Apache Spark MLlib

Machine Learning Library for a parallel computing framework

Review by Renat Bekbolatov (June 4, 2015)

Spark MLlib is an open-source machine learning library that works on top of Spark. It is included in the standard Spark distribution and provides data structures and distributed implementations of many machine learning algorithms. This project is in active development, has a growing number of contributors and many adopters in the industry.

This review will attempt to look at Spark MLlib from the perspective of parallelization and scalability. In preparation of this report, MLlib public documentation, library guide and codebase (development snapshot version of 1.4) was referenced[1]. While MLlib also offers Python and Java API, we will focus only on Scala API.

There will be some technical terms that are outside the scope of this report, and we will not look at them in details. Instead we will explain just enough to show how a specific MLlib component is implemented and, wherever possible, provide references for further independent study if needed.

1 Execution Model

Iterations

Iterative methods are at the core of Spark MLlib. Given a problem, we guess an answer, then iteratively improve the guess until some condition is met (e.g. Krylov subspace methods). Improving an answer typically involves passing through all of the distributed data and aggregating some partial result on the driver node. This partial result is some model, for instance, an array of numbers. Condition can be some sort of convergence of the sequence of guesses or reaching the maximum number of allowed iterations. This common pattern is shown in Fig 1.

In a distributed environment, we have one node that runs the driver program with sequential flow and worker nodes that run tasks in parallel. Our computation begins at the driver node. We pick an initial guess for the answer. The answer is some number, an array of numbers, possibly a matrix. More abstractly, it is a data structure for some model. This is what will be broadcast to all the worker nodes for processing, so any data structure used here is serializable.

In each iteration, a new job is created and worker nodes receive a copy of the most recent model and the function (closure) that needs to be executed on the data partition. Computation of model partials is performed in parallel on the workers and when all of the partial results are ready they are combined and a single aggregated model is available on the driver. This execution flow is very similar to BSP[13]: distribute tasks, synchronize on task completions, transfer data, repeat.

Whenever possible, worker nodes cache input data by keeping it in memory, possibly in serialized form to save space, between iterations[1]. This helps with performance, because we avoid going to disk for each iteration. Even serialization/deserialization is typically cheaper than writing to or reading from disk.

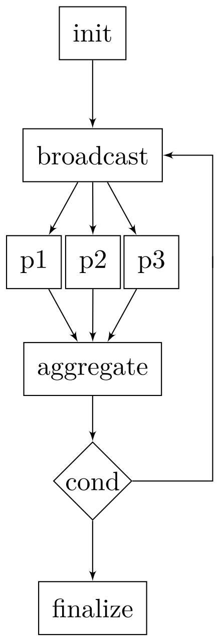

Figure 1: Iterations

When considering model partials, it is important to keep the size of the transferred data in mind. This will be serialized and broadcast in each iteration. If we want our algorithm implementation to be scalable, we must make sure that the amount of data sent doesn't grow too fast with the problem size or dimensions.

Each iteration can be viewed as single program, multiple data (SPMD), because we are performing identical program on partitions of data.

BSP

When analyzing performance of an iterative algorithm on Spark, it appears that it is sufficient to use

[1]This is in contrast to Spark predecessors, for instance MapReduce/Hadoop, where data must be read from disk on each iteration.

BSP[13] model. Individual pair-wise remote memory accesses are below the abstraction layer and models such as *CTA* and *LogP* do not provide additional benefit when operating with Spark abstractions[12, 14].

At the beginning of each iteration we broadcast tasks and wait for all the partitions to compute their partial results before combining them. This is similar to super-steps with barrier synchronization in *BSP*. Communication between processors is typically driver-to-all, and all-to-driver patterns[1].

The most important factors of algorithm analysis on Spark are data partition schemes and individual partition sizes, partial model sizes and communication latency.

2 Infrastructure

Core Linear Algebra

Spark MLlib uses *Breeze*[2] and *jblas*[3] for basic linear algebra. Both of these libraries are enriched wrappers around *netlib*[4], which make it possible to use *netlib* packages through Scala or Java. *Breeze* is a numerical processing library that loads native *netlib* libraries if it they are available and gracefully falls back to use their JVM versions. JVM versions are provided by F2J library. Library *jblas* also uses *netlib* BLAS and LAPACK under the hood.

Using *netlib* for numeric computations not only helps us improve speed of execution, better use of processor caches, it also helps us deal with issues of precision and stability of calculations[2].

Local Data Types

Core MLlib data types are tightly integrated with *Breeze* data types. The most important local data type is a *Vector* - which is stored internally as an array of doubles. Two implementations are available: *DenseVector* and *SparseVector*[3].

An extension of *Vector* is a concept of a *LabeledPoint* - which is an array of features: *Vector*, along with a label (class) assignment, represented by a double. This is a useful construct for problems of supervised learning.

A local matrix is also available in MLlib: *Matrix* - and as the case with *Vector*, it is stored as an array of doubles, has two implementations: *DenseMatrix* and *SparseMatrix*[4].

RDD: Distributed Data

MLlib builds upon Spark RDD construct. To use MLlib functions, one would transform original data into an RDD of vectors. Newer Spark MLlib functionality allows operations on a higher-level abstraction, borrowed from other popular data processing frameworks: *DataFrame*. In a later section (3.1, Library Functions: Distributed Matrices) we will discuss distributed matrices in detail.

RDD[T] can be seen as a distributed list containing elements of type *T*. One can transform RDDs by transforming contained values in some way, and force materialization of some results of such transformations through *actions*. Let us briefly go through a few basic transformations and actions.

Let us recap a couple of basic RDD operations. Consider an RDD of integer numbers, A, stored in 3 partitions:

$$A \leftarrow [[0,1,2,3],[4,5,6],[7,8,9]]$$

Here our first partition contains [0, 1, 2, 3], second contains [4, 5, 6], and the last partition has [7, 8, 9]. Now, we can create another RDD, B, as follows:

$$B \leftarrow A.map\{n \Rightarrow n+1\}$$

B is now $[[1,2,3,4],[5,6,7],[8,9,10]]$. We could perform a reduction operation, say calculate the sum of all elements in B with an action operation *reduce*:

$$sum \leftarrow B.reduce\{(a,b) \Rightarrow a+b\}$$

Calling *reduce* starts a computation that will touch each element in each partition and combine results into a single value that will be available on the driver node.

Aggregations

Action *reduce* might be the simplest non-trivial aggregation operation. It requires that the type of the

[1]Communication will be explained in section *Infrastructure*, where *aggregation* and *shuffle* procedures are explained.

[2]Talk by Sam Halliday on high-performance linear algebra in Scala, where he describes how one can access functionality of *netlib* native routines on JVM: https://skillsmatter.com/skillscasts/5849-high-performance-linear-algebra-in-scala

[3]Internally, both of these provide quick conversions to and from *Breeze* equivalents

[4]Again, internally, *Breeze* matrix conversions are also provided

output of the reduction function is same as the type of the values stored in RDD. In each partition, this reduction function will be applied to values in that partition and the running partial result. Then each of the partial results is sent to the driver node and the same reduction application is performed on those partial results locally.

If we wanted to have an output of a different type, we could use an action *aggregate*, to which we could supply both the append-like and merge-like functions: one for adding individual values to the partial result, and the other for merging partial results. With *aggregate*, as with *reduce*, we will go through each value in a partition and perform an append-like operation to come up with a partial result. Then each of the partial results is sent to the driver and now the merge-like operation is performed on those partial results locally on the single driver node.

Actions *reduce* and *aggregate* are not the only ways to aggregate values stored in an RDD[1]. In fact, in MLlib, it is a common pattern to aggregate in trees: *treeReduce* and *treeAggregate*, which attempt to combine data in a tree-like reduction: initial partitions are leaf nodes and final result is the root node. Users can specify how deep the tree should be and Spark will minimize the communication cost, repartitioning and reducing the data at each tree aggregation step[2].

Tree aggregation is useful when merging of model partials requires considerable amount of computation and if there are a large number of partitions, where it might be cheaper to spread out this merge computations onto the worker nodes. In any case, merge operation must be commutative and associative[3].

In any aggregation, the step of transferring data will have to go through writing to disk and a remote read from another node. More generally, this operation is called *shuffle*.

Shuffle

Shuffle is an inter-node communication between stages of a Spark job, potentially an all-to-all communication. Initially, on each node, we start transforming our RDD data locally, and proceed doing so as long as we do not need data from other partitions.

As soon as we need data from other nodes, we block and retrieve data from those remote nodes that hold data for a new data partition that is assigned to this node (Fig 2).

How expensive is Spark shuffle? On each node, it needs to write a portion of data designated for some other partition in some separate file, later to be pulled by the node to which this piece of data belongs. Let us imagine the *worst* case scenario, where re-partitioning is spreading each previous partition into all other partitions. Then on *each node*, we will have to write *all* data to disk, and *each node* will have to pull data from *each other node* - all of the data will be moved and all of the data cached in memory on each node become useless, because now each node has to work with new data.

Shuffle operation is performed during many standard RDD operations. For example, if we attempt to join two RDDs of key/-value pairs on their keys, we are re-partitioning our data into a common key partitioning and thus perform a shuffle.

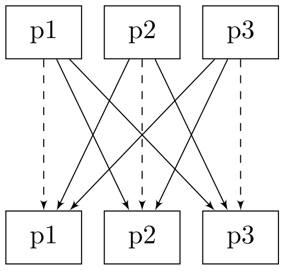

Figure 2: Shuffle

To build efficient programs on Spark, it is important to partition data with consideration of required data transfers. For example, this is done in *treeAggregate* described earlier: at each level of the tree, on average only half of the current data gets transferred while the other half stays on the same node for further processing. MLlib implementation is aware of this concern.

3 Library Functions

Let us look at available MLlib functionality in detail. This section roughly follows MLlib's core lower level `mllib` package organization. We will go through most of the library functionality, discuss implementation details and comment on notable features, strengths and limitations of implementation, mainly focusing on parallelization and scalability.

3.1 Distributed Matrices

Package `linalg.distributed` contains some implementations of distributed matrices[4]. The package provides a root interface for distributed matrices:

[1]See Spark documentation for more action operations.
[2]By default, Spark uses 2-level trees to aggregate data.
[3]making the set of aggregations a commutative semi-group.

[4]`org.apache.spark.mllib.linalg.distributed`

DistributedMatrix. For the remainder of this section, let us consider a $m \times n$ matrix M with elements v_{ij}, column vectors $c_1, c_2, ..., c_n$ and row vectors $r_1, r_2, ..., r_m$.

We use *RowMatrix* to partition matrix data into row vector ranges: an RDD of row vectors r_i. To block-partition the matrix we use *BlockMatrix*. Sometimes it is convenient to use *CoordinateMatrix* to construct distributed matrices from distributed matrix entries (i, j, v_{ij}). *CoordinateMatrix* provides methods to convert itself into *RowMatrix* or *BlockMatrix* forms.

RowMatrix is the workhorse of Spark distributed matrix computations. Typically, distributed operations are implemented for *RowMatrix* and operations on other implementations would first convert to *RowMatrix* and then reference these implementations.

All of the following implementations are for *RowMatrix*, with the exception of eigenvector finding and distributed matrix multiplication. Eigenvectors can be computed for a matrix without having it materialized, i.e. we only need to provide a function that will calculate products of this matrix by some input vector, and so we do not need to actually have any RDD representing this matrix. Distributed matrix multiplication is implemented only for *BlockMatrix* form. Let us look at matrix operations in more details.

Gram Matrix

Calculation of Gram matrix

$$G = M^\dagger M, \ s.t.\ G_{ij} = \langle c_i, c_j \rangle$$

is also parallelizable. This is done by aggregation of $\sum_{i=1}^{m} r_i^\dagger r_i$ over all rows r_i. As with most other computations on arrays, under the hood, each matrix/vector operation is performed by BLAS routines. One Spark job will compute Gram matrix and make it available on the driver node. Each worker node will have to use $O(n^2)$ space to store intermediate computed data and communicate it during aggregation step. Aggregation results in a single local matrix on the driver node.

Again, just as with eigenvector search, this means that materialization of Gram matrix through this computation is not scalable in the number of columns, but it is scalable in the number of rows. For both of these distributed operations, parallelization across rows before aggregation seems to be optimal: each worker node is busy computing partial results for some partitions.

Eigenvectors

In Spark MLlib, finding eigenvectors is possible only for symmetric matrices with real values. Calculation is done with Arnoldi iterations, provided by ARPACK[1]. The materialized matrix itself is usually not necessary, only the callback function that computes its multiplication by a vector. This allows for distributed processing where multiplication operation is performed in parallel by worker nodes and the results from each partition can be associatively aggregated on the driver node. However, since the iterations are performed locally on the driver node (all intermediate data is stored locally), this method still requires space of $4n(k+1)$ double values on a single node, where n is the number of columns and k is the number of leading eigenvalues that was requested to be computed. This computation is scalable in the number of rows, but not in the number of columns.

Singular Value Decomposition

Let's assume we want to find SVD decomposition

$$M = U \Sigma V^\dagger$$

where $U_{m\times k}$ and $V_{n\times k}$ are orthogonal matrices and $\Sigma_{k\times k}$ is a diagonal matrix. To do this, we find eigenvectors of

$$M^\dagger M = V \Sigma^2 V^\dagger$$

yielding V and Σ, of dimensions $n \times k$ and $k \times k$ respectively, which allow them to completely fit in memory for smaller n and k values. Then we can also compute U of dimension $m \times k$, which typically will not fit in memory on a single machine and needs to be distributed, since m is the dimension representing the number of rows in the *RowMatrix* representation.

Depending on how large n and k (number of values to keep) are, singular value decomposition will be calculated differently. If n is small (less than 100), or if k is larger than $n/2$ and n is not bigger than 15,000 threshold, let us call this ***condition for local eigenvalue computation***, then we compute Gram matrix $G = M^\dagger M$ in distributed fashion once, as shown above, and obtain a local matrix G. Then if k is less than $n/3$, we compute its eigenvectors locally, as explained above, using ARPACK. If k is greater than or equal to $n/3$, we use *BREEZE* method to compute SVD directly using Gram matrix multiplication callback function[2].

[1] More information on Arnoldi iterations and ARPACK software can be found in [8, 7, 6].

[2] *BREEZE* uses LAPACK for this operation

If on the other hand, if ***condition for local eigenvalue computation*** doesn't hold, then we avoid materializing Gram matrix, and instead, iteratively compute eigenvalues of $M^{\dagger}M$, in each iteration[1] we compute

$$v \mapsto M^{\dagger}Mv$$

in parallel, transmitting data with size only on the order of $O(n)$. As with eigenvalue search and Gram matrix computation, this will require a new map/aggregate job and the associated communication costs in each iteration.

SVD computation in Spark MLlib scales (distributes computation) well with number of rows (m), but not as well with the number of columns (n): one must be aware that a vector of size n must fit in memory. For example, when SVD is applied to a term-document matrix, number of columns (typically number of terms) should be trimmed down to something manageable on a single machine - e.g. 10K[2].

Covariance Matrix

Treating each row vector r_i as an observation with m values, we can use Gram matrix to calculate covariances. Gram matrix is calculated in parallel and is made available on the driver node. In a distributed way, we also compute means for each column:

$$\mu_i = \sum_k c_i^k / m$$

and use the formula

$$Cov(X,Y) = E[XY] - E[X]E[Y]$$

to compute covariances without care for precision (no BLAS here). This makes Spark covariance computation for *RowMatrix* numerically unstable and MLlib users should be aware of this.

Principal Component Analysis

Spark MLlib calculates PCA by computing SVD for covariance matrix locally on the driver node[3] and taking the desired number of singular values. Here we inherit numerical instability of covariance matrix computation operation, which is described above. Also, here we cannot arbitrarily scale with the number of dimensions because we have to perform this on a single node. Principal component analysis and singular value decomposition are the main methods for dimensionality reduction in Spark MLlib.

Pair-wise Cosign Similarities

This operation calculates cosine similarities between all pairs of columns. It performs an algorithm DIMSUM (dimension-independent similarity computation using MapReduce). The algorithm effectively performs computations on sampled data[4].

Local Matrix Multiplication

Spark MLlib support transformation of a given distributed matrix M, by multiplication with a local (not-distributed) matrix A on the right side: MA. Matrix A is serialized and broadcast to all worker nodes and each portion of the resulting matrix MA is calculated by worker nodes independently, resulting in a new *RowMatrix*.

Distributed Matrix Multiplication

Distributed matrix multiplication is not available for *RowMatrix*, but it is implemented for *BlockMatrix* form. Ideally, we want this to be scalable not only in the number of rows, but also in the number of columns. This requires us to partition in both dimensions, which is achieved with *BlockMatrix*[5].

This multiplication between two block matrices M and N requires that they have compatible partitioning schemes: number of columns in blocks of M must match the number of rows in blocks of N. Co-grouping blocks of A and B by (i_A, j_B, k), where i_A are row indices of blocks in A, j_B are column indices of blocks in B, k runs through column indices of A (same as row indices of B), and reducing over all k to common key (i_A, j_B), we come up with blocks of AB. Co-grouping requires a shuffle step[6], where each target block (i_A, j_B) of AB receives a row of blocks from A and a column of blocks from B: so in total from about $O(numColBlocks_A + numRowBlocks_B)$ partitions, which makes the total communication size during

[1]Arnoldi iteration method. Local computations are handled by ARPACK, we only provide a callback function that will compute $v \mapsto M^{\dagger}Mv$ in parallel and return computed multiplication back to eigenvector computation routine.

[2]example of this is shown in book *Advanced Analytics with Spark*[21]

[3]Breeze over LAPACK

[4]Description of DIMSUM algorithm can be found here: http://arxiv.org/abs/1206.2082

[5]Current published version of Spark MLlib (1.3) provides only block partitioned version, in future it might provide matrices with other partitioning, such as cyclic or hybrid.

[6]worst case is all-to-all communication

shuffle $O(numColBlocks_A \times numRowBlocks_B \times (numColBlocks_A + numRowBlocks_B) \times colsPerBlock_A \times rowsPerBlock_B$. This further simplifies to

$$O(2n_A^3/colsPerBlock_A)$$

double values transmitted between nodes[1].

This shows the trade-off between communication size and computation size on nodes: we can decrease communication size by increasing the number of columns per block in A ($colsPerBlock_A$), and in doing so increase the computation and memory requirements on each node: extreme case is a non-partitioned matrix, where all computation is performed on a single node. On the other hand, we can allocate smaller computation work on individual nodes by decreasing the block size (in this case columns per block in A), allowing for bigger communication size during shuffle phase.

3.2 Convex Optimization

Many ML problems can be viewed as convex optimization problems. MLlib provides parallel implementations for two basic optimization algorithms: SGD and L-BFGS[10, 11].

Package `optimization`[2] contains a trait *Optimizer*, which requires only one method

$$optimize(RDD[(Double, Vector)], Vector) : Vector$$

This method takes labeled input data points ($RDD[(Double, Vector)]$) and an initial weights vector, and performs an optimization yielding the resulting weights vector.

Out of the box MLlib provides two optimizers: Stochastic Gradient Descent and L-BFGS.

Stochastic Gradient Descent

For SGD, we provide a method that calculates gradients (trait *Gradient*) and a function that updates the weights (trait *Updater*), possibly adding gradient contributions from some regularization[3]. Let us look at *Gradient* and *Updater* in more detail:

[1]it is likely that there will be multiple partitions will be allocated per node, so this is worst case scenario, where each partition is computed on a distinct worker node. Also it includes the blocks that this partition already has - its own block, but this is not important asymptotically.

[2]`org.apache.spark.mllib.optimization`

[3]In future, *Updater* will be split into *Regularizer* and *StepUpdater*

Gradient: given a labeled data point and weight vector, calculate gradient and weight vector update. MLlib provides 3 implementations: Hinge, Least Squares, Logistic.

Updater: given a calculated loss gradient, calculates the next iteration of model vector, by adding regularization gradient and adjusting for step size. Out of the box, MLlib comes with implementations with no-regularization or regularizations based on L_1, L_2.

SGD computation follows the iterative execution model described earlier[4]. We start with a guess for a weight vector. Then, in each iteration, we broadcast that current weights vector along with the gradient computation. Using this information, we compute gradient contributions from data points in each partition. We perform this calculation on a subset of available data in partition, facilitated by RDD sampling functionality[5]. At the end of the iteration, partial gradients are aggregated on the driver node and an *Updater* is applied, where additional regularizer gradients are added, thus yielding a new estimate for the weights vector.

Iteration termination condition is simply the number of iterations, with default value of 100.

L-BFGS

L-BFGS is a quasi-Newton method that in each iteration uses second-order adjustment to find direction towards improved guess and hence converges faster than SGD, which only uses first-order approximation[10]. Inverse Hessian (matrix of second-order partial derivatives) is not materialized, but instead is approximated using previous gradient evaluations.

Breeze drives iterations of the calculation, and through callback functions interacts with Spark when we need to calculate loss value and gradient for a given weight vector in parallel.

It is preferred to use L-BFGS because of its faster convergence, however some current (1.3) *Updater* implementations that were originally designed for SGD are not suitable for L-BFGS (e.g. L_1) [1].

[4]see section *Execution Model*

[5]RDD.sample(...) method

3.3 Generalized Linear Model

Generalized linear algorithms can be solved as convex optimization problems. We can pick an optimizer, provide it gradient computation method and regularization function, and solving that optimization problem, we get solutions to regression problems.

Algorithm	Loss	Regularizer
Linear Regression	Least Squares	None
Logistic Regression	Logistic	L_2
Ridge Regression	Least Squares	L_2
Lasso Regression	Least Squares	L_1
Linear SVM	Hinge	L_2

Figure 3: Generalized Linear Algorithms

Combinations of the provided gradients and regularizations automatically give us some useful algorithms from the *Generalized Linear Algorithms* family in the form of stochastic gradient descent (Fig 3).

Spark MLlib 1.3 provides SGD implementations of these algorithms. For logistic regression, it also provides L-BFGS implementation.

3.4 Classification Algorithms

Let us look at available classification algorithms. Most of them rely on generalized linear algorithms, as explained earlier. Some rely on tree-based methods and we will discuss them separately in section *Tree methods*.

In package `classification`[1] we have a trait *ClassificationModel* which predicts a class for a given vector. There are three implementations that are trained with *Logistic Regression, Linear SVM* or *Naive Bayes*. As we have seen earlier, Logistic Regression and Linear SVM are members of generalized linear algorithms family. MLlib provides their implementations with SGD. In addition to that, we also have L-BFGS implementation for logistic regression.

These generalized linear model classifications support only 2-class classification, which is a major drawback for users. It is possible to reduce the problem of multiclass classification to combinations of binary classifiers with schemes such as one-vs-all, or one-vs-one, or others. Work is currently under way in ML pipeline API[2] to introduce generic multiclass to binary reduction schemes. Tree-based methods and Naive Bayes support multiclass classification.

Provided Naive Bayes algorithm implementation is for a Multinomial Naive Bayes[9] model with probability of a vector $x = (t_j)$ to belong to class C_i equal to:

$$p(C_i|x) = p(C_i)\prod_j p(t_j|C_i)$$

We compute $p(C_i)$ and $p(t_j|C_i)$ in parallel, by scanning through all data points and collecting totals, and thus deducing the frequencies, which are used as probability estimates. Transferred data is on the order of $O(m \times P)$, where m is the number of features and P is the number of partitions. This algorithm implementation treats feature values as counts and requires that all values of all features are non-negative.

MLlib classification algorithms support only linear kernels, which is a considerable limitation for many applications. Primary reason for this is that non-linear kernels are difficult to parallelize. This can be (only) partially mitigated by introducing feature interactions. Also, as mentioned earlier, a rich set of multi-class decision-tree based classification algorithms is also provided in Spark MLlib.

3.5 Regression Algorithms

Most regression algorithms[3], with the exception of two, come from the above-mentioned generalized linear models and all are implemented with SGD: *Linear Regression, Lasso/Ridge Regressions.* Within the same package, an additional regression algorithm is provided: *Isotonic Regression.* Outside of this package, we have decision tree-based regression algorithms.

MLlib isotonic regression is implemented using parallelized pool adjacent violators (PAV) algorithm and is available only for one-dimensional feature vectors. Sequential PAV implementation and parallelization are described in [15, 16].

3.6 Tree Methods

Spark MLlib provides several tree-based classification and regression algorithms in package `tree`[4], which is probably one of the more complicated algorithm packages in core MLlib today and also could be one the most useful for classification and regression problems. Two main algorithms are ensemble algorithms

[1]`org.apache.spark.mllib.classification`

[2]ML pipeline API will be discussed in section *ML Pipeline API*

[3]Package `org.apache.spark.mllib.regression`

[4]`org.apache.spark.mllib.tree`

that are composed of one or more simple decision trees: *Random Forest* and *Gradient Boosted Trees.*

Tree algorithms are driven through configuration objects of class *Strategy* or *BoostedStrategy*, where users can specify numerous algorithm parameters allowing for a great deal of flexibility.

For example, there are abstractions *Impurity* and *Loss* associated with tree algorithms. Trait *Impurity* captures the score of class purity used in calculation of information gain, for example, during tree growth stage when we perform decision node splits. Three implementations are provided which are based on variance, Gini, and entropy. Trait *Loss* is used to communication loss function in Gradient Boosted Trees, and has three implementations: *LogLoss*, *AbsoluteError* and *SquaredError*. Then, there are abstractions for the type of a feature (categorical or continuous), quantile strategy, ensemble combining strategy, algorithm type (classification or regression), and so on - a modular design allowing many component combinations.

Base Decision Tree Algorithm

A base single decision tree algorithm functionality is implemented in an object *DecisionTree*. It provides a common shared implementation of fitting one decision tree that is used by both *Random Forest* and *Gradient Boosted Trees* algorithms.

Two important methods here are: *findSplitsBins*, which picks splits for all features taking into account the type of features (e.g. categorical or continuous) and the task at hand (regression, binary classification, multiclass classification) and *findBestSplits*, which finds the best splits for given nodes. Operation *findSplitsBins* works with a sample of input data for continuous value features and safely handles categorical data. This operation is scalable. With *findBestSplits* we pick best splits at each tree node in parallel.

Random Forests

Random Forest is an ensemble algorithm that is composed of one or more simple Decision Trees[17], where prediction is based on majority vote or an average value of predictions of all decision trees in the ensemble. It can be used for both classification and regression.

We start by picking sub-samples and feature sets for each tree in the ensemble. Then we distribute the recursive computation of best splits in all trees. To reduce computation needed, we perform splits on binned data where possible. At the end of this distributed tree construction, we have an ensemble of trees that is ready to do predictions.

When we increase the number of trees in random forest (e.g. to reduce variance of our model), or just add more training data, we are able to evenly distribute the computation load across all cluster nodes, which makes it a good choice for large problems.

Gradient Boosted Trees

Gradient Boosted Trees give us a different way of combining individual decision trees. We build up a sequence of smaller trees with boosting method: each new tree contributes a little bit of new information to the final decision, that previous trees misclassified[18]. This is done by increasing the weight of those misclassified data points at each iteration (*boosting* of some weights).

With gradient boosted trees, each tree is smaller than in random forest, but we need to build up trees sequentially. Calculation of loss function and reweighting is done in parallel, which helps with parallelization of each individual iteration.

3.7 Clustering Algorithms

MLlib provides four clustering algorithms: *K-means*, *Latent Dirichlet Allocation*, *Gaussian Mixture*, and *Power Iteration Clustering.* K-means and Gaussian mixture algorithms find more general usages, while power iteration clustering and latent Dirichlet allocation are associated with graph and document data respectively.

K-means

In *K-means* algorithms, we pick some initial set of cluster centers and in each iteration we compute a better set of centers. This is done through relabeling the data points by picking the closest of the current centers and picking class means of data points, based on the new class assignments, as our new centers. Error function that we care about in K-means algorithms is *Within Set Sum of Squared Error (WSSSE).*

Algorithm runs in the typical [broadcast, compute partials, collect, ...] pattern that was described in section *Iterations*, making this computation trivially parallelizable and scalable[1].

Users can control parameters such as number of iterations, convergence threshold, number of times to re-run algorithm. The last parameter is important because there is no guarantee of a single minimum.

Gaussian Mixture

Generalization of K-means algorithm, *Gaussian Mixture Model*, is implemented in Spark MLlib. Gaussian Mixture Model objects of class *GaussianMixtureModel* (GMM) are encoded by one array of weights and one array of multivariate Gaussian objects, which themselves are pairs of two other items: one double representing mean, and one local matrix representing covariances.

Gaussian Mixture algorithms tries to find the set of Gaussian distributions of sub-populations within the main population sample that we are working on. It does so with *Expectation Maximization* (EM) algorithm[19], where at each step we try to come up with a better estimated set of Gaussians.

Input data is in the form of RDD of local vectors. First step is to convert local vectors into Breeze vectors and cache them in memory. This is the data that our iterations will run against. We pick a small random sample of data for each cluster and calculate means and the covariance matrix for chosen clusters - this will serve as our initial guess.

In Expectation Maximization technique, each iteration consists of two steps: step (E) calculate log-likelihood with current model parameters, step (M) find the parameters maximizing likelihood.

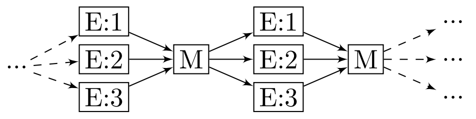

Figure 4: *Expectation Maximization* iterations

The first step is performed in parallel, as the current model is broadcast to all nodes, where cluster contributions for each input point will be computed into partial results. Partial results are *expectation sums*, data structures with size on the order of kM^2, where k is the number of clusters and M is the number of features. These partial expectation sums are be aggregated on the driver node. Second step is performed completely on the driver node, where these expectation sums are used to create a new estimate for distributions[2].

Since step (E) is completely distributed and it is the only step where we need to look at all of the data points, this algorithm computation is scalable with the number of data points.

However, because we have to operate on expectations sums on each node and their sizes are on the order of $O(kM^2)$ (contains all covariance information for all clusters), this implementation is not scalable with the number of clusters and number of features, though this limitation most likely won't come up often as an issue in practice.

Power Iteration Clustering

Power Iteration Clustering[22] will cluster indexed data points v_i with respect to provided symmetric similarity function $s(i, j)$. One way to view input to this clustering is as (undirected) graph vertices with similarity provided as edge weights. Then similarity function, the main user input to this algorithm, can be seen as an adjacency matrix A with

$$A_{ij} = s(v_i, v_j)$$

which is transformed into a random walk normalized Laplacian W:

$$W = D^{-1}A$$

where degree matrix D is a diagonal matrix with

$$D_{ii} = \Sigma_j A_{ij}$$

Approximation of eigenvector corresponding to the largest eigenvalue will be used as the intermediate vertex weights vector $v^\intercal$. We find the largest eigenvalues of this matrix W through an iterative process (power iteration), where each new estimate for an eigenvector is a multiplication of previous estimate v by matrix W (c is a normalizing factor for stability):

$$v^\intercal \to cWv^\intercal$$

As one can notice, parallelization is trivially achievable here. Partial products can be calculated in distributed fashion and aggregated to produce new weights $v^\intercal$.

It is worth noting, that in Spark MLlib, this is actually done with GraphX vertex messaging functionality. This process in the end amounts to the same processing as is done in distributed matrix-vector multiplication.

[1]MLlib also has a non-parallel K-means version. By default, a parallel version described here is used.

[2]uses BLAS *syr* function.

Convergence condition used is on the rate of change in differences between successive weight vector estimates $v^\intercal$ - algorithm can be stopped when *acceleration* of changes is small enough[22].

Multiple iterations of this algorithm lead to weight assignments $v^\intercal$ that are then passed into K-means algorithm for separation into clusters, also a parallelized procedure.

Latent Dirichlet Allocation

Just as we used iterative Expectation Maximization technique to find the best fitting mixture of Gaussians, we can find best fits for topics in documents, when looking at both documents and topics as bag-of-words objects.

In Spark MLlib, search for the best fitting topic model with Latent Dirichlet Allocation is reduced to an optimization problem. Two implementations are provided - one is based on Expectation Maximization[24], which uses GraphX, and the other is an online algorithm that operates on samples of data and performs online variational Bayes LDA[23].

The first optimization algorithm creates a bipartite graph representing term-document relationships. On each vertex we store an array that contains information about topic counts (array of doubles). It starts with random soft assignments of tokens to topics and repartitioning of the graph, so that edges are grouped by document. Then we perform typical expectation maximization iterations, where at the end of each iteration we have updated topic counts on each vertex.

The online optimization algorithm processes a small sample of the corpus on each iteration, and updates the term-topic distribution adaptively for the terms appearing in that sample. As input data, it takes an RDD of document IDs along with a vector of terms for that document (bag of words representation) and doesn't modify it for processing, unlike the first graph-based EM implementation.

Both of the algorithms parallelize well with the number of documents. As for the number of terms, it appears that the second implementation has to keep data structures with size on the order of the number of terms in memory and thus can present a scalability bottleneck.

3.8 Recommendation Algorithms

Currently, Spark MLlib provides only one recommendation algorithm - *Alternating Least Squares* (ALS) - it is based on the *collaborative filtering* technique, where two sides (factors) of recommendation (e.g. users and products) are hypothetically connected through a set of unobserved (latent) factors[20].

If $A = A_{ij}$ is such a matrix with each row representing some user's purchases and each column representing some product purchases, then we propose that it is a product of two matrices X and $Y^\dagger$, where both X and Y have k columns:

$$A = XY^\dagger$$

This means that somehow we can capture information about users and products individually in a vector in some k-dimensional space. It could be, for instance, a preference for some genre of music, or some other topic which is not openly given in input data. Then generated recommendations work as similarity measure between users and products in this discovered basis, e.g. inner product (cosine similarity):

$$Rec(user_a, prod_b) = \Sigma_i X_{ai} Y_{bi}$$

Typically, input data to this algorithm is a large and sparse matrix. For example, if we look at users and their purchased or rated products, users could be represented by rows and products by columns: each rating (explicit information - user rated the product) or each purchase (implicit information - we assume that user liked the product to some degree) is represented by an entry in this sparse matrix.

When we supply an input RDD of all entries in this matrix A (e.g. *ratings*), we iterate a fixed number of times, by alternating computation of X from A and Y, and Y from A and X, using the familiar minimum least squares computation:

$$X = AY(Y^\dagger Y)^{-1}$$

and

$$Y^\dagger = (X^\dagger X)^{-1} X^\dagger A$$

making this training procedure scalable and well-parallelized.

For the final computed model, Spark MLlib implementation creates an object of type *MatrixFactorizationModel*, which is a model that represents the result of matrix factorization $XY^\dagger$.

Provided algorithm can be trained with implicit ratings, e.g. user X purchased product A, or explicit ratings, e.g. user X rated product A at 4.5/5.0. Both versions run similar iterations under the hood.

One scalability limitations are that the current implementation supports only IDs (number of rows/-columns) of type 32-bit integers. Another drawback is that the model is large, it is a distributed matrix of size on the order of the number of users and products. This might be a concern for real-time analytics applications, because in order to produce recommendations, one will need to have this model in memory on all nodes and run a Spark job to come up with recommendations of products for a given user, or users for a given product.

3.9 Frequent Pattern Mining

Spark MLlib provides a parallel implementation an algorithm for mining frequent item-sets: *FP-growth*[25, 26]. Input data comes in as an RDD of arrays of items (arbitrary type of items). Initially, we calculate individual item frequencies by simply using a *reduceByKey* operation. Then we utilize a specialized suffix-tree data structure, FP-Tree, to run through the elements of the RDD and build all association frequencies. This ends up being performed completely in parallel. The only possible bottleneck is the size of the tree, which must completely fit in memory on any individual node.

3.10 Feature Engineering

Besides standard ML algorithms, we are also provided several feature engineering tools: Word2Vec, TF/IDF, feature normalizer/scaler, feature selection. Standard scaler will scale and shift data to make its mean zero and standard deviation of unity. Normalizer will scale the values of a vector to make its norm unity. Chi-squared feature selector will perform Pearson's test for independence between categorical features and label, and select only the features closest to the label. All these three functionalities are trivially parallelized. Let us look at Word2Vec and TF/IDF.

Word2Vec

Given a set of documents, this procedure will create a vector representation of each occurring term, in such a way that "similar" terms will be closer in this vector space. The concept of similarity between terms here is when another term could replace this term given the context, something similar to the concept of a synonym.

Spark MLlib implementation uses skip-gram model with hierarchical softmax method. Algorithm will perform a series of iterations, where in each iteration we will go through all sentences and update two arrays of size equal to the vocabulary size (number of all terms).

During processing of each sentence, we go through each word and for each word we will need to loop over $log(|Vocabulary|)$ elements in the Huffman encoding (tree) of all terms. This is the hierarchical softmax portion used to calculate conditional term probabilities.

Assuming each node can hold two arrays and a tree containing data for each term in vocabulary, then the rest is easy parallelized along with input sentences distribution.

TF/IDF

Term frequency-inverse document frequency is calculated in two parts. First we need to compute term frequencies and then we also need to compute inverse document frequencies. In Spark MLlib, hashing is used to generate a unique id for a term. This helps with parallelization, as no single map of term indices is necessary, but this comes at a cost of hash collisions. Term frequencies are calculated trivially with a single pass over sentences, while inverse document frequencies are calculated in two passes: one to count occurrences, and second to normalize them.

3.11 Other Functionality

Let us just mention several useful functionalities provided in Spark MLlib that are naturally parallelized or do not require parallelization.

Basic Statistics

Summary statistics can obviously be calculated in parallel. *Kernel density* estimation is possible on given limited set of points for which it needs to be computed. Here we simply aggregate contributions from all elements in the dataset at these points.

It is possible to generate RDDs with random numbers from certain distribution families, but it is not clear how to properly align it with existing RDD data. *Random RDD* creation basically works by shipping a specific random number generator to partitions and modifying its seed using partition id.

Pearson and Spearman *correlations* are provided: Pearson correlation is computed from covariance matrix generated from rows of a RowMatrix of incoming data, while in Spearman correlation computation, we first construct grouped ranks by iterating over all values, and then calculate Pearson correlation.

Stratified sampling is possible by providing desired class distributions and going through each record deciding whether to include it in the sample or not. Exact version of this sampling requires more processing, but is still parallelized.

Chi-squared independence test is parallelized - only a single pass through distributed input data is necessary. Intermediate data will be aggregated to construct contingency tables used in further processing.

Evaluation Metrics

Several evaluation metrics are available for models: BinaryClassificationMetrics, MulticlassMetrics, RegressionMetrics, RankingMetrics. All of metrics functions are naturally parallel in nature, since they use some computed model to calculate predictions for each point in the data set in parallel.

PMML Export

Spark MLlib provides a trait `PMMLExportable` to designate models that can be exported to PMML [1]. Implementation of PMML export is a very recent development and is provided only for a basic set of MLlib models. It is currently under development to include more models. This is performed locally on the driver node.

4 Streaming Support

One of the desired qualities in a machine learning library is ability to learn from streaming data.

One special case is models that are built using SGD. Whenever model weights are updated with stochastic gradient descent, we can use streaming data to train this model continuously. Spark MLlib provides implementation of this sort for binary classification: `StreamingLogisticRegressionWithSGD`.

For streaming clustering, we have an algorithm `StreamingKMeans` available in the library.

This is where the list of streaming ML algorithms provided by default ends.

5 spark.ml

Support for high-level use of MLlib is provided with `spark.ml`, which organizes machine learning work with *ML pipelines*. By its meta-processing nature, this processing itself does not need to be parallelized, while individual stages of the pipeline will have their own independent parallelization.

Dataset

Spark ML operates on a *ML Dataset*, a concept of data set which is built on top of Spark SQL *DataFrame* structures that can use custom binary formats to store columnar data more efficiently. DataFrame performance received a lot of attention from Spark developers, making newer implementations CPU-efficient, cache-aware.

Pipelines

Two main concepts in spark.ml are ML pipeline stateless processing stage types: *Transformers* and *Estimators*. Transformer stages transform DataFrames into new DataFrames. For instance, some transformer stage can apply a model to data and calculate predictions. Estimator stages take DataFrames and create Transformers. An example of this would be a learning algorithm that produces a model that can make predictions on new input data. A sequence of Transformers and Estimators forms a *Pipeline*.

6 Comparisons

Let us briefly discuss how Spark MLlib is similar to or different from some other existing machine learning libraries[2]. We can analyze and compare alternatives using the following four axis: availability of algorithms, parallelization and distribution of workload, visualization capabilities, and flexibility.

[1]Quoting from Wikipedia, Predictive Model Markup Language (PMML) is an XML-based file format developed by the Data Mining Group to provide a way for applications to describe and exchange models produced by data mining and machine learning algorithms.

[2]A comprehensive review and comparison of all machine learning frameworks and libraries is not provided here. Instead only 4 major alternatives are discussed

More mature ML libraries and frameworks, such as R, WEKA and scikit-learn, provide a multitude of algorithms, while Spark MLlib is currently not as rich. For example, it doesn't provide an out-of-the box ready-to-use multiclass SVM constructions.

Spark MLlib is still in early stages and it is not yet a common tool for data scientists today. This is in comparison to established leaders such as R and python-based libraries.

Compared to other libraries, Spark MLlib offers better parallelization - almost all of the algorithms that are implemented, are implemented so that work gets distributed across the cluster nodes. Its machine learning algorithms scale linearly with the amount of data. In contrast, execution models of R and python-based libraries rely on having the dataset in memory on a single machine[1].

One exception to this would be *Apache Mahout* project, which is a library on machine learning algorithms that works on top of Apache Hadoop (MapReduce). With the adoption of Spark and shown inefficiencies of MapReduce framework when dealing with iterative algorithms, it appears that Mahout algorithms will have to migrate to Spark framework, or somehow will be superseded by Spark MLlib.

MLlib	R	scikit-learn	WEKA	Mahout	
	⋆	⋆	⋆		Number of algorithms
⋆				⋆	Algorithm parallelization
	⋆	⋆	⋆		Visualization functionality
⋆		⋆	⋆		Data processing flexibility
⋆	⋆	⋆			Ease of prototyping
⋆	⋆	⋆	⋆		Execution Speed
	⋆	⋆	⋆	⋆	Project Maturity

Figure 5: Machine Learning framework evaluations by category. Better framework/library is subjectively chosen by author for each category and marked with a star.

Strong support for visualization is typically provided in established analytics frameworks, while in Spark MLlib Scala version it is only nascent if at all exists[2]. Python-based version (pySpark) does allow for iPython like notebook. One of the projects that aims to bring this interface to Scala is *Apache Zeppelin.*

With WEKA users can interact through GUI and command-line interface, or through Java API. GUI does not offer as rich graphics experience as R and Python, but having Java API allows for some automated and more flexible handling of data. While WEKA provides language support, typically writing Java code is not as straightforward as R, Python or Scala. This makes WEKA less desirable for quick prototyping and exploratory analysis.

One advantage of Spark MLlib probably lies in the fact that Scala is a full-fledged programming language, which allows for more flexible processing of data. This is especially important when user have to perform non-trivial logic when pre-process data and engineering features, where language expressiveness helps users achieve specific functionality faster. R provides quite a bit of programming functionality, but it is not as flexible as an actual programming language. Python-based library benefits from a complete language support and a rich set of libraries written in Python.

7 Conclusions

Apache Spark MLlib library is a young project with a lot of active development. It offers parallel implementations of many important algorithms: regression, classification, clustering, recommendations, topic modeling, frequent pattern mining.

Distributed convex optimization implementations of stochastic gradient descent and L-BFGS methods are provided in Spark MLlib and used internally for linear model fits.

At lower level, it offers stable linear algebra computations that can be run efficiently on native BLAS, providing distributed matrix computations: multiplication, eigenvalues for symmetric matrices, SVD decomposition. On higher level, it provides ML pipelines API which make it possible to compose lower level functionality.

In addition to machine learning algorithms and higher level framework API, it also provides a lot of useful utility functionalities for manipulating data and designing features.

[1]There has been work on integrating R and Spark in a project called *SparkR.* With this tool, R command `lapply` started on the driver node will be distributed to to worker nodes, each of which running R locally, via Spark's `map` transformation. However, most ML algorithms that are implemented in independent R packages are not automatically parallelized with SparkR.

[2]A project *breeze-viz* is currently under development to allow easy creation of visualizations similar to R and python.

References

[1] Machine Learning Library (MLlib) Guide, *https://spark.apache.org/docs/latest/mllib-optimization.html*

[2] Breeze: numerical processing library for Scala, *https://github.com/scalanlp/breeze*

[3] jblas: linear algebra for Java, *http://jblas.org/*

[4] netlib: collection of mathematical software, *http://www.netlib.org/*

[5] Jeffrey Dean and Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, Google, Inc 2004

[6] W.E. Arnoldi, *The principle of minimized iterations in the solution of the matrix eigenvalue problem* Quart. Appl. Math., 9, pp. 17-29 1951

[7] R.B. Lehoucq, D.C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods* SIAM, Philadelphia, 1989

[8] D.C. Sorensen, *Implicit Application of Polynomial Filters in a k-step Arnoldi method.* SIAM J. Matrix Anal. Appl., 13:357-385, 1992

[9] Naive Bayes in Text Classification *http://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classification-1.html*

[10] J. Nocedal *Updating Quasi-Newton Matrices with Limited Storage* Mathematics of Computation 35, pp. 773-782. 1980

[11] D.C. Liu and J. Nocedal *On the Limited mem Method for Large Scale Optimization* Mathematical Programming B, 45, 3, pp. 503-528. 1989

[12] Snyder, Lawrence, *Type architectures, shared memory, and the corollary of modest potential*, Ann. Rev. Comput. Sci 289-317, 1986

[13] Valiant, L. G., *A bridging model for parallel computation*, Communications of the ACM 33(8):103-111, 1990

[14] Culler, D. et al, *LogP: Towards a Realistic Model of Parallel Computation*, Proceedings of IV ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, May 1993

[15] Tibshirani, Ryan J., Holger Hoefling, and Robert Tibshirani, *Nearly-isotonic regression*, Technometrics 53.1 (2011): 54-61. 2011

[16] Kearsley, Anthony J., Richard A. Tapia, and Michael W. Trosset, *An approach to parallelizing isotonic regression.*, Applied Mathematics and Parallel Computing. Physica-Verlag HD, 1996. 141-147. 1996

[17] Breiman, Leo, *Random Forests*, Machine Learning 45 (1): 5-32 2001

[18] Friedman, J. H., *Greedy Function Approximation: A Gradient Boosting Machine.*, 1999

[19] Dempster, A.P.; Laird, N.M.; Rubin, D.B., *Maximum Likelihood from Incomplete Data via the EM Algorithm.*, Journal of the Royal Statistical Society, Series B 39 (1): 1-38 1977

[20] Koren, Yehuda and Bell, Robert and Volinsky, Chris, *Matrix Factorization Techniques for Recommender Systems*, IEEE Computer Society Press, J. Computer, 42: 30-37 2009

[21] Sandy Ryza, Uri Laserson, Sean Owen, Josh Wills, *Advanced Analytics with Spark*, O'Reilly, 2015

[22] Frank Lin, William W. Cohen, *Power Iteration Clustering*, Proceedings of the 27th International Conference on Machine Learning, Haifa, Israel, 2010

[23] Hoffman, Blei and Bach, *Online Learning for Latent Dirichlet Allocation*, NIPS, 2010

[24] Asuncion, Welling, Smyth, and Teh. *On Smoothing and Inference for Topic Models*, UAI, 2009

[25] Han et al., *Mining frequent patterns without candidate generation*, Proceedings of the 2000 ACM SIGMOD international conference on Management of data, 2000

[26] Li, Haoyuan and Wang, Yi and Zhang, Dong and Zhang, Ming and Chang, Edward Y., *Pfp: Parallel Fp-growth for Query Recommendation*, Proceedings of the 2008 ACM Conference on Recommender Systems, 2008