SPARK'S STANDARD LIBRARIES

ARUN DEBRAY
AUGUST 15, 2014

CONTENTS

1. MLLIB AND THE SINGULAR VALUE DECOMPOSITION

This talk was given by Reza Zadeh, at Stanford's Institute for Computational and Mathematical Engineering (ICME). He contributes to the machine-learning code in Spark, and is one of the over fifty contributors to MLlib.

MLlib came out of AMPLab, in UC Berkeley, and has been shipped with Spark since September 2013. It has a lot of standard machine learning algorithms already implemented, e.g. logitic regression, naïve Bayes, linear SVM, k-means, decision trees, and so on. And there are a lot of ways to play with optimizations to these. Since MLlib is so recent, more algorithms are still being added.

Example Invocations.

```
// k-means clustering
// load and parse the data
val data = sc.textFile("kmeans_data.txt")
val parsedData = data.map(_.sploit(' ').map(_.toDouble)).cache()

// cluster into 2 clusters
val clusters = KMeans.train(parsedData, 2, numIterations = 20)
```

```
// compute principal components of a row matrix
// this depends on the size of the matrix (remember, this is _big_ data)
val points: RDD[Vector] = ...
// this row matrix class is the most mature
val mat = RowRDDMatrix(points)
val pc = mat.computePrincipalComponents(20)

// project points to a low-dimensional space
val projected = map.multiply(pc).rows

// then, train a -means model on the projected data
val model = KMeans.train(projected, 10)
```

Alternating least-squares is one of the oldest and most optimized algorithms in MLlib.

```
// load and parse the data
// ...

// Build the recommendation model using ALS (alternating least-squares)
val model = ALS.train(ratings, 1, 20, 0.01)

val usersProducts = ratings.map { case Rating(user, produt, rate) => user, product }
```

Notice that all of these algorithms take advantage of the fact that MLlib can iterate over data repeatedly and quickly.

Most of the algorithms in MLlib are implemented by reducing to convex optimization (as in the least-squares implementation, below) or a singular value decomposition.

```
data = spark.textFile(...).map(readPoint).cache()

w = numpy.random.rand(D)

for i in range(iterations):
    gradient = data.map(lambda p:
        (1 / (1 + exp(-p.y * w.dot(p.x)))) * p.y * p.x
    ).reduce(lambda a, v: a + b)
    w = -gradient
```

Notice that this, or a PageRank example based on an SVD, would be a lot less reasonable in MapReduce; lots of things need to be read and written to/from memory. However, notice the calls to cache the data: this means that the lists of neighbors in the network are kept in RAM. Furthermore, redundant work is avoided in hashing by using partitioning. For example, one could assign all URLs for a given domain (in the PageRank example set) to a single machine, to reduce the amount of communication between different computer.

```
links = spark.textFile(...) # and so on: parse the data

ranks = links.mapValues(lambda v: 1.0) # RDD of (id, rank)

for i in range(ITERATIONS):
    ranks = links.join)ranks).flatMap(
        lambda (id, (links, rank)): ...
    ).reduceByKey(lambda a, b: a + b)
```

The point is, this quick algorithm is much more efficient than the corresponding one in MapReduce.

Singular Value Decomposition. There are two cases for the SVD: the first is where the matrix is tall and skinny, and the other where it's more square. Thee lead to two different algorithms, but `computeSVD` takes care of deciding which one to call.

For a tall and skinny $m \times n$ matrix so ($m \gg n$), $A^{\mathrm{T}}A$ is much smaller than A (it's $n \times n$), and dense (and often, small enough that a single computer can take care of it). But it also holds all dot products between columns of A, so $A = U\Sigma V^{\mathrm{T}}$, and thus $A^{\mathrm{T}}A = V\Sigma^2 V^{\mathrm{T}}$, so we can retrieve V and Σ. Then, they can be inverted easily, so one more map-reduce retrieves U.

This algorithm doesn't work as well when the matrix is ill-conditioned (i.e. it might not be the most incredibly precise in some contexts). This is OK for many contexts, especially Web ones, but the developers are working on adding a more accurate version.

If the matrix is square, we do something different. A Fortran77 package (!) called ARPACK is a very mature package for computing eigenvalue decomposition. These tend to be difficult if not impossible to rewrite as a distributed algorithm, but fortunately there's a JNI interface via Netlib-Java, and Scala runs on the JVM…

This is still single-core, which can be worked around by distributing the matrix-vector multiplications, because ARPACK can solve a lot of things with its reverse communication interface, i.e. only passing in matrix-vector multiplications. So it's still fast, yet distributed.

There's lots of research into distributing programs algorithmically, but this technique of reducing to matrix-vector multiplications has lots of hope for generalization.

Now, the caveat is that in Mlib 1.0, the currently released version, only the first algorithm exists. But in MLlib 1.1, both of these will be available, and this is likely to be released in a few weeks (the code is already frozen).

All-pairs Similarity Computation. This is an algorithm that will be in MLlib 1.2. The problem is: in some large number (e.g. 10^6) of vectors, which ones are pairwise similar to each other? Considering all $\binom{1000000}{2}$ pairs is a bad idea; this grows quadratically. The solution is called DIMSUM, and will be implemented as part of the `RowMatrix` class.

Similarity is defined as the cosine-similarity, which is a normalized dot product (so 1 if very similar, and 0 if very dissimilar).

The naïve way to do this is a map-reduce, which produces the normalized dot product for each pair. The mapper does the following: for all pairs a_{ij}, a_{jk} in r_i: emit $(j,k) \to a_{ij}a_{jk}$. Then, the reducer sums what it receivers, returning

the normalized dot product. Given (i, j) and entries $\langle v_1, \ldots, v_R \rangle$, it returns

$$\mathbf{c}_i^{\mathrm{T}} \mathbf{c}_j = \sum_{i=1}^{R} v_i.$$

The shuffle size is $O(mL^2)$, and the largest reduce key is $O(m)$; both of these values are very bad: they're intractable when $m \geq 10^{12}$ or $L \geq 100$. Thus, let's use DIMSUM. The reducer is the same, and we don't have time to go into the mapper, but look at the slides.

Intuitively, the mapper flips a coin to determine whether or not to output information. This algorithm produces an estimate, but (as part of the speaker's thesis) it converges, and this can be proven. Here are some results:

- The shuffle size is $O(nL\gamma)$ ($\gamma \approx \log n$).
- The largest reduce key is $O(\gamma)$.

Most of the audience, and the scope of the talk, is not mathematically inclined, but the point is that it'll be shipped in MLlib so one can use it without having to understand all of the proofs. Such a use case would, for example, ask for all pairs of vectors that have a similarity coefficient of at least 0.5. In the real world, Twitter uses this to determine which users are similar to others, so that people receive recommended people to follow.

Where we're going next. There's lots of ongoing work in MLlib, e.g. working on a stats library, multiclass decision trees, and so on. The algorithms mentioned above are also in progress, and with some students at Stanford, they're working on building some convex-optimization tools.

MLlib interacts nicely with the other libraries which will be discussed today; for example, in Spark 1.1, one can train linear models in a streaming fashion, i.e. interacting with the Streaming library. This involves fiddling with the algorithms a bit, and isn't yet done for everything; this is a work in progress.

MLlib can also work with SQL, and with GraphX.

```
val graph = Graph(pages, links)
val pageRank: RDD[(Long, Double)] = graph.staticPageRank(10).vertices

/ load page labels (spam or not), content features
val labelAndFeatures: RDD[(Long, (Double, Seq((Int, Double))))] = ...
val training: RDD[LabeledPoint] =
    labelAndFeatures.join(pageRank).map {
        case (id, (label, features), pageRank)) =>
            labeledPoint(label, Vectors.sparse(features ++ (1000, pageRank)))
}

// then, you can train a stochastic gradient descent on it, or such]
```

Since Spark is open-source, then there's lots of research going on at universities. This relates to the fact that implementing convex optimization is hard — but there's a package called CvxPy that came out of Stanford to handle convex optimization in Python, and makes distributing them very easy.

2. Automating the MLlib Pipeline

This talk was given by Ameet Talwalkar, who is at DataBricks.

There are several problems inherent in machine learning, akin to mazes and cages. Machine learning is difficult for end users, and for researchers, it's really hard to make scalable. For end users, we want to simplify machine learning so one doesn't need several classes in statitics to get things to work.

The point of MLbase is to simplify both development and deployment, and is built on Apache Spark. In fact, it's also built on MLlib, which was just talked about, and then on top of MLI, an API to simplify the development (hide the distributed aspect from the end user, a little bit, to make it easier to use). Finally, the last layer, the work in progress, is MLOpt, a declarative layer which attempts to automate these construction. Thee last two are highly experimental research, but are leading to interesting things happening in MLlib (which is production-ready).

As we saw, MLlib is one of the fastest-growing parts of the Spark ecosystem — even though it was only shipped 11 months ago, there have been dozens of contributors, a new language (Python) added on top of Scala and Java, and very good documentation, etc.

The goal of MLI is to shield ML developers from low-level details. This led to a paper, etc., but the ideas are making it into current and future releases of MLlib. The next release of MLlib is being tested now, and has a Python

decision tree API along with other high-level goodies; over the longer term, the goal is for MLlib to contain scalable implementation of standard ML algorithms, but also the underlying optimization primitives. The goal is to also have support for developing the ML pipeline (making it easier to use these algorithms).

This is an open-source project, so feedback from users or developers is strongly encouraged.

The grand vision here is a very simple API: the user declaratively specifies a task (e.g. via a SQL query), and receives a result; but under the hood; the database optimizer handles everything else. This would be some "MQL," a machine learning query language,a nd this is something that people have been working towards for a while.

The thing is, of course, this isn't easy, or we wouldn't be talking about it. For example, in the context of gradient-based algorithms (or any other iterative approach), training a model requires iterating over the data many times, repeating until convergence. This model doesn't take all that long to train on Spark, so we've solved a problem!

But there are lots of models: lots of algorithms with lots of hyper-parameters (e.g. the learning rate, regularization), and then featurization: how do we extract features from text? This is yet another problem.

The standard approach to determining the hyperparameters is to try everything! Imagine a two-dimensional hyperparameter space with a single model (e.g. SVM), with the two hyperparameters of learning rate and regularization. So people make a grid of ome finenes determine by their resources and patience, and thendetermine which of these is best.

Great, except that each grid point is a full model, so this takes a long time, especially when there are many hyperparameters or many families of models. This rapidly becomes a *huge* search problem, though in specific small cases one can just do it once with a for loop and wait not all that long.

Finally, how do we define "best" for machine learning models? This has several answers, but the user should be able to specify which one they want.

A better approahc is to batch-process each model, leading to better resource utilization. Another solution is to only partially calculate answers and use this to determine which models to fully train. These optimizations are grounded in the fact that it's significantly faster to process data than read it from memory, so batch processing (for example) is much faster. In Spark, for instance, this leads to 2× to 5× speedups on small sizes, but makes much less of a difference when the feature size is larger.

But this thrns out to be related to lots of matrix-vector multiplications (matrix of models, vectors of data), whcich led to rewriting it in terms of matrix-matrix multiplies, which is 5 to 6 times faster (not as good as the theory predicted, but still very impressive).

There are several different types of algorithmic speedups, e.g. doing gradient descent or whatever, but today we'll talk about early stopping. Sometimes, we can know early on that a model is no good, and should move past it sooner. For example, if one has a model whose error decreases with slope about m_1 over iterations, and another model has error decreasing much more slowly, then the second one can be abandoned; it's unlikely to catch up to the first one.

Unsurprisingly, early stopping saves a lot of time and passes over the data, leading to a speedup of abotu 5×... but what about the accuracy? It turns out to not decrese the accuracy significantly, so it's super useful.

Finally, what method are we using to search through the space? There are some derivative-free optimization techniques which are particularly useful (since we don't know much about the smooth structure of the space, or it might be expensive to compute), such as grid or random, Nelder-Mead, Powell's method, and even a few Bayesian models, called things such as SMAC, TPE, and Spearment. These generally start with random points and walks that are gradually refined.

The results: Powell's method and Nelder-Mead don't work so well, as gradient descent-based methods don't work so well on these heavily-constrained spaces; random search does pretty well, and TPE works best so far.

When they're combined all together, this turns an overnight computation into something that takes about 30 minutes. Larger problems may still need to go overnight, but the naïve approach would take a week!

There's lots of future wrork here. the pipeline is more complicated than was presented here, and each step adds hyperparameters, making the problem much more difficult. Some new techniques that they plan to take advantage of are ensembling (combining the best models), sampling, and better parallelism for smaller data sets. Furthermore, this model has some issues with false positives influencing the algorithm; this is being worked on.