*. We then partition these row-blocks to obtain folds in row-block format, and then reblock again to covert to the original block format.

Figure 6 (A) illustrates the RB approach, which requires 3 MapReduce jobs. We start with the input dataset and run a MapReduce job to reblock it into row-blocks. The mapper groups each cell by its row id and the reducer has the information to reconstruct the complete row. In the second MapReduce job, which does the actual partitioning on the row-blocks, we construct the output folds based on random sampling. For μ-*holdout*, we randomly sample a number in $[0,1]$ and hold out the row (send to test dataset) if the number is less than μ and hold in the tuple (send to train dataset) otherwise. The reducer uses counters to reconstruct the indexes of the rows on the output folds. This is repeated (per tuple) for as many folds/iterations as required. Similarly, for k-fold, the partitioning mapper assigns each row block to 1 out of k test folds (picked randomly), and sends it to the train fold for the remaining $k-1$ folds. Finally, in the third MapReduce job, we again use the Reblock operation in SystemML on all output fold matrices to reconvert them to the original block format. We generate multiple outputs from a MapReduce job in Hadoop using the *MultipleOutputs* package in Hadooop.

Note that RB approach runs overall in linear time (in the amount of data) since we operate on each tuple independently during the partitioning job. However, the RB approach cannot efficiently handle the third kind of row-partitioning needed – sampling with replacement, required for for bootstrap CV and bagging EL. It has been shown in prior work that with replacement sampling is non-trivial to parallelize correctly and efficiently [28]. Correct with replacement sampling will be linear time for each row if done in parallel, making it quadratic overall and unsuitable for large-scale data. Thus, the baseline RB approach cannot be used for bootstrap CV and bagging EL. To solve this

problem, we propose a novel approach of implementing distributed sampling using database-style joins.

Join-based Approaches. For these approaches, we precompute the results of the random sampling to obtain the would-be row-ids in the output for each input row *before* actually splitting the data. This (input row-id, output row-id) mapping is written out to an *IDTable* in a logically central location (the control node) as a distributed sequence file, and is joined with the input matrix using database-style joins over MapReduce. Two variants of the join-based approach are supported. The *HashMap method (HM)* maintains in memory the IDTable and performs a map-side join, while the *Join-Reblock method (JR)* performs a reduce-side join [36].

Figures 6(b,c) illustrate HM and JR, respectively. HM requires a single MapReduce job to go from input data to outputs folds (all in block format), effectively merging the partitioning and reblocking into a single step. JR requires two MapReduce jobs, one to effect partitioning using reduce-side join, and one to reblock the output folds back to block format. In HM, the IDTable is sent to each mapper node using the *DistributedCache* feature of Hadoop. Each mapper loads the IDTable into memory, and then looks up the output fold and index of each row in the input matrix block. Each reducer aggregates one block of one output matrix, thus obtaining all matrices directly in block format. In JR, one kind of mapper outputs cells of the input matrix based on row-id and the other mapper outputs entries of the IDTable based on row-id. The reducer aggregates the row-blocks in each matrix, and subsequently joins it with the IDTable entry for that row, producing output folds in row-block format. We perform a SystemML Reblock operation to obtain the output matrices in block format.

We now describe how the IDTable is used for sampling with replacement, which cannot be handled by RB. Figure 6(d) shows the original input matrix, visualized as a table with row-ids as keys. The IDTable contains as many rows as the sample size (based on μ), and each column represents one fold. The entries in a column are basically a sequential list of input matrix row-ids, sampled with replacement. In the illustration, row 2 of the input occurs twice in the output (as rows 1 and 4). Since the IDTable is written out incrementally, only a portion is held in memory during construction. The JR approach splits the entries of the IDTable and send entries to the reducer based on input row-ids. However, the HM approach constructs an in-memory inverted hashmap while reading the IDTable (mapping from input row-id to list of future row-ids), and send entries to the reducer based on the output fold and block numbers.

Leveraging the IDTable to handle partitioning (for k-fold CV) and sampling without replacement (for holdout methods) is straightforward. Each IDTable row corresponds to each input matrix row (Figure 6(d)). The experimental evaluation in Section 7 demonstrates that HM and JR approaches are faster than RB for partitioning and sampling without replacement for a wide range of input data characteristics. We observe that the ensemble evaluation step for RSM requires information about which columns were sampled to construct each ensemble model. Since the IDTable provides exactly this information, we only use HM and JR methods for RSM.

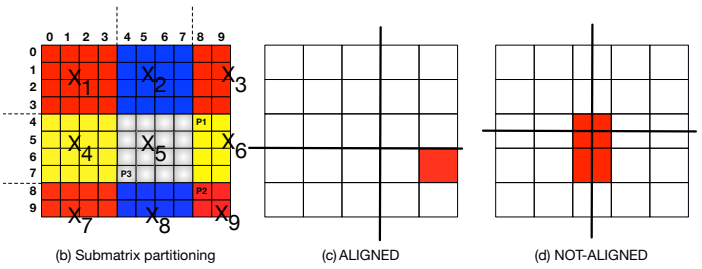


Figure 8: (a) Illustrating submatrix partitioning. The grey portion corresponds to A, yellow corresponds to B, blue corresponds to C and the red portion corresponds to the D matrix. (b) When the partitioning boundary aligns with the block boundaries. Each small rectange indicates a block. (c) Case when the block boundaries do not align with the partition boundary.

4.2 Submatrix Partitioning

We now describe methods to partition the input matrix into submatrices to support cross-validation for unsupervised models such as PCA, SVD and NMF. As described in Section 2.2, the rows and columns of the input matrix must first be randomly shuffled before being partitioned into submatrices. We now describe both the shuffling and partitioning steps.

Shuffling: In this step, we shuffle the rows and columns of the matrix using an approach similar to the HashMap method (HM) described in the previous section. We populate a hash table with randomly selected new row (column) indexes for existing rows (columns), then perform a map-side join of this hash table with the input dataset to create a randomly shuffled matrix. The hash table fits into memory since the number of its entries is small (#rows + #columns). Since SystemML uses a sparse matrix representation, only non-zero entries in the matrix are shuffled. The output of the shuffling step is a matrix in cell format and may be subsequently blocked using a Reblock step.

Partitioning: After shuffling, we partition the matrix into contiguous boundaries. We use the example of 3×3 Gabriel holdouts on a 10×10 matrix as shown in Figure 8(a). Of the 9 folds generated, the figure shows the fold in which columns $4, 5, 6, 7$ and rows $4, 5, 6, 7$ are held out and the rest are held in. For each cell in the input matrix, we determine (1) the output matrix to which it belongs and (2) its index in the output matrix. For instance, the cell $P_1 = (5, 8)$ in the original matrix belongs to the output matrix B. Its index in B is $(1, 4)$ since we need to include all the yellow entries in B. Similarly, the cell $P_2 = (8, 8)$ in the original matrix belongs to D and its index in D is $(4, 4)$. Also, the cell P_3 belongs to the matrix A and its index in A is $(2, 1)$.

Although the above algorithm is described in terms of iterating over every cell in the original matrix, we typically use the blocked format and iterate over matrix blocks which is much more efficient. Only the block id (b_i, b_j) for each input block requires modification. One potential issue with block format is that the block boundary may not align with the submatrix partitions. Consider the example of 2×2 Gabriel holdouts on a 5×5 block matrix as shown in Fig-

Cost	Reblock-Based (RB)	Join-Reblock (JR)	HashMap (HM)
I/O (b)	$24mn + 24mnk$	$8m(n+k) + 24mnk$	$8m(n + kP + nk)$ $(+8mkP\ distribution)$
Shuffle (b)	$8mn + 16mnk$	$8m(n+k) + 8mnk$	$8mnk$ $(+8mkP\ distribution)$
Shuffle (kv)	$m(n+k) + mnk$	$m(n+k) + mnk$	mnk

Figure 7: Cost model for row-partitioning: m: # rows in input matrix, n: # columns, k: # folds, P: # mapper nodes. The first row is I/O cost in bytes, the second row his shuffle cost in bytes, and third row is the number of key-value pairs shuffled in the MapReduce jobs of each approach.

ures 8(b,c). If the partition boundary aligns with the block boundary, as in Figures 8(b), we can directly iterate over each block. For the not-aligned case (Figure 8(c)), we have two alternatives. The first alternative is to use the cell-based approach and iterate over the cells in each block, which is inefficient. The second alternative is to *nudge* the partition boundaries such that they align with the block boundaries. As illustrated in Figure 8(c), we move the partition boundaries (to the right and below) such that it matches that of Figure 8(b). The size difference for matrices obtained using this heuristic is minimal and we experimentally observed no bias to CV errors when used on real-world topic modeling datasets [19].

4.3 Cell partitioning algorithms

We now present algorithms for constructing folds by partitioning the cells of the matrix. In this case, we only support holdout cross-validation, i.e., we randomly holdout a small fraction of the cells in the matrix. Since SystemML [11] uses a sparse matrix representation, only non-zero elements in the matrix are explicitly stored. Hence, iterating over the cells of the matrix does not uniformly partition the matrix. Instead, we generate bit matrices corresponding to the held out portion of the matrix. Each bit matrix B has entries either 0 or 1. If $B_{ij} = 1$, then the corresponding entry in the data matrix A is held out (i.e., test component), other wise, the entry is held in (i.e., in the train component). Note that the training algorithm for CV using cell-based partitioning needs to be written based on these bit matrices. [19] developed efficient strategies to learn *weighted NMF* models using such bit matrices.

5. COST-BASED OPTIMIZATION FOR CV AND EL

This section describes how SystemML selects the appropriate plan for evaluating the declarative cross-validation and ensemble learning operators from the space of possible plans. The user-defined functions implementing training (model building), testing (using ensemble), and error aggregation (prediction combination) in CV (EL) are already optimized by SystemML. Thus, this section focuses on selecting the most efficient method for the `GENSPLITS` task from the available methods and the scheduling of tasks to evaluate the CV and EL operators (i.e., `GENSPLITS`, the UDFs for training, test, and error aggregation).

`GENSPLITS` cost model: In Section 4.1, we propose three possible implementations of row-based split generation. To determine the most efficient method at runtime, we develop a cost model based on the data and system characteristics most influencing the performance of these methods. Thus, SystemML can use the cost model to automatically select at runtime the most efficient method.

The costs for the three row-based split generation algorithms described in Section 4.1 are shown in Figure 7. We measure the costs for I/O in terms of total number of bytes, and for shuffling in terms of both total number of bytes and number of key-value pairs that flow through the jobs. As we can see from the table, there is no clear winner among the three approaches in all cases. The HashMap (HM) approach has the lowest shuffle costs in most cases; however, it has a significant I/O cost since the map table needs to be read by each of the mappers (distribution costs). Furthermore, HM is limited by the memory size in each mapper (i.e., the HashMap of size $8mk$ bytes must fit into the memory of the mapper, where m is the number of rows and k is the number of folds / ensemble models). Overall, the Reblock (RB) approach has the highest costs in most cases (for large-scale data, usually $m, n >> k, P$), with the Join-Reblock (JR) method having performance between the RB and HM. Either the HM or JR method is required for RSM and sampling with replacement, with the JR method not having the memory limitation of HM.

We currently use the following approach for the optimizer to pick the method. First, we check if hashmap for HM can fit in the memory of each mapper; if so, we use the HM strategy. Otherwise, we compute the shuffle costs for JR and RB methods and select the method with the smaller shuffle cost. If the shuffle costs are are comparable, we use the method with the smaller I/O cost. As we see later in the experiments, for the datasets we considered, the optimizer usually picks the HM method due to large available memory. However, if memory on each mapper is limited, the optimizer typically picks JR, as suggested by the cost formulae. Our experimental evaluation in the next section validates the cost model captures the characteristics which dominate the performance of the pa

For submatrix partitioning (Section 4.2), we showed the block-based method is more efficient for all datasets and only a single method for cell partitioning was presented in Section 4.3.

Scheduling: Since we know the exact semantics for CV and EL (i.e., split generation, multiple training and model evaluation) we can effectively parallelize and schedule computation as necessary. As described, the `GENSPLITS` operator constructs all splits by replicating the dataset for each fold or ensemble model training set. However, this may not be feasible in practice, especially when considering very large-scale 10000-fold cross-validation or ensembles with thousands of models. To support this case, we make a straightforward modification to `GENSPLITS` to only generate per execution as many folds as feasible, based on available storage on the

HDFS.

Similarly, SystemML can evaluate in parallel multiple folds for CV and EL. Each CV fold produces two UDF calls, a training function call and testing function call. Since the CV folds are independent, an arbitrary number of these UDFs can be scheduled in parallel based on available system resources. After all folds are evaluated, the final error aggregation UDF is invoked. EL is parallelized in a similar manner, since the construction of each ensemble model is independent.

6. IMPLEMENTATION DETAILS

We briefly mention other relevant implementation details for the CV and EL language operators.

Runtime compilation:
In addition to the DML language constructs, we introduced additional HOPs and LOPs to support CV and EL. We compile and optimize the train and test user-defined functions at runtime, for each fold or ensemble model after obtaining the sizes of the matrices resulting from the `GENSPLITS` step. The overhead of repeated compilation is negligible (milliseconds) relative to function execution time.

Pseudo random sampling on Hadoop
SystemML uses random sampling to compute the various folds for cross-validation. Since SystemML is based on a parallel platform Hadoop, care must be taken to ensure that the random sampling is unbiased and perfectly pseudo random. For this purpose, we use `WELL1024` [23], a long period random number generator (PRNG) with period of approximately 2^{1000}. Each period can be split into 2^{300} streams each of length 2^{700}, and efficient skipping over substreams is supported as well. We incorporate WELL1024 PRNG into SystemML as follows. Each mapper is assigned a unique identifier and a particular number of substreams based on the type of partitioning required. For example, suppose that each mapper requires 2 streams (k-fold stratified cv with domain size = 2). In that case, if we get to the mapper with id = 4, we first skip 6 times to get the first stream for the mapper and skip again to get the second stream for the mapper. Experimental results indicate the time taken for such skipping is typically subsecond for over 10,000 substreams.

Supporting Stratification
As mentioned in Section 2, supervised learning tasks require the folds to be stratified based on the class label. We support stratification for row-based partitioning tasks using multiple seeds. We illustrate with an example. Suppose that we want to classify emails as spam or genuine. In this case, we use 2 seeds, one for each email type. For each row, we examine its class label and sample from the appropriate seed.

7. EXPERIMENTS

The goal of our experiments is to study the performance and scalability of the various algorithms for data partitioning under different data and system characteristics.

Experimental Setup. The experiments were conducted with Hadoop 0.20 on a 40-core cluster with 5 local machines as worker nodes. Each machine has 8 cores with hyperthreading, 32 GB RAM and 2 TB storage. We set each node to run XX concurrent mappers and XX concurrent reducers. The datasets are synthetic, and for given dimensionality and sparsity, the data generator creates random matrices with uniformly distributed non-zero cells. A matrix block size of 1000×1000 is used.

7.1 Row Partitioning

We first compare the performance and scalability of the three row partitioning schemes – Reblock-Based (RB), Join-Reblock (JR) and HashMap (HM). We vary the input matrix characteristics, viz., number of rows, number of columns and sparsity, fixing two at a time, as well as the number of folds.

Number of Rows. Figure 9 plots the runtimes for k-fold CV and bagging EL against the number of rows. The number of columns is fixed at 10000 with sparsity at 0.1% and 5 output folds. The fraction (relative size of train fold) for bagging is 0.3. We see that all three methods (RB, JR and HM) scale linearly with the number of rows. As expected, HM performs the fastest (nearly 3x faster than RB), and JR is only slightly faster than RB. However, it should be noted that as the number of rows goes up, HM will eventually crash due to insufficient memory, while JR and RB continue to scale. Also, RB is not applicable to bagging since it involves with replacement sampling as explained in Section 4.1.

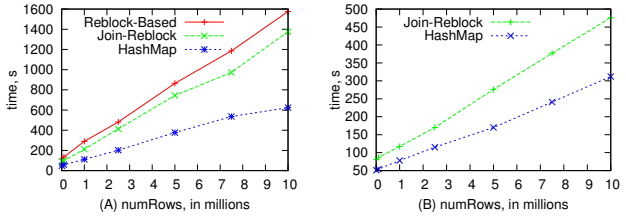


Figure 9: Performance of Row-partitioning schemes: (a) k-fold CV (b) Bagging EL against number of rows

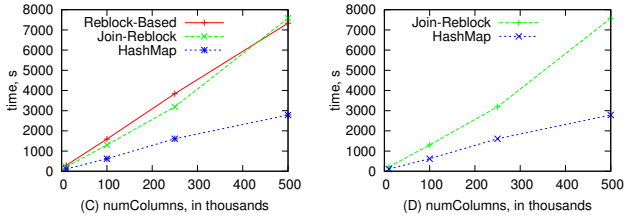


Figure 10: Performance of Row-partitioning schemes: (a) k-fold CV (b) RSM EL against number of columns

Number of Columns. Figure 10 plots the runtimes for k-fold CV and RSM EL (which partitions on columns) against the number of columns, fixing the number of rows at 1 million and sparsity at 0.1%. The number of folds is again 5 and the fraction for RSM is set at 30%. Again, we see a linear scaleup for all methods, but JR performs almost the same as RB as number of columns goes higher. This is because the reduce-side join becomes more CPU-intensive owing to the larger row sizes. Again, RB doesn't handle RSM as explained in Section 4.1.

Sparsity. Figure 11 (a) plots the runtimes for holdout EL against the sparsity of the input matrix, fixing the other variables. Since the number of cells that flow in the MapReduce jobs is linearly dependent on the sparsity, we see that all the methods scale linearly with sparsity.

Folds. Figure 11 (b) plots the runtimes for Holdout CV against the number of folds, fixing the other variables. Again, as expected, we see a linear increase in the runtime for all the three methods.

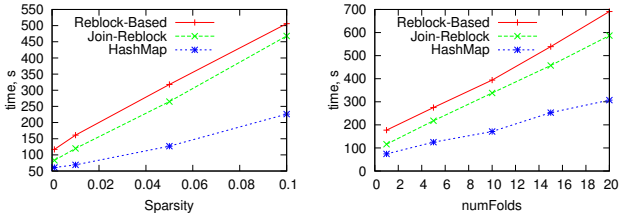


Figure 11: Performance of Row-partitioning schemes: (a) Holdout EL against sparsity (b) Holdout CV against number of folds

7.2 Submatrix Partitioning

Next, we compare the performance of the two submatrix partitioning schemes – Cell-Based (P1) and Block-Based (P2). In this experiment, we illustrate the advantages of blocking the input matrix for submatrix partitioning. We start with a matrix in cell format and partition it using two different plans. In the first approach P1, we execute 2×2 partitioning using the cell-based approach (Section 4.2. In the second approach P2, we use a two step process in which we initially block the matrix using a reblock job and subsequently use the block approach for partitioning. We measure the time taken for each case. For the second case, we compute the time taken for both the reblocking and the partitioning. The results are shown in 12 (a). We execute two experiments with different column sizes. In each experiment, we vary the number of rows in the matrix. As shown in the figure, the time taken in P_2 is much less than P_1. We would like to note here that this may seem counter intuitive since we are actually making two passes over the data. The reason for this is that in the first case, we are replicating cells (while shuffling) where as in the second case we are only replicating blocks, which are much fewer in number.

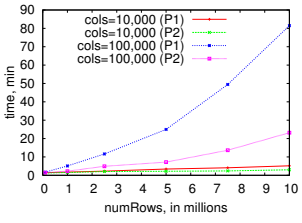


Figure 12: Performance of submatrix-partitioning schemes

7.3 Speed-up and Scale-up

Here, we study the speed-up and scale-up behavior of the row partitioning methods. We plot the runtimes against the number of reducers in Figure 13(a). We then vary the dataset size proportional to the number of reducers and plot the runtimes in Figure 13 (B). We (should) see that the speed-up is linear as we increase the number of reducers. However, the scale-up is not flat since there will be more network overheads as the number of reducers increases.

Figure 13: Speedup and Scaleup: TODO

8. RELATED WORK

Large-scale machine learning has become an important research task given the *application pull* – spam detection, personalized search, recommendation application etc. and *technology push* – availability of large-scale data processing tools such as MapReduce and multicore systems. Examples of such systems include Apache Mahout [1], Google prediction API [12], SystemML [11], GraphLab [25], Pegasus [20], Spark [38], and DryadLINQ [17]. Further, there has been a great deal lot of literature including [7, 8]. Our work extends one of this large scale machine learning systems, SystemML, to support cross-validation and ensemble learning as a first-class declarative operator.

Sections 2.2 and 2.3 provide background on cross-validation and ensemble learning. Typically, a standalone package will support CV or EL for a particular machine learning technique. E.g., the `randomforest` package in R implements random forest ensembles for decision trees. Approaches to parallelize CV and EL in a MapReduce setting typically address a single technique as well. [27] parallelizes AdaBoost and LogitBoost for ensembles, while PLANET [28] builds ensembles of *decision trees* over very large scale data using MapReduce. In contrast, we develop declarative operators for generic meta learning for all kinds of models and ensemble techniques. Our approach can efficiently support both the parallel boosting in [27] and random forests, along with CV and EL for many other kinds of models.

9. CONCLUSIONS

Massive amounts of data generated on the web are used to routinely train very complex machine learning models for solving a range of problems such as spam detection, recommender systems, personalized search and advertising. Consequently, verifying the quality of machine learned models has become an important task. Cross-validation is a versatile tool that has been extensively used for solving this problem, as well as a number of related problems such as model-selection and feature selection. In this paper, we build a system that provides the ability to cross-validate all possible machine learning models (including unsupervised models) over very large-scale datasets using the MapReduce framework. As we show in the paper, owing to the multitude of methods to cross-validate a given machine learning model (for e.g., partitioning techniques) and the various potential implementation strategies, we need to develop a declarative system that can choose the best possible plan for a given cross-validation task. In future, we intend to extend our system for performing other meta-learning tasks such as boosting, stacking and other ensemble strategies. We also plan to develop scalable cross-validation strategies for non i.i.d samples such as relational learning models.

10. REFERENCES

[1] The Mahout Project. Web Site. `http://mahout.apache.org/`.

[2] R. Bell, Y. Koren, and C. Volinsky. All Together Now: A Perspective on the Netflix Prize. *CHANCE*, 23:24–24, 2010.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *JMLR*, 3:993–1022, 2003.

[4] U. Braga-Neto and E. R. Dougherty. Is Cross-validation Valid for Small-sample Microarray Classification? *Bioinformatics*, 20(3), 2004.

[5] L. Breiman. Bagging Predictors. *Machine Learning*, 24:123–140, 1996. 10.1007/BF00058655.

[6] E. Candes and T. Tao. The Power of Convex Relaxation: Near-optimal Matrix Completion. In *IEEE Trans. Information Theory 56(5)*, pages 2053–2080, 2009.

[7] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. In *NIPS*, 2006.

[8] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2):1481–1492, 2009.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[10] Y. Freund and R. E. Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *JCSS*, 55(1):119–139, 1997.

[11] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, pages 231–242, 2011.

[12] Google Prediction API. Web Site. `http://code.google.com/apis/predict/`.

[13] I. Guyon and A. Elisseeff. An Introduction to Variable and Feature Selection. *JMLR*, 3:1157–1182, 2003.

[14] A. Y. Halevy, P. Norvig, and F. Pereira. The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[15] T. K. Ho. The Random Subspace Method for Constructing Decision Forests. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20:832–844, August 1998.

[16] A. Isaksson, M. Wallman, H. Göransson, and M. G. Gustafsson. Cross-validation and Bootstrapping are Unreliable in Small Sample Classification. *Pattern Recognition Letters*, 29(14), 2008.

[17] M. Isard and Y. Yu. Distributed Data-parallel Computing using a High-Level Programming Language. In *SIGMOD*, pages 987–994, 2009.

[18] G. K. Le Biplot Util dexplaration de Donnees Multidimensionellles. In *J. Roy. Stat. Soc. Series*, 2002.

[19] B. Kanagal and V. Sindhwani. Rank Selection in Low-rank Matrix Approximations. *NIPS*, 2010.

[20] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A Peta-Scale Graph Mining System. In *ICDM*, pages 229–238, 2009.

[21] R. Kohavi and D. Sommerfield. Feature Subset Selection Using the Wrapper Method: Overfitting and Dynamic Search Space Topology. In *KDD*, pages 192–197, 1995.

[22] H.-P. Kriegel, P. Kröger, and A. Zimek. Clustering High-dimensional Data: A Survey on Subspace Clustering, Pattern-based Clustering, and Correlation Clustering. *TKDD*, 3:1:1–1:58, March 2009.

[23] P. L'Ecuyer. Random Numbers for Simulation. *Commun. ACM*, 33, October 1990.

[24] D. D. Lee and H. S. Seung. Learning the Parts of Objects by Non-negative Matrix Factorization. In *Nature*, pages 788–791, 401(6755) 1999.

[25] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *UAI*, 2010.

[26] A. Niculescu-Mizil, C. Perlich, G. Swirszcz, V. Sindhwani, Y. Liu, P. Melville, D. Wang, J. Xiao, J. Hu, M. Singh, W. X. Shang, and Y. F. Zhu. KDD Cup 2009 Winning the KDD Cup Orange Challenge with Ensemble Selection. *JMLR: Workshop and Conference Proceedings 7: 23-34*, 2010.

[27] I. Palit and C. K. Reddy. Scalable and Parallel Boosting with MapReduce. *IEEE TKDE*, 2011.

[28] B. Panda, J. Herbach, S. Basu, and R. J. Bayardo. PLANET: Massively Parallel Learning of Tree Ensembles with Mapreduce. *PVLDB*, 2(2):1426–1437, 2009.

[29] Parallel Computing Toolbox. Web Site. `http://www.mathworks.com/products/parallel-computing/`.

[30] P. O. Perry and A. B. Owen. Bi-cross-validation of the SVD and the Non-negative Matrix Factorization. In *Annals of Applied Statistics*, pages 564–594, 2009.

[31] R. Polikar. Ensemble Learning. *Scholarpedia*, 4(1):2776, 2009.

[32] Revolution Analytics. Web Site. `http://www.revolutionanalytics.com/`.

[33] T. Scheffer and T. Joachims. Expected Error Analysis for Model Selection. In *ICML*, pages 361–370, 1999.

[34] N. Srebro and T. Jaakkola. Weighted Low-Rank Approximations. In *ICML*, pages 720–727, 2003.

[35] The R Project for Statistical Computing. Web Site. `http://www.r-project.org/`.

[36] T. White. *Hadoop: The Definitive Guide*. OŔeilly Media, 2009.

[37] H. Wold. Cross-validatory Estimation of the Number of Components in Factor and Principal Component Models. In *Technometrics*, 1978.

[38] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.