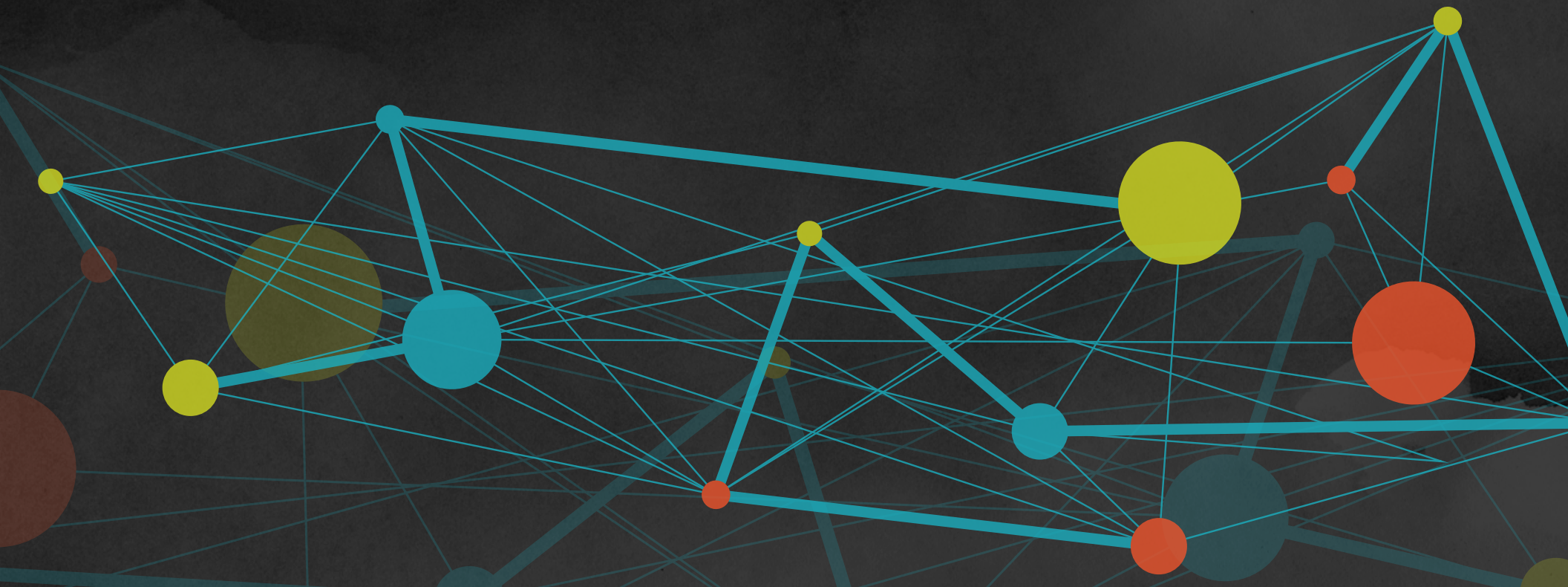
databricks

Mastering Advanced Analytics with Apache® Spark™

Highlights from the Databricks Blog



Mastering Advanced Analytics with Apache® Spark™

Highlights from the Databricks Blog

By Joseph Bradley, Hossein Falaki, Tim Hunter, Eric Liang, Feynman Liang, Xiangrui Meng, Burak Yavuz, and Reza Zadeh

Special thanks to our guest authors Jiajin Zhang and Dandan Tu from Huawei; Yuhao Yang from Intel; Jeremy Freeman from the Janelia Research Campus; Manish Amde from Origami Logic; Ankur Dave, Evan Sparks, and Shivaram Venkataraman from UC Berkeley; and Doris Xin from UIUC.

2nd in a series from Databricks:

Databricks
160 Spear Street, 13th Floor
San Francisco, CA 94105
Contact Us

About Databricks

Databricks' vision is to empower anyone to easily build and deploy advanced analytics solutions. The company was founded by the team who created Apache® Spark™, a powerful open source data processing engine built for sophisticated analytics, ease of use, and speed. Databricks is the largest contributor to the open source Apache Spark project providing 10x more code than any other company. The company has also trained over 20,000 users on Apache Spark, and has the largest number of customers deploying Spark to date. Databricks provides a just-in-time data platform, to simplify data integration, real-time experimentation, and robust deployment of production applications. Databricks is venture-backed by Andreessen Horowitz and NEA. For more information, contact info@databricks.com.

Introduction

The continued growth in data coupled with the need to make increasingly complex decisions against that data is creating massive hurdles that prevent organizations from deriving insights in a timely manner using traditional analytical approaches. Whether you're scrutinizing the clickstream from millions of visitors to optimize online ad placements, or sifting through billions of transactions to identify signs of fraud, the need for advanced analytics - such as machine learning and graph processing - to automatically glean insights from enormous volumes of data is more evident than ever.

Apache Spark, the de facto standard for big data processing and data sciences across all industries provides both machine learning and graph processing libraries, allowing companies to tackle complex problems easily with the power of highly scalable, clustered computers.

Our engineers, including the creators of Spark and Spark committers, and members of the Spark PMC, endeavor to bring state-of-the-art advanced machine learning and graph computation techniques to Spark with their contributions to MLlib, GraphX, and other libraries. Through the Databricks Blog, they regularly highlight new Spark releases and features relevant to advanced analytics, provide technical tutorials on MLlib components, in addition to sharing practical implementation tools and tips.

This ebook, the second of a series, offers a collection of the most popular technical blog posts that provide an introduction to machine learning on Spark and highlight many of the major developments around Spark MLlib and GraphX. It also dives into advanced topics such as clustering, graph processing and others — and details on how you can use SparkR to analyze data at scale with the R language. Whether you are just getting started with Spark or are already a Spark power user, this ebook will arm you with the knowledge to be successful on your next Spark project.

Read all the books in this Series:

Apache Spark Analytics Made Simple
Mastering Advanced Analytics with Apache Spark
Lessons for Large-Scale Machine Learning Deployments on Apache Spark
Building Real-Time Applications with Spark Streaming

Section 1:
An Introduction to Machine Learning in Apache Spark

New Features in MLlib in Spark 1.0

July 16, 2014 | by Xiangrui Meng

MLlib is a Spark component focusing on machine learning. It became a standard component of Spark in version 0.8 (Sep 2013). The initial contribution was from Berkeley AMPLab. Since then, 50+ developers from the open source community have contributed to its codebase. With the release of Spark 1.0, I'm glad to share some of the new features in MLlib. Among the most important ones are:

- sparse data support
- regression and classification trees
- distributed matrices
- PCA and SVD
- L-BFGS optimization algorithm
- new user guide and code examples

This is the first in a series of blog posts about features and optimizations in MLlib. We will focus on one feature new in 1.0 — sparse data support.

Large-scale ≈ Sparse

When I was in graduate school, I wrote "large-scale sparse least squares" in a paper draft. My advisor crossed out the word "sparse" and left a comment: "Large-scale already implies sparsity, so you don't need to mention it twice." It is a good argument. Sparse datasets are indeed very common in the big data world, where the sparsity may come from many sources, e.g.,

- feature transformation: one-hot encoding, interaction, and binarization,
- large feature space: n-grams,
- missing data: rating matrix.

Take the Netflix Prize qualifying dataset as an example. It contains around 100 million ratings generated by 480,189 users on 17,770 movies. Therefore, the rating matrix contains only 1% nonzeros. If an algorithm can utilize this sparsity, it can see significant improvements.

Exploiting Sparsity

In Spark 1.0, MLlib adds full support for sparse data in Scala, Java, and Python (previous versions only supported it in specific algorithms like alternating least squares). It takes advantage of sparsity in both storage and computation in methods including SVM, logistic regression, Lasso, naive Bayes, k-means, and summary statistics.

To give a concrete example, we ran k-means clustering on a dataset that contains more than 12 million examples with 500 feature dimensions. There are about 600 million nonzeros and hence the density is about 10%. The result is listed in the following table:

	sparse	**dense**
storage	7GB	47GB
time	58s	240s

So, not only did we save 40GB of storage by switching to the sparse format, but we also received a 4x speedup. If your dataset is sparse, we strongly recommend you to try this feature.

Getting Started

Both sparse and dense feature vectors are supported via the Vector interface. A sparse vector is represented by two parallel arrays: indices and values. Zero entries are not stored. A dense vector is backed by a double array representing its entries. For example, a vector [1., 0., 0., 0., 0., 0., 3.] can be represented in the sparse format as (7, [0, 6], [1., 3.]), where 7 is the size of the vector, as illustrated below:

$$\text{dense}: \underbrace{1.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{3.}$$

$$\text{sparse}: \begin{cases} \text{size}: 7 \\ \text{indices}: \underbrace{0}\ \underbrace{6} \\ \text{values}: \underbrace{1.}\ \underbrace{3.} \end{cases}$$

Take the Python API as an example. MLlib recognizes the following types as dense vectors:

- NumPy's `array`,
- Python's list, e.g., `[1, 2, 3]`.

and the following as sparse vectors:

- MLlib's `SparseVector`,
- SciPy's `csc_matrix` with a single column.

We recommend using NumPy arrays over lists for efficiency, and using the factory methods implemented in Vectors to create sparse vectors.

```
import numpy as np
import scipy.sparse as sps
from pyspark.mllib.linalg import Vectors

# Use a NumPy array as a dense vector.
dv1 = np.array([1.0, 0.0, 3.0])
# Use a Python list as a dense vector.
dv2 = [1.0, 0.0, 3.0]
# Create a SparseVector.
sv1 = Vectors.sparse(3, [0, 2], [1.0, 3.0])
# Use a single-column SciPy csc_matrix as a sparse vector.
sv2 = sps.csc_matrix((np.array([1.0, 3.0]), np.array([0, 2]),
    np.array([0, 2])), shape = (3, 1))
```

K-means takes an RDD of vectors as input. For supervised learning, the training dataset is represented by an RDD of labeled points. A LabeledPoint contains a label and a vector, either sparse or dense. Creating a labeled point is straightforward.

```
from pyspark.mllib.linalg import SparseVector
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense vector.
pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])

# Create a labeled point with a negative label and a sparse vector.
neg = LabeledPoint(0.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

MLlib also supports reading and saving labeled data in LIBSVM format. For more information on the usage, please visit the MLlib guide and code examples.

When to Exploit Sparsity

For many large-scale datasets, it is not feasible to store the data in a dense format. Nevertheless, for medium-sized data, it is natural to ask when we should switch from a dense format to sparse. In MLlib, a sparse vector requires 12nnz+4 bytes of storage, where nnz is the number of nonzeros, while a dense vector needs 8n bytes, where n is the vector size. So storage-wise, the sparse format is better than the dense format when more than 1/3 of the elements are zero. However, assuming that the data can be fit into memory in both formats, we usually need sparser data to observe a speedup, because the sparse format is not as efficient as the dense format in computation. Our experience suggests a sparsity of around 10%, while the exact switching point for the running time is indeed problem-dependent.

Sparse data support is part of Spark 1.0, which is available for download right now at http://spark.apache.org/. We will cover more new features in MLlib in a series of posts. So stay tuned.

Statistics Functionality in Spark 1.1

August 27, 2014 | by Doris Xin, Burak Yavuz, Xiangrui Meng and Hossein Falaki

One of our philosophies in Spark is to provide rich and friendly built-in libraries so that users can easily assemble data pipelines. With Spark, and MLlib in particular, quickly gaining traction among data scientists and machine learning practitioners, we're observing a growing demand for data analysis support outside of model fitting. To address this need, we have started to add scalable implementations of common statistical functions to facilitate various components of a data pipeline. We're pleased to announce Spark 1.1. ships with built-in support for several statistical algorithms common in exploratory data pipelines:

- **correlations**: data dependence analysis
- **hypothesis testing**: goodness of fit; independence test
- **stratified sampling**: scaling training set with controlled label distribution
- **random data generation**: randomized algorithms; performance tests

As ease of use is one of the main missions of Spark, we've devoted a great deal of effort to API designing for the statistical functions. Spark's statistics APIs borrow inspirations from widely adopted statistical packages, such as R and SciPy.stats, which are shown in a recent O'Reilly survey to be the most popular tools among data scientists.

Correlations

Correlations provide quantitative measurements of the statistical dependence between two random variables. Implementations for correlations are provided under `mllib.stat.Statistics`.

MLlib	• `corr(x, y = None, method = "pearson" | "spearman")`
R	• `cor(x, y = NULL, method = c("pearson", "kendall", "spearman"))`
SciPy	• `pearsonr(x, y)` • `spearmanr(a, b = None)`

As shown in the table, R and SciPy.stats present us with two very different directions for correlations API in MLlib. We ultimately converged on the R style of having a single function that takes in the correlation method name as a string argument out of considerations for extensibility as well as conciseness of the API list. The initial set of methods contains "pearson" and "spearman", the two most commonly used correlations.

Hypothesis testing

Hypothesis testing is essential for data-driven applications. A test result shows the statistical significance of an event unlikely to have occurred by chance. For example, we can test whether there is a significant association between two samples via independence tests. In Spark 1.1, we implemented chi-squared tests for goodness-of-fit and independence:

MLlib	• `chiSqTest(observed: Vector, expected: Vector)` • `chiSqTest(observed: Matrix)` • `chiSqTest(data: RDD[LabeledPoint])`
R	• `chisq.test(x, y = NULL, correct = TRUE, p = rep(1/length(x), length(x)), rescale.p = FALSE, simulate.p.value = FALSE)`
SciPy	• `chisquare(f_obs, f_exp = None, ddof = 0, axis = 0)`

When designing the chi-squared test APIs, we took the greatest common denominator of the parameters in R's and SciPy's APIs and edited out some of the less frequently used parameters for API simplicity. Note that as with R and SciPy, the input data types determine whether the goodness of fit or the independence test is conducted. We added a special case support for input type `RDD[LabeledPoint]` to enable feature selection via chi-squared independence tests.

Stratified sampling

It is common for a large population to consist of various-sized subpopulations (strata), for example, a training set with many more positive instances than negatives. To sample such populations, it is advantageous to sample each stratum independently to reduce the total variance or to represent small but important strata. This sampling design is called stratified sampling. Unlike the other statistics functions, which reside in MLlib, we placed the stratified sampling methods in Spark Core, as sampling is widely used in data analysis. We provide two versions of stratified sampling, `sampleByKey` and `sampleByKeyExact`. Both apply to an RDD of key-value pairs with key indicating the stratum, and both take a map from users that specifies the sampling probability for each stratum. Neither R nor SciPy provides built-in support for stratified sampling.

MLlib	• `sampleByKey(withReplacement, fractions, seed)` • `sampleByKeyExact(withReplacement, fractions, seed)`

Similar to `RDD.sample`, `sampleByKey` applies Bernoulli sampling or Poisson sampling to each item independently, which is cheap but doesn't guarantee the exact sample size for each stratum (the size of the stratum times the corresponding sampling probability). `sampleByKeyExact` utilizes scalable sampling algorithms that guarantee the exact sample size for each stratum with high probability, but it requires multiple passes over the data. We have a separate method name to underline the fact that it is significantly more expensive.

Random data generation

Random data generation is useful for testing of existing algorithms and implementing randomized algorithms, such as random projection. We provide methods under `mllib.random.RandomRDDs` for generating RDDs that contains i.i.d. values drawn from a distribution, e.g., uniform, standard normal, or Poisson.

MLlib	• `normalRDD(sc, size, [numPartitions, seed])` • `normalVectorRDD(sc, numRows, numCols, [numPartitions, seed])`
R	• `rnorm(n, mean=0, sd=1)`
SciPy	• `randn(d0, d1, …, dn)` • `normal([loc, scale, size])` • `standard_normal([size])`

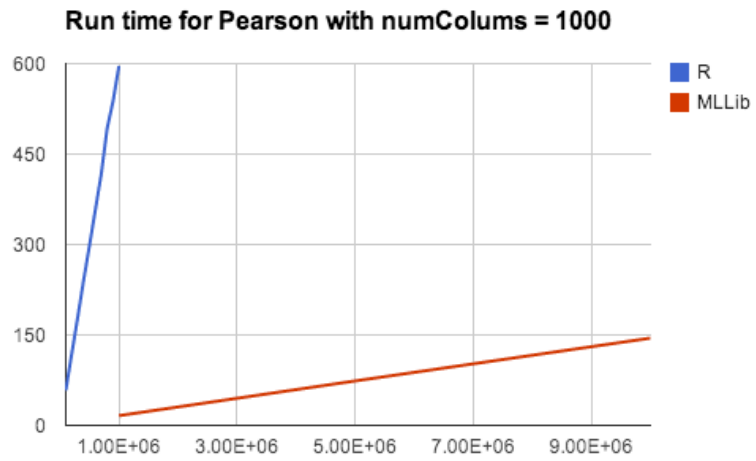
The random data generation APIs exemplify a case where we added Spark-specific customizations to a commonly supported API. We show a side-by-side comparison of MLlib's normal distribution data generation APIs vs. those of R and SciPy in the table above. We provide 1D (`RDD[Double]`) and 2D (`RDD[Vector]`) support, compared to only 1D in R and arbitrary dimension in NumPy, since both are prevalent in MLlib functions. In addition to the Spark specific parameters, such as SparkContext and number of partitions, we also allow users to set the seed for reproducibility. Apart from the built-in distributions, users can plug in their own via `RandomDataGenerator`.

What about SparkR?

At this point you may be asking why we're providing native support for statistics functions inside Spark given the existence of the SparkR project. As an R package, SparkR is a great lightweight solution for empowering familiar R APIs with distributed computation support. What we're aiming to accomplish with these built-in Spark statistics APIs is cross language support as well as seamless integration with other components of Spark, such as Spark SQL and Streaming, for a unified data product development platform. We expect these features to be callable from SparkR in the future.

Concluding remarks

Besides a set of familiar APIs, statistics functionality in Spark also brings R and SciPy users huge benefits including scalability, fault tolerance, and seamless integration with existing big data pipelines. Both R and SciPy run on a single machine, while Spark can easily scale up to hundreds of machines and distribute the computation. We compared the running times of MLlib's Pearson's correlation on a 32-node cluster with R's, not counting the time needed for moving data to the node with R installed. The results shown below demonstrates a clear advantage of Spark over R on performance and scalability.



As the Statistics APIs are experimental, we'd love feedback from the community on the usability of these designs. We also welcome contributions from the community to enhance statistics functionality in Spark.

Scalable Collaborative Filtering with Spark MLlib

July 23, 2014 | by Burak Yavuz, Xiangrui Meng and Reynold Xin

Recommendation systems are among the most popular applications of machine learning. The idea is to predict whether a customer would like a certain item: a product, a movie, or a song. Scale is a key concern for recommendation systems, since computational complexity increases with the size of a company's customer base. In this blog post, we discuss how Spark MLlib enables building recommendation models from billions of records in just a few lines of Python (Scala/Java APIs also available).

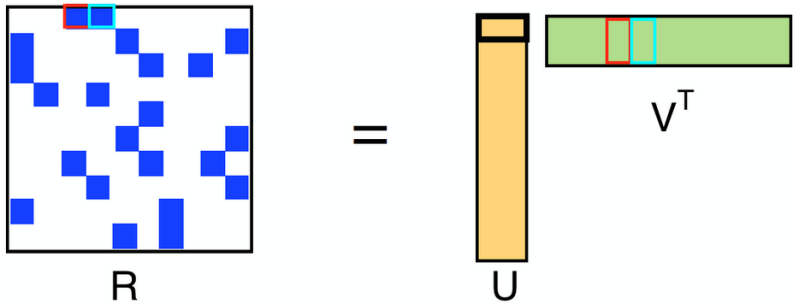
```
from pyspark.mllib.recommendation import ALS

# load training and test data into (user, product, rating) tuples
def parseRating(line):
  fields = line.split()
  return (int(fields[0]), int(fields[1]), float(fields[2]))
training = sc.textFile("...").map(parseRating).cache()
test = sc.textFile("...").map(parseRating)

# train a recommendation model
model = ALS.train(training, rank = 10, iterations = 5)

# make predictions on (user, product) pairs from the test data
predictions = model.predictAll(test.map(lambda x: (x[0], x[1])))
```

What's Happening under the Hood?

Recommendation algorithms are usually divided into:

(1) **Content-based filtering**: recommending items similar to what users already like. An example would be to play a Megadeth song after a Metallica song.

(2) **Collaborative filtering**: recommending items based on what similar users like, e.g., recommending video games after someone purchased a game console because other people who bought game consoles also bought video games.



Spark MLlib implements a collaborative filtering algorithm called **Alternating Least Squares (ALS)**, which has been implemented in many machine learning libraries and widely studied and used in both academia and industry. ALS models the rating matrix (R) as the multiplication of low-rank user (U) and product (V) factors, and learns these factors by minimizing the reconstruction error of the observed ratings. The unknown ratings can subsequently be computed by multiplying these

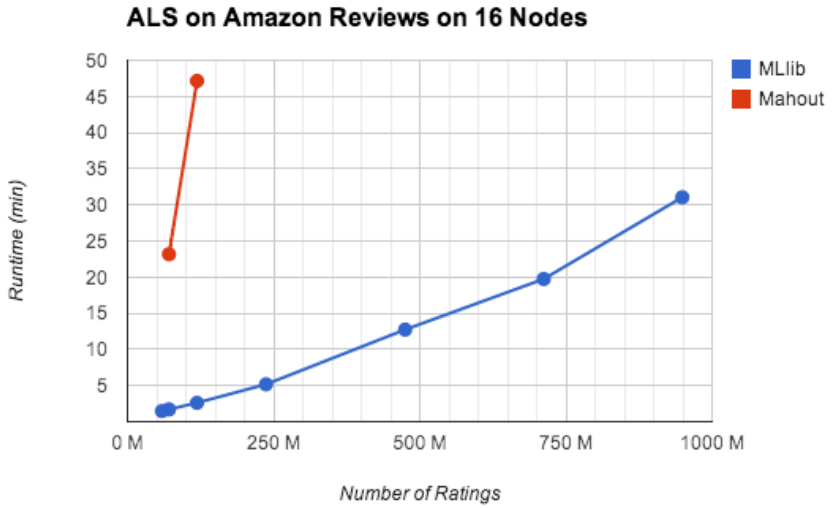
factors. In this way, companies can recommend products based on the predicted ratings and increase sales and customer satisfaction.

ALS is an iterative algorithm. In each iteration, the algorithm alternatively fixes one factor matrix and solves for the other, and this process continues until it converges. MLlib features a blocked implementation of the ALS algorithm that leverages Spark's efficient support for distributed, iterative computation. It uses native LAPACK to achieve high performance and scales to billions of ratings on commodity clusters.

Scalability, Performance, and Stability

Recently we did an experiment to benchmark ALS implementations in Spark MLlib at scale. The benchmark was conducted on EC2 using m3.2xlarge instances set up by the Spark EC2 script. We ran Spark using out-of-the-box configurations. To help understand state-of-the-art, we also built Mahout from GitHub and tested it. This benchmark is reproducible on EC2 using the scripts at https://github.com/databricks/als-benchmark-scripts.

We ran 5 iterations of ALS on scaled copies of the Amazon Reviews dataset, which contains 35 million ratings collected from 6.6 million users on 2.4 million products. For each user, we create pseudo-users that have the same ratings. That is, for every rating as (userId, productId, rating), we generate (userId+i, productId, rating) where 0 <= i < s and s is the scaling factor.



The current version of Mahout runs on Hadoop MapReduce, whose scheduling overhead and lack of support for iterative computation substantially slows down ALS. Mahout recently announced switching to Spark as the execution engine, which will hopefully address the performance concerns.

Spark MLlib demonstrated excellent performance and scalability, as demonstrated in the chart above. MLlib can also scale to much larger datasets and to larger number of nodes, thanks to its fault-tolerance design. With 50 nodes, we ran 10 iterations of MLlib's ALS on 100 copies of the Amazon Reviews dataset in only 40 minutes. And with EC2 spot instances the total cost was less than $2. Users can use Spark MLlib to reduce the model training time and the cost for ALS, which is historically very expensive to run because the algorithm is very communication intensive and computation intensive.

# ratings	# users	# products	time
3.5 billion	660 million	2.4 million	40 mins

It is our belief at Databricks and the broader Spark community that machine learning frameworks need to be performant, scalable, and be able to cover a wide range of workloads including data exploration and feature extraction. MLlib integrates seamlessly with other Spark components, delivers best-in-class performance, and substantially simplifies operational complexity by running on top of a fault-tolerant engine. That said, our work is not done and we are working on making machine learning easier. Stay tuned for more exciting features.

Note: The blog post was updated on July 24, 2014 to reflect a new performance optimization that will be included in Spark MLlib 1.1. The runtime for 3.5B ratings went down from 90 mins in MLlib 1.0 to 40 mins in MLlib 1.1.

ML Pipelines: A New High-Level API for MLlib

January 7, 2015 | by Xiangrui Meng, Joseph Bradley, Evan Sparks and Shivaram Venkataraman

Get the Notebook

MLlib's goal is to make practical machine learning (ML) scalable and easy. Besides new algorithms and performance improvements that we have seen in each release, a great deal of time and effort has been spent on making MLlib *easy*. Similar to Spark Core, MLlib provides APIs in three languages: Python, Java, and Scala, along with user guide and example code, to ease the learning curve for users coming from different backgrounds. In Spark 1.2, Databricks, jointly with AMPLab, UC Berkeley, continues this effort by introducing a pipeline API to MLlib for easy creation and tuning of practical ML pipelines.

A practical ML pipeline often involves a sequence of data pre-processing, feature extraction, model fitting, and validation stages. For example, classifying text documents might involve text segmentation and cleaning, extracting features, and training a classification model with cross-validation. Though there are many libraries we can use for each stage, connecting the dots is not as easy as it may look, especially with large-scale datasets. Most ML libraries are not designed for distributed computation or they do not provide native support for pipeline creation and tuning. Unfortunately, this problem is often ignored in academia, and it has received largely ad-hoc treatment in industry, where development tends to occur in manual one-off pipeline implementations.

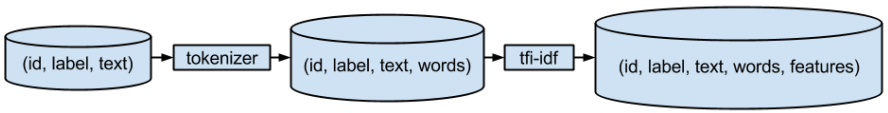
In this post, we briefly describe the work done to address ML pipelines in MLlib, a joint effort between Databricks and AMPLab, UC Berkeley, and inspired by the scikit-learn project and some earlier work on MLI.

Dataset abstraction

In the new pipeline design, a dataset is represented by Spark SQL's SchemaRDD and an ML pipeline by a sequence of dataset transformations. **(Update**: SchemaRDD was renamed to DataFrame in Spark 1.3). Each transformation takes an input dataset and outputs the transformed dataset, which becomes the input to the next stage. We leverage on Spark SQL for several reasons: data import/export, flexible column types and operations, and execution plan optimization.

Data import/export is the start/end point of an ML pipeline. MLlib currently provides import/export utilities for several application-specific types: LabeledPoint for classification and regression, Rating for collaborative filtering, and so on. However, realistic datasets may contain many types, such as user/item IDs, timestamps, or raw records. The current utilities cannot easily handle datasets with combinations of these types, and they use inefficient text storage formats adopted from other ML libraries.

Feature transformations usually form the majority of a practical ML pipeline. A feature transformation can be viewed as appending new columns created from existing columns. For example, text tokenization breaks a document up into a bag of words, and tf-idf converts a bag of words into a feature vector, while during the transformations the labels need to be preserved for model fitting. More complex feature transformations are quite common in practice. Hence, the dataset needs to support columns of different types, including dense and sparse vectors, and operations that create new columns from existing ones.



In the example above, id, text, and words are carried over during transformations. They are unnecessary for model fitting, but useful in prediction and model inspection. It doesn't provide much information if the prediction dataset only contains the predicted labels. If we want to inspect the prediction results, e.g., checking false positives, it is quite useful to look at the predicted labels along with the raw input text and tokenized words. The columns needed at each stage are quite different. It would be ideal that the underlying execution engine can optimize for us and only load the required columns.

Fortunately, Spark SQL already provides most of the desired functions and we don't need to reinvent the wheel. Spark SQL supports import/export SchemaRDDs from/to Parquet, an efficient columnar storage format, and easy conversions between RDDs and SchemaRDDs. It also supports pluggable external data sources like Hive and Avro. Creating (or declaring to be more precise) new columns from existing columns is easy with user-defined functions. The materialization of SchemaRDD is lazy. Spark SQL knows how to optimize the execution plan based on the columns requested, which fits our needs well. SchemaRDD supports standard data types. To make it a better fit for ML, we worked together with the Spark SQL team and added Vector type as a user-defined type that supports both dense and sparse feature vectors.

We show a simple Scala code example for ML dataset import/export and simple operations. More complete dataset examples in Scala and Python can be found under the `examples/` folder of the Spark repository. We refer users to Spark SQL's user guide to learn more about SchemaRDD and the operations it supports.

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.util.MLUtils


// Load a LIBSVM file into an RDD[LabeledPoint].
val labeledPointRDD: RDD[LabeledPoint] =
MLUtils.loadLibSVMFile(sc, "/path/to/libsvm")

// Convert from RDD to DF and save it as a Parquet.
labeledPointRDD.toDF().write.parquet("/path/to/parquet")

// Load the Parquet file back into a DataFrame
val labeledPointDF = sqlContext.read.parquet("/path/to/parquet")
// Collect the feature vectors and print them.
labeledPointDF.select('features).collect().foreach(println)
```

Pipeline

The new pipeline API lives under a new package named “spark.ml”. A pipeline consists of a sequence of stages. There are two basic types of pipeline stages: Transformer and Estimator. A Transformer takes a dataset as input and produces an augmented dataset as output. E.g., a tokenizer is a Transformer that transforms a dataset with text into an dataset with tokenized words. An Estimator must be first fit on the input dataset to produce a model, which is a Transformer that transforms the input dataset. E.g., logistic regression is an Estimator that trains on a dataset with labels and features and produces a logistic regression model.

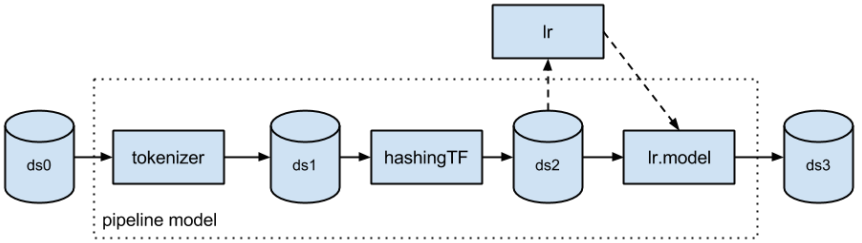
Creating a pipeline is easy: simply declare its stages, configure their parameters, and chain them in a pipeline object. For example the following code creates a simple text classification pipeline consisting of a tokenizer, a hashing term frequency feature extractor, and logistic regression.

```
val tokenizer = new Tokenizer()
  .setInputCol("text")
  .setOutputCol("words")
val hashingTF = new HashingTF()
  .setNumFeatures(1000)
  .setInputCol(tokenizer.getOutputCol)
  .setOutputCol("features")
val lr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.01)
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, hashingTF, lr))
```

The pipeline itself is an Estimator, and hence we can call fit on the entire pipeline easily.

```
val model = pipeline.fit(trainingDataset)
```

The fitted model consists of the tokenizer, the hashing TF feature extractor, and the fitted logistic regression model. The following diagram draws the workflow, where the dash lines only happen during pipeline fitting.

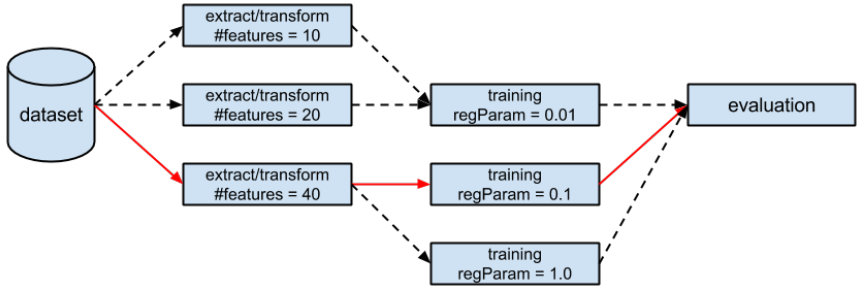


The fitted pipeline model is a transformer that can be used for prediction, model validation, and model inspection.

```
model.transform(testDataset)
  .select('text, 'label, 'prediction)
  .collect()
  .foreach(println)
```

One unfortunate characteristic of ML algorithms is that they have many hyperparameters that must be tuned. These hyperparameters – e.g. degree of regularization – are distinct from the model parameters being

optimized by MLlib. It is hard to guess the best combination of hyperparameters without expert knowledge on both the data and the algorithm. Even with expert knowledge, it may become unreliable as the size of the pipeline and the number of hyperparameters grows. Hyperparameter tuning (choosing parameters based on performance on held-out data) is usually necessary to obtain meaningful results in practice. For example, we have two hyperparameters to tune in the following pipeline and we put three candidate values for each. Therefore, there are nine combinations in total (four shown in the diagram below) and we want to find the one that leads to the model with the best evaluation result.



We support cross-validation for hyperparameter tuning. We view cross-validation as a meta-algorithm, which tries to fit the underlying estimator with user-specified combinations of parameters, cross-evaluate the fitted models, and output the best one. Note that there is no specific requirement on the underlying estimator, which could be a pipeline, as long as it could be paired with an Evaluator that outputs a scalar metric from predictions, e.g., precision. Tuning a pipeline is easy:

```
// Build a parameter grid.
val paramGrid = new ParamGridBuilder()
  .addGrid(hashingTF.numFeatures, Array(10, 20, 40))
  .addGrid(lr.regParam, Array(0.01, 0.1, 1.0))
  .build()

// Set up cross-validation.
val cv = new CrossValidator()
  .setNumFolds(3)
  .setEstimator(pipeline)
  .setEstimatorParamMaps(paramGrid)
  .setEvaluator(new BinaryClassificationEvaluator)

// Fit a model with cross-validation.
val cvModel = cv.fit(trainingDataset)
```

It is important to note that users can embed their own transformers or estimators into an ML pipeline, as long as they implement the pipeline interfaces. The API makes it easy to use and share code maintained outside MLlib. More complete code examples in Java and Scala can be found under the 'examples/' folder of the Spark repository. We refer users to the spark.ml user guide for more information about the pipeline API.

Concluding remarks

The blog post describes the ML pipeline API introduced in Spark 1.2 and the rationale behind it. The work is covered by several JIRAs: SPARK-3530, SPARK-3569, SPARK-3572, SPARK-4192, and SPARK-4209. We refer users to the design docs posted on each JIRA page for more information about the design choices. And we would like to thank everyone who participated in the discussion and provided valuable feedback.

That being said, the pipeline API is experimental in Spark 1.2 and the work is still far from done. For example, more feature transformers can help users quickly assemble pipelines. We would like to mention some ongoing work relevant to the pipeline API:

- SPARK-5097: Adding data frame APIs to SchemaRDD
- SPARK-4586: Python API for ML pipeline
- SPARK-3702: Class hierarchy for learning algorithms and models

The pipeline API is part of Spark 1.2, which is available for download at http://spark.apache.org/. We look forward to hearing back from you about it, and we welcome your contributions and feedback.

Get the Notebook

New Features in Machine Learning Pipelines in Spark 1.4

July 29, 2015 | by Joseph Bradley, Xiangrui Meng and Burak Yavuz

Spark 1.2 introduced Machine Learning (ML) Pipelines to facilitate the creation, tuning, and inspection of practical ML workflows. Spark's latest release, Spark 1.4, significantly extends the ML library. In this post, we highlight several new features in the ML Pipelines API, including:

- A stable API — Pipelines have graduated from Alpha!
- New feature transformers
- Additional ML algorithms
- A more complete Python API
- A pluggable API for customized, third-party Pipeline components

If you're new to using ML Pipelines, you can get familiar with the key concepts like Transformers and Estimators by reading our previous blog post.

New Features in Spark 1.4

With significant contributions from the Spark community, ML Pipelines are much more featureful in the 1.4 release. The API includes many common feature transformers and more algorithms.

New Feature Transformers

A big part of any ML workflow is massaging the data into the right features for use in downstream processing. To simply feature extraction, Spark provides many feature transformers out-of-the-box. The table below outlines most of the feature transformers available in Spark 1.4 along with descriptions of each one. Much of the API is inspired by scikit-learn; for reference, we provide names of similar scikit-learn transformers where available.

Transformer	Description	scikit-learn
Binarizer	Threshold numerical feature to binary	Binarizer
Bucketizer	Bucket numerical features into ranges	
ElementwiseProduct	Scale each feature/column separately	
HashingTF	Hash text/data to vector. Scale by term frequency	FeatureHasher

Transformer	Description	scikit-learn
IDF	Scale features by inverse document frequency	TfidfTransformer
Normalizer	Scale each row to unit norm	Normalizer
OneHotEncoder	Encode k-category feature as binary features	OneHotEncoder
PolynomialExpansion	Create higher-order features	PolynomialFeatures
RegexTokenizer	Tokenize text using regular expressions	(part of text methods)
StandardScaler	Scale features to 0 mean and/or unit variance	StandardScaler
StringIndexer	Convert String feature to 0-based indices	LabelEncoder
Tokenizer	Tokenize text on whitespace	(part of text methods)
VectorAssembler	Concatenate feature vectors	FeatureUnion
VectorIndexer	Identify categorical features, and index	
Word2Vec	Learn vector representation of words	

* Only 3 of the above transformers were available in Spark 1.3 (HashingTF, StandardScaler, and Tokenizer).

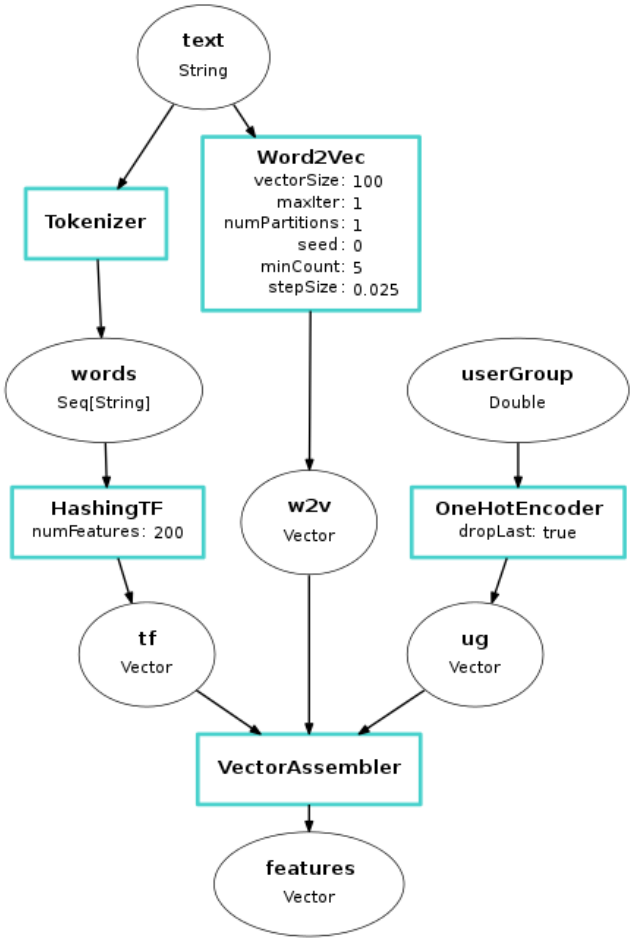
The following code snippet demonstrates how multiple feature encoders can be strung together into a complex workflow. This example begins with two types of features: *text* (String) and *userGroup* (categorical). For example:

text	userGroup
When I bought this lamp, I had no idea what I was getting into. It's amazing the kind of low quality you find online.	3
dude the laptops really cool and i gotta say its much better than the other one i got	2
I had to get a gift for my dad, and I saw this kite. It reminded me of when I was a kid.	5

We generate text features using both hashing and the Word2Vec algorithm, and then apply a one-hot encoding to *userGroup*. Finally, we combine all features into a single feature vector which can be used by ML algorithms such as Logistic Regression.

```
from pyspark.ml.feature import *
from pyspark.ml import Pipeline
tok = Tokenizer(inputCol="text", outputCol="words")
htf = HashingTF(inputCol="words", outputCol="tf", numFeatures=200)
w2v = Word2Vec(inputCol="text", outputCol="w2v")
ohe = OneHotEncoder(inputCol="userGroup", outputCol="ug")
va = VectorAssembler(inputCols=["tf", "w2v", "ug"],
outputCol="features")
pipeline = Pipeline(stages=[tok,htf,w2v,ohe,va])
```

The following diagram shows the full pipeline. Pipeline stages are shown as blue boxes, and DataFrame columns are shown as bubbles.



Better Algorithm Coverage

In Spark 1.4, the Pipelines API now includes trees and ensembles: Decision Trees, Random Forests, and Gradient-Boosted Trees. These are some of the most important algorithms in machine learning. They can be used for both regression and classification, are flexible enough to handle many types of applications, and can use both continuous and categorical features.

The Pipelines API also includes Logistic Regression and Linear Regression using Elastic Net regularization, an important statistical tool mixing L1 and L2 regularization.

Spark 1.4 also introduces OneVsRest (a.k.a. One-Vs-All), which converts any binary classification "base" algorithm into a multiclass algorithm. This flexibility to use any base algorithm in OneVsRest highlights the versatility of the Pipelines API. By using DataFrames, which support varied data types, OneVsRest can remain oblivious to the specifics of the base algorithm.

More Complete Python API

ML Pipelines have a near-complete Python API in Spark 1.4. Python APIs have become much simpler to implement after significant improvements to internal Python APIs, plus the unified DataFrame API. See the Python API docs for ML Pipelines for a full feature list.

Customizing Pipelines

We have opened up APIs for users to write their own Pipeline stages. If you need a custom feature transformer, ML algorithm, or evaluation metric in your workflow, you can write your own and plug it into ML Pipelines. Stages communicate via DataFrames, which act as a simple, flexible API for passing data through a workflow.

The key abstractions are:

- **Transformer**: This includes feature transformers (e.g., OneHotEncoder) and trained ML models (e.g., LogisticRegressionModel).
- **Estimator**: This includes ML algorithms for training models (e.g., LogisticRegression).
- **Evaluator**: These evaluate predictions and compute metrics, useful for tuning algorithm parameters (e.g., BinaryClassificationEvaluator).

To learn more, start with the overview of ML Pipelines in the ML Pipelines Programming Guide.

Looking Ahead

The roadmap for Spark 1.5 includes:

- ***API***: More complete algorithmic coverage in Pipelines, and more featureful Python API. There is also initial work towards an MLlib API in Spark R.
- ***Algorithms***: More feature transformers (such as CountVectorizer, DiscreteCosineTransform, MinMaxScaler, and NGram) and algorithms (such as KMeans clustering and Naive Bayes).
- ***Developers***: Improvements for developers, including to the feature attributes API and abstractions.

ML Pipelines do not yet cover all algorithms in MLlib, but the two APIs can interoperate. If your workflow requires components from both APIs, all you need to do is convert between RDDs and DataFrames. For more information on conversions, see the DataFrame guide.

Acknowledgements

Thanks very much to the community contributors during this release! You can find a complete list of JIRAs for ML Pipelines with contributors on the Apache Spark JIRA.

Learning More

To get started, download Spark and check out the ML Pipelines User Guide! Also try out the ML package code examples. Experts can get started writing their own Transformers and Estimators by looking at the DeveloperApiExample code snippet.

To contribute, follow the MLlib 1.5 Roadmap JIRA. Good luck!

Improved Frequent Pattern Mining in Spark 1.5: Association Rules and Sequential Patterns

September 28, 2015 | by Feynman Liang, Jiajin Zhang, Dandan Tu and Xiangrui Meng

We would like to thank Jiajin Zhang and Dandan Tu from Huawei for contributing to this blog.

Discovering frequent patterns hiding in a big dataset has application across a broad range of use cases. Retailers may be interested in finding items that are frequently purchased together from a large transaction database. Biologists may be interested in frequent DNA or amino acid sequences. In Spark 1.5, we have significantly improved Spark's frequent pattern mining capabilities by adding algorithms for association rule generation and sequential pattern mining.

Association rules generation

Association rules are generated from frequent itemsets, subsets of items that appear frequently across transactions. Frequent itemset mining was first added in Spark 1.3 using the Parallel FP-growth algorithm. Spark 1.4 adds a new Python API for FP-growth. We refer readers to our previous blog post for more details.

In addition to identifying frequent itemsets, we are often interested in learning association rules.

For example, in a retailer's transaction database, a rule {toothbrush, floss} => {toothpaste} with a *confidence value* 0.8 would indicate that 80% of customers who buy a toothbrush and floss also purchase a toothpaste in the same transaction. The retailer could then use this information, put both toothbrush and floss on sale, but raise the price of toothpaste to increase overall profit.

Spark 1.5 adds support for distributed generation of association rules. Rule generation is done via a simple method call to FPGrowthModel with a min confidence value, for example:

```
val transactions: RDD[Array[String]] = ...
val model = new FPGrowth()
  .setMinSupport(0.2)
  .setNumPartitions(10)
  .run(transactions)
val minConfidence = 0.8
model.generateAssociationRules(minConfidence).collect().foreach
{
  rule =>
    println(rule.antecedent.mkString(",") + " => " +
      rule.consequent.mkString(",")
  )
}
```

Sequential pattern mining

Unlike frequent itemsets, where the items in a transaction are unordered, sequential pattern mining takes the order of items into account. In many use cases ranging from text mining to DNA sequence motif discovery, we care about the order in which items appear in a pattern.

Sequential pattern mining is widely used in Huawei, a Fortune Global 500 telecommunications company. For example, a mobile network of millions of users could generate several hundred GBs of session signaling data per day. Among the signaling sequences, we want to extract frequent sequential patterns like routing updates, activation failures, and broadcasting timeouts that could potentially lead to customer complaints. By identifying those patterns in real traffic, we can proactively reach out to customers with potential issues and help improve their experience.

Thanks to a collaboration between Databricks and Huawei, especially to Huawei for initiating the effort, sharing their use cases, and making significant code contribution, we are proud to announce support for parallel sequential pattern mining in Spark 1.5. This latest version ships with a parallel implementation of the PrefixSpan algorithm originally described by Pei et al.

Example: mining frequent sequential sign language patterns

To demonstrate PrefixSpan, we will mine frequent sequential patterns from the American Sign Language database provided by Boston University. Running PrefixSpan to discover frequent sequential patterns requires only a few lines of code:

```
val sequences: RDD[Array[Array[String]]] = ...
val prefixSpan = new PrefixSpan()
  .setMinSupport(0.6)
  .setMaxPatternLength(10)
val patterns = prefixSpan.run(sequences)
```

From this, we discover that common sequential patterns in the database include:

```
(head pos: tilt fr/bk - front), (eye aperture - ONSET), (POS - Verb)
(head pos: turn - ONSET), (eye aperture - ONSET), (POS - Verb)
(head pos: tilt fr/bk - ONSET), (eye aperture - ONSET), (POS - Verb)
(eye brows - ONSET), (eye aperture - ONSET), (POS - Verb)
(head pos: tilt fr/bk - front), (POS - Noun), (POS - Verb)
```

where each item indicates a sign or a gesture. For details, please see this gist.

Implementation

We followed the PrefixSpan algorithm but made modifications to parallelize the algorithm in a novel way for running on Spark. At a high level, our algorithm iteratively extends the lengths of prefixes until its associated projected database (i.e. the set of all sequences with that given prefix) is small enough to fit on a single machine. We then process each of these projected databases locally and combine the results to yield all of the sequential patterns.

What's next?

The improvements to frequent pattern mining have been a collaboration between many Spark contributors. This work is pushing the limits on distributed pattern mining. Ongoing work includes: model import/export for FPGrowth and PrefixSpan, a Python API for PrefixSpan, optimizing PrefixSpan for single-item itemsets, etc. To get involved, please check the MLlib 1.6 roadmap.

MLlib Highlights in Spark 1.6

January 21, 2016 | by Joseph Bradley and Xiangrui Meng

With the latest release, Apache Spark's Machine Learning library includes many improvements and new features. Users can now save and load ML Pipelines, use extended R and Python APIs, and run new ML algorithms. This blog post highlights major developments in MLlib 1.6 and mentions the roadmap ahead.

Many thanks to the 90+ contributors during this release, as well as the community for valuable feedback. MLlib's continued success is due to your hard work!

Pipeline persistence

After training an ML model, users often need to deploy it in production on another cluster or system. Model persistence allows users to save a model trained on one system and load that model on a separate production system. With MLlib 1.6, models and even entire pipelines can be saved and loaded.

This new persistence functionality is part of the Pipelines API, which integrates with DataFrames and provides tools for constructing ML workflows. To refresh on Pipelines, see our original blog post. With this latest release, a user can train a pipeline, save it, and reload exactly the same pipeline at a later time or on another Spark cluster. Users can also persist untrained pipelines and individual models.

Saving and loading a full pipeline may be done with single lines of code:

```
val model = pipeline.fit(data)
model.save("s3://my-location/myModel")
val sameModel = PipelineModel.load("s3://my-location/
myModel")
```

To learn more, check out a simple example in this notebook. Persistence is available in Scala and Java, and Python support will be added in the next release.

New ML algorithms

MLlib 1.6 adds several new ML algorithms for important application areas.

- **Survival analysis** has many applications, such as modeling and predicting customer churn. (How long will a customer stay with our product, and what can be done to increase that lifetime?) MLlib now has log-linear models for survival analysis.

- **Streaming hypothesis testing** can be used for A/B testing to choose between models or to do canary testing of a new model. We now provide testing using the Spark Streaming framework.

- **Summary statistics** in DataFrames help users to quickly understand their data. Spark 1.6 adds new statistics such as variance, standard deviation, correlations, skewness, and kurtosis.

- **Bisecting k-means clustering** is an accelerated clustering algorithm, useful for identifying patterns and groups within unlabeled data. See an example here.

- **New feature transformers** in MLlib 1.6 include ChiSqSelector (feature selection), QuantileDiscretizer (adaptive discretization of features), and SQLTransformer (SQL operations within ML Pipelines).

See the corresponding sections in the ML User Guide for examples.

ML in SparkR

Spark 1.5 introduced MLlib in SparkR with Generalized Linear Models (GLMs) as described in the blog post Generalized Linear Models in SparkR and R Formula Support in MLlib. These efforts give R users access to distributed Machine Learning algorithms, using familiar APIs. Spark 1.6 expands this functionality with two key features.

Model summary: R users who inspect Spark Linear Regression models using `summary(model)` will see more statistics, including deviance residuals and coefficient standard errors, t-values, and p-values.

Feature interactions: R users can build more expressive GLMs using feature interactions (using the R ":" operator).

The code snippet below demonstrates these new features; see the full example in this notebook.

```
# Create a Spark Dataframe from the "iris" dataset
df <- createDataFrame(sqlContext, iris)

# Fit a model using a new interaction feature created from base features.
model2 <- glm(Sepal_Length ~ Sepal_Width : Species, data = df, family =
"gaussian")

# Summarize the model, just as with native R models
summary(model2)

$devianceResiduals
Min       Max
-1.167923 1.523675

$coefficients
                                Estimate  Std. Error t value  Pr(>|t|)
(Intercept)                     3.357888  0.3300034  10.17531 0
Sepal_Width:Species_versicolor 0.9299109 0.1197601  7.764782 1.317835e-12
Sepal_Width:Species_virginica  1.084014  0.1116593  9.70823  0
Sepal_Width:Species_setosa     0.4832626 0.09682807 4.990935 1.68765e-06
```

Model summary statistics and feature interactions in R formulae are also available from the Scala, Java, and Python APIs.

Testable example code (for developers)

For developers, one of the most useful additions to MLlib 1.6 is testable example code. The code snippets in the user guide can now be tested more easily, which helps to ensure examples do not break across Spark versions.

Specifically, pieces of code in the "examples" folder can be inserted into the user guide automatically. This means that scripts can check to ensure that examples compile and run, rather than requiring developers to check the user guide examples manually. Thanks to the many contributors to this effort!

Looking ahead

We hope to have more MLlib contributors than ever during the upcoming release cycle.

Given very positive feedback about ML Pipelines, we will continue to expand and improve upon this API. Python support for Pipeline persistence is a top priority.

Generalized Linear Models (GLMs) and the R API are key features for data scientists. In future releases, we plan to extend functionality with more models families and link functions, better model inspection, and more MLlib algorithms available in R.

For more details on the roadmap, please see the MLlib 2.0 Roadmap JIRA. We also recommend that users follow Spark Packages, where many new ML algorithms are hosted.

We always welcome new contributors! To get started, check out the MLlib 2.0 Roadmap JIRA and the Wiki on Contributing to Spark.

Learning more

For details on the MLlib 1.6 release, see the release notes.
For guides, examples, and API docs, see the MLlib User Guide and the Pipelines guide.

Good luck, and thanks for your continued support of MLlib!

Section 2:
Advanced Topics: Clustering, Trees, Graph Processing, and More

Distributing the Singular Value Decomposition with Spark

July 21, 2014 | by Li Pu and Reza Zadeh

Guest post by Li Pu from Twitter and Reza Zadeh from Databricks on their recent contribution to Spark's machine learning library.

Singular Value Decomposition (SVD) is one of the cornerstones of linear algebra and has widespread application in many real-world modeling situations. Problems such as recommender systems, linear systems, least squares, and many others can be solved using the SVD. It is frequently used in statistics where it is related to principal component analysis (PCA) and to correspondence analysis, and in signal processing and pattern recognition. Another usage is latent semantic indexing in natural language processing.

Decades ago, before the rise of distributed computing, computer scientists developed the single-core ARPACK package for computing the eigenvalue decomposition of a matrix. Since then, the package has matured and is in widespread use by the numerical linear algebra community, in tools such as SciPy, GNU Octave, and MATLAB.

An important feature of ARPACK is its ability to allow for arbitrary matrix storage formats. This is possible because it doesn't operate on the matrices directly, but instead acts on the matrix via prespecified operations, such as matrix-vector multiplies. When a matrix operation is required, ARPACK gives control to the calling program with a request for a matrix-vector multiply. The calling program must then perform the multiply and return the result to ARPACK.

By using the distributed-computing power of Spark, we can distribute the matrix-vector multiplies, and thus exploit the years of numerical computing expertise that have gone into building ARPACK, and the years of distributing computing expertise that have gone into Spark.

Since ARPACK is written in Fortran77, it cannot immediately be used on the Java Virtual Machine. However, through the netlib-java and breeze interfaces, we can use ARPACK on the JVM. This also means that low-level hardware optimizations can be exploited for any local linear algebraic operations. As with all linear algebraic operations within MLlib, we use hardware acceleration whenever possible.

We are building the linear algebra capabilities of MLlib. Currently in Spark 1.0 there is support for Tall and Skinny SVD and PCA, and as of Spark 1.1, we will have support for SVD and PCA via ARPACK.

Example Runtime

A very popular matrix in the recommender systems community is the Netflix Prize Matrix. The matrix has 17,770 rows, 480,189 columns, and 100,480,507 non-zeros. Below we report results on several larger matrices (up to 16x larger) we experimented with at Twitter.

With the Spark implementation of SVD using ARPACK, calculating wall-clock time with 68 executors and 8GB memory in each, looking for the top 5 singular vectors, we can factorize larger matrices distributed in RAM across a cluster, in a few seconds.

Matrix size	Number of nonzeros	Time per iteration (s)	Total time (s)
23,000,000 x 38,000	51,000,000	0.2	10
63,000,000 x 49,000	440,000,000	1	50
94,000,000 x 4,000	1,600,000,000	0.5	50

Apart from being fast, SVD is also easy to run. Here is a code snippet showing how to run it on sparse data loaded from a text file.

Efficient Similarity Algorithm Now in Spark, Thanks to Twitter

October 20, 2014 | by Reza Zadeh

Our friends at Twitter have contributed to MLlib, and this post uses material from Twitter's description of its open-source contribution, with permission. The associated pull request was released in Spark 1.2.

Introduction

We are often interested in finding users, hashtags and ads that are very similar to one another, so they may be recommended and shown to users and advertisers. To do this, we must consider many pairs of items, and evaluate how "similar" they are to one another.

We call this the "all-pairs similarity" problem, sometimes known as a "similarity join." We have developed a new efficient algorithm to solve the similarity join called "Dimension Independent Matrix Square using MapReduce," or DIMSUM for short, which made one of Twitter's most expensive batch computations 40% more efficient.

To describe the problem we're trying to solve more formally, when given a dataset of sparse vector data, the all-pairs similarity problem is to find all similar vector pairs according to a similarity function such as cosine similarity, and a given similarity score threshold.

Not all pairs of items are similar to one another, and yet a naive algorithm will spend computational effort to consider even those pairs of items that are not very similar. The brute force approach of considering all pairs of items quickly breaks, since its computational effort scales quadratically.

For example, for a million vectors, it is not feasible to check all roughly trillion pairs to see if they're above the similarity threshold. Having said that, there exist clever sampling techniques to focus the computational effort on only those pairs that are above the similarity threshold, thereby making the problem feasible. We've developed the DIMSUM sampling scheme to focus the computational effort on only those pairs that are highly similar, thus making the problem feasible.

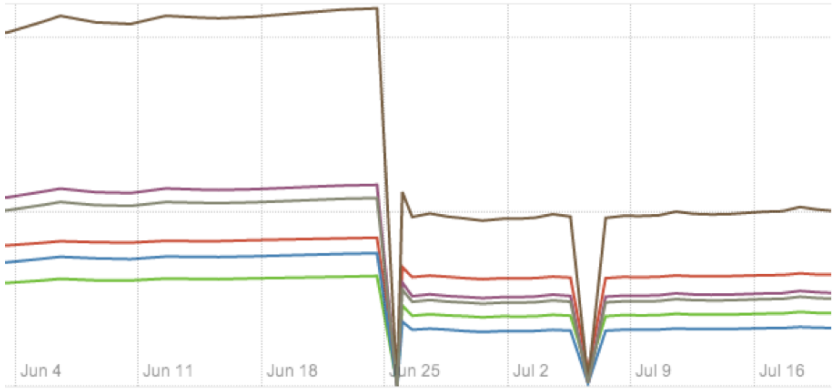
Intuition

The main insight that allows gains in efficiency is sampling columns that have many non-zeros with lower probability. On the flip side, columns that have fewer non-zeros are sampled with higher probability. This sampling scheme can be shown to provably accurately estimate cosine similarities, because those columns that have many non-zeros have more trials to be included in the sample, and thus can be sampled with lower probability.

There is an in-depth description of the algorithm on the Twitter Engineering blog post.

Experiments

We run DIMSUM on a production-scale ads dataset. Upon replacing the traditional cosine similarity computation in late June, we observed 40% improvement in several performance measures, plotted below.



Usage from Spark

The algorithm is available in MLlib as a method in RowMatrix. This makes it easy to use and access:

```
// Arguments for input and threshold
val filename = args(0)
val threshold = args(1).toDouble

// Load and parse the data file.
val rows = sc.textFile(filename).map { line =>
  val values = line.split(' ').map(_.toDouble)
  Vectors.dense(values)
}
val mat = new RowMatrix(rows)

// Compute similar columns perfectly, with brute force.
val simsPerfect = mat.columnSimilarities()

// Compute similar columns with estimation using DIMSUM
val simsEstimate = mat.columnSimilarities(threshold)
```

Here is an example invocation of DIMSUM. This functionality will be available as of Spark 1.2.

Additional information can be found in the GigaOM article covering the DIMSUM algorithm.

Scalable Decision Trees in MLlib

September 29, 2014 | by Manish Amde and Joseph Bradley

Get the Notebook

This is a post written together with one of our friends at Origami Logic. Origami Logic provides a Marketing Intelligence Platform that uses Spark for heavy lifting analytics work on the backend.

Decision trees and their ensembles are industry workhorses for the machine learning tasks of classification and regression. Decision trees are easy to interpret, handle categorical and continuous features, extend to multi-class classification, do not require feature scaling and are able to capture non-linearities and feature interactions.

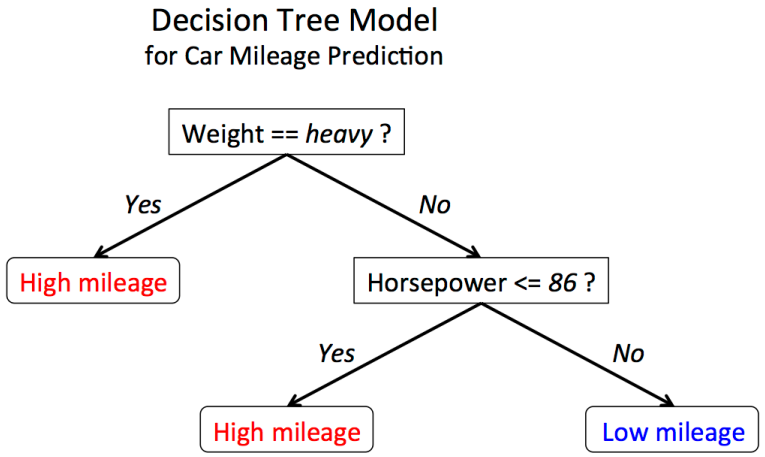
Due to their popularity, almost every machine learning library provides an implementation of the decision tree algorithm. However, most are designed for single-machine computation and seldom scale elegantly to a distributed setting. Spark is an ideal platform for a scalable distributed decision tree implementation since Spark's in-memory computing allows us to efficiently perform multiple passes over the training dataset.

About a year ago, open-source developers joined forces to come up with a fast distributed decision tree implementation that has been a part of the Spark MLlib library since release 1.0. The Spark community has actively improved the decision tree code since then. This blog post describes the implementation, highlighting some of the important optimizations and presenting test results demonstrating scalability.

New in Spark 1.1: MLlib decision trees now support multiclass classification and include several performance optimizations. There are now APIs for Python, in addition to Scala and Java.

Algorithm Background

At a high level, a decision tree model can be thought of as hierarchical if-else statements that test feature values in order to predict a label. An example model for a binary classification task is shown below. It is based upon car mileage data from the 1970s! It predicts the mileage of the vehicle (high/low) based upon the weight (heavy/light) and the horsepower.



A model is learned from a training dataset by building a tree top-down. The if-else statements, also known as splitting criteria, are chosen to maximize a notion of information gain — it reduces the variability of the labels in the underlying (two) child nodes compared the parent node. The learned decision tree model can later be used to predict the labels for new instances.

These models are interpretable, and they often work well in practice. Trees may also be combined to build even more powerful models, using ensemble tree algorithms. Ensembles of trees such as random forests and boosted trees are often top performers in industry for both classification and regression tasks.

Simple API

The following example shows how a decision tree in MLlib can be easily trained using a few lines of code using the new Python API in Spark 1.1. It reads a dataset, trains a decision tree model and then measures the training error of the model. Java and Scala examples can be found in the Spark documentation on DecisionTree.

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.tree import DecisionTree
from pyspark.mllib.util import MLUtils

# Load and parse the data file into an RDD of LabeledPoint.
# Cache data since we will use it again to compute training error.
data = MLUtils.loadLibSVMFile(sc, 'mydata.txt').cache()

# Train a DecisionTree model.
model = DecisionTree.trainClassifier(
    data,
    numClasses=2,
    categoricalFeaturesInfo={}, # all features are continuous
    impurity='gini',
    maxDepth=5,
    maxBins=32)

# Evaluate model on training instances and compute training error
predictions = model.predict(data.map(lambda x: x.features))
labelsPredictions = data.map(lambda lp: lp.label).zip(predictions)
trainErr = labelsPredictions.filter(lambda (v, p): v != p).count() \
    / float(data.count())
print('Training Error = ' + str(trainErr))
print('Learned classification tree model:')
print(model)
```

Optimized Implementation

Spark is an ideal compute platform for a scalable distributed decision tree implementation due to its sophisticated DAG execution engine and in-memory caching for iterative computation. We mention a few key optimizations.

Level-wise training: We select the splits for all nodes at the same level of the tree simultaneously. This level-wise optimization reduces the number of passes over the dataset exponentially: we make one pass for each level, rather than one pass for each node in the tree. It leads to significant savings in I/O, computation and communication.

Approximate quantiles: Single machine implementations typically use sorted unique feature values for continuous features as split candidates for the best split calculation. However, finding sorted unique values is an expensive operation over a distributed dataset. The MLlib decision tree uses quantiles for each feature as split candidates. It's a standard tradeoff for improving decision tree performance without significant loss of accuracy.

Avoiding the map operation: The early prototype implementations of the decision tree used both map and reduce operations when selecting best splits for tree nodes. The current code uses significantly less computation and communication by exploiting the known structure of the pre-computed split candidates to avoid the map step.

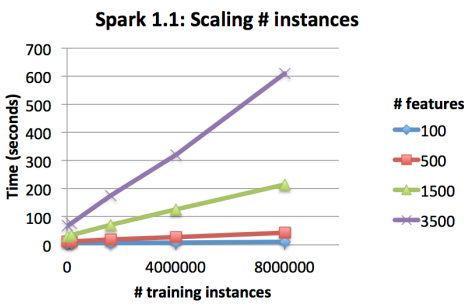
Bin-wise computation: The best split computation discretizes features into bins, and those bins are used for computing sufficient statistics for splitting. We precompute the binned representations of each instance, saving computation on each iteration.

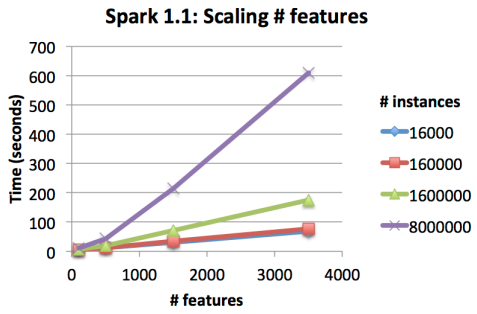
Scalability

We demonstrate the scalability of MLlib decision trees with empirical results on various datasets and cluster sizes.

Scaling with dataset size

The two figures below show the training times of decision trees as we scale the number of instances and features in the dataset. The training times increased linearly, highlighting the scalability of the implementation.

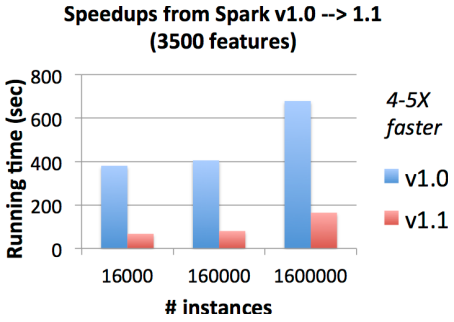


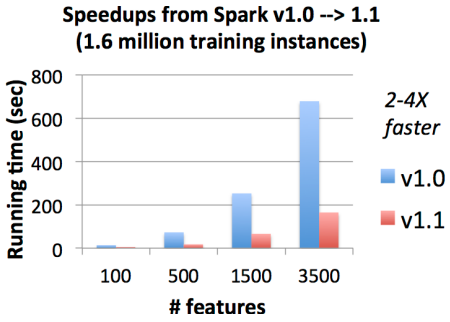


These tests were run on an EC2 cluster with a master node and 15 worker nodes, using r3.2xlarge instances (8 virtual CPUs, 61 GB memory). The trees were built out to 6 levels, and the datasets were generated by the spark-perf library.

Spark 1.1 speedups

The next two figures show improvements in Spark 1.1, relative to the original Spark 1.0 implementation. On the same datasets and cluster, the new implementation is 4-5X faster on many datasets!





What's Next?

The tree-based algorithm development beyond release 1.1 will focus primarily on ensemble algorithms such as random forests and boosting. We will also keep optimizing the decision tree code for performance and plan to add support for more options in the upcoming releases.

Further Reading

- See examples and the API in the MLlib decision tree documentation.
- Watch the decision tree presentation from the 2014 Spark Summit.
- Check out video and slides from another talk on decision trees at a Sept. 2014 SF Scala/Bay Area Machine Learning meetup.

Acknowledgements

The Spark MLlib decision tree work was initially performed jointly with Hirakendu Das (Yahoo Labs), Evan Sparks (UC Berkeley AMPLab), and Ameet Talwalkar and Xiangrui Meng (Databricks). More contributors have joined since then, and we welcome your input too!

Get the Notebook

Random Forests and Boosting in MLlib

January 21, 2015 | by Joseph Bradley and Manish Amde

Get the Notebook

This is a post written together with Manish Amde from Origami Logic.

Spark 1.2 introduced Random Forests and Gradient-Boosted Trees (GBTs) into MLlib. Suitable for both classification and regression, they are among the most successful and widely deployed machine learning methods. Random Forests and GBTs are *ensemble learning algorithms*, which combine multiple decision trees to produce even more powerful models. In this post, we describe these models and the distributed implementation in MLlib. We also present simple examples and provide pointers on how to get started.

Ensemble Methods

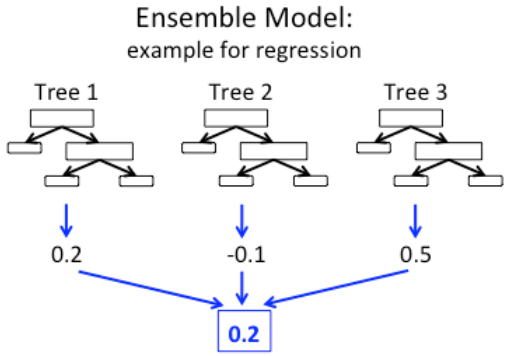
Simply put, ensemble learning algorithms build upon other machine learning methods by combining models. The combination can be more powerful and accurate than any of the individual models.

In MLlib 1.2, we use Decision Trees as the base models. We provide two ensemble methods: Random Forests and Gradient-Boosted Trees (GBTs). The main difference between these two algorithms is the order in which each component tree is trained.

Random Forests train each tree independently, using a random sample of the data. This randomness helps to make the model more robust than a single decision tree, and less likely to overfit on the training data.

GBTs train one tree at a time, where each new tree helps to correct errors made by previously trained trees. With each tree added, the model becomes even more expressive.

In the end, both methods produce a weighted collection of Decision Trees. The ensemble model makes predictions by combining results from the individual trees. The figure below shows a simple example of an ensemble with three trees.



In this example regression ensemble, each tree predicts a real value. These three predictions are then combined to produce the ensemble's final prediction. Here, we combine predictions using the mean (but the algorithms use different techniques depending on the prediction task).

Distributed Learning of Ensembles

In MLlib, both Random Forests and GBTs partition data by instances (rows). The implementation builds upon the original Decision Tree code, which distributes learning of single trees (described in an earlier blog post). Many of our optimizations are based upon Google's PLANET project, one of the major published works on learning ensembles of trees in the distributed setting.

Random Forests: Since each tree in a Random Forest is trained independently, multiple trees can be trained in parallel (in addition to the parallelization for single trees). MLlib does exactly that: A variable number of sub-trees are trained in parallel, where the number is optimized on each iteration based on memory constraints.

GBTs: Since GBTs must train one tree at a time, training is only parallelized at the single tree level.

We would like to highlight two key optimizations used in MLlib:

- **Memory**: Random Forests use a different subsample of the data to train each tree. Instead of replicating data explicitly, we save memory by using a TreePoint structure which stores the number of replicas of each instance in each subsample.

- **Communication**: Whereas Decision Trees are usually trained by selecting from all features at each decision node in the tree, Random Forests often limit the selection to a random subset of features at each node. MLlib's implementation takes advantage of this subsampling to reduce communication: e.g., if only 1/3 of the features are used at each node, then we can reduce communication by a factor of 1/3.

For more details, see the Ensembles Section in the MLlib Programming Guide.

Using MLlib Ensembles

We demonstrate how to learn ensemble models using MLlib. The following Scala examples show how to read in a dataset, split the data into training and test sets, learn a model, and print the model and its test accuracy. Refer to the MLlib Programming Guide for examples in Java and Python. Note that GBTs do not yet have a Python API, but we expect it to be in the Spark 1.3 release (via Github PR 3951).

Random Forest Example

```
import org.apache.spark.mllib.tree.RandomForest
import org.apache.spark.mllib.tree.configuration.Strategy
import org.apache.spark.mllib.util.MLUtils

// Load and parse the data file.
val data =
  MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// Split data into training/test sets
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))

// Train a RandomForest model.
val treeStrategy = Strategy.defaultStrategy("Classification")
val numTrees = 3 // Use more in practice.
val featureSubsetStrategy = "auto" // Let the algorithm choose.
val model = RandomForest.trainClassifier(trainingData,
  treeStrategy, numTrees, featureSubsetStrategy, seed = 12345)

// Evaluate model on test instances and compute test error
val testErr = testData.map { point =>
  val prediction = model.predict(point.features)
  if (point.label == prediction) 1.0 else 0.0
}.mean()
println("Test Error = " + testErr)
println("Learned Random Forest:n" + model.toDebugString)
```

Gradient-Boosted Trees Example

```
import org.apache.spark.mllib.tree.GradientBoostedTrees
import org.apache.spark.mllib.tree.configuration.BoostingStrategy
import org.apache.spark.mllib.util.MLUtils

// Load and parse the data file.
val data =
  MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// Split data into training/test sets
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))

// Train a GradientBoostedTrees model.
val boostingStrategy =
  BoostingStrategy.defaultParams("Classification")
boostingStrategy.numIterations = 3 // Note: Use more in practice
val model =
  GradientBoostedTrees.train(trainingData, boostingStrategy)

// Evaluate model on test instances and compute test error
val testErr = testData.map { point =>
  val prediction = model.predict(point.features)
  if (point.label == prediction) 1.0 else 0.0
}.mean()
println("Test Error = " + testErr)
println("Learned GBT model:n" + model.toDebugString)
```

Scalability

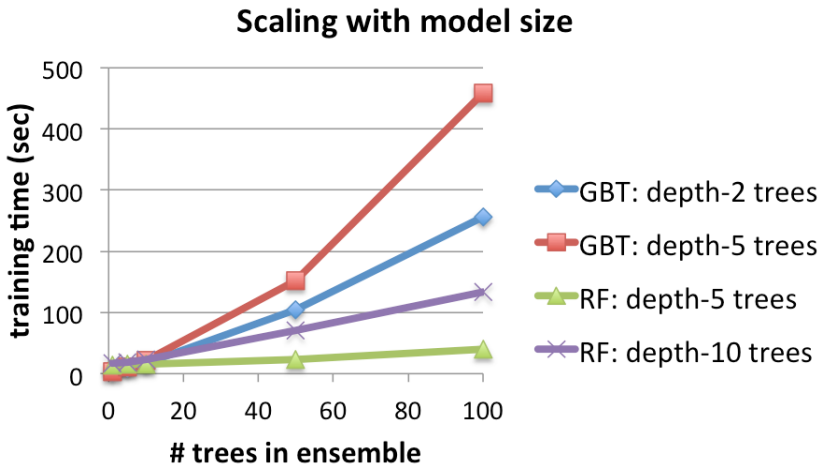
We demonstrate the scalability of MLlib ensembles with empirical results on a binary classification problem. Each figure below compares Gradient-Boosted Trees ("GBT") with Random Forests ("RF"), where the trees are built out to different maximum depths.

These tests were on a regression task of predicting song release dates from audio features (the YearPredictionMSD dataset from the UCI ML repository). We used EC2 r3.2xlarge machines. Algorithm parameters were left as defaults except where noted.

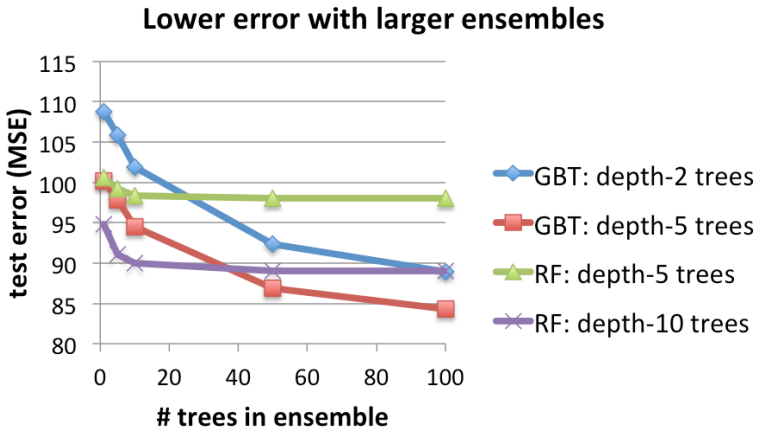
Scaling model size: Training time and test error

The two figures below show the effect of increasing the number of trees in the ensemble. For both, increasing trees require more time to learn (first figure) but also provide better results in terms of test Mean Squared Error (MSE) (second figure).

Comparing the two methods, Random Forests are faster to train, but they often require deeper trees than GBTs to achieve the same error. GBTs can further reduce the error with each iteration, but they can begin to overfit (increase test error) after too many iterations. Random Forests do not overfit as easily, but their test error plateaus.



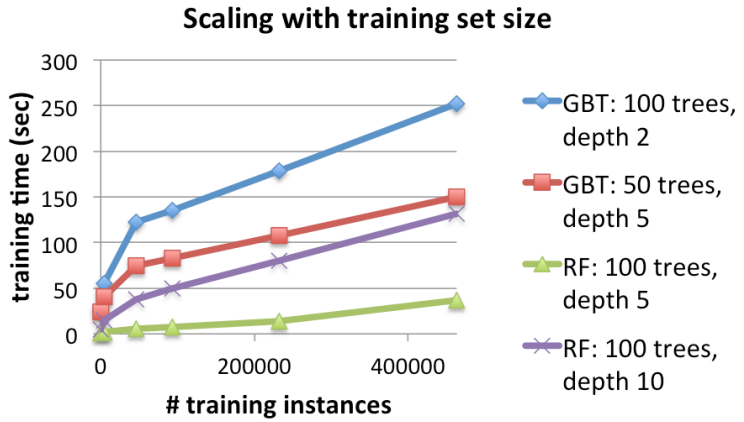
Below, for a basis for understanding the MSE, note that the left-most points show the error when using a single decision tree (of depths 2, 5, or 10, respectively).

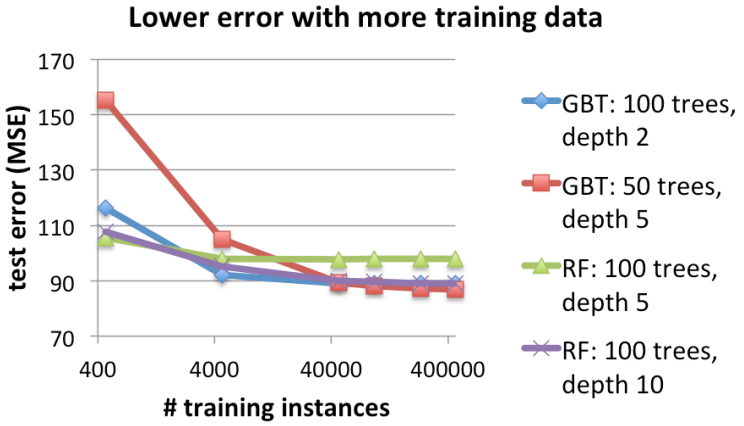


Details: 463,715 Training Instances. 16 Workers.

Scaling training dataset size: Training time and test error

The next two figures show the effect of using larger training datasets. With more data, both methods take longer to train but achieve better test results.

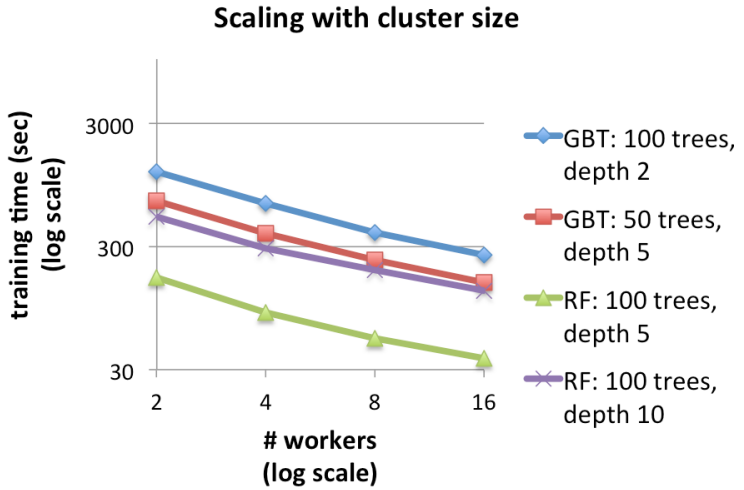




Details: 16 Workers.

Strong scaling: Faster training with more workers

This final figure shows the effect of using a larger compute cluster to solve the same problem. Both methods are significantly faster when using more workers. For example, GBTs with depth-2 trees train about 4.7 times faster on 16 workers than on 2 workers, and larger datasets produce even better speedups.



463,715 Training Instances.

What's Next?

GBTs will soon include a Python API. The other top item for future development is pluggability: ensemble methods can be applied to almost any classification or regression algorithm, not only Decision Trees. The Pipelines API introduced by Spark 1.2's experimental spark.ml package will allow us to generalize ensemble methods to be truly pluggable.

Further Reading

- See examples and the API in the MLlib ensembles documentation.
- Learn more background info about the decision trees used to build ensembles in this previous blog post.

Acknowledgements

MLlib ensemble algorithms have been developed collaboratively by the authors of this blog post, Qiping Li (Alibaba), Sung Chung (Alpine Data Labs), and Davies Liu (Databricks). We also thank Lee Yang, Andrew Feng, and Hirakendu Das (Yahoo) for help with design and testing. We will welcome your contributions too!

Get the Notebook

Introducing Streaming k-means in Spark 1.2

January 28, 2015 | by Jeremy Freeman

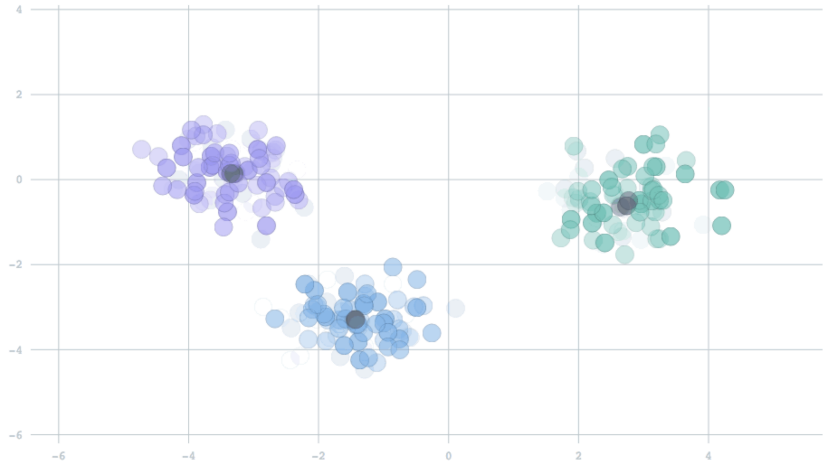
Many real world data are acquired sequentially over time, whether messages from social media users, time series from wearable sensors, or — in a case we are particularly excited about — the firing of large populations of neurons. In these settings, rather than wait for all the data to be acquired before performing our analyses, we can use streaming algorithms to identify patterns over time, and make more targeted predictions and decisions.

One simple strategy is to build machine learning models on static data, and then use the learned model to make predictions on an incoming data stream. But what if the patterns in the data are themselves dynamic? That's where streaming algorithms come in.

A key advantage of Spark is that its machine learning library (MLlib) and its library for stream processing (Spark Streaming) are built on the same core architecture for distributed analytics. This facilitates adding extensions that leverage and combine components in novel ways without reinventing the wheel. We have been developing a family of streaming machine learning algorithms in Spark within MLlib. In this post we describe streaming k-means clustering, included in the recently released Spark 1.2.

Algorithm

The goal of k-means is to partition a set of data points into k clusters. The now classic k-means algorithm — developed by Stephen Lloyd in the 1950s for efficient digital quantization of analog signals — iterates between two steps. First, given an initial set of k cluster centers, we find which cluster each data point is closest to. Then, we compute the average of each of the new clusters and use the result to update our cluster centers. At each of these steps — re-assigning and updating — we are making the points within each cluster more and more similar to one another (more formally, we are in both steps shrinking the within-cluster-sum-of-squares). By iterating between these two steps repeatedly, we can usually converge to a good solution.



See the animated gif here

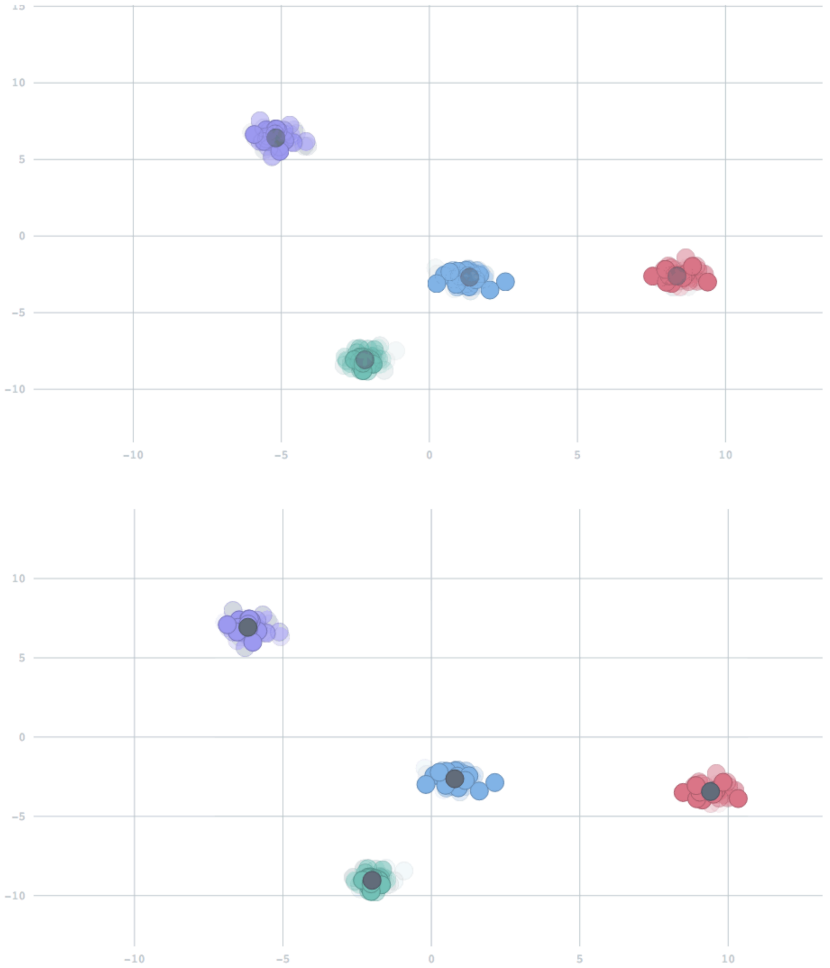
In the streaming setting, our data arrive in batches, with potentially many data points per batch. The simplest extension of the standard k-means algorithm would be to begin with cluster centers — usually random locations, because we haven't yet seen any data — and for each new batch of data points, perform the same two-step operation described above. Then, we use the new centers to repeat the procedure on the next batch. Here is a movie showing the behavior of this algorithm for two-dimensional data streaming from three clusters that are slowly drifting over time. The centers track the true clusters and adapt to the changes over time.

Forgetfulness

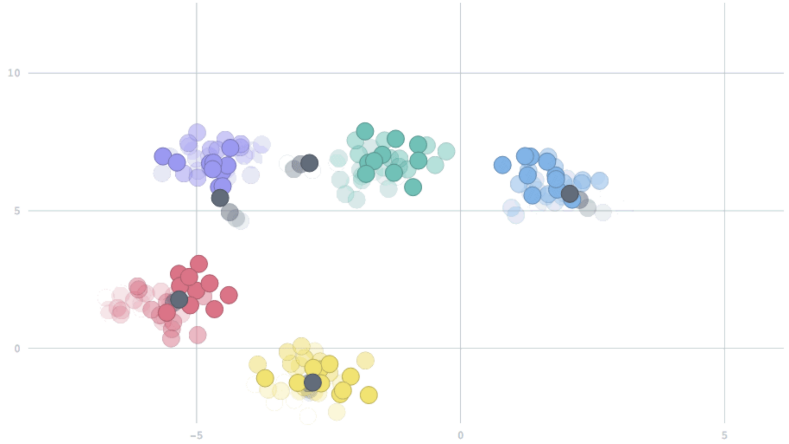
If the source of the data is constant — the same three clusters forever — the above streaming algorithm will converge to a similar solution as if k-means was run offline on the entire accumulated data set. In fact, in this case the streaming algorithm is identical to a well-known offline k-means algorithm, "mini-batch" k-means, which repeatedly trains on random subsets of the data to avoid loading the entire data set into memory.

However, what if the sources of data are changing over time? How can we make our model reflect those changes?

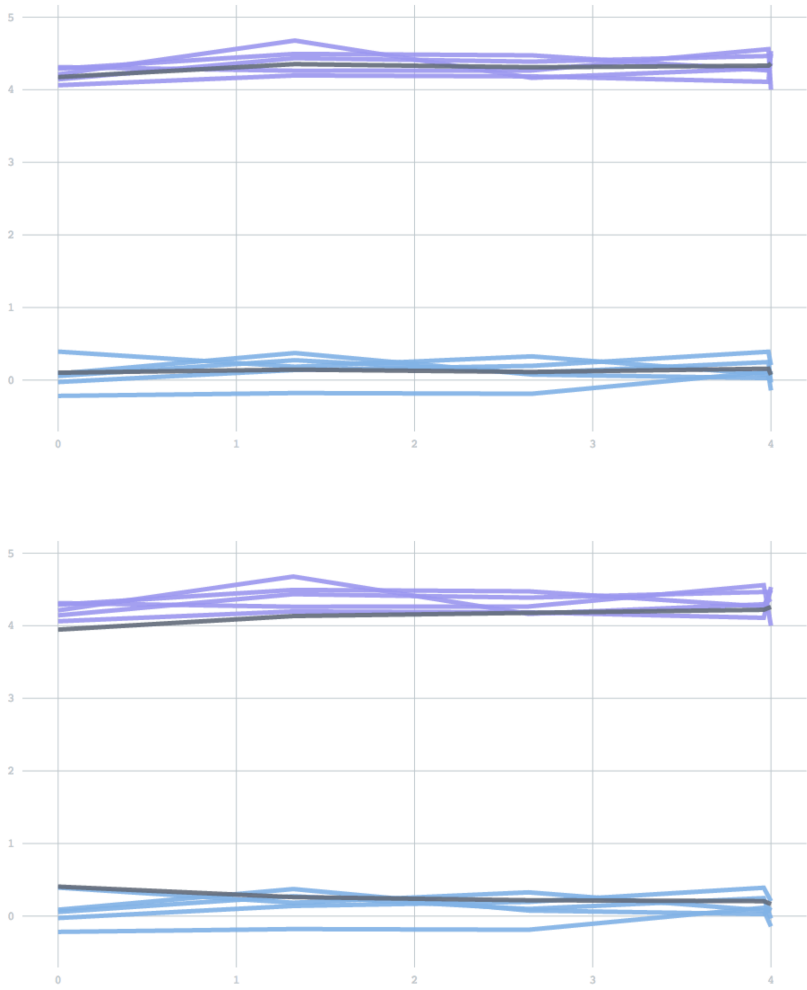
For this setting, we have extended the algorithm to support **forgetfulness**, allowing the model to adapt to changes over time. The key trick is to add a new parameter that balances the relative importance of new data versus past history. One setting of this parameter will be equivalent to the scenario described above, where all data from the beginning of time are treated equally. At the other extreme, only the most recent data will be used. Settings in between will combine the present with a partial reflection of the past. Here is an animation showing two settings of this forgetfulness parameter, in streams where the centers change half-way through. Watch how the cluster centers quickly adjust to the new locations in the second case, but take a while to shift in the first.



With the appropriate setting of the parameter, we can have cluster centers that smoothly adapt to dynamic changes in the data. In this animation, watch five clusters drift over time, and the centers track them.



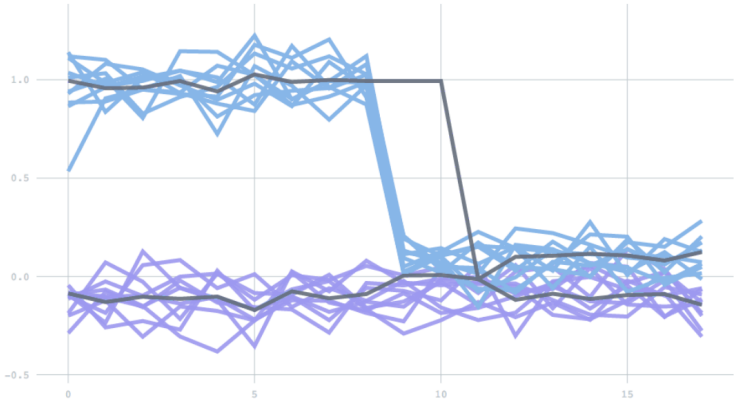
Mathematically, forgetfulness amounts to adding an extra parameter to the **update rule**: the equation describing how to update centers given a new batch of data. However, as a scalar value between 0 to 1, it is not a particularly intuitive parameter. So instead, we expose a **half-life**, which describes the time it takes before past data contributes to only one half of the current model. To demonstrate, we'll use a one-dimensional version of the examples above. We start with data drawn from two clusters, and then switch to data from two different clusters. The half life determines how many batches it will take for the contribution from the initial set of points to reduce to half. You can see the effect of changing the half-life in the time it takes for the clusters adjust. With a half-life of 0.5 batches the change finishes in about 1 batch, but with a half-life of 5 it takes about 10 batches.



See the animated gif here

Users may want to think about their half-life in terms of either the number of batches (which have a fixed duration in time), or the number of points. If you have 1000 data points in one batch and 10 in the other, perhaps you want those 1000 to have a proportionately larger impact. On the other hand, you might want to remain stable across fluctuations in data points, and instead treat all periods of time equally. To solve this, we've introduced the concept of a **time unit** that can be specified as either batches or points. Given a user-specified half life and time unit, the algorithm automatically calculates the appropriate forgetfulness behavior.

A final feature included is a check to eliminate **dying** clusters. If there is a dramatic change in the data generating process, one of the estimated clusters may suddenly be far from any data, and stay stuck in its place. To prevent this scenario, clusters are checked for such behavior on each batch. A cluster detected as dying is eliminated, and the largest cluster is split in two. In this one-dimensional demo, two clusters are initially far apart, but then one changes to be much closer to the other. At first the incorrect cluster persists (top line), but soon it disappears, and the other cluster splits to correctly lock on to the new cluster centers.



See the animated gif here

Getting started

To get started using streaming k-means yourself, read more about streaming k-means and try the example code. To generate your own visualizations of streaming clustering like the ones shown here, and explore the range of settings and behaviors, check out the code in the spark-ml-streaming package.

Looking forward

Many algorithms and analyses can benefit from streaming implementations. Along with streaming linear regression (as of 1.1) and streaming clustering (as of 1.2), we plan to add streaming versions of of factorization and classification in future releases, incorporate them into the new Python Streaming API, and use our new forgetfulness parameterization across the algorithms as a unified way to control dynamic model updating.

Special thanks to Xiangrui Meng, Tathagata Das, and Nicholas Sofroniew (for work on algorithm development) and Matthew Conlen (for visualizations).

Topic modeling with LDA: MLlib meets GraphX

March 25, 2015 | by Joseph Bradley

An updated version of this gist can be found in the "Topic Modeling with Latent Dirichlet Allocation" notebook here.

Topic models automatically infer the topics discussed in a collection of documents. These topics can be used to summarize and organize documents, or used for featurization and dimensionality reduction in later stages of a Machine Learning (ML) pipeline.

With Spark 1.3, MLlib now supports *Latent Dirichlet Allocation (LDA)*, one of the most successful topic models. LDA is also the first MLlib algorithm built upon GraphX. In this blog post, we provide an overview of LDA and its use cases, and we explain how GraphX was a natural choice for implementation.

Topic Models

At a high level, topic modeling aims to find structure within an unstructured collection of documents. After learning this "structure," a topic model can answer questions such as: What is document X discussing? How similar are documents X and Y? If I am interested in topic Z, which documents should I read first?

LDA

Topic modeling is a very broad field. Spark 1.3 adds Latent Dirichlet Allocation (LDA), arguably the most successful topic model to date. Initially developed for both text analysis and population genetics, LDA has since been extended and used in many applications from time series to image analysis. First, let's describe LDA in terms of text analysis.

What are topics? LDA is not given topics, so it must infer them from raw text. LDA defines a topic as a distribution over words. For example, when we run MLlib's LDA on a dataset of articles from 20 newsgroups, the first few topics are:

Topic 1		Topic 2		Topic 3	
term	**weight**	**term**	**weight**	**term**	**weight**
game	0.014	space	0.021	drive	0.021
team	0.011	nasa	0.006	card	0.015
hockey	0.009	earth	0.006	system	0.013
play	0.008	henry	0.005	scsi	0.012
games	0.007	launch	0.004	hard	0.011

Looking at these highest-weighted words in 3 topics, we can quickly understand what each topic is about: sports, space exploration, and computers. LDA's success largely stems from its ability to produce interpretable topics.

Use Cases

In addition to inferring these topics, LDA infers a distribution over topics for each document. E.g., document X might be 60% about "space exploration," 30% about "computers" and 10% about other topics.

These topic distributions can be used in many ways:

- **Clustering**: Topics are cluster centers and documents are associated with multiple clusters (topics). This clustering can help organize or summarize document collections.
    - See this generated summary of *Science* articles from Prof. Blei and Lafferty. Click on a topic to see a list of articles about that topic.
- **Feature generation**: LDA can generate features for other ML algorithms to use. As mentioned above, LDA infers a distribution over topics for each document; with K topics, this gives K numerical features. These features can then be plugged into algorithms such as Logistic Regression or Decision Trees for prediction tasks.
- **Dimensionality reduction:** Each document's distribution over topics gives a concise summary of the document. Comparing documents in this reduced feature space can be more meaningful than comparing in the original feature space of words.

Using LDA in MLlib

We give a short example of using LDA. We describe the process here and provide the actual code in this Github gist*. Our example first loads and pre-processes documents. The most important part of preprocessing is choosing a vocabulary. In our example, we split text into terms (words) and then remove (a) non-alphabetic terms, (b) short terms with < 4 characters, and (c) the most common 20 terms (as "stopwords"). In general, it is important to adjust this preprocessing for your particular dataset.

We then run LDA using 10 topics and 10 iterations. It is often important to choose the number of topics based on your dataset. Using other options as defaults, we train LDA on the Spark documentation Markdown files (spark/docs/*.md).

We end up with 10 topics. Here are 5 hand-picked topics, each with its most important 5 terms. Note how each corresponds nicely to a component of Spark! (The quoted topic titles were added by hand to make this clear).

"Spark Core"		"GraphX"		"MLlib"		"SQL"		"Streaming"	
term	weight	term	weight	term	weight	term	weight	term	weight
cluster	0.014	graph	0.029	model	0.023	schema	0.012	kinesis	0.014
mesos	0.013	vertex	0.018	training	0.021	dataframe	0.01	more	0.009
driver	0.008	vertices	0.013	features	0.014	table	0.01	streaming	0.009
executor	0.008	edges	0.011	feature	0.012	hive	0.009	java	0.008
executors	0.008	graphx	0.009	load	0.01	create	0.009	dstream	0.007

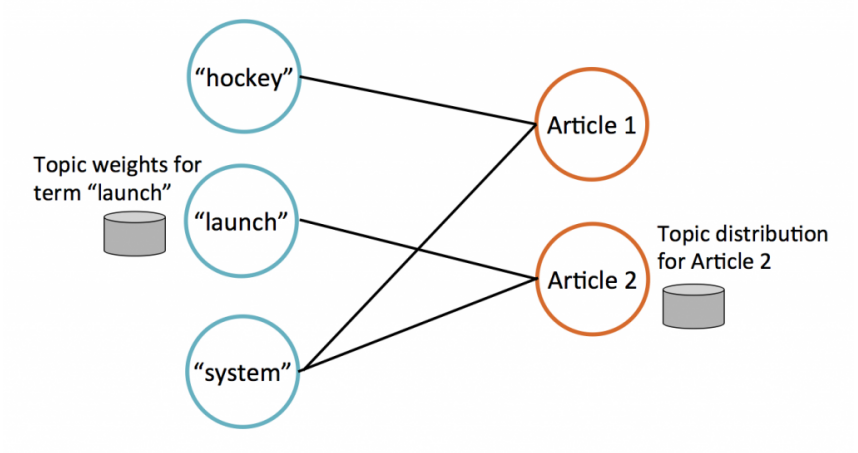
LDA has Scala and Java APIs in Spark 1.3. The Python API will be added soon.

The gist code was designed for earlier versions of Spark. An updated version of this code can be found in the "Topic Modeling with Latent Dirichlet Allocation" notebook here.

Implementation: GraphX

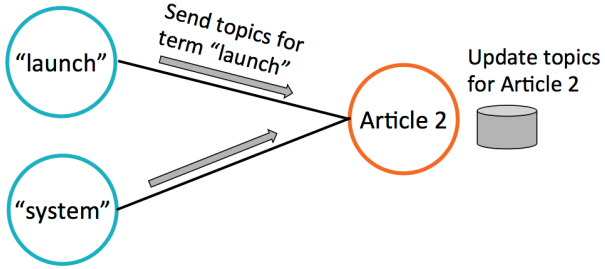
There are many algorithms for learning an LDA model. We chose Expectation-Maximization (EM) for its simplicity and fast convergence. Because EM for LDA has an implicit graphical structure, building LDA on top of GraphX was a natural choice.

LDA has 2 main types of data: terms (words) and documents. We store this data on a bipartite graph (illustrated below) which has term vertices (left) and document vertices (right). Each term vertex stores weights indicating which topics that term is relevant to; likewise, each document vertex stores its current estimate of the topics discussed in the document.



Whenever a term appears in a document, the graph has an edge between the corresponding term vertex and document vertex. E.g., in the figure above, Article 1 contains the terms "hockey" and "system."

These edges also illustrate the algorithm's communication. On each iteration, every vertex updates its data (topic weights) by collecting data from its neighbors. Below, Article 2 updates its topic estimates by collecting data from connected term vertices.



GraphX was thus a natural choice for LDA. As MLlib grows, we expect more graph-structured learning algorithms in the future!

Scalability

Parallelization of LDA is not straightforward, and there have been many research papers proposing different strategies. The key problem is that all methods involve a large amount of communication. This is evident in the graph description above: terms and documents need to update their neighbors with new data on each iteration, and there are *many* neighbors.

We chose the Expectation-Maximization algorithm partly because it converges to a solution in a small number of iterations. Fewer iterations means less communication.

Before adding LDA to Spark, we ran tests on a large Wikipedia dataset. Here are the numbers:

- Training set size: 4.6 million documents
- Vocabulary size: 1.1 million terms
- Training set size: 1.1 billion tokens (~239 words/document)
- 100 topics
- 16-worker EC2 cluster
- Timing results: 176 sec/iteration on average over 10 iterations

What's Next?

Spark contributors are currently developing additional LDA algorithms: online Variational-Bayes (a fast approximate algorithm) and Gibbs sampling (a slower but sometimes more accurate algorithm). We are also adding helper infrastructure such as Tokenizers for automatic data preparation and more prediction functionality.

To see examples and learn the API details, check out the MLlib documentation.

Acknowledgements

The development of LDA has been a collaboration between many Spark contributors:

Joseph K. Bradley, Joseph Gonzalez, David Hall, Guoqiang Li, Xiangrui Meng, Pedro Rodriguez, Avanesov Valeriy, and Xusen Yin.

Additional resources

Learn more about topic models and LDA with these overviews:

- Overview of topic models: D. Blei and J. Lafferty. "Topic Models." In A. Srivastava and M. Sahami, editors, Text Mining: Classification, Clustering, and Applications. Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, 2009.

- Wikipedia on LDA, with mathematical details

Get in-depth background from these research papers:

- Original LDA papers
  - Blei, Ng, and Jordan. "Latent Dirichlet Allocation." JMLR, 2003.
- Application: text document analysis
  - Pritchard et al. "Inference of population structure using multilocus genotype data." Genetics 155: 945–959, 2000.
    - Application: population genetics analysis
- Paper which clearly explains several algorithms, including EM: Asuncion, Welling, Smyth, and Teh. "On Smoothing and Inference for Topic Models." UAI, 2009.

Large Scale Topic Modeling: Improvements to LDA on Spark

September 22, 2015 | by Feynman Liang, Yuhao Yang and Joseph Bradley

This blog was written by Feynman Liang and Joseph Bradley from Databricks, and Yuhao Yang from Intel.

What are people discussing on Twitter? To catch up on distributed computing, what news articles should I read? These are questions that can be answered by topic models, a technique for analyzing the topics present in collections of documents. This blog post discusses improvements in Spark 1.4 and 1.5 for topic modeling using the powerful Latent Dirichlet Allocation (LDA) algorithm.

Spark 1.4 and 1.5 introduced an online algorithm for running LDA incrementally, support for more queries on trained LDA models, and performance metrics such as likelihood and perplexity. We give an example here of training a topic model over a dataset of 4.5 million Wikipedia articles.

Topic models and LDA

Topic models take a collection of documents and automatically infer the topics being discussed. For example, when we run Spark's LDA on a dataset of 4.5 million Wikipedia articles, we can obtain topics like those in the table below.

Topic 1		Topic 2		Topic 3		Topic 4		Topic 5	
president	0.026	district	0.057	world	0.042	company	0.038	airport	0.031
state	0.015	village	0.048	gold	0.036	business	0.017	aircraft	0.019
member	0.011	population	0.038	championships	0.028	management	0.009	engine	0.018
committee	0.011	bar	0.034	silver	0.028	services	0.008	convert	0.016
served	0.010	municipality	0.030	bronze	0.013	companies	0.008	air	0.016

Table 1: Example LDA Topics Learned from Wikipedia Articles Dataset

In addition, LDA tells us which topics each document is about; document X might be 30% about Topic 1 ("politics") and 70% about Topic 5 ("airlines"). Latent Dirichlet Allocation (LDA) has been one of the most successful topic models in practice. See our previous blog post on LDA to learn more.

A new online variational learning algorithm

Online variational inference is a technique for learning an LDA model by processing the data incrementally in small batches. By processing in small batches, we are able to easily scale to very large datasets. MLlib implements an algorithm for performing online variational inference originally described by Hoffman et al.

Performance comparison

The table of topics shown previously were learned using the newly developed online variational learning algorithm. If we compare timing results, we can see a significant speedup in using the new online algorithm over the old EM algorithm:

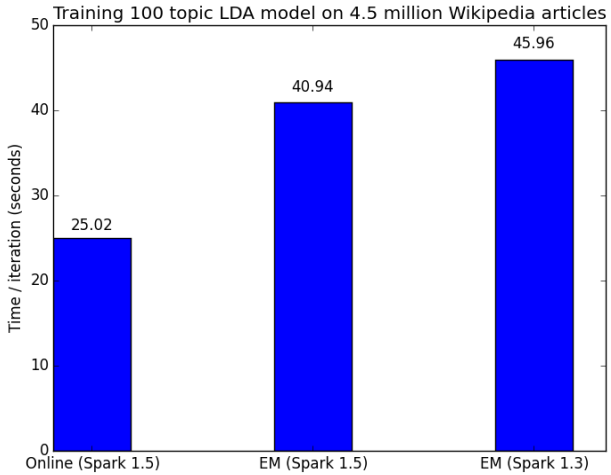


Figure 1: Online Learning Algorithm Learns Faster Than Earlier Em Algorithm

Experiment details

We first preprocessed the data by filtering common English stop words and limiting the vocabulary to the 10,000 most common words. We then trained a 100 topic LDA model for 100 iterations using the online LDA optimizer. We ran our experiments using Databricks on a 16 node AWS r3.2xlarge cluster with data stored in S3. For the actual code, see this Github gist.

Improved predictions, metrics, and queries

Predict topics on new documents

In addition to describing topics present in the training set, Spark 1.5 makes the trained LDA models more useful by allowing users to predict topics for a new test document.

Evaluate your model with likelihood and perplexity

After learning an LDA model, we are often interested in how well the model fits the data. We have added two new metrics to evaluate this: likelihood and perplexity.

Make more queries

This new release also adds several new queries users can perform on a trained LDA model. For example, we can now obtain the top k topics for each document (“What is this document discussing?”) as well as the top documents per topic (“To learn about topic X, what documents should I read?”).

Tips for running LDA

- Make sure to run for enough iterations. Early iterations may return useless (e.g. extremely similar) topics, but running for more iterations dramatically improves the results. We have noticed this is especially true for EM.

- To handle stop words specific to your data, a common workflow is to run LDA, look at topics, identify stop words that show up in the topics, filter them out, and run LDA again.

- Picking the number of topics is an art; there are algorithms to choose automatically, but domain expertise is critical to getting good results.

- The Pipelines API feature transformers are very useful for preprocessing text to prepare it for LDA; see Tokenizer, StopwordsRemover and CountVectorizer in particular.

What's next?

Spark contributors are actively working on improving our LDA implementation. Some works in progress include: Gibbs sampling (a slower but sometimes more accurate algorithm), streaming LDA algorithms, and hierarchical Dirichlet processes (for automatically choosing the number of topics).

Acknowledgements

The development of LDA has been a collaboration between many Spark contributors.

Feynman Liang, Yuhao Yang, Joseph K. Bradley, and others made recent improvements, and many others contributed to the earlier work.

Introducing GraphFrames

March 3, 2016 | by Joseph Bradley, Tim Hunter, Ankur Dave and Xiangrui Meng

Get the Notebook

We would like to thank Ankur Dave from UC Berkeley AMPLab for his contribution to this blog post.

Databricks is excited to announce the release of GraphFrames, a graph processing library for Apache Spark. Collaborating with UC Berkeley and MIT, we have built a graph library based on DataFrames. GraphFrames benefit from the scalability and high performance of DataFrames, and they provide a uniform API for graph processing available from Scala, Java, and Python.

What are GraphFrames?

GraphFrames support general graph processing, similar to Apache Spark's GraphX library. However, GraphFrames are built on top of Spark DataFrames, resulting in some key advantages:

- **Python, Java & Scala APIs:** GraphFrames provide uniform APIs for all 3 languages. For the first time, all algorithms in GraphX are available from Python & Java.

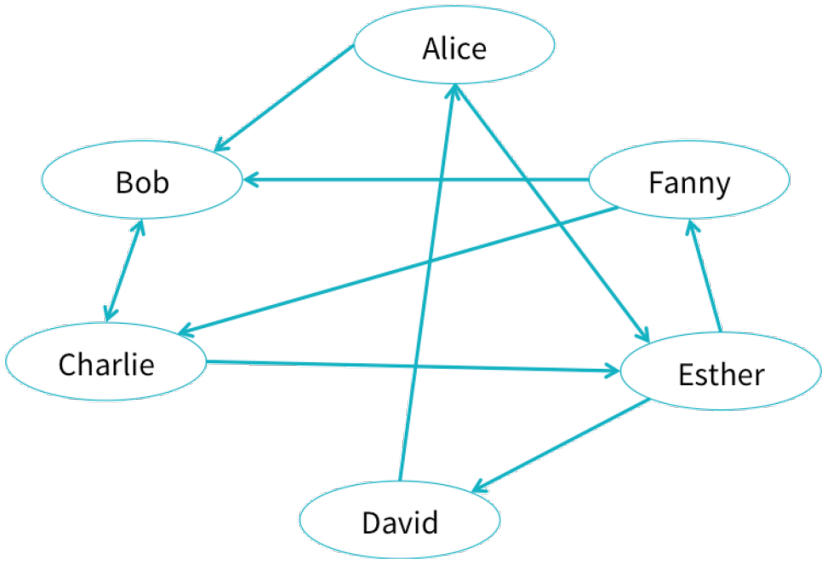
- **Powerful queries:** GraphFrames allow users to phrase queries in the familiar, powerful APIs of Spark SQL and DataFrames.

- **Saving & loading graphs:** GraphFrames fully support DataFrame data sources, allowing writing and reading graphs using many formats like Parquet, JSON, and CSV.

In GraphFrames, vertices and edges are represented as DataFrames, allowing us to store arbitrary data with each vertex and edge.

An example social network

Say we have a social network with users connected by relationships. We can represent the network as a graph, which is a set of vertices (users) and edges (connections between users). A toy example is shown below.



Click on the figure to see the full example notebook

We might then ask questions such as “Which users are most influential?” or “Users A and B do not know each other, but should they be introduced?” These types of questions can be answered using graph queries and algorithms.

GraphFrames can store data with each vertex and edge. In a social network, each user might have an age and name, and each connection might have a relationship type.

id	name	age
a	Alice	34
b	Bob	36
c	Charlie	30
d	David	29
e	Esther	32
f	Fanny	36

src	dst	relationship
a	e	friend
f	b	follow
c	e	friend
a	b	friend
b	c	follow
c	b	follow
f	c	follow
e	f	follow
e	d	friend
d	a	friend

Click on the table to see the full example notebook

Simple queries are simple

GraphFrames make it easy to express queries over graphs. Since GraphFrame vertices and edges are stored as DataFrames, many queries are just DataFrame (or SQL) queries.

Example:
How many users in our social network have “age” > 35?
We can query the `vertices` DataFrame:

```
g.vertices.filter("age > 35”)
```

Example:
How many users have at least 2 followers?
We can combine the built-in inDegrees method with a DataFrame query.

```
g.inDegrees.filter("inDegree >= 2”)
```

Graph algorithms support complex workflows

GraphFrames support the full set of algorithms available in GraphX, in all 3 language APIs. Results from graph algorithms are either DataFrames or GraphFrames. For example, what are the most important users? We can run PageRank:

```
results = g.pageRank(resetProbability=0.15, maxIter=10)
display(results.vertices)
```

id	name	age	pagerank
a	Alice	34	0.5837691183776904
b	Bob	36	1.1381260736817564
d	David	29	0.5271874389390149
e	Esther	32	0.9212415144929111
c	Charlie	30	1.298428953431926
f	Fanny	36	0.5271874389390149

Click on the table to see the full example notebook

GraphFrames also support new algorithms:

- Breadth-first search (BFS): Find shortest paths from one set of vertices to another
- Motif finding: Search for structural patterns in a graph

Motif finding lets us make powerful queries. For example, to recommend whom to follow, we might search for triplets of users A,B,C where A follows B and B follows C, but A does not follow C.

```
# Motif: A->B->C but not A->C
results = g.find("(A)-[]->(B); (B)-[]->(C); !(A)-[]->(C)")
# Filter out loops (with DataFrame operation)
```

A	C
▾ object id: e name: Esther age: 32	▾ object id: c name: Charlie age: 30
▸ {"id":"f","name":"Fanny","age":36}	▸ {"id":"e","name":"Esther","age":32}
▸ {"id":"a","name":"Alice","age":34}	▸ {"id":"f","name":"Fanny","age":36}
▸ {"id":"a","name":"Alice","age":34}	▸ {"id":"c","name":"Charlie","age":30}
▸ {"id":"e","name":"Esther","age":32}	▸ {"id":"a","name":"Alice","age":34}
▸ {"id":"a","name":"Alice","age":34}	▸ {"id":"d","name":"David","age":29}
▸ {"id":"b","name":"Bob","age":36}	▸ {"id":"e","name":"Esther","age":32}

Click on the table to see the full example notebook

The full set of GraphX algorithms supported by GraphFrames is:

- PageRank: Identify important vertices in a graph
- Shortest paths: Find shortest paths from each vertex to landmark vertices
- Connected components: Group vertices into connected subgraphs

- Strongly connected components: Soft version of connected components
- Triangle count: Count the number of triangles each vertex is part of
- Label Propagation Algorithm (LPA): Detect communities in a graph

GraphFrames integrate with GraphX

GraphFrames fully integrate with GraphX via conversions between the two representations, without any data loss. We can convert our social network to a GraphX graph and back to a GraphFrame.

```
val gx: Graph[Row, Row] = g.toGraphX()
val g2: GraphFrame = GraphFrame.fromGraphX(gx)
```

See the GraphFrame API docs for more details on these conversions.

What's next?

Graph-specific optimizations for DataFrames are under active research and development. Watch Ankur Dave's Spark Summit East 2016 talk to learn more. We plan to include some of these optimizations in GraphFrames for its next release!

Get started with these tutorial notebooks in Scala and Python in the Databricks Community Edition beta program. If you do not have access to the beta yet, join the beta waitlist.

Download the GraphFrames package from the Spark Packages website. GraphFrames are compatible with Spark 1.4, 1.5, and 1.6.

Learn more in the User Guide and API docs.

The code is available on Github under the Apache 2.0 license. We welcome contributions! Check the Github issues for ideas to work on.Working with R on Apache Spark

Get the Notebook

Section 3:
Working with R on Apache Spark

Announcing SparkR: R on Spark

June 9, 2015 | by Shivaram Venkataraman

I am excited to announce that the upcoming Apache Spark 1.4 release will include SparkR, an R package that allows data scientists to analyze large datasets and interactively run jobs on them from the R shell.

R is a popular statistical programming language with a number of extensions that support data processing and machine learning tasks. However, interactive data analysis in R is usually limited as the runtime is single-threaded and can only process data sets that fit in a single machine's memory. SparkR, an R package initially developed at the AMPLab, provides an R frontend to Apache Spark and using Spark's distributed computation engine allows us to run large scale data analysis from the R shell.

Project History

The SparkR project was initially started in the AMPLab as an effort to explore different techniques to integrate the usability of R with the scalability of Spark. Based on these efforts, an initial developer preview of SparkR was first open sourced in January 2014. The project was then developed in the AMPLab for the next year and we made many performance and usability improvements through open source contributions to SparkR. SparkR was recently merged into the Apache Spark project and will be released as an alpha component of Apache Spark in the 1.4 release.

SparkR DataFrames

The central component in the SparkR 1.4 release is the SparkR DataFrame, a distributed data frame implemented on top of Spark. Data frames are a fundamental data structure used for data processing in R and the concept of data frames has been extended to other languages with libraries like Pandas etc. Projects like dplyr have further simplified expressing complex data manipulation tasks on data frames. SparkR DataFrames present an API similar to dplyr and local R data frames but can scale to large data sets using support for distributed computation in Spark.

The following example shows some of the aspects of the DataFrame API in SparkR. (You can see the full example at https://gist.github.com/shivaram/d0cd4aa5c4381edd6f85)

```
# flights is a SparkR data frame. We can first print the column
# names, types, flights
#DataFrame[year:string, month:string, day:string,
dep_time:string, dep_delay:string, #arr_time:string,
arr_delay:string, carrier:string, tailnum:string,
flight:string, origin:string, #dest:string, air_time:string,
distance:string, hour:string, minute:string]

# Print the first few rows using `head`
head(flights)

# Filter all the flights leaving from JFK
jfk_flights <- filter(flights, flights$origin == "JFK")


# Collect the DataFrame into a local R data frame (for plotting
etc.)
local_df <- collect(jfk_flights)
```

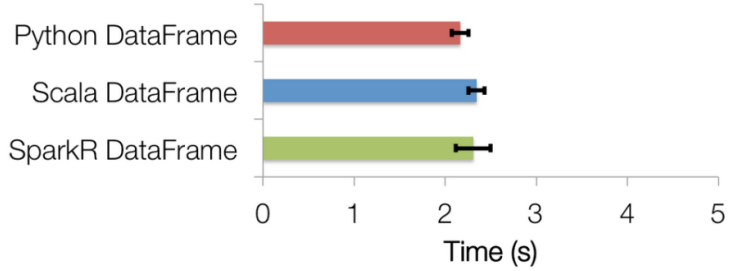
For a more comprehensive introduction to DataFrames you can see the SparkR programming guide at http://people.apache.org/~pwendell/spark-releases/latest/sparkr.html

Benefits of Spark integration

In addition to having an easy to use API, SparkR inherits many benefits from being tightly integrated with Spark. These include:

Data Sources API: By tying into Spark SQL's data sources API SparkR can read in data from a variety of sources include Hive tables, JSON files, Parquet files etc.

Data Frame Optimizations: SparkR DataFrames also inherit all of the optimizations made to the computation engine in terms of code generation, memory management. For example, the following chart compares the runtime performance of running group-by aggregation on 10 million integer pairs on a single machine in R, Python and Scala (it uses the same dataset as https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html). From the graph we can see that using the optimizations in the computation engine makes SparkR performance similar to that of Scala / Python.



Scalability to many cores and machines: Operations executed on SparkR DataFrames get automatically distributed across all the cores and machines available on the Spark cluster. As a result SparkR DataFrames can be used on terabytes of data and run on clusters with thousands of machines.

Looking forward

We have many other features planned for SparkR in upcoming releases: these include support for high level machine learning algorithms and making SparkR DataFrames a stable component of Spark.

The SparkR package represents the work of many contributors from various organizations including AMPLab, Databricks, Alteryx and Intel. We'd like to thank our contributors and users who tried out early versions of SparkR and provided feedback. If you are interested in SparkR, do check out our talks at Spark Summit.

Generalized Linear Models in SparkR and R Formula Support in MLlib

October 5, 2015 | by Eric Liang and Xiangrui Meng

Get the Notebook

Spark 1.5 adds initial support for distributed machine learning over SparkR DataFrames. To provide an intuitive interface for R users, SparkR extends R's native methods for fitting and evaluating models to use MLlib for large-scale machine learning. In this blog post, we cover how to work with generalized linear models in SparkR, and how to use the new R formula support in MLlib to simplify machine learning pipelines. This work was contributed by Databricks in Spark 1.5. We'd also like to thank Alteryx for providing input on early designs.

Generalized Linear Models

Generalized linear models unify various statistical models such as linear and logistic regression through the specification of a model family and link function. In R, such models can be fitted by passing an R model formula, family, and training dataset to the `glm()` function. Spark 1.5 extends `glm()` to operate over Spark DataFrames, which are distributed data collections managed by Spark. We also support elastic-net regularization for these models, the same as in R's `glmnet` package.

Fitting Models

Since we extend R's native methods for model fitting, the interface is very similar. R lets you specify the modeling of a response variable in a compact symbolic form. For example, the formula `y ~ f0 + f1` indicates the response `y` is modeled linearly by variables `f0` and `f1`. In 1.5 we support a subset of the R formula operators available. This includes the `+` (inclusion), `-` (exclusion), `.` (include all), and intercept operators. To demonstrate glm in SparkR, we will walk through fitting a model over a 12 GB dataset (with over 120 million records) in the example below. Datasets of this size are hard to train on a single machine due to their size.

Preprocessing

The dataset we will operate on is the publicly available airlines dataset, which contains twenty years of flight records (from 1987 to 2008). We are interested in predicting airline arrival delay based on the flight departure delay, aircraft type, and distance traveled.

First, we read the data from the CSV format using the spark-csv package and join it with an auxiliary planes table with details on individual aircraft.

```
> airlines <- read.df(sqlContext, path="/home/ekl/airlines",
    source="com.databricks.spark.csv", header="true", inferSchema="true")
> planes <- read.df(sqlContext, "/home/ekl/plane_info",
    source="com.databricks.spark.csv", header="true", inferSchema="true")
> joined <- join(airlines, planes, airlines$TailNum == planes$tailnum)
```

We use functionality from the DataFrame API to apply some preprocessing to the input. As part of the preprocessing, we decide to drop rows containing null values by applying `dropna()` to the DataFrame.

```
> training <- dropna(joined)
> showDF(select(training,
    “aircraft_type”, “Distance”, “ArrDelay”, “DepDelay”))

 aircraft_type              | Distance | DepDelay  | ArrDelay
----------------------------|----------|-----------|----------
 "Balloon"                  | 23       | 18        | 20
 "Fixed Wing Multi-Engine"  | 815      | 2         | -2
 "Fixed Wing Single-Engine" | 174      | 0         | 1
```

Training

The next step is to use MLlib by calling `glm()` with a formula specifying the model variables. We specify the Gaussian family here to indicate that we want to perform linear regression. MLlib caches the input DataFrame and launches a series of Spark jobs to fit our model over the distributed dataset.

```
> model <- glm(ArrDelay ~ DepDelay + Distance +
aircraft_type,
    family = "gaussian", data = training)
```

Note that parameter “lambda” can be used with glm to add regularization and “alpha” to adjust elastic-net constant.

Evaluation

As with R’s native models, coefficients can be retrieved using the `summary()` function.

```
> summary(model)
$coefficients
                                                Estimate
(Intercept)                                -0.5155037863
DepDelay                                    0.9776640253
Distance                                   -0.0009826032
aircraft_type__Fixed Wing Multi-Engine      0.3348238914
aircraft_type__Fixed Wing Single-Engine     0.2296622061
aircraft_type__Balloon                      0.5374569269
```

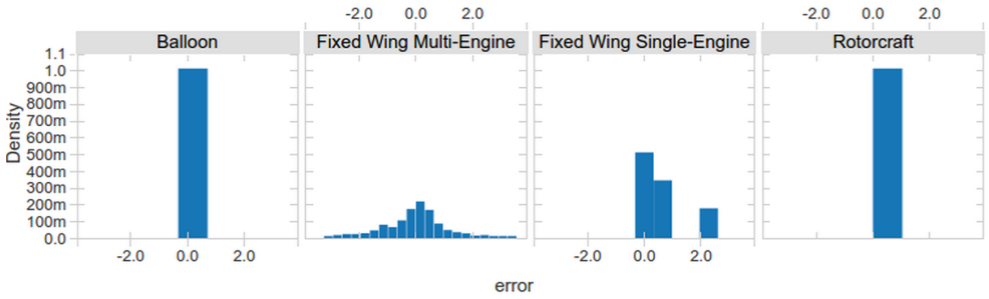
Note that the `aircraft_type` feature is categorical. Under the hood, SparkR automatically performs one-hot encoding of such features so that it does not need to be done manually. Beyond String and Double type features, it is also possible to fit over MLlib Vector features, for compatibility with other MLlib components.

To evaluate our model we can also use `predict()` just like in R. We can pass in the training data or another DataFrame that contains test data.

```
> preds <- predict(model, training)
> errors <- select(
    preds, preds$label, preds$prediction, preds$aircraft_type,
    alias(preds$label - preds$prediction, "error"))
```

Since the returned DataFrame contains the original columns in addition to the label, features, and predicted value, it is easy to inspect the result. Here we take advantage of the built-in visualizations from Databricks to examine the error distribution with respect to the aircraft type.

```
> display(sample(errors, F, .0001))
```



In summary, SparkR now provides seamless integration of DataFrames with common R modeling functions, making it simple for R users to take advantage of MLlib's distributed machine learning algorithms.

To learn more about SparkR and its integration with MLlib, see the latest SparkR documentation.

R formula support in other languages

SparkR implements the interpretation of R model formulas as an MLlib feature transformer, for integration with the ML Pipelines API. The RFormula transformer provides a convenient way to specify feature transformations like in R.

To see how the RFormula transformer can be used, let's start with the same airlines dataset from before. In Python, we create an RFormula transformer with the same formula used in the previous section.

```
>>> import pyspark.ml.feature.RFormula
>>> formula = RFormula(
        formula="ArrDelay ~ DepDelay + Distance + aircraft_type")
```

After the transformation, a DataFrame with features and label column appended is returned. Note that we have to call `fit()` on a dataset before we can call `transform()`. The `fit()` step determines the mapping of categorical feature values to vector indices in the output, so that the fitted RFormula can be used across different datasets.

```
>>> formula.fit(training).transform(training).show()
+--------------+---------+---------+---------+--------------------+------+
| aircraft_type| Distance| DepDelay| ArrDelay|            features| label|
+--------------+---------+---------+---------+--------------------+------+
|       Balloon|       23|       18|       20| [0.0,0.0,23.0,18.0]|  20.0|
|  Multi-Engine|      815|        2|       -2| [0.0,1.0,815.0,2.0]|  -2.0|
| Single-Engine|      174|        0|        1| [1.0,0.0,174.0,0.0]|   1.0|
+--------------+---------+---------+---------+--------------------+------+
```

Any ML pipeline can include the RFormula transformer as a pipeline stage, which is in fact how SparkR implements `glm()`. After we have created an appropriate RFormula transformer and an estimator for the desired model family, fitting a GLM model takes only one step:

```
>>> import pyspark.ml.Pipeline
>>> import pyspark.ml.regression.LinearRegression

>>> estimator = LinearRegression()
>>> model = Pipeline(stages=[formula,
estimator]).fit(training)
```

When the pipeline executes, the features referenced by the formula will be encoded into an output feature vector for use by the linear regression stage.

We hope that RFormula will simplify the creation of ML pipelines by providing a concise way of expressing complex feature transformations. Starting in Spark 1.5 the RFormula transformer is available for use in Python, Java, and Scala.

What's next?

In Spark 1.6 we are adding support for more advanced features of R model formulas, including feature interactions, more model families, link functions, and better summary support.

As part of this blog post, we would like to thank Dan Putler and Chris Freeman from Alteryx for useful discussions during the implementation of this functionality in SparkR, and Hossein Falaki for input on content.

Get the Notebook

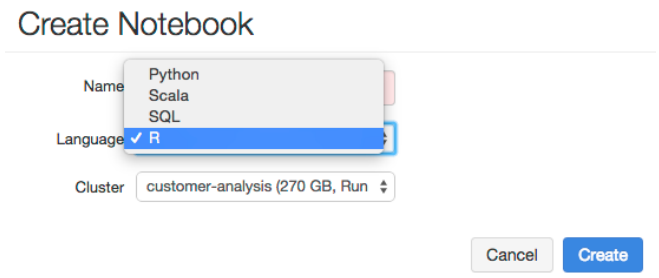
Introducing R Notebooks in Databricks

July 13, 2015 | by Hossein Falaki

To dive deeper into R Notebooks in Databricks, check out the 'Data Science with Apache Spark and R in Databricks' notebooks based on the Million Songs Dataset:

Spark 1.4 was released on June 11 and one of the exciting new features was SparkR. I am happy to announce that we now support R notebooks and SparkR in Databricks, our hosted Spark service. Databricks lets you easily use SparkR in an interactive notebook environment or standalone jobs.

R and Spark nicely complement each other for several important use cases in statistics and data science. Databricks R Notebooks include the SparkR package by default so that data scientists can effortlessly benefit from the power of Apache Spark in their R analyses. In addition to SparkR, any R package can be easily installed into the notebook. In this blog post, I will highlight a few of the features in our R Notebooks.



To get started with R in Databricks, simply choose R as the language when creating a notebook. Since SparkR is a recent addition to Spark, remember to attach the R notebook to any cluster running Spark version 1.4 or later. The SparkR package is imported and configured by default. You can run Spark queries in R:

Using SparkR you can access and manipulate very large data sets (e.g., terabytes of data) from distributed storage (e.g., Amazon S3) or data warehouses (e.g., Hive).

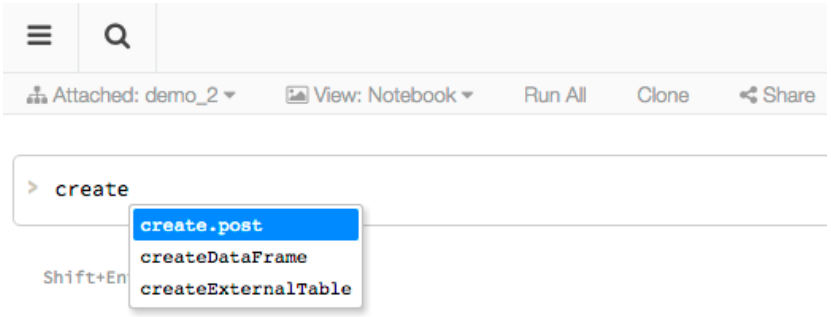
```
airlinesDF <- read.df(sqlContext, path="dbfs:/databricks-datasets/airlines",
   source="com.databricks.spark.csv", header="true")
registerTempTable(airlinesDF, "airlines")
```

SparkR offers distributed DataFrames that are syntax compatible with R data frames. You can also collect a SparkR DataFrame to local data frames.

```
delays <- collect(sql(sqlContext, "select avg(Distance) as distance,
  avg(ArrDelay) as arrivalDelay,
  avg(DepDelay) as departureDelay,
  Origin,
  Dest,
  UniqueCarrier as carrier from airlines group by Origin, Dest,
UniqueCarrier"))
```

For an overview of SparkR features see our recent blog post. Additional details on SparkR API can be found on the Spark website.

Autocomplete and Libraries



Databricks R notebooks offer autocomplete similar to the R shell. Pressing TAB will complete the code or present available options if multiple exist.

You can install any R library in your notebooks using `install.packages()`. Once you import the new library, autocomplete will also apply to the newly introduced methods and objects.

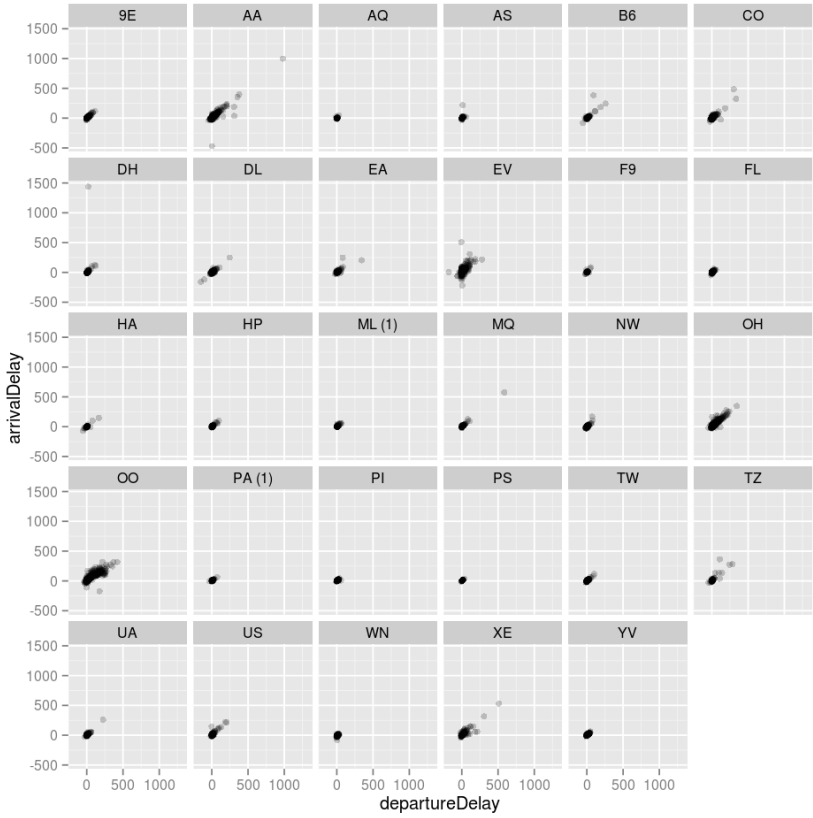
Interactive Visualization

At Databricks we believe visualization is a critical part of data analysis. As a result we embraced R's powerful visualization and complemented it with many additional visualization features.

Inline plots

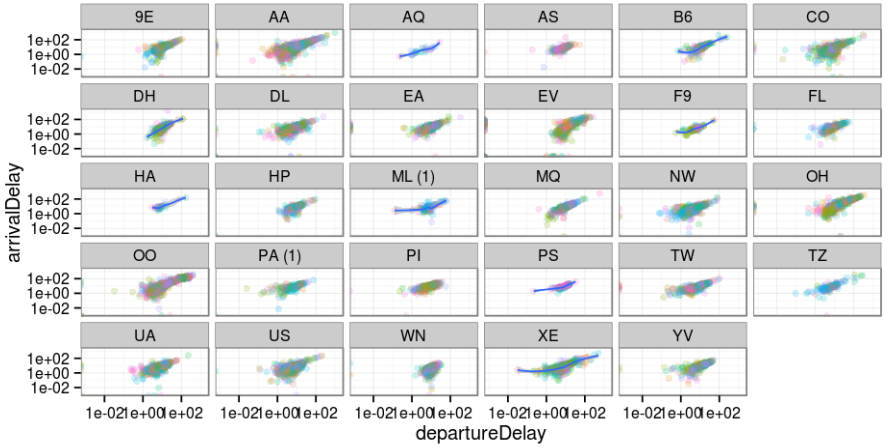
In R Notebooks you can use any R visualization library, including base plotting, ggplot, Lattice, or any other plotting library. Plots are displayed inline in the notebook and can be conveniently resized with the mouse.

```
library(ggplot2)
p <- ggplot(delays, aes(departureDelay, arrivalDelay)) +
  geom_point(alpha = 0.2) + facet_wrap(~carrier)
p
```



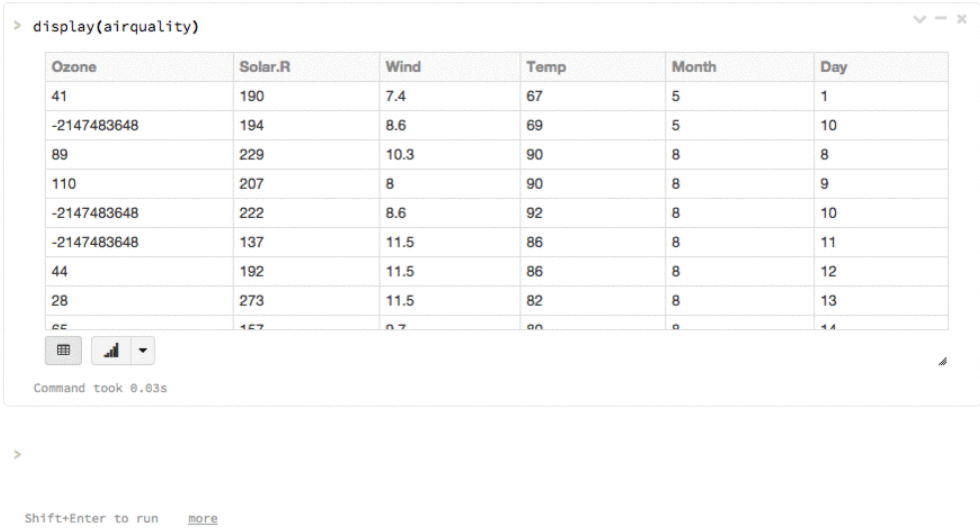
You can set options to change aspect ratio and resolution of inline plots.

```
options(repr.plot.height = 500, repr.plot.res = 120)
p + geom_point(aes(color = Dest)) + geom_smooth() +
  scale_x_log10() + scale_y_log10() + theme_bw()
```



One-click visualizations

You can use Databricks's built-in **`display()`** function on any R or SparkR DataFrame. The result will be rendered as a table in the notebook, which you can then plot with one click without writing any custom code.

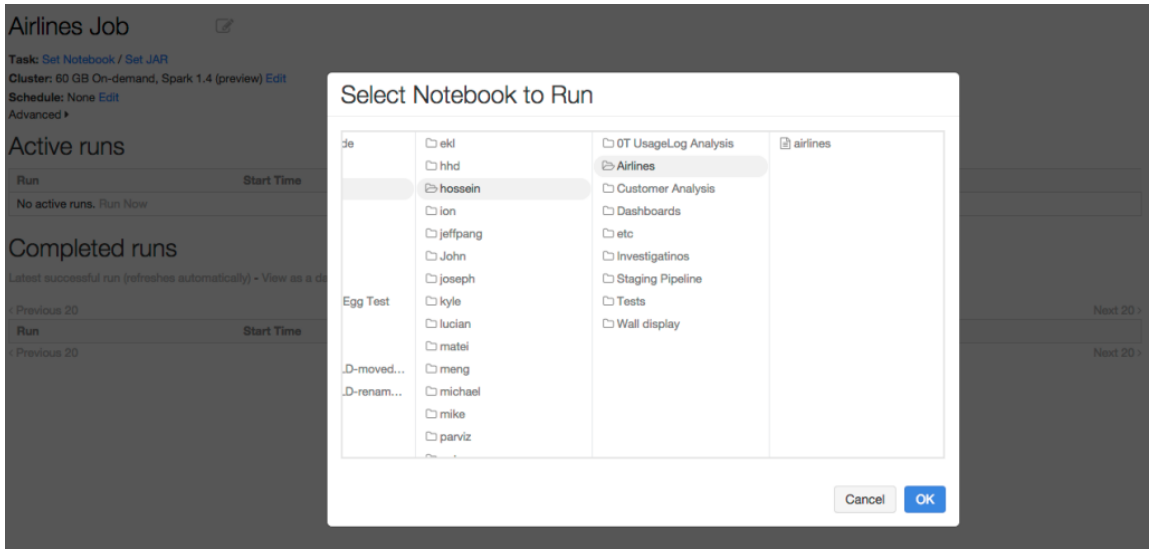


Advanced interactive visualizations

Similar to other Databricks notebooks, you can use **`displayHTML()`** function in R notebooks to render any HTML and Javascript visualization.

Running Production Jobs

Databricks is an end-to-end solution to make building a data pipeline easier – from ingest to production. The same concept applies to R Notebooks as well: You can schedule your R notebooks to run as jobs on existing or new Spark clusters. The results of each job run, including visualizations, are available to browse, making it much simpler and faster to turn the work of data scientists into production.



Summary

R Notebooks in Databricks let anyone familiar with R take advantage of the power of Spark through simple Spark cluster management, rich one-click visualizations, and instant deployment to production jobs. We believe SparkR and R Notebooks will bring even more people to the rapidly growing Spark community.

To dive deeper into R Notebooks in Databricks, check out the 'Data Science with Apache Spark and R in Databricks' notebooks based on the Million Songs Dataset:

Data Science with Apache Spark and R in Databricks
1. Parsing songs data
2. Exploring songs data
3. Modeling Songs

Conclusion

Our mission at Databricks is to dramatically simplify big data processing and data sciences so organizations can immediately start working on their data problems, in an environment accessible to data scientists, engineers, and business users alike. We hope the collection of blog posts we've curated in this eBook will provide you with the insights and tools to help you solve your biggest data problems.

If you enjoyed the technical content in this eBook, visit the Databricks Blog for more technical tips, best practices, and case studies from the Spark experts at Databricks.

Read all the books in this Series:

Apache Spark Analytics Made Simple
Mastering Advanced Analytics with Apache Spark
Lessons for Large-Scale Machine Learning Deployments on Apache Spark
Building Real-Time Applications with Spark Streaming

To learn more about Databricks, check out some of these resources:

Databricks Primer
How-To Guide: The Easiest Way to Run Spark Jobs
Solution Brief: Making Data Warehousing Simple
Solution Brief: Making Machine Learning Simple
White Paper: Simplifying Spark Operations with Databricks

To try out Databricks for yourself, sign-up for a 14-day free trial today!