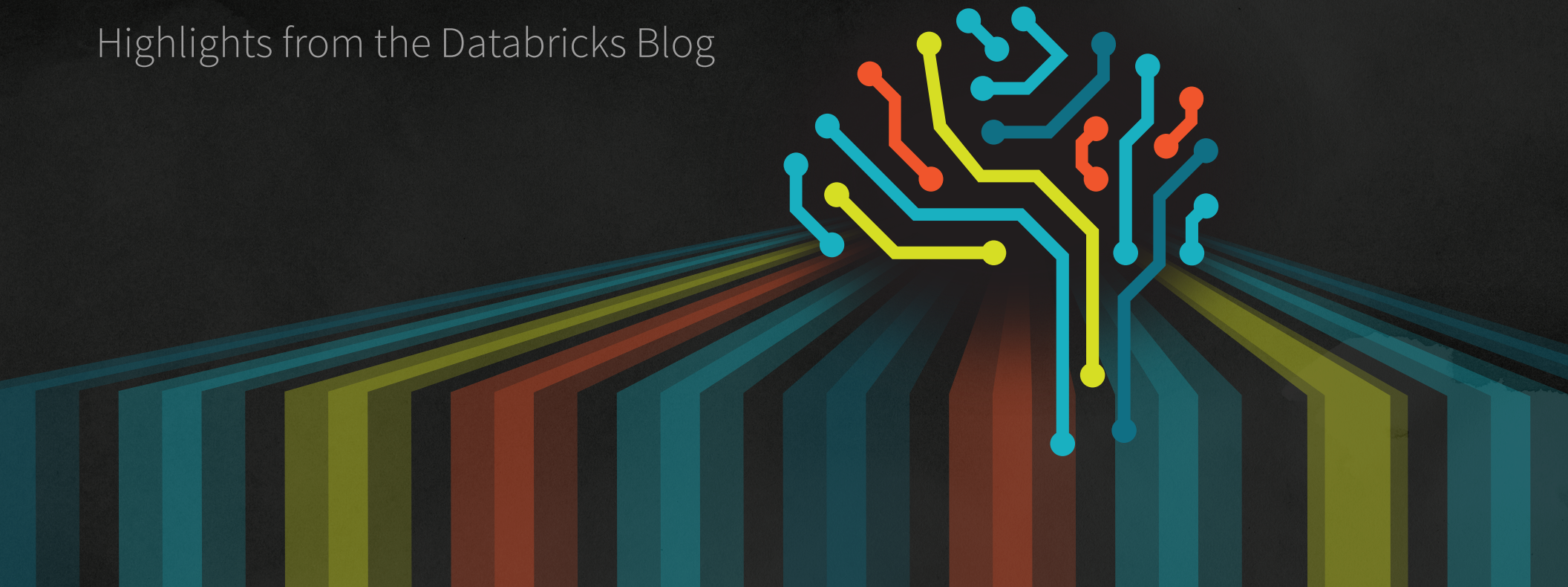
databricks

Lessons for Large-Scale Machine Learning Deployments on Apache® Spark™

Highlights from the Databricks Blog



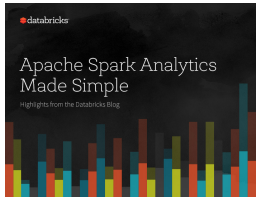
Lessons from Large-Scale Machine Learning Deployments on Apache® Spark™

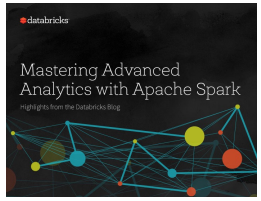
Highlights from the Databricks Blog

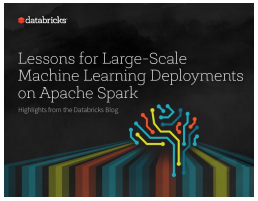
By Jake Bellacera, Joseph Bradley, Bill Chambers, Wayne Chan, Hossein Falaki, Tim Hunter, Denny Lee, Feynman Liang, Kavitha Mariappan, Xiangrui Meng, Dave Wang, Raela Wang, Reynold Xin, and Burak Yavuz

Special thanks to our guest authors Andy Huang and Wei Wu from Alibaba Taobao; Eugene Zhulenev from Collective; Deborah Siegel from the Northwest Genome Center and the University of Washington; and Saman Keshmiri and Christopher Hoshino-Fish from Sellpoints.

3rd 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

Apache® Spark™, the de facto standard for big data processing and data science provides comprehensive machine learning and graph processing libraries, allowing companies to easily tackle complex advanced analytics problems. The Databricks engineering team include the creators of Apache Spark, members of the Spark PMC, and Spark committers, who endeavor to bring state-of-the-art machine learning and graph computation techniques to Spark by contributing code to MLlib, GraphX, and many other libraries. Through the Databricks Blog, they also regularly highlight new Spark releases and features relevant to advanced analytics and provide technical deep-dives on Spark components and the broader ecosystem.

This ebook, the third of a series, picks up where the second book left off on the topic of advanced analytics, and jumps straight into practical tips for performance tuning and powerful integrations with other machine learning tools — including the popular deep learning framework TensorFlow and the python library scikit-learn. The second section of the book is devoted to address the roadblocks in developing machine learning algorithms on Apache Spark — from simple visualizations to modeling audiences with Apache Spark machine learning pipelines. Finally, the ebook showcases a selection of Spark machine learning use cases from ad tech, retail, financial services, and many other industries.

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: Performance Tuning and Practical Integration

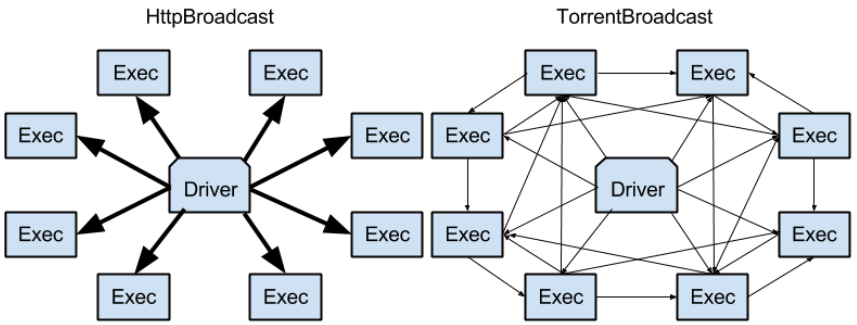
Spark 1.1: MLlib Performance Improvements

September 22, 2014 | by Burak Yavuz and Xiangrui Meng

With an ever-growing community, Spark has had it's 1.1 release. MLlib has had its fair share of contributions and now supports many new features. We are excited to share some of the performance improvements observed in MLlib since the 1.0 release, and discuss two key contributing factors: torrent broadcast and tree aggregation.

Torrent broadcast

The beauty of Spark as a unified framework is that any improvements made on the core engine come for free in its standard components like MLlib, Spark SQL, Streaming, and GraphX. In Spark 1.1, we changed the default broadcast implementation of Spark from the traditional `HttpBroadcast` to `TorrentBroadcast`, a BitTorrent like protocol that evens out the load among the driver and the executors. When an object is broadcasted, the driver divides the serialized object into multiple chunks, and broadcasts the chunks to different executors. Subsequently, executors can fetch chunks individually from other executors that have fetched the chunks previously.



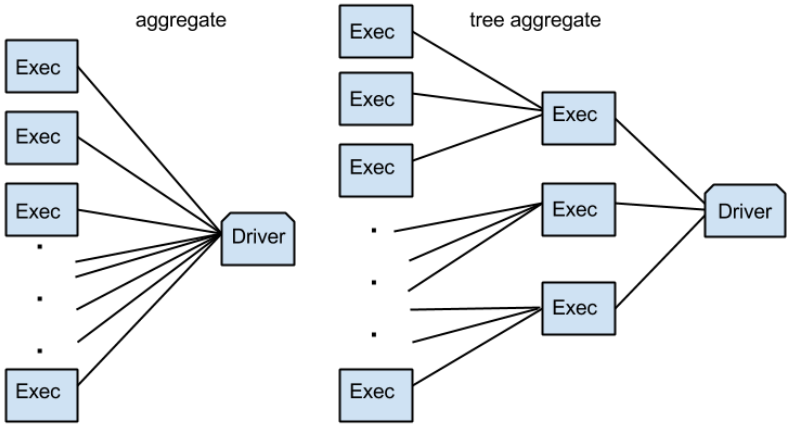
How does this change in Spark Core affect MLlib's performance?

A common communication pattern in machine learning algorithms is the one-to-all broadcast of intermediate models at the beginning of each iteration of training. In large-scale machine learning, models are usually huge and broadcasting them via http can make the driver a severe bottleneck because all executors (workers) are fetching the models from the driver. With the new torrent broadcast, this load is shared among executors as well. It leads to significant speedup, and MLlib takes it for free.

Tree aggregation

Similar to broadcasting models at the beginning of each iteration, the driver builds new models at the end of each iteration by aggregating partial updates collected from executors. This is the basis of the MapReduce paradigm. One performance issue with the `reduce` or `aggregate` functions in Spark (and the original MapReduce) is that the

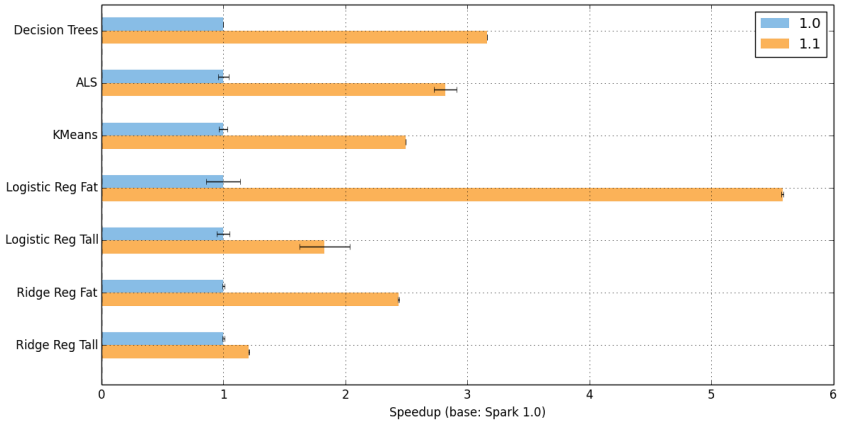
aggregation time scales linearly with respect to the number of partitions of data (due to the CPU cost in merging partial results and the network bandwidth limit).



In MLlib 1.1, we introduced a new aggregation communication pattern based on multi-level aggregation trees. In this setup, model updates are combined partially on a small set of executors before they are sent to the driver, which dramatically reduces the load the driver has to deal with. Tests showed that these functions reduce the aggregation time by an order of magnitude, especially on datasets with a large number of partitions.

Performance improvements

Changing the way models are broadcasted and aggregated has a huge impact on performance. Below, we present empirical results comparing the performance on some of the common machine learning algorithms in MLlib. The x-axis can be thought of the speedup the 1.1 release has over the 1.0 release. Speedups between 1.5-5x can be observed across all algorithms. The tests were performed on an EC2 cluster with 16 slaves, using m3.2xlarge instances. The scripts to run the tests are a part of the "spark-perf" test suite which can be found on https://github.com/databricks/spark-perf.



For Ridge Regression And Logistic Regression, The Tall Identifier Corresponds To A Tall-Skinny Matrix (1,000,000 X 10,000) And Fat Corresponds To A Short-Fat Matrix (10,000 X 1,000,000).

Performance improvements in distributed machine learning typically come from a combination of communication pattern improvements and algorithmic improvements. We focus on the former in this post, and algorithmic improvements will be discussed later. So download the latest version of Apache Spark now, enjoy the performance improvements, and stay tuned for future posts.

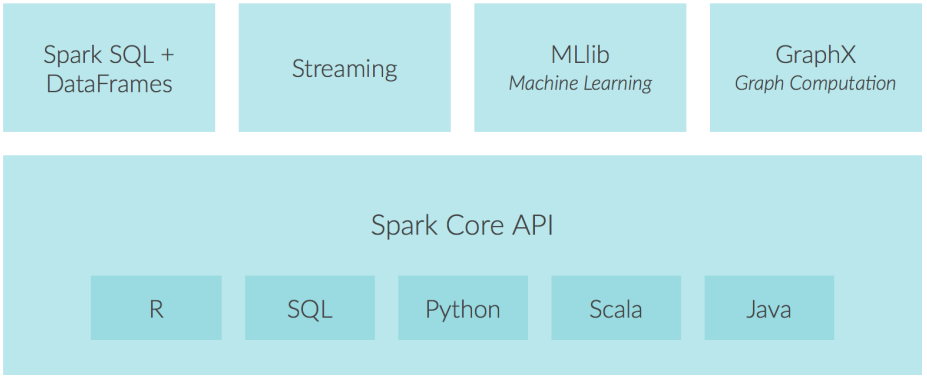
Recent performance improvements in Apache Spark: SQL, Python, DataFrames, and More

April 24, 2015 | by Reynold Xin

In this post, we look back and cover recent performance efforts in Spark. In a follow-up blog post next week, we will look forward and share with you our thoughts on the future evolution of Spark's performance.

2014 was the most active year of Spark development to date, with major improvements across the entire engine. One particular area where it made great strides was performance: Spark set a new world record in 100TB sorting, beating the previous record held by Hadoop MapReduce by three times, using only one-tenth of the resources; it received a new SQL query engine with a state-of-the-art optimizer; and many of its built-in algorithms became five times faster.

Back in 2010, we at the AMPLab at UC Berkeley designed Spark for interactive queries and iterative algorithms, as these were two major use cases not well served by batch frameworks like MapReduce. As a result, early users were drawn to Spark because of the significant performance improvements in these workloads. However, performance optimization is a never-ending process, and as Spark's use cases have grown, so have the areas looked at for further improvement. User feedback and detailed measurements helped the Apache Spark developer community to prioritize areas to work in. Starting with the core engine, I will cover some of the recent optimizations that have been made.



The Spark Ecosystem

Core engine

One unique thing about Spark is its user-facing APIs (SQL, streaming, machine learning, etc.) run over a common core execution engine. Whenever possible, specific workloads are sped up by making optimizations in the core engine. As a result, these optimizations speed up *all* components. We've often seen very surprising results this way: for example, when core developers decreased latency to introduce Spark Streaming, we also saw SQL queries become two times faster.

In the core engine, the major improvements in 2014 were in communication. First, *shuffle* is the operation that moves data point-to-point across machines. It underpins almost all workloads. For example, a SQL query joining two data sources uses shuffle to move tuples that should be joined together onto the same machine, and product recommendation algorithms such as ALS use shuffle to send user/product weights across the network.

The last two releases of Spark featured a new sort-based shuffle layer and a new network layer based on Netty with zero-copy and explicit memory management. These two make Spark more robust in very large-scale workloads. In our own experiments at Databricks, we have used this to run petabyte shuffles on 250,000 tasks. These two changes were also the key to Spark setting the current world record in large-scale sorting, beating the previous Hadoop-based record by 30 times in per-node performance.

In addition to shuffle, core developers rewrote Spark's *broadcast* primitive to use a BitTorrent-like protocol to reduce network traffic. This speeds up workloads that need to send a large parameter to multiple machines, including SQL queries and many machine learning algorithms. We have seen more than five times performance improvements for these workloads.

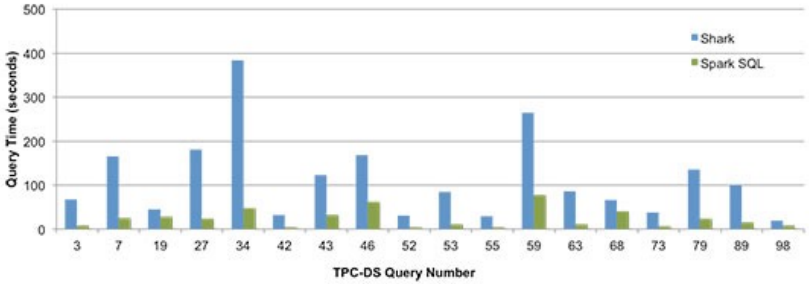
Python API (PySpark)

Python is perhaps the most popular programming language used by data scientists. The Spark community views Python as a first-class citizen of the Spark ecosystem. When it comes to performance, Python programs historically lag behind their JVM counterparts due to the more dynamic nature of the language.

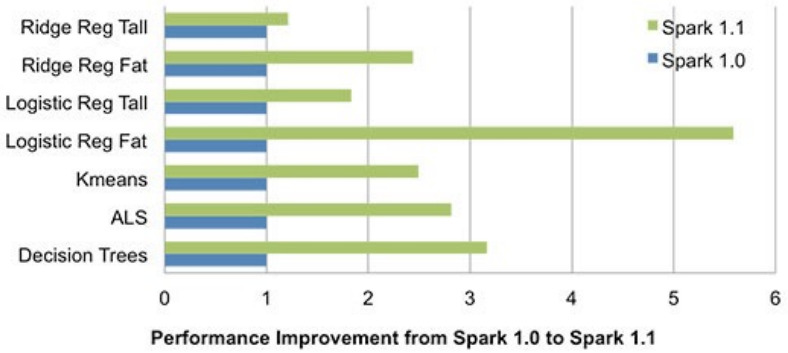
Spark's core developers have worked extensively to bridge the performance gap between JVM languages and Python. In particular, PySpark can now run on *PyPy* to leverage the just-in-time compiler, in some cases improving performance by a factor of 50. The way Python processes communicate with the main Spark JVM programs have also been redesigned to enable *worker reuse*. In addition, broadcasts are handled via a more optimized serialization framework, enabling PySpark to broadcast data larger than 2GB. The latter two have made general Python program performance two to 10 times faster.

SQL

One year ago, Shark, an earlier SQL on Spark engine based on Hive, was deprecated and we at Databricks built a new query engine based on a new query optimizer, Catalyst, designed to run natively on Spark. It was a controversial decision, within the Apache Spark developer community as well as internally within Databricks, because building a brand new query engine necessitates astronomical engineering investments. One year later, more than 115 open source contributors have joined the project, making it one of the most active open source query engines.



Shark vs. Spark SQL



Despite being less than a year old, Spark SQL is outperforming Shark on almost all benchmarked queries. In TPC-DS, a decision-support benchmark, Spark SQL is outperforming Shark often by an order of magnitude, due to better optimizations and code generation.

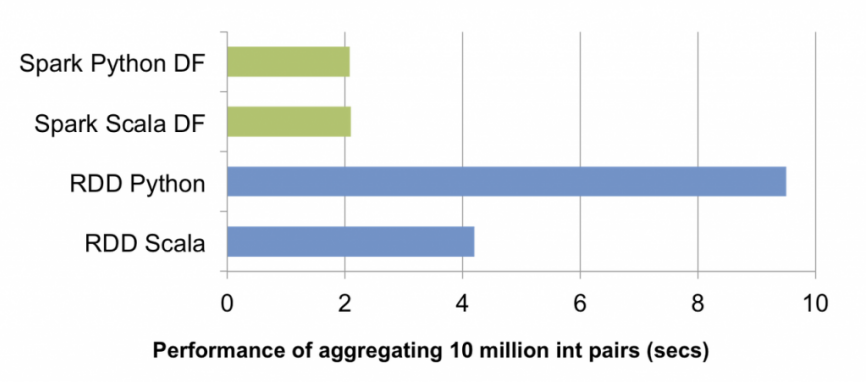
Machine learning (MLlib) and Graph Computation (GraphX)

From early on, Spark was packaged with powerful standard libraries that can be optimized along with the core engine. This has allowed for a number of rich optimizations to these libraries. For instance, Spark 1.1 featured a new communication pattern for aggregating machine learning models using multi-level aggregation trees. This has reduced the model aggregation time by an order of magnitude. This new communication pattern, coupled with the more efficient broadcast implementation in core, results in speeds 1.5 to five times faster across all algorithms.

In addition to optimizations in communication, *Alternative Least Squares (ALS)*, a common collaborative filtering algorithm, was also re-implemented 1.3, which provided another factor of two speedup for ALS over what the above chart shows. In addition, all the built-in algorithms in GraphX have also seen 20% to 50% runtime performance improvements, due to a new optimized API.

DataFrames: Leveling the Field for Python and JVM

In Spark 1.3, we introduced a new *DataFrame* API. This new API makes Spark programs more concise and easier to understand, and at the same time exposes more application semantics to the engine. As a result, Spark can use Catalyst to optimize these programs.



Through the new DataFrame API, Python programs can achieve the same level of performance as JVM programs because the Catalyst optimizer compiles DataFrame operations into JVM bytecode. Indeed, performance sometimes beats hand-written Scala code.

The Catalyst optimizer will also become smarter over time, picking better logical optimizations and physical execution optimizations. For example, in the future, Spark will be able to leverage schema information to create a custom physical layout of data, improving cache locality and reducing garbage collection. This will benefit both Spark SQL and DataFrame programs. As more libraries are converting to use this new DataFrame API, they will also automatically benefit from these optimizations.

The goal of Spark is to offer a single platform where users can get the best distributed algorithms for any data processing task. We will continue to push the boundaries of performance, making Spark faster and more powerful for more users.

Note: An earlier version of this blog post appeared on O'Reilly Radar.

Deep Learning with Spark and TensorFlow

January 25, 2016 | by Tim Hunter

Neural networks have seen spectacular progress during the last few years and they are now the state of the art in image recognition and automated translation. TensorFlow is a new framework released by Google for numerical computations and neural networks. In this blog post, we are going to demonstrate how to use TensorFlow and Spark together to train and apply deep learning models.

You might be wondering: what's Spark's use here when most high-performance deep learning implementations are single-node only? To answer this question, we walk through two use cases and explain how you can use Spark and a cluster of machines to improve deep learning pipelines with TensorFlow:

1. **Hyperparameter Tuning:** use Spark to find the best set of hyperparameters for neural network training, leading to 10X reduction in training time and 34% lower error rate.

2. **Deploying models at scale:** use Spark to apply a trained neural network model on a large amount of data.

Hyperparameter Tuning

An example of a deep learning machine learning (ML) technique is artificial neural networks. They take a complex input, such as an image or an audio recording, and then apply complex mathematical transforms on these signals. The output of this transform is a vector of numbers that is easier to manipulate by other ML algorithms. Artificial neural networks perform this transformation by mimicking the neurons in the visual cortex of the human brain (in a much-simplified form).

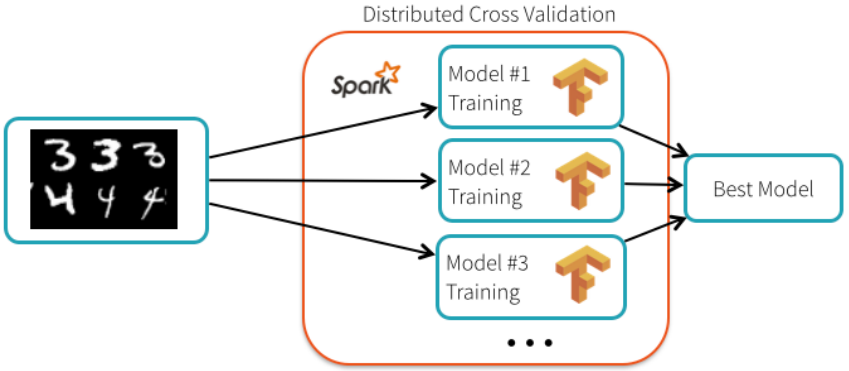
Just as humans learn to interpret what they see, artificial neural networks need to be trained to recognize specific patterns that are 'interesting'. For example, these can be simple patterns such as edges, circles, but they can be much more complicated. Here, we are going to use a classical dataset put together by NIST and train a neural network to recognize these digits:

The TensorFlow library automates the creation of training algorithms for neural networks of various shapes and sizes. The actual process of building a neural network, however, is more complicated than just running some function on a dataset. There are typically a number of very important hyperparameters (configuration parameters in layman's terms) to set, which affects how the model is trained. Picking the right parameters leads to high performance, while bad parameters can lead to prolonged training and bad performance. In practice, machine learning practitioners rerun the same model multiple times with different hyperparameters in order to find the best set. This is a classical technique called hyperparameter tuning.

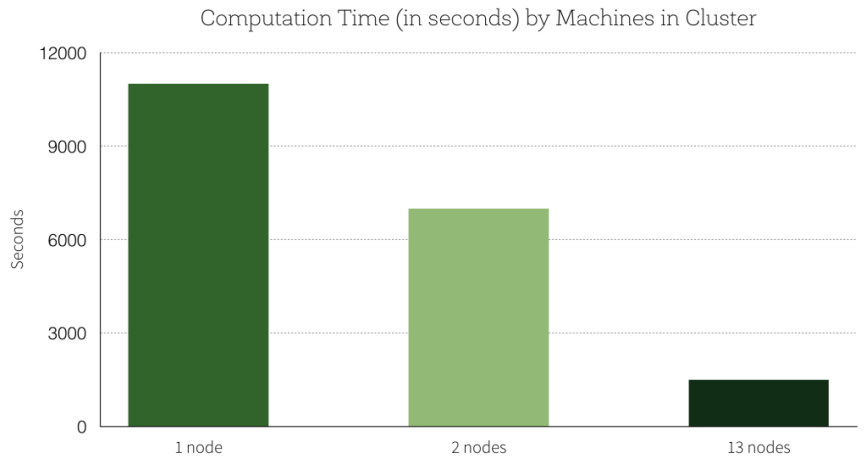
When building a neural network, there are many important hyperparameters to choose carefully. For example:

- **Number of neurons in each layer:** Too few neurons will reduce the expression power of the network, but too many will substantially increase the running time and return noisy estimates.
- **Learning rate:** If it is too high, the neural network will only focus on the last few samples seen and disregard all the experience accumulated before. If it is too low, it will take too long to reach a good state.

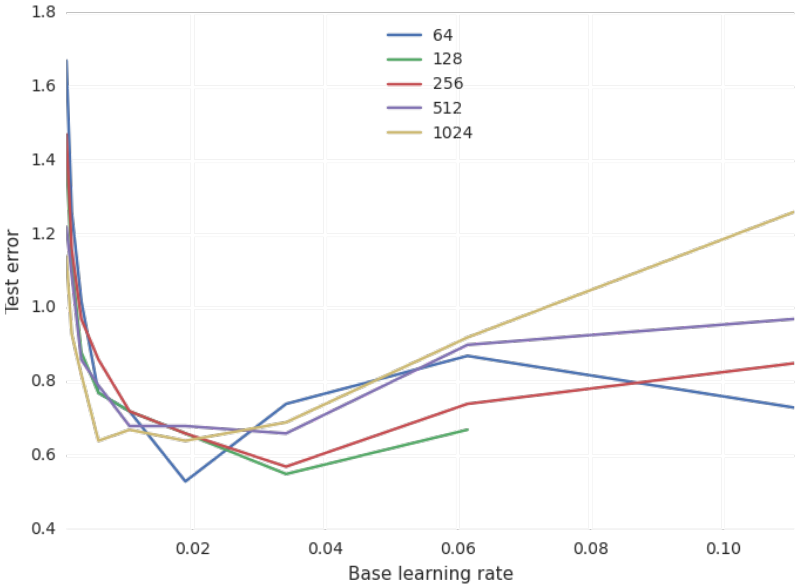
The interesting thing here is that even though TensorFlow itself is not distributed, the hyperparameter tuning process is "embarrassingly parallel" and can be distributed using Spark. In this case, we can use Spark to broadcast the common elements such as data and model description, and then schedule the individual repetitive computations across a cluster of machines in a fault-tolerant manner.



How does using Spark improve the accuracy? The accuracy with the default set of hyperparameters is 99.2%. Our best result with hyperparameter tuning has a 99.47% accuracy on the test set, which is a **34% reduction of the test error**. Distributing the computations scaled linearly with the number of nodes added to the cluster: using a 13-node cluster, we were able to train 13 models in parallel, which translates into a **7x speedup** compared to training the models one at a time on one machine. Here is a graph of the computation times (in seconds) with respect to the number of machines on the cluster:



More important though, we get insights into the sensibility of the training procedure to various hyperparameters of training. For example, we plot the final test performance with respect to the learning rate, for different numbers of neurons:



This shows a typical tradeoff curve for neural networks:

- The learning rate is critical: if it is too low, the neural network does not learn anything (high test error). If it is too high, the training process may oscillate randomly and even diverge in some configurations.
- The number of neurons is not as important for getting a good performance, and networks with many neurons are much more sensitive to the learning rate. This is Occam's Razor principle: simpler model tend to be "good enough" for most purposes. If you have the time and resource to go after the missing 1% test error, you must be willing to invest a lot of resources in training, and to find the proper hyperparameters that will make the difference.

By using a sparse sample of parameters, we can zero in on the most promising sets of parameters.

How do I use it?

Since TensorFlow can use all the cores on each worker, we only run one task at one time on each worker and we batch them together to limit contention. The TensorFlow library can be installed on Spark clusters as a regular Python library, following the instructions on the TensorFlow website. The following notebooks below show how to install TensorFlow and let users rerun the experiments of this blog post:

- Distributed processing of images using TensorFlow
- Testing the distribution processing of images using TensorFlow

Deploying Models at Scale

TensorFlow models can directly be embedded within pipelines to perform complex recognition tasks on datasets. As an example, we show how we can label a set of images from a stock neural network model that was already trained.

The model is first distributed to the workers of the clusters, using Spark's built-in broadcasting mechanism:

```
with gfile.FastGFile( 'classify_image_graph_def.pb', 'rb') as f:
  model_data = f.read()
model_data_bc = sc.broadcast(model_data)
```

Then this model is loaded on each node and applied to images. This is a sketch of the code being run on each node:

```
def apply_batch(image_url):
  # Creates a new TensorFlow graph of computation and imports the model
  with tf.Graph().as_default() as g:
    graph_def = tf.GraphDef()
    graph_def.ParseFromString(model_data_bc.value)
    tf.import_graph_def(graph_def, name='')

    # Loads the image data from the URL:
    image_data = urllib.request.urlopen(img_url, timeout=1.0).read()

    # Runs a tensor flow session that loads the
    with tf.Session() as sess:
      softmax_tensor = sess.graph.get_tensor_by_name('softmax:0')
      predictions = sess.run(softmax_tensor, {'DecodeJpeg/contents:0':
image_data})
      return predictions
```

This code can be made more efficient by batching the images together.

Here is an example of image:

And here is the interpretation of this image according to the neural network, which is pretty accurate:

```
('coral reef', 0.88503921),
   ('scuba diver', 0.025853464),
   ('brain coral', 0.0090828091),
   ('snorkel', 0.0036010914),
   ('promontory, headland, head, foreland', 0.0022605944)])
```

Looking forward

We have shown how to combine Spark and TensorFlow to train and deploy neural networks on handwritten digit recognition and image labeling. Even though the neural network framework we used itself only works in a single-node, we can use Spark to distribute the hyperparameter tuning process and model deployment. This not only cuts down the training time but also improves accuracy and gives us a better understanding of various hyperparameters' sensibility.

While this support is only available on Python, we look forward to providing deeper integration between TensorFlow and the rest of the Spark framework.

Auto-Scaling scikit-learn with Spark

February 8, 2016 | by Tim Hunter and Joseph Bradley

Data scientists often spend hours or days tuning models to get the highest accuracy. This tuning typically involves running a large number of independent Machine Learning (ML) tasks coded in Python or R. Following some work presented at Spark Summit Europe 2015, we are excited to release Scikit-learn integration package for Spark that dramatically simplifies the life of data scientists using Python. This package automatically distributes the most repetitive tasks of model tuning on a Spark cluster, without impacting the workflow of data scientists:

- When used on a single machine, Spark can be used as a substitute to the default multithreading framework used by scikit-learn (Joblib).

- If a need comes to spread the work across multiple machines, no change is required in the code between the single-machine case and the cluster case.

Scale data science effortlessly

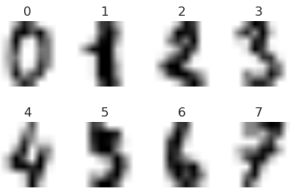
Python is one of the most popular programming languages for data exploration and data science, and this is in no small part due to high quality libraries such as Pandas for data exploration or scikit-learn for machine learning. Scikit-learn provides fast and robust implementations of standard ML algorithms such as clustering, classification, and regression.

Scikit-learn's strength has typically been in the realm of computing on a single node, though. For some common scenarios, such as parameter tuning, a large number of small tasks can be run in parallel. These scenarios are perfect use cases for Spark.

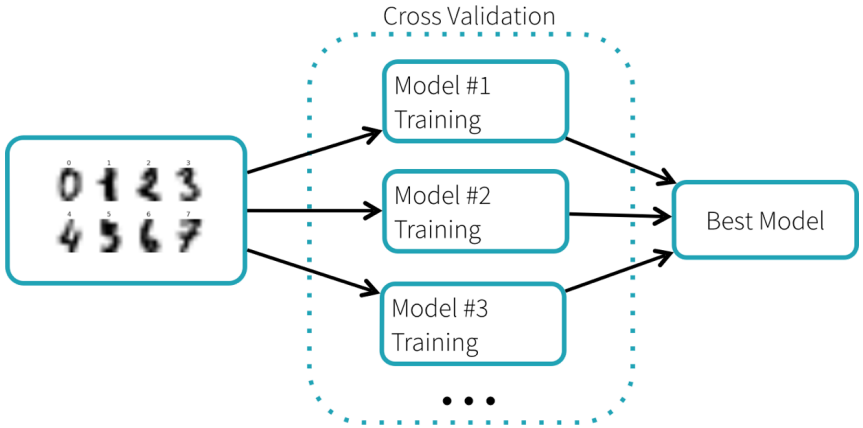
We explored how to integrate Spark with scikit-learn, and the result is the Scikit-learn integration package for Spark. It combines the strengths of Spark and scikit-learn *with no changes to users' code*. It re-implements some components of scikit-learn that benefit the most from distributed computing. Users will find a Spark-based cross-validator class that is fully compatible with scikit-learn's cross-validation tools. By swapping out a single class import, users can distribute cross-validation for their existing scikit-learn workflows.

Distribute tuning of Random Forests

Consider a classical example of identifying digits in images. Here are a few examples of images taken from the popular digits dataset, with their labels:



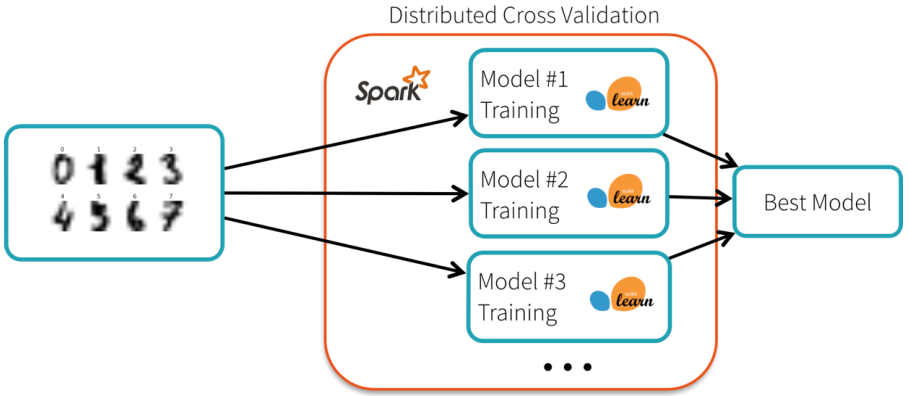
We are going to train a random forest classifier to recognize the digits. This classifier has a number of parameters to adjust, and there is no easy way to know which parameters work best, other than trying out many different combinations. Scikit-learn provides GridSearchCV, a search algorithm that explores many parameter settings automatically. GridSearchCV uses selection by cross-validation, illustrated below. Each parameter setting produces one model, and the best-performing model is selected.



The original code, using only scikit-learn, is as follows:

```
from sklearn import grid_search, datasets
from sklearn.ensemble import RandomForestClassifier
from sklearn.grid_search import GridSearchCV
digits = datasets.load_digits()
X, y = digits.data, digits.target
param_grid = {"max_depth": [3, None],
              "max_features": [1, 3, 10],
              "min_samples_split": [1, 3, 10],
              "min_samples_leaf": [1, 3, 10],
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"],
              "n_estimators": [10, 20, 40, 80]}
gs = grid_search.GridSearchCV(RandomForestClassifier(),
param_grid=param_grid)
gs.fit(X, y)
```

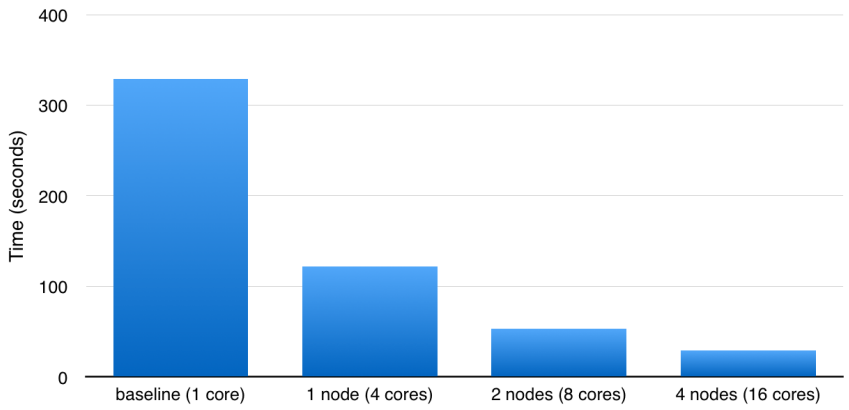
The dataset is small (in the hundreds of kilobytes), but exploring all the combinations takes about 5 minutes on a single core. The scikit-learn package for Spark provides an alternative implementation of the cross-validation algorithm that distributes the workload on a Spark cluster. Each node runs the training algorithm using a local copy of the scikit-learn library, and reports the best model back to the master:



The code is the same as before, except for a one-line change:

```
from sklearn import grid_search, datasets
from sklearn.ensemble import RandomForestClassifier
# Use spark_sklearn's grid search instead:
from spark_sklearn import GridSearchCV
digits = datasets.load_digits()
X, y = digits.data, digits.target
param_grid = {"max_depth": [3, None],
              "max_features": [1, 3, 10],
              "min_samples_split": [1, 3, 10],
              "min_samples_leaf": [1, 3, 10],
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"],
              "n_estimators": [10, 20, 40, 80]}
gs = grid_search.GridSearchCV(RandomForestClassifier(),
param_grid=param_grid)
gs.fit(X, y)
```

This example runs under 30 seconds on a 4-node cluster (which has 16 CPUs). For larger datasets and more parameter settings, the difference is even more dramatic.



Get started

If you would like to try out this package yourself, it is available as a Spark package and as a PyPI library. To get started, check out this example notebook on Databricks.

In addition to distributing ML tasks in Python across a cluster, Scikit-learn integration package for Spark provides additional tools to export data from Spark to python and vice-versa. You can find methods to convert Spark DataFrames to Pandas dataframes and numpy arrays. More details can be found in this Spark Summit Europe presentation and in the API documentation.

We welcome feedback and contributions to our open-source implementation on Github (Apache 2.0 license).

Section 2:
Machine Learning Scenarios

Simplify Machine Learning on Spark with Databricks

June 4, 2015 | by Denny Lee

Try the Dataframes and Machine Learning Notebooks in Databricks Community Edition

As many data scientists and engineers can attest, the majority of the time is spent not on the models themselves but on the supporting infrastructure. Key issues include on the ability to easily visualize, share, deploy, and schedule jobs. More disconcerting is the need for data engineers to re-implement the models developed by data scientists for production. With Databricks, data scientists and engineers can simplify these logistical issues and spend more of their time focusing on their data problems.

Simplify Visualization

An important perspective for data scientists and engineers is the ability to quickly visualize the data and the model that is generated. For example, a common issue when working with linear regression is to determine the model's goodness of fit. While statistical evaluations such as Mean Squared Error are fundamental, the ability to view the data scatterplot in relation to the regression model is just as important.

Training the models

Using a dataset comparing the population (x) with label data of median housing prices (y), we can build a linear regression model using Spark MLlib's Linear Regression with Stochastic Gradient Descent (LinearRegressionWithSGD). Spark MLlib is a core component of Apache Spark that allows data scientists and data engineers to quickly experiment and build data models – and bring them to production. Because we are experimenting with SGD, we will need to try out different iterations and learning rates (i.e. alpha or step size).

An easy way to start experimenting with these models is to create a Databricks notebook in your language of choice (python, scala, Spark SQL) and provide contextual information via markdown text. The screenshot below is two cells from an example DBC notebook where the top cell contains markdown comments while the bottom cell contains pyspark code to train two models.

***Figure 1**: Screenshot of Databricks Notebook training two models with Linear Regression with SGD*

Evaluating the models

Once the models are trained, with some additional pyspark code, you can quickly calculate the mean squared error of these two models:

```
valuesAndPreds = parsedData.map(lambda p: (p.label,
model.predict(p.features)))
MSE = valuesAndPreds.(lambda (v, p): (v - p)**2).mean()
print("Mean Squared Error = " + str(MSE))
```

The definition of the models and MSE results are in the table below.

	# of iterations	Step Size	MSE
Model A	100	0.01	1.25095190484
Model B	1500	0.1	0.205298649734

While the evaluation of statistics most likely indicates that Model B has a better goodness of fit, the ability to visually inspect the data will make it easier to validate these results.

Visualizing the models

With Databricks, there are numerous visualization options that you can use with your Databricks notebooks. In addition to the default visualizations automatically available when working with Spark DataFrames, you can also use matplotlib, ggplot, and d3.js – all embedded with the same notebook.

In our example, we are using ggplot (the python code is below) so we can not only provide a scatter plot of the original dataset (in blue), but also graph line plots of the two models where Model A is in red and Model B is in green.

```
p = ggplot(pydf, aes('x','y')) + \
    geom_point(color='blue') + \
    geom_line(pydf, aes('x','y2'), color='red') + \
    geom_line(pydf, aes('x','y3'), color='green')
display(p)
```

Embedded within the same notebook is the median housing prices ggplot scatterplot figure where the x-axis is the normalized population and y-axis is the normalized median housing price; Model A is in red while Model B is in green.

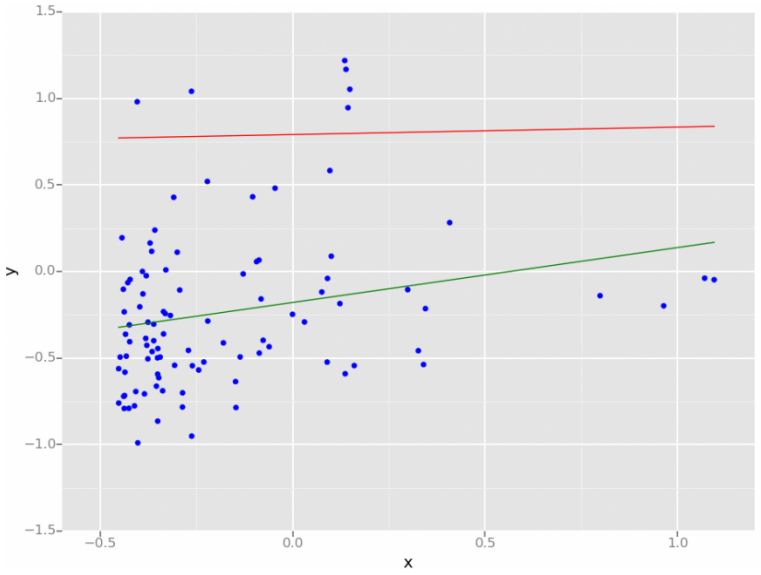


Figure 2: *Screenshot of a ggplot scatterplot embedded within a Databricks notebook*

As you can see from the above figure, the green line (Model B) has a better goodness of fit compared to the red line (Model A). While the evaluation statistics pointed toward this direction, the ability to quickly visualize the data and the models within the same notebook allows the data scientist to spend more time understanding and optimizing their models.

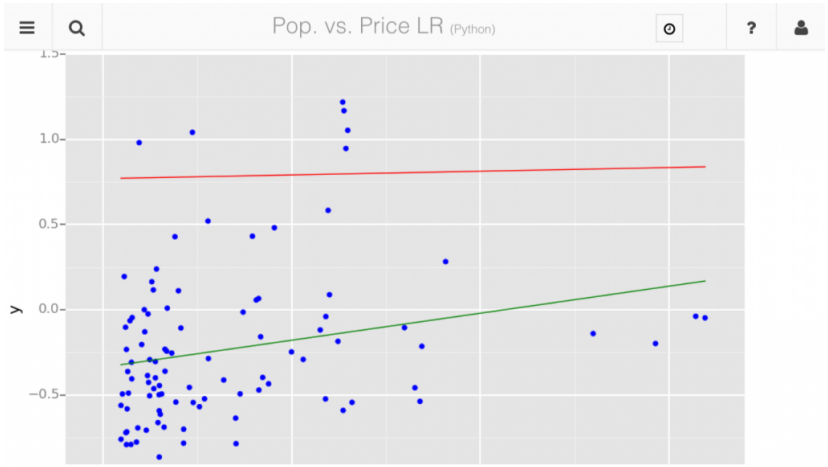


Figure 3: *Databricks notebook view of a the same linear regression SGD model via matplotlib on an iPhone 6.*

Simplify Sharing

Another crucial aspect of data sciences is the collaborative effort needed to solve data problems. With many developers, engineers, and data scientists often working in different time zones, schedules, and/or locations, it is important to have an environment that is designed for collaboration.

Portability

With Databricks, you can make it easier to collaborate with your team. You can share your Databricks notebooks by sharing its URL so that any web browser on any device can view your notebooks.

Non-proprietary

While these notebooks are optimized for Databricks, you can export these notebooks to python, scala, and SQL files so you can use them in your own environments. A common use-case for this approach is that data scientists and engineers will collaborate and experiment in Databricks and then apply their resulting code into their on-premises environment.

Share Definitions

As a data scientist or data engineer working with many different datasets, keeping up with all of the changes in schema and locations itself can be a full time job. To help keep this under control, Databricks includes centralized table definitions. Instead of searching for include files that contain the schema, go the tables tab within Databricks and you can define all of your tables in one place. This way as a data engineer updates the schema or source location for these table, these changes a re immediately available to all notebooks.

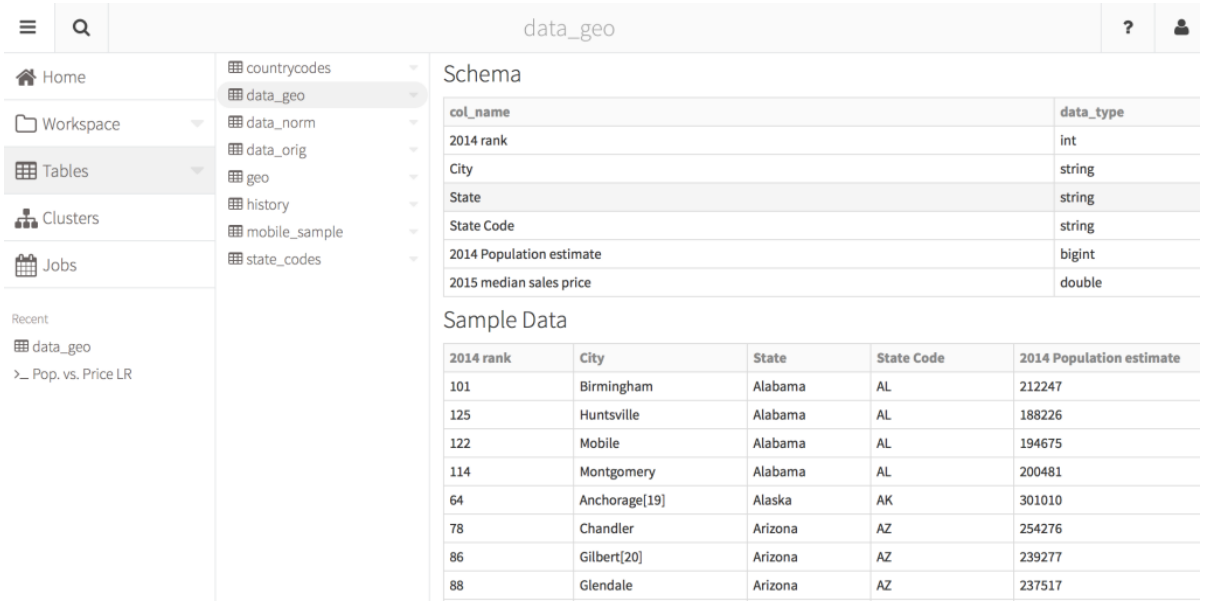


Figure 4: *View of table definitions (schema and sample data) all from one place.*

Collaborate

As notebooks are being created and shared, users can comment on the code or figures so they can provide input to the notebooks without making any changes to them. This way you can lock the notebooks to prevent accidental changes and still accept feedback.

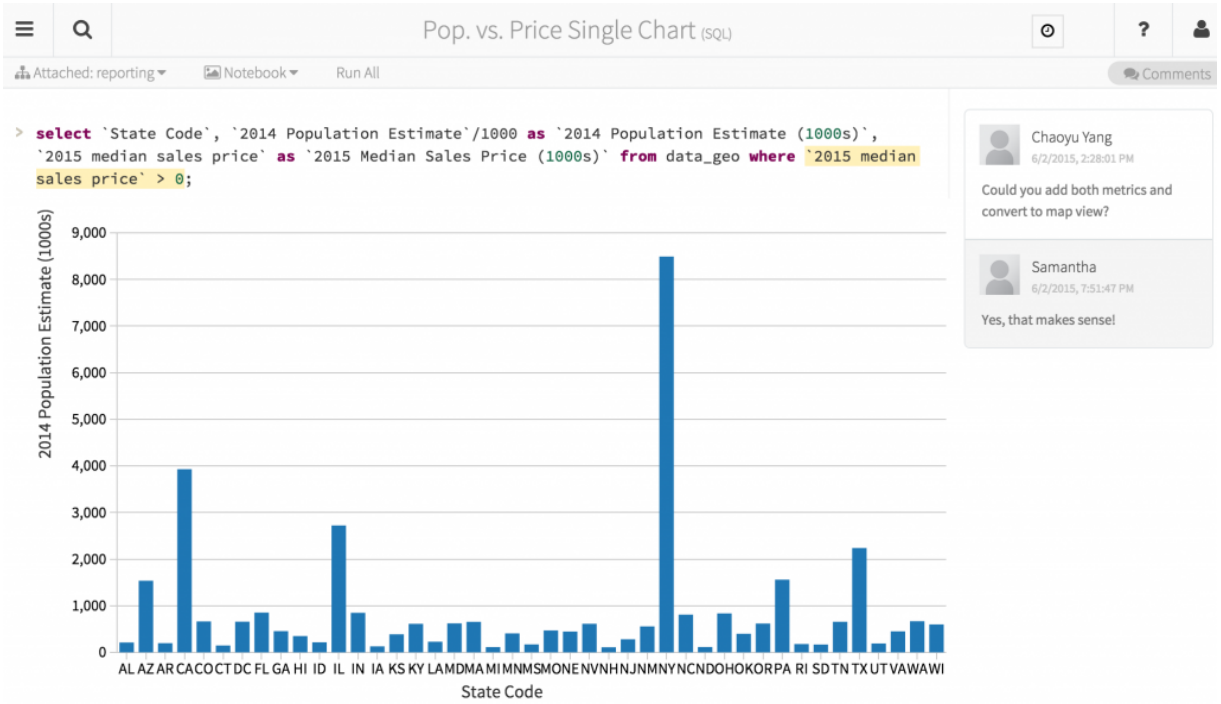


Figure 5: *Users commenting on a Databricks notebook to more easily facilitate feedback*

Simplify Deployment

One of the key advantages of Databricks is that the model developed by data scientists can be run in production. This is a huge advantage as it reduces the development cycle and tremendously simplifies the maintenance. In contrast, today data scientists develop the model using single machine tools such as R or Python and then have data engineers re-implement the model for production.

Simplify Infrastructure

As a data engineer, there are many steps and configurations to deploy Apache Spark in production. Some examples include (but are not limited to):

- Configuring High Availability and Disaster Recovery for your Spark clusters
- Building the necessary manifests to spin up and down clusters
- Configuring Spark to utilize local SSDs for fast retrieval
- Upgrading or patching your Spark clusters to the latest version of the OS or Apache Spark

With Databricks, the management of your Spark clusters are taken care by dedicated Databricks engineers who are supported by the developers and committers of the Apache Spark open source project. These clusters are configured for optimal performance and balance the issues surrounding resource scheduling, caching, and garbage collection.

Once deployed, you can quickly view what clusters are available and their current state including the libraries and notebooks that are attached to the cluster(s). Concerns around high availability, disaster recovery, manifests to build and deploy clusters, service management, configurations, patching, and upgrades are all managed on your behalf using your own (or your company's) AWS account.

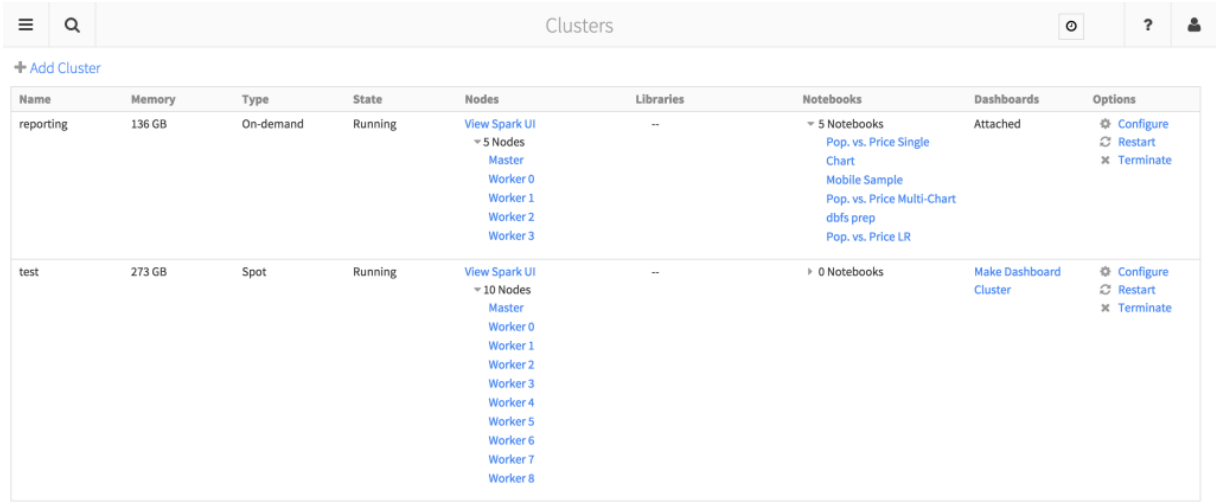


Figure 6: *Databricks Cluster view for easier management of your Databricks infrastructure*

Simplify Job Scheduling

Traditionally, transitioning from code development to production is a complicated task. It typically involves separate personnel and processes to build the code and push it into production. But Databricks has a powerful Jobs feature for running applications in production. You can take the notebook you had just created and run it as a periodic job – scheduling it minute, hourly, daily, weekly, or monthly intervals. It also has a smart cluster allocation feature that allows you to run your notebook on an existing cluster or on an on-demand cluster. You can also receive email notifications for your job as well as configure retries and timeouts.

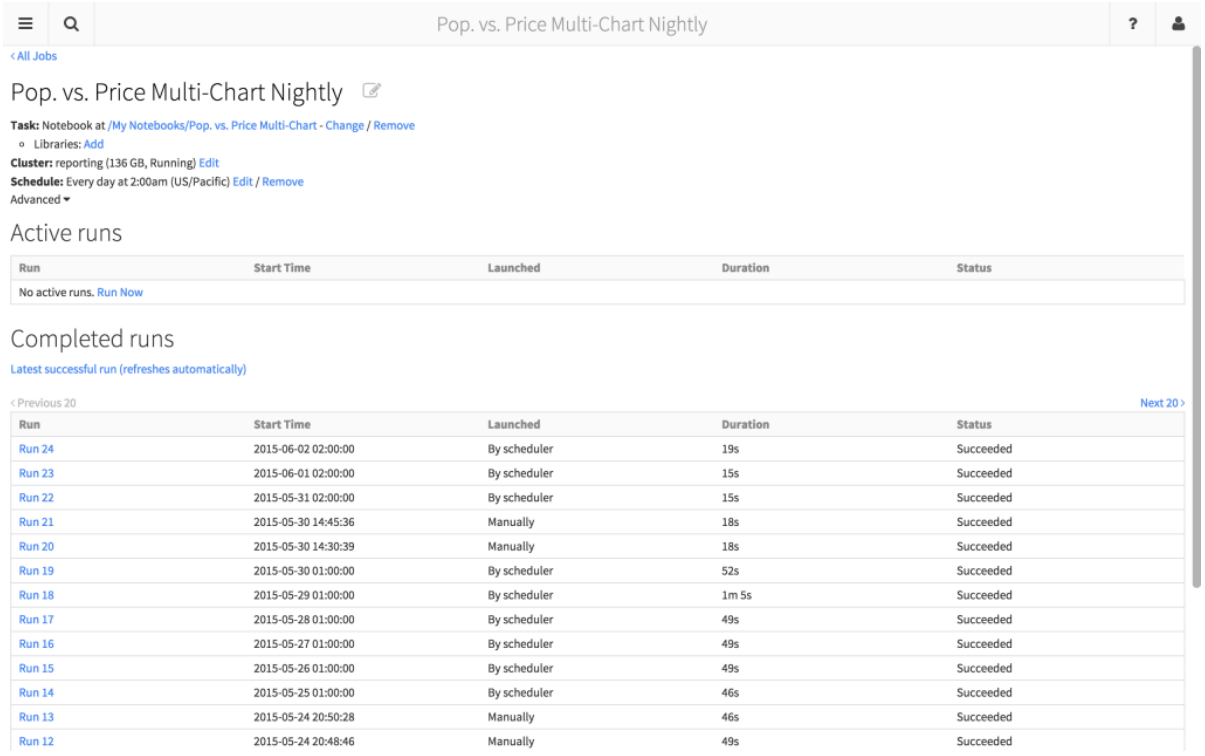


Figure 7: *View of the Population vs. Price Multi-Chart Notebook Nightly Job*

As well, you can upload and execute any Spark JAR compiled against any Spark installation within the Jobs feature. Therefore any previous work can be used immediately instead of recreating and rebuilding the code-base.

Try Databricks

We created Databricks to make it easier for data scientists and data engineers to focus on experimenting and training their models, quickly deploy and schedule jobs against those models, easily collaborate and share their learnings, and easily share the schema and definitions for their datasets. Let us manage the cluster, configure it for optimal performance, perform upgrades and patches, and ensure high availability and disaster recovery.

Machine Learning with Spark MLlib is a lot more fun when you get to spend most of your time doing Machine Learning.

Try Databricks today with a trial account.
databricks.com/try-databricks

Visualizing Machine Learning Models

October 27, 2015 | by Joseph Bradley, Feynman Liang, Tim Hunter and Raela Wang

You've built your machine learning models and evaluated them with error metrics, but do the numbers make sense? Being able to visualize models is often a vital step in advanced analytics as it is usually easier to understand a diagram than numbers in a table.

Databricks has a built-in `display()` command that can display DataFrames as a table and create convenient one-click plots. Recently, we have extended the `display()` command to visualize machine learning models as well.

In this post, we will look at how easy visualization can be with Databricks — a quick `display()` command can give you immediate feedback about complex models!

Linear Models: Fitted vs Residuals

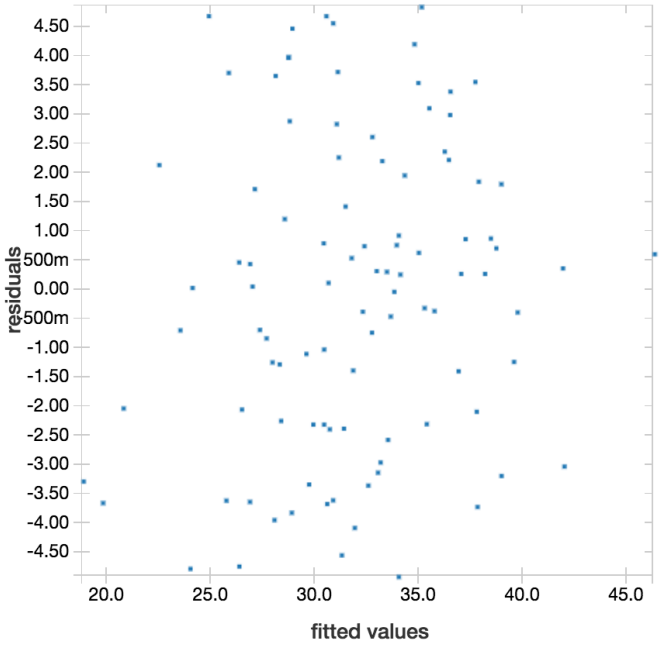
Scala-only

The Fitted vs Residuals plot is available for Linear Regression and Logistic Regression models. The Databricks' Fitted vs Residuals plot is analogous to R's "Residuals vs Fitted" plots for linear models.

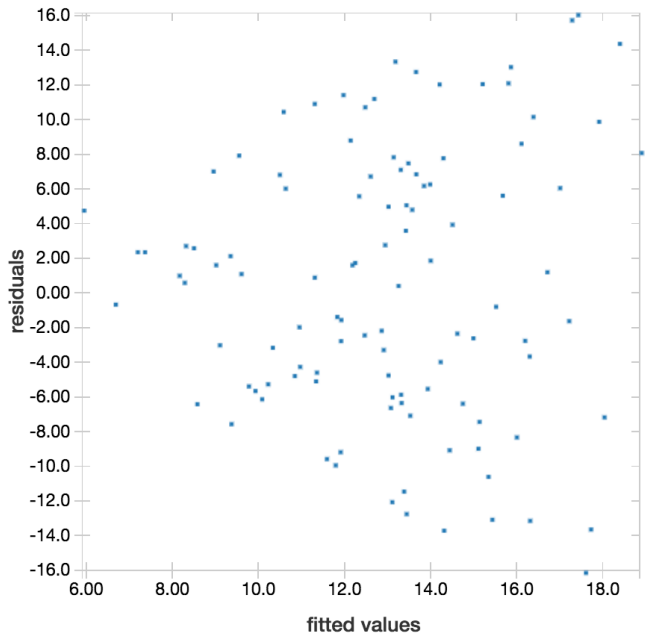
Here, we will look at how these plots are used with Linear Regression.

Linear Regression computes a prediction as a weighted sum of the input variables. The Fitted vs Residuals plot can be used to assess a linear regression model's goodness of fit.

```
display(linearModel, data, plotType="fittedVsResiduals")
```



The previous image is an example of a fitted vs residuals plot for a linear regression model that is returning good predictions. A good linear model will usually have residuals distributed randomly around the residuals=0 line with no distinct outliers and no clear trends. The residuals should also be small for the whole range of fitted values.



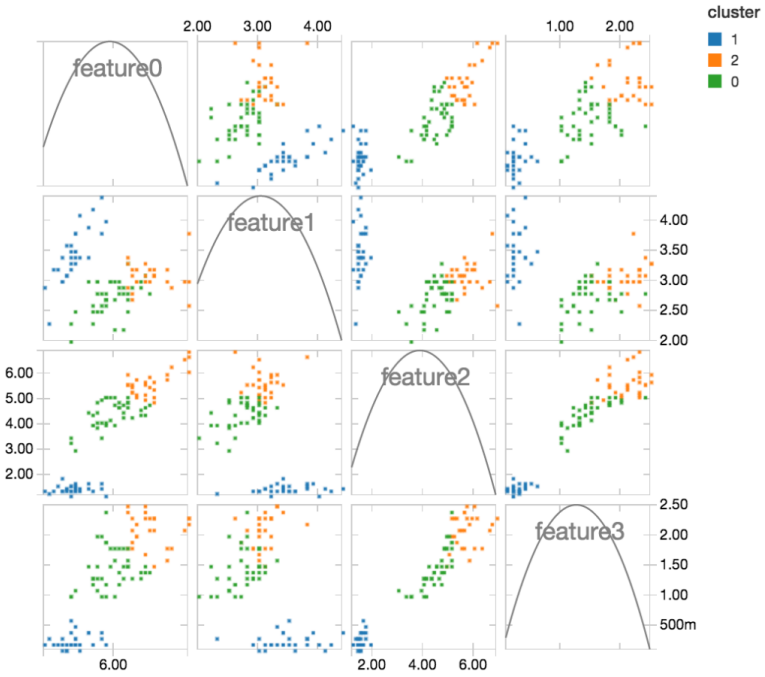
In comparison, this visualization is a warning sign for this linear regression model: the range of residuals increases as the fitted values increase. This could mean we should evaluate using relative error instead of absolute error.

K-means Clustering: Visualizing Clusters

K-means tries to separate data points into clusters by minimizing the sum of squared errors between data points and their nearest cluster centers.

We can now visualize clusters and plot feature grids to identify trends and correlations. Each plot in the grid corresponds to 2 features, and data points are colored by their respective cluster labels. The plots can be used to visually assess how well your data have been clustered.

```
display(kMeansModel, data)
```



From these plots, we notice that clusters 0 and 2 are sometimes overlapping with each other for some features, whereas cluster 1 is always cleanly separated from the rest. Features 2 and 3 are particularly useful for distinguishing cluster 1.

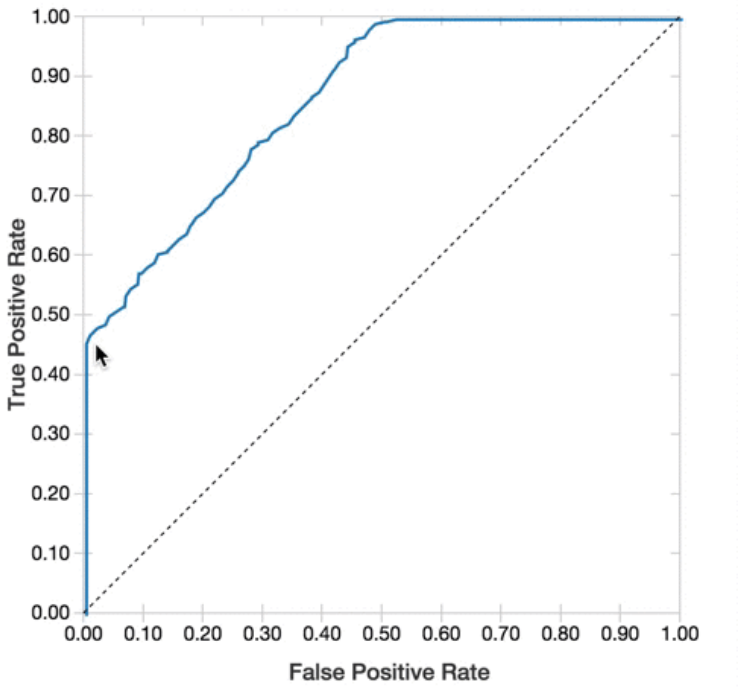
Logistic Regression: ROC Curves

Scala-only, with clusters running Spark 1.5 or higher

Logistic Regression is widely used for binary classification, where a logistic function is used to model the class probabilities of your data.

Logistic Regression converts a numerical class probability into a binary (0/1) label using a threshold, and adjusting the threshold allows you to adjust the probability cutoff for predicting 0 vs. 1. To review how your model performs over various thresholds, you can easily plot your model's ROC Curve with the `display()` command. The plot will also interactively display threshold values on mouseover.

```
display(logisticModel, data, plotType="ROC")
```



The dotted diagonal line represents how a model will perform if it randomly guesses every prediction, and the (0.00,1.00) point in the top left corner represents a perfect classification. From the above curve, it is clear that our model is doing much better than random guessing, and we can adjust the threshold based on how much we value true positive vs. false positive predictions.

To see an example of these visualizations and try out the interactive display, check out the exported notebook here.

What's Next?

The plots listed above as Scala-only will soon be available in Python notebooks as well. Note, K-means clustering visualization is already available in Python; you can find an example at Databricks Guide > Visualization of Machine Learning Models (Python). There is also the decision tree visualization in Scala; you can find an example at Databricks Guide > Visualization of Machine Learning (Scala).

Stay tuned for Machine Learning Pipeline visualizations!

On-Time Flight Performance with GraphFrames for Apache Spark

March 16, 2016 | by Denny Lee, Joseph Bradley, Bill Chambers and Jake Bellacera

This blog includes the full On-Time Flight Performance with GraphFrames notebook which includes more extensive examples you can try with the Databricks Community Edition for free.

Introduction

Graph structures are a more intuitive approach to many classes of data problems. Whether traversing social networks, restaurant recommendations, or flight paths, it is easier to understand these data problems within the context of graph structures: vertices, edges, and properties. For example, the analysis of flight data is a classic graph problem as airports are represented by *vertices* and flights are represented by *edges*. As well, there are numerous *properties* associated with these flights including but not limited to departure delays, plane type, and carrier.

In this post, we will use GraphFrames (as recently announced in Introducing GraphFrames) within Databricks notebooks to quickly and easily analyze flight performance data organized in graph structures. Because we're using graph structures, we can easily ask a number of questions that are not as intuitive as tabular structures such as finding structural motifs, airport ranking using PageRank, and shortest paths between cities. GraphFrames leverage the distribution and expression capabilities of the DataFrame API to both simplify your queries and leverage the performance optimizations of the Spark SQL engine. In addition, with GraphFrames, graph analysis is available in Python, Scala, and Java.

Install the GraphFrames Spark Package

To use GraphFrames, you will first need to install the GraphFrames Spark Packages. Installing packages in Databricks is a few simple steps

Note, to reference GraphFrames within spark-shell, pyspark, or spark-submit:

```
$SPARK_HOME/bin/spark-shell --packages graphframes:graphframes:
0.1.0-spark1.6
```

Preparing the Flight Datasets

The two sets of data that make up our graphs are the **`airports`** dataset (vertices) which can be found at OpenFlights Airport, airline and route data and the **`departuredelays`** dataset (edges) which can be found at Airline On-Time Performance and Causes of Flight Delays: On_Time Data.

After installing the GraphFrames Spark Package, you can import it and create your vertices, edges, and GraphFrame (in PySpark) as noted to the right.

```
# Import graphframes (from Spark-Packages)
from graphframes import *

# Create Vertices (airports) and Edges (flights)
tripVertices = airports.withColumnRenamed("IATA",
"id").distinct()
tripEdges = departureDelays.select("tripid", "delay", "src",
"dst", "city_dst", "state_dst")

# This GraphFrame builds upon the vertices and edges based on our
trips (flights)
tripGraph = GraphFrame(tripVertices, tripEdges)
```

For example, the tripEdges contains the flight data identifying the *origin* IATA airport code (src) and the *destination* IATA airport code (dst), city (city_dst), and state (state_dst) as well as the departure delays (delay).

```
> # Edges
  #  The edges of our graph are the flights between airports
  display(tripEdges)
```

▸ (1) Spark Jobs

tripid	delay	src	dst	city_dst	state_dst
3010630	4	BFL	IAH	Houston	TX
3020630	-3	BFL	IAH	Houston	TX
3021124	27	BFL	IAH	Houston	TX
3030630	-3	BFL	IAH	Houston	TX
3031124	34	BFL	IAH	Houston	TX
3040630	-8	BFL	IAH	Houston	TX
3041124	105	BFL	IAH	Houston	TX
3050630	-11	BFL	IAH	Houston	TX
3051124	5	BFL	IAH	Houston	TX

Showing the first 1000 rows.

Simple Queries against the tripGraph GraphFrame

Now that you have created your tripGraph GraphFrame, you can run a number of simple queries to quickly traverse and understand your GraphFrame. For example, to **understand the number of airports and trips** in your GraphFrame, run the PySpark code below.

```
print "Airports: %d" % tripGraph.vertices.count()
print "Trips: %d" % tripGraph.edges.count()
```

Which returns the output:

```
Airports: 279
Trips: 1361141
```

Because GraphFrames are DataFrame-based Graphs in Spark, you can write highly expressive queries leveraging the DataFrame API. For example, the query below allows us to filter flights (edges) for delayed flights (delay > 0) originating from SFO airport where we calculate and sort by the average delay, i.e. **What flights departing from SFO are most likely to have significant delays?**

```
tripGraph.edges\
  .filter("src = 'SFO' and delay > 0")\
  .groupBy("src", "dst")\
  .avg("delay")\
  .sort(desc("avg(delay)"))
```

Reviewing the output, you will quickly identify there are significant average delays to Will Rogers World Airport (OKC), Jackson Hole (JAC), and Colorado Springs (COS) from SFO in this dataset.

```
> display(tripGraph.edges.filter("src = 'SFO' and delay > 0").groupBy("src",
  "dst").avg("delay").sort(desc("avg(delay)")))
```

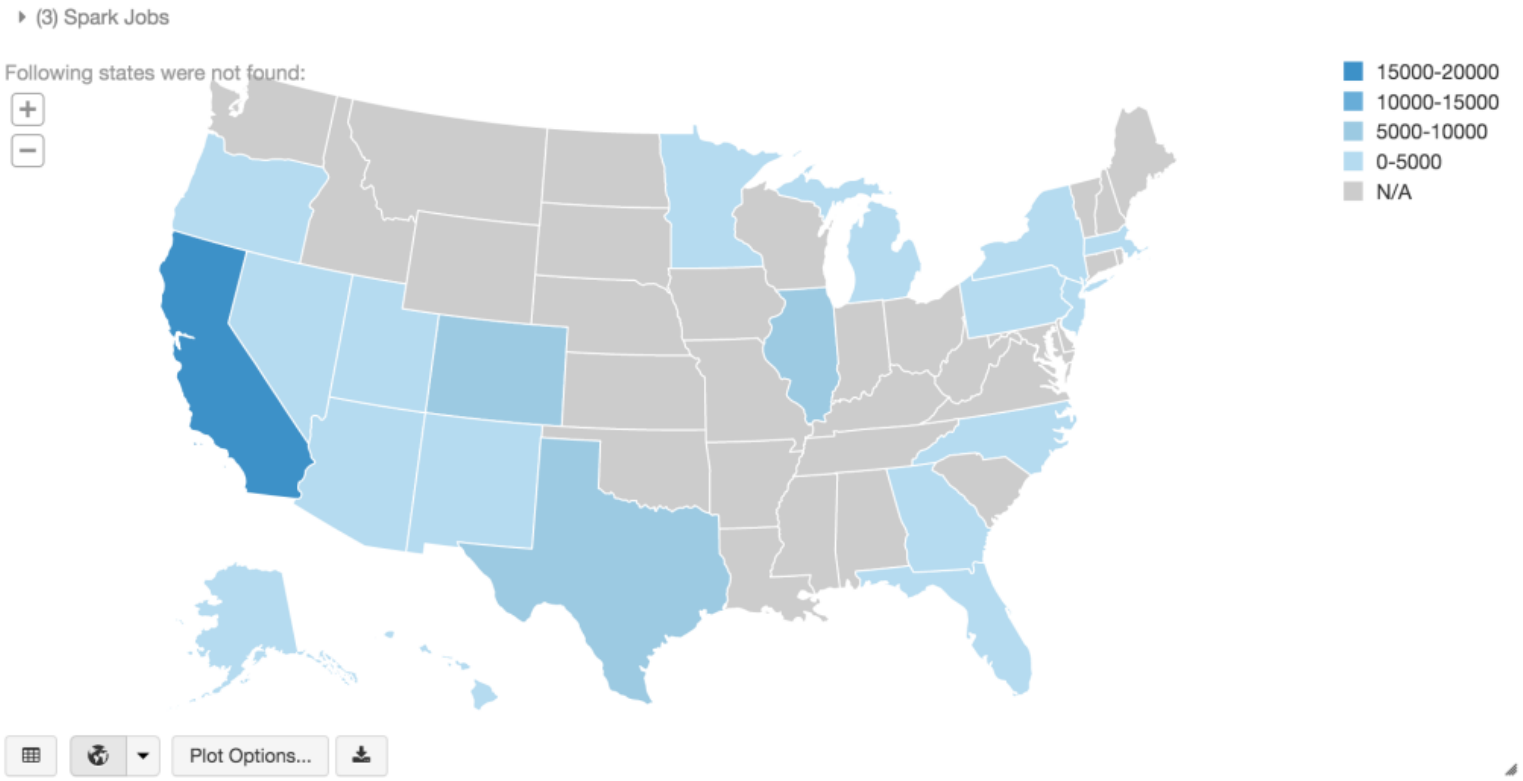
▸ (1) Spark Jobs

src	dst	avg(delay)
SFO	OKC	59.073170731707314
SFO	JAC	57.13333333333333
SFO	COS	53.976190476190474
SFO	OTH	48.09090909090909
SFO	SAT	47.625
SFO	MOD	46.80952380952381

With Databricks notebooks, we can also quickly visualize geographically:

What destination states tend to have significant delays departing from SEA?

```
> # States with the longest cummulative delays (with individual delays > 100 minutes) (origin: Seattle)
  display(tripGraph.edges.filter("src = 'SEA' and delay > 100"))
```



Using Motif Finding to understand flight delays

To more easily understand the complex relationship of city airports and their flights with each other, we can use motifs to find patterns of airports (i.e. vertices) connected by flights (i.e. edges). The result is a DataFrame in which the column names are given by the motif keys.

For example, to ask the question **What delays might we blame on SFO?**, you can generate the simplified motif below.

```
motifs = tripGraphPrime.find("(a)-[ab]->(b); (b)-[bc]->(c)")\
  .filter("(b.id = 'SFO') and (ab.delay > 500 or bc.delay > 500)
and bc.tripid > ab.tripid and bc.tripid > ab.tripid + 10000")
display(motifs)
```

With SFO as the connecting city (b), we are looking for all flights [ab] from any origin city (a) that will connect to SFO (b) prior to flying [bc] to any destination city (c). We are filtering it such that the delay for either flight ([ab] or [bc]) is greater than 500 minutes and the second flight (bc) occurred within approximately a day of the first flight (ab).

Below is an abridged subset from this query where the columns are the respective motif keys.

a	ab	b	bc	c
Houston (IAH)	IAH -> SFO (-4) [1011126]	San Francisco (SFO)	SFO -> JFK (536) [1021507]	New York (JFK)
Tuscon (TUS)	TUS -> SFO (-5) [1011126]	San Francisco (SFO)	SFO -> JFK (536) [1021507]	New York (JFK)

With this motif finding query, we have quickly determined that passengers in this dataset left Houston and Tuscon for San Francisco on time or a little early [1011126]. But for any of those passengers that were flying to New York through this connecting flight in SFO [1021507], they were delayed by 536 minutes.

Using PageRank to find the most important airport

Because GraphFrames is built on GraphX, there are a number of built-in algorithms that we can leverage right away. PageRank was popularized by the Google Search Engine and created by Larry Page. To quote Wikipedia:

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

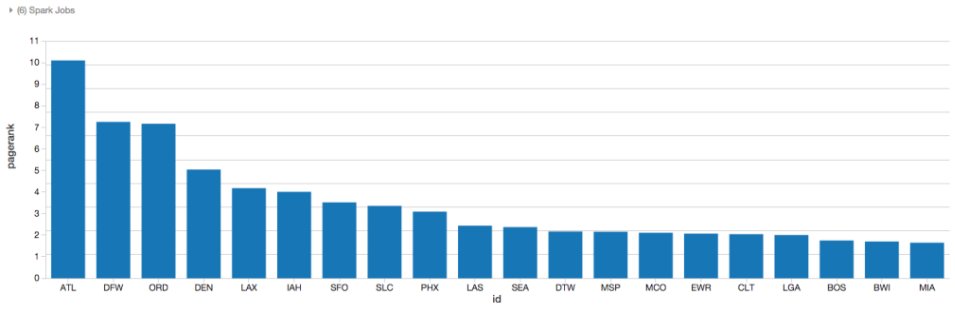
While the above example refers to web pages, what's awesome about this concept is that it readily applies to any graph structure whether it is created from web pages, bike stations, or airports and the interface is as simple as calling a method. You'll also notice that GraphFrames will return the PageRank results as a new column appended to the vertices DataFrame for a simple way to continue our analysis after running the algorithm!

As there are a large number of flights and connections through the various airports included in this dataset, we can use the PageRank algorithm to have Spark traverse the graph iteratively to compute a rough estimate of how important each airport is.

```
# Determining Airport ranking of importance using pageRank
ranks = tripGraph.pageRank(resetProbability=0.15, maxIter=5)

display(ranks.vertices.orderBy(ranks.vertices.pagerank.desc()).lim
it(20))
```

As noted in the chart below, using the PageRank algorithm, Atlanta is considered one of the most important airports based on the quality of connections (i.e. flights) between the different vertices (i.e. airports); corresponding to the fact that Atlanta is the busiest airport in the world by passenger traffic.



Determining flight connections

With so many flights between various cities, you can use the `GraphFrames.bfs` (Breadth First Search) method to find the paths between two cities. The query below attempts to find the path between San Francisco (SFO) and Buffalo (BUF) with a maximum path length of 1 (i.e direct flight). The results set is empty (i.e. no direct flights between SFO and BUF).

```
filteredPaths = tripGraph.bfs(
  fromExpr = "id = 'SFO'",
  toExpr = "id = 'BUF'",
  maxPathLength = 1)
display(filteredPaths)
```

So let's extend the query to have a `maxPathLength = 2`, that is having one connecting flight between SFO and BUF.

```
filteredPaths = tripGraph.bfs(
  fromExpr = "id = 'SFO'",
  toExpr = "id = 'BUF'",
  maxPathLength = 2)
display(filteredPaths)
```

An abridged subset of the paths from SFO to BUF can be seen in the table below.

from	v1	to
SFO	MSP (Minneapolis)	BUF
SFO	EWR (Newark)	BUF
SFO	JFK (New York)	BUF
SFO	ORD (Chicago)	BUF
SFO	ATL (Atlanta)	BUF
SFO	LAS (Las Vegas)	BUF
SFO	BOS (Boston)	BUF
…	…	…

Visualizing Flights Using D3

To get a powerful visualization of the flight paths and connections in this dataset, we can leverage the Airports D3 visualization within our Databricks notebook. By connecting our GraphFrames, DataFrames, and D3 visualizations, we can visualize the scope of all of the flight connections as noted below for all on-time or early departing flights within this dataset. The blue circles represent the vertices (i.e. airports) where the size of the circle represents the number of edges (i.e. flights) in and out of those airports. The black lines are the edges themselves (i.e. flights) and their respective connections to the other vertices (i.e. airports). Note for any edges that go offscreen, they are representing vertices (i.e. airports) in the states of Hawaii and Alaska.

See the full animation here.

Next: Try for yourself

You can view the full On-Time Flight Performance with GraphFrames notebook which includes more extensive examples. You can also import the notebook into your Databricks account to execute the notebook end-to-end with these simple few steps. Don't have a Databricks account? Try Databricks Community Edition to get access to Apache Spark for free.

Mining Ecommerce Graph Data with Spark at Alibaba Taobao

August 14, 2014 | by Andy Huang and Wei Wu

This is a guest blog post from our friends, formerly at Alibaba Taobao.

Alibaba Taobao operates one of the world's largest e-commerce platforms. We collect hundreds of petabytes of data on this platform and use Spark to analyze these enormous amounts of data. Alibaba Taobao probably runs some of the largest Spark jobs in the world. For example, some Spark jobs run for weeks to perform feature extraction on petabytes of image data. In this blog post, we share our experience with Spark and GraphX from prototype to production at the Alibaba Taobao Data Mining Team.

Every day, hundreds of millions of users and merchants interact on Alibaba Taobao's marketplace. These interactions can be expressed as complicated, large scale graphs. Mining data requires a distributed data processing engine that can support fast interactive queries as well as sophisticated algorithms.

Spark and GraphX embed a standard set of graph mining algorithms, including PageRank, triangle counting, connected components, shortest path. The implementation of these algorithms focuses on reusability.

Users can implement variants of these algorithms in order to exploit performance optimization opportunities for specific workloads. In our experience, the best way to learn GraphX is to read and understand the source code of these algorithms.

Alibaba Taobao started prototyping with GraphX in Spark 0.9 and went into production in May 2014 around the time that Spark 1.0 was released.

One thing to note is that GraphX is still evolving quickly. Although the user-facing APIs are relatively stable, the internals have seen fairly large refactoring and improvements from 0.8 to 1.0. Based on our experience, each minor version upgrade provided 10 – 20% performance improvements even without modifying our application code.

Graph Inspection Platform

Graph-based structures model the many relationships between our users and items in our store. Our business and product teams constantly need to make decisions based on the value and health of each relationship. Before Spark, they used their intuition to estimate such properties, resulting in decisions which were not a good fit with reality. To solve this problem, we developed a platform to scientifically quantify all these metrics in order to provide evidence and insights for product decisions.

This platform requires constantly re-iterating the set of metrics it provides to users, depending on product demand. The interactive nature of both Spark and GraphX proves very valuable in building this platform. Some of the metrics this platform measures are:

Degree Distribution: Degree distribution measures the distribution of vertex degrees (e.g. how many users have 50 friends). It also provides valuable information on the number of high degree vertices (so-called super vertices). Often our downstream product infrastructure needs to accommodate super vertices in a special manner (because they have a high impact on propagation algorithms), and thus it is crucial to understand their distribution among our data. GraphX's VertexRDD provides built-in support for both in-degrees and out-degrees.

Second Degree Neighbors: Modeling social relationships often requires measuring the second-degree neighbor distribution. For example, in an instant messaging platform we developed, the number of "retweets" correlates with the number of second degree neighbors (e.g. number of friends of friends). While GraphX does not yet provide built-in support for counting second degree neighbors, we implemented it using two rounds of propagations: the first round propagates each vertex's ID to its neighbors, and the second round propagates all IDs from neighbors to second degree neighbors. After the two rounds of propagations, each vertex calculates the number of second degree neighbors using a hash set.

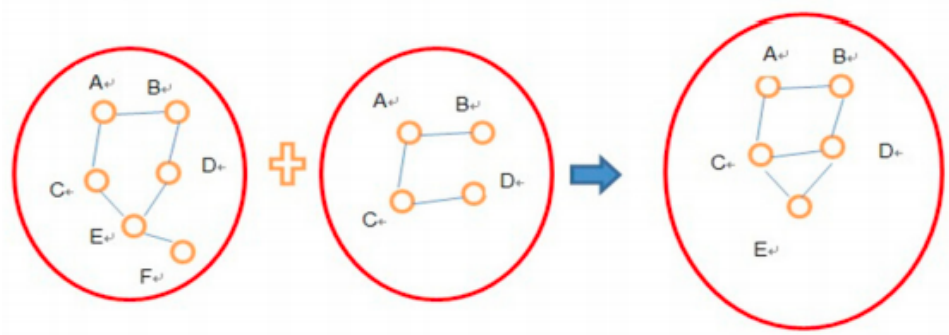
One thing to note in this calculation is that we use the aforementioned degree distribution to remove super vertices from the second degree neighbor calculation. Otherwise, these super vertices would create too many messages, leading to high computation skew and high memory usage.

Connected Components: Connected components refer to some set of subgraphs that are "connected", i.e. there exists a path connecting any pair of vertices in the subgraph. Connected component is very useful in dividing a large graph into multiple, smaller graphs, and then operations that are computationally too expensive to run on the large graph. This algorithm can also be adapted to discover tightly connected networks.

We are developing more metrics using both built-in functions provided by Spark and GraphX, as well as new ones implemented internally. This platform nurtures a new culture such that our product decisions are no longer based on instinct and intuition, but rather on metrics mined from data.

Multi-graph Merging

The Graph Inspection Platform provides us with different properties for modeling relationships. Each relationship structure has its own strengths and weaknesses. For example, some relationship structure provides more valuable information in connected components, while another other structure might work better for interactions. We often make decisions based on multiple different properties and structural representations of the same underlying graph. Based on GraphX, we developed a multi-graph merging framework that creates "intersections" of multiple graphs.



The above figure illustrates the algorithm to merge graph A and graph B to create graph C: edges are created in graph C if any of its vertices exist in graph A or graph B.

This merging framework was implemented using the outerJoinVertices operator provided by GraphX. In addition to naively merging two graphs, the framework can also assign different weights to the input graphs. In practice, our analysis pipelines often merge multiple graphs in a variety of ways and run them on the Graph Inspection Platform.

Belief Propagation

Weighted belief propagation is a classic way of modeling graph data, often used to predict a user's influence or credibility. The intuition is simple: highly credited users often interact with other highly credited users, while lower credited users often interact with other lower credited users. Although the algorithm is simple, historically we did not attempt to run these on our entire graph due to the computation cost to scale this to hundreds of millions of users and billions of interactions. Using GraphX, we are able to scale this analysis to the entire graphs we have.

Each run of the algorithm requires 3 iterations, and each iteration requires 8 iterations in GraphX's Pregel API. After a total of 30 iterations in Pregel, the AUC (area under the curve) increased from 0.6 to 0.9, which is a very satisfactory prediction rate.

While we are still in the early stages of our journey with GraphX, already today we have been able to generate impressive insights with graph modeling and analysis that would have been very hard to accomplish at our scale without GraphX. We plan to enrich and further develop our various platforms and frameworks to include an even wider array of metrics and apply them to tag/topic inference, demographics inference, transaction prediction, which will in turn improve our various recommendation systems' effectiveness.

Authors

Andy Huang leads the data mining team at Taobao. He is a very early adopter of Spark, using it production since Spark 0.5.

Wei Wu is an engineer at Taobao's data mining team. His interests span distributed systems, large-scale machine learning and data mining.

This Guest Blog Post Is A Translation Of Part Of An Article Published By Csdn Programmer Magazine.

Audience Modeling With Spark ML Pipelines

October 20, 2015 | by Eugene Zhulenev

This is a guest blog from Eugene Zhulenev on his experiences with Engineering Machine Learning and Audience Modeling at Collective.

At Collective, we heavily rely on machine learning and predictive modeling to run our digital advertising business. All decisions about what ad to show at this particular time to this particular user are made by machine learning models (some of them are in real-time while some of them are offline).

We have a lot of projects that uses machine learning, the common name for all of them can be **Audience Modeling**, as they all are trying to predict audience conversion (*CTR, Viewability Rate, etc…*) based on browsing history, behavioral segments, and other type of predictors.

For most of our new development, we use Spark and Spark MLLib. However, while it is an awesome project, we found that there are some widely used tools and libraries that are missing in Spark. To add those missing features that we would really like to have in Spark, we created Spark Ext (Spark Extensions Library).

Spark Ext on Github: https://github.com/collectivemedia/spark-ext

I'm going to show a simple example of combining Spark Ext with Spark ML pipelines for predicting user conversions based on geo and browsing history data.

Spark ML pipeline example: SparkMlExtExample.scala

Predictors Data

I'm using a dataset with 2 classes, that will be used for solving classification problem (user converted or not). It's created with dummy data generator so that these 2 classes can be easily separated. It's pretty similar to real data that usually is available in digital advertising.

Browsing History Log

History of websites that were visited by a user.

```
Cookie          | Site          | Impressions
--------------- |-------------- | -------------
wKgQaV0lHZanDrp | live.com      | 24
wKgQaV0lHZanDrp | pinterest.com | 21
rfTZLbQDwbu5mXV | wikipedia.org | 14
rfTZLbQDwbu5mXV | live.com      | 1
rfTZLbQDwbu5mXV | amazon.com    | 1
r1CSY234HTYdvE3 | youtube.com   | 10
```

Geo Location Log

Latitude/Longitude impression history.

```
Cookie          | Lat     | Lng       | Impressions
--------------- |---------| --------- | ------------
wKgQaV0lHZanDrp | 34.8454 | 77.009742 | 13
wKgQaV0lHZanDrp | 31.8657 | 114.66142 | 1
rfTZLbQDwbu5mXV | 41.1428 | 74.039600 | 20
rfTZLbQDwbu5mXV | 36.6151 | 119.22396 | 4
r1CSY234HTYdvE3 | 42.6732 | 73.454185 | 4
r1CSY234HTYdvE3 | 35.6317 | 120.55839 | 5
20ep6ddsVckCmFy | 42.3448 | 70.730607 | 21
20ep6ddsVckCmFy | 29.8979 | 117.51683 | 1
```

Transforming Predictors Data

As you can see the predictors data (sites and geo) is in *long* format. Each `cookie` has multiple rows associated with it; in general, it is not a good fit for machine learning. We'd like `cookie` to be a primary key while all other data should form the `feature vector`.

Gather Transformer

Inspired by R `tidyr` and `reshape2` packages, we convert a *long* `DataFrame` with values for each key into a *wide* `DataFrame` and apply an aggregation function if the single key has multiple values.

```
val gather = new Gather()
      .setPrimaryKeyCols("cookie")
      .setKeyCol("site")
      .setValueCol("impressions")
```

```
Cookie          | Sites
----------------|-----------------------------------------------
wKgQaV0lHZanDrp | [
                |  { site: live.com, impressions: 24.0 },
                |  { site: pinterest.com, impressions: 21.0 }
                | ]
rfTZLbQDwbu5mXV | [
                |  { site: wikipedia.org, impressions: 14.0 },
                |  { site: live.com, impressions: 1.0 },
                |  { site: amazon.com, impressions: 1.0 }
                | ]
```

Google S2 Geometry Cell Id Transformer

The S2 Geometry Library is a spherical geometry library, very useful for manipulating regions on the sphere (commonly on Earth) and indexing geographic data. Basically, it assigns a unique cell id for each region on the earth.

Good article about S2 library: Google's S2, geometry on the sphere, cells, and Hilbert curve

For example, you can combine S2 transformer with `Gather` to convert `lat/lon` to `K-V` pairs, where the key will be `S2` cell id. Depending on a level you can assign all people in Greater New York area (level = 4) into one cell, or you can index them block by block (level = 12).

```
// Transform lat/lon into S2 Cell Id
val s2Transformer = new S2CellTransformer()
  .setLevel(5)
  .setCellCol("s2_cell")

// Gather S2 CellId log
val gatherS2Cells = new Gather()
  .setPrimaryKeyCols("cookie")
  .setKeyCol("s2_cell")
  .setValueCol("impressions")
  .setOutputCol("s2_cells")

val gatheredCells =
gatherS2Cells.transform(s2Transformer.transform(geoDf))
```

```
Cookie           | S2 Cells
-----------------|-----------------------------------------------
wKgQaV0lHZanDrp  | [
                 |  { s2_cell: d5dgds, impressions: 5.0 },
                 |  { s2_cell: b8dsgd, impressions: 1.0 }
                 | ]
rfTZLbQDwbu5mXV  | [
                 |  { s2_cell: d5dgds, impressions: 12.0 },
                 |  { s2_cell: b8dsgd, impressions: 3.0 },
                 |  { s2_cell: g7aeg3, impressions: 5.0 }
                 | ]
```

Assembling Feature Vector

K-V pairs from the result of **Gather** are cool, and groups all the information about the cookie into a single row. However, they cannot be used as input for machine learning. To be able to train a model, the predictors data need to be represented as a vector of doubles. This is easy to do if all the features are continuous and numeric. But if some of them are categorical or in **gathered** shape, this is not a trivial task.

Gather Encoder

Encodes categorical key-value pairs using dummy variables.

```
// Encode S2 Cell data
val encodeS2Cells = new GatherEncoder()
  .setInputCol("s2_cells")
  .setOutputCol("s2_cells_f")
  .setKeyCol("s2_cell")
  .setValueCol("impressions")
  .setCover(0.95) // dimensionality reduction
```

```
Cookie           | S2 Cells
-----------------|-----------------------------------------------
wKgQaV0lHZanDrp  | [
                 |  { s2_cell: d5dgds, impressions: 5.0 },
                 |  { s2_cell: b8dsgd, impressions: 1.0 }
                 | ]
rfTZLbQDwbu5mXV  | [
                 |  { s2_cell: d5dgds, impressions: 12.0 },
                 |  { s2_cell: g7aeg3, impressions: 5.0 }
                 | ]
```

Transformed into

```
Cookie           | S2 Cells Features
-----------------|-----------------------
wKgQaV0lHZanDrp  | [ 5.0  ,  1.0 , 0   ]
rfTZLbQDwbu5mXV  | [ 12.0 ,  0   , 5.0 ]
```

Note that it's 3 unique cell id values, that gives 3 columns in the final feature vector.

Optionally apply dimensionality reduction using `top` transformation:

- Top coverage, is selecting categorical values by computing the count of distinct users for each value, sorting the values in descending order by the count of users, and choosing the top values from the resulting list such that the sum of the distinct user counts over these values covers c percent of all users (e.g. selecting top sites covering 99% of users).

Spark ML Pipelines

Spark ML Pipeline – is new high-level API for Spark MLLib.

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. Read More.

In Spark ML it's possible to split an ML pipeline into multiple independent stages, group them together in a single pipeline and run it with Cross-Validation and Parameter Grid to find the best set of parameters.

Put It All together with Spark ML Pipelines

Gather encoder is a natural fit into Spark ML Pipeline API.

```
// Encode site data
val encodeSites = new GatherEncoder()
  .setInputCol("sites")
  .setOutputCol("sites_f")
  .setKeyCol("site")
  .setValueCol("impressions")

// Encode S2 Cell data
val encodeS2Cells = new GatherEncoder()
  .setInputCol("s2_cells")
  .setOutputCol("s2_cells_f")
  .setKeyCol("s2_cell")
  .setValueCol("impressions")
  .setCover(0.95)

// Assemble feature vectors together
val assemble = new VectorAssembler()
  .setInputCols(Array("sites_f", "s2_cells_f"))
  .setOutputCol("features")

// Build logistic regression
val lr = new LogisticRegression()
  .setFeaturesCol("features")
  .setLabelCol("response")
  .setProbabilityCol("probability")

// Define pipeline with 4 stages
val pipeline = new Pipeline()
  .setStages(Array(encodeSites, encodeS2Cells, assemble, lr))

val evaluator = new BinaryClassificationEvaluator()
  .setLabelCol(Response.response)

val crossValidator = new CrossValidator()
  .setEstimator(pipeline)
  .setEvaluator(evaluator)

val paramGrid = new ParamGridBuilder()
  .addGrid(lr.elasticNetParam, Array(0.1, 0.5))
  .build()

crossValidator.setEstimatorParamMaps(paramGrid)
crossValidator.setNumFolds(2)

println(s"Train model on train set")
val cvModel = crossValidator.fit(trainSet)
```

Conclusion

The Spark ML API makes machine learning much easier. Spark Ext is a good example of how it is possible to create custom transformers/ estimators that later can be used as a part of a bigger pipeline, and can be easily shared/reused by multiple projects.

Full code for example application is available on Github.

Interactive Audience Analytics With Spark and HyperLogLog

October 13, 2015 | by Eugene Zhulenev

This is a guest blog from Eugene Zhulenev on his experiences with Engineering Machine Learning and Audience Modeling at Collective.

At Collective, we are working not only on cool things like Machine Learning and Predictive Modeling but also on reporting that can be tedious and boring. However at our scale even simple reporting application can become a challenging engineering problem. This post is based on a talk that I gave at NY-Scala Meetup. Slides are available here.

Example application is available on github:
https://github.com/collectivemedia/spark-hyperloglog

Impression Log

We are building reporting application that is based on an impression log. It's not exactly the way how we get data from out partners, it's pre-aggregated by Ad, Site, Cookie. And even in this pre-aggregated format it takes hundreds of gigabytes per day on HDFS.

```
Ad           | Site          | Cookie          | Impressions | Clicks | Segments
-------------|---------------|-----------------|-------------|--------|--------------------------------
bmw_X5       | forbes.com    | 13e835610ff0d95 | 10          | 1      | [a.m, b.rk, c.rh, d.sn, ...]
mercedes_2015 | forbes.com    | 13e8360c8e1233d | 5           | 0      | [a.f, b.rk, c.hs, d.mr, ...]
nokia        | gizmodo.com   | 13e3c97d526839c | 8           | 0      | [a.m, b.tk, c.hs, d.sn, ...]
apple_music  | reddit.com    | 1357a253f00c0ac | 3           | 1      | [a.m, b.rk, d.sn, e.gh, ...]
nokia        | cnn.com       | 13b23555294aced | 2           | 1      | [a.f, b.tk, c.rh, d.sn, ...]
apple_music  | facebook.com  | 13e8333d16d723d | 9           | 1      | [a.m, d.sn, g.gh, s.hr, ...]
```

Each cookie id has assigned segments which are just 4-6 letters code, that represents some information about the cookie, that we get from 3rd party data providers such as Blukai.

```
- a.m  : Male
- a.f  : Female
- b.tk : $75k-$100k annual income
- b.rk : $100k-$150k annual income
- c.hs : High School
- c.rh : College
- d.sn : Single
- d.mr : Married
```

For example if a cookie has been assigned `a.m` segment, it means that we think (actually the data provider thinks) that this cookie belongs to a male. The same thing for annual income level and other demographics information.

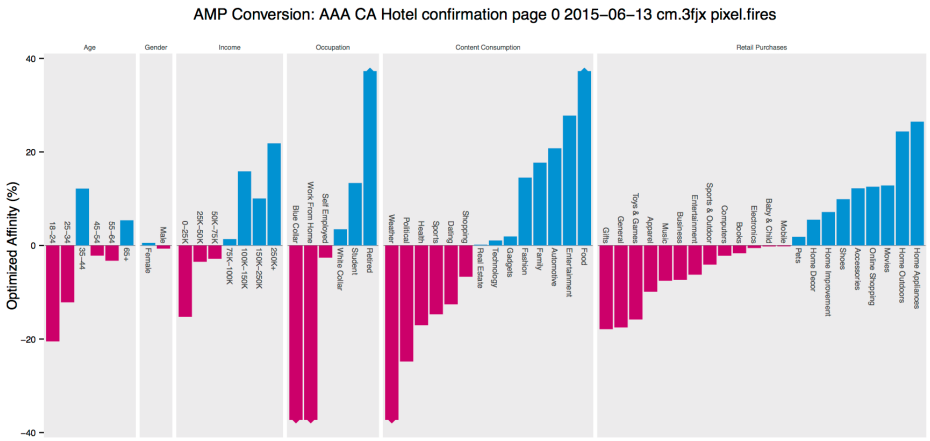
We don't have precise information, to whom exactly a particular cookie belongs nor what is their real annual income level. These segments are essentially probabilistic, nevertheless we can get very interesting insights from this data.

What we can do with this data

Using this impression log we can answer some interesting questions

- We can calculate a given group's prevalence in a campaign's audience, eg. what role do **males** play in the optimized audience for a **Goodyear Tires** campaign?
- What is **male/female** ratio for people who have seen **bmw_X5** ad on **forbes.com**
- Income distribution for people who have seen an Apple Music ad
- Nokia clicks distribution across different education levels

Using these basic questions we can create an "Audience Profile" that describes what type of audience is prevailing in an optimized campaign or partner website.



Blue bars mean that this particular segment tends to view ad/visit website more than on average, and red bar mean less. For example for **Goodyear Tires** we expect to see more **male** audience than **female**.

Solving problems with SQL

SQL looks like an easy choice for this problem, however as I already mentioned we have hundreds of gigabytes of data every day, and we need to get numbers based on 1-year history in seconds. Hive/Impala simply cannot solve this problem.

```
select count(distinct cookie_id) from impressions
    where site = 'forbes.com'
    and ad = 'bmw_X5'
    and segment contains 'a.m'
```

Unfortunately, we have almost infinite combinations of filters that users can define, so it's not feasible to pre-generate all possible reports. Users can use any arbitrary ad, site, campaign, order filter combinations, and may want to know audience intersection with any segment.

Audience cardinality approximation with HyperLogLog

We came up with a different solution; instead of providing precise results for every query, we are providing approximated numbers with very high precision. Usually, the error rate is around 2% which for this particular application is really good. We don't need to know an exact number of male/female cookies in the audience. To be able to say what audience is prevailing, approximated numbers are more than enough.

We use HyperLogLog, which is an algorithm for the count-distinct problem, approximating the number of distinct elements (cardinality). It uses finite space and has configurable precision. It able to estimate cardinalities of >10^9 with a typical accuracy of 2%, using 1.5kB of memory.

```
trait HyperLogLog {
    def add(cookieId: String): Unit
    //   |A|
    def cardinality(): Long
    //   |A ∪ B|
    def merge(other: HyperLogLog): HyperLogLog
    //   |A ∩ B| = |A| + |B| - |A ∪ B|,
    def intersect(other: HyperLogLog): Long
}
```

Here is a rough API that is provided by **HyperLogLog**. You can add a new cookieId to it, get cardinality estimation of unique cookies that were already added to it, merge it with another **HyperLogLog**, and finally get an intersection. It's important to notice that after the **intersect** operation, the **HyperLogLog** object is lost, and you only have approximated intersection cardinality. Therefore, usually **HyperLogLog** intersection is the last step in the computation.

I suggest you to watch the awesome talk by Avi Bryant where he discusses not only HyperLogLog but lot's of other approximation data structures that can be useful for big-data analytics: http://www.infoq.com/presentations/abstract-algebra-analytics.

From cookies to HyperLogLog

We split out original impression log into two tables.

For the ad impressions table, we remove segment information and aggregate cookies, impressions and clicks by Ad and Site. **HyperLogLog** can be used in an aggregation function similar to how use the **sum** operation. Zero is an empty **HyperLogLog** while the plus operation is **merge** (btw it's exactly properties required by **Monoid**)

Ad	Site	Cookies HLL	Impressions	Clicks
bmw_X5	forbes.com	HyperLogLog@23sdg4	5468	35
bmw_X5	cnn.com	HyperLogLog@84jdg4	8943	29

For the segments table, we remove ad and site information, and aggregate data by segment.

```
Segment       | Cookies HLL        | Impressions | Clicks
------------- | ------------------ | ----------- | ------
Male          | HyperLogLog@85sdg4 | 235468      | 335
$100k-$150k   | HyperLogLog@35jdg4 | 569473      | 194
```

Percent of college and high school education in the BMW campaign

If you can imagine that we can load these tables into **Seq**, then audience intersection becomes a really straightforward task, that can be solved by a couple lines of functional scala operations.

```
case class Audience(ad: String, site: String, hll: HyperLogLog,
imp: Long, clk: Long)

case class Segment(name: String, hll: HyperLogLog, imp: Long, clk:
Long)

val adImpressions: Seq[Audience] = ...
val segmentImpressions: Seq[Segment] = ...

val bmwCookies: HyperLogLog = adImpressions
    .filter(_.ad = "bmw_X5")
    .map(_.hll).reduce(_ merge _)

val educatedCookies: HyperLogLog = segmentImpressions
    .filter(_.segment in Seq("College", "High School"))
    .map(_.hll).reduce( _ merge _)

val p = (bmwCookies intersect educatedCookies) /
bmwCookies.count()
```

Spark DataFrames with HyperLogLog

Obviously we can't load all the data into a scala **Seq** on single machine, because it's huge. Even after removing cookie level data and transforming it into **HyperLogLog** objects, it's around 1-2 gigabytes of data for a single day.

So we have to use some distributed data processing framework to solve this problem, and we chose Spark.

What are Spark DataFrames

- Inspired by R data.frame and Python/Pandas DataFrame
- Distributed collection of rows organized into named columns
- Used to be SchemaRDD in Spark < 1.3.0

High-Level DataFrame Operations

- Selecting required columns
- Filtering
- Joining different data sets
- Aggregation (count, sum, average, etc)

You can start by referring to the Spark DataFrame guide or DataBricks blog post.

Ad impressions and segments in DataFrames

We store all of our data on HDFS using Parquet data format, and that's how it looks after it's loaded into Spark DataFrames.

```
val adImpressions: DataFrame = sqlContext.parquetFile("/aa/
audience")

adImpressions.printSchema()
// root
//   | -- ad: string (nullable = true)
//   | -- site: string (nullable = true)
//   | -- hll: binary (nullable = true)
//   | -- impressions: long (nullable = true)
//   | -- clicks: long (nullable = true)

val segmentImpressions: DataFrame = sqlContext.parquetFile("/aa/
segments")

segmentImpressions.printSchema()
// root
//   | -- segment: string (nullable = true)
//   | -- hll: binary (nullable = true)
//   | -- impressions: long (nullable = true)
//   | -- clicks: long (nullable = true)
```

`HyperLogLog` is essentially a huge `Array[Byte]` with some clever hashing and math, so it's straightforward to store it on HDFS in serialized form.

Working with a Spark DataFrame

We wanted to know the answer for the question: "Percent of college and high school education in the BMW campaign".

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.HLLFunctions._

val bmwCookies: HyperLogLog = adImpressions
    .filter(col("ad") === "bmw_X5")
    .select(mergeHll(col("hll")).first() // -- sum(clicks)

val educatedCookies: HyperLogLog = hllSegments
    .filter(col("segment") in Seq("College", "High School"))
    .select(mergeHll(col("hll")).first()

val p = (bmwCookies intersect educatedCookies) /
bmwCookies.count()
```

It looks pretty familiar, not too far from example based on scala `Seq`. Only one unusual operation that you might notice if you have some experience with Spark is `mergeHLL`. It's not available in Spark by default, it is a custom `PartialAggregate` function that can compute aggregates for serialized `HyperLogLog` objects.

Writing your own Spark aggregation function

To write you own aggregation function you need to define a function that will be applied to each row in `RDD` partition, in this example it's called `MergeHLLPartition`. Then you need to define the function that will take results from different partitions and merge them together, for `HyperLogLog` it's called `MergeHLLMerge`. And finally you need to tell Spark how you want it to split your computation across `RDD` (DataFrame is backed by `RDD[Row]`)

```
case class MergeHLLPartition(child: Expression)
  extends AggregateExpression with trees.UnaryNode[Expression]
{ ... }

case class MergeHLLMerge(child: Expression)
  extends AggregateExpression with trees.UnaryNode[Expression]
{ ... }

case class MergeHLL(child: Expression)
  extends PartialAggregate with trees.UnaryNode[Expression] {

  override def asPartial: SplitEvaluation = {
    val partial = Alias(MergeHLLPartition(child),
"PartialMergeHLL")()

    SplitEvaluation(
      MergeHLLMerge(partial.toAttribute),
      partial :: Nil
    )
  }
}

def mergeHLL(e: Column): Column = MergeHLL(e.expr)
```

After that, writing aggregations becomes a really easy task, and your expressions will look like “native” DataFrame code, which is really nice, and super easy to read and reason about.

Also it works much faster than solving this problem with scala transformations on top of `RDD[Row]`, as Spark catalyst optimizer can execute an optimized plan and reduce the amount of data that needs to be shuffled between spark nodes.

And finally, it’s so much easier to manage mutable state. Spark encourage you to use immutable transformations, and it’s really cool until you need extreme performance from your code. For example, if you are using something like `reduce` or `aggregateByKey` you don’t really know when and where your function instantiated, when it’s done with `RDD` partition, nor when the results are transferred to another Spark node for a merge operation. With `AggregateExpression` you have explicit control over mutable state, and it’s totally safe to accumulate mutable state during execution for a single partition. At the end when you’ll need to send data to another node where you can create immutable copy.

In this particular case, using a mutable `HyperLogLog` merge implementation helped to speed up computation times by almost 10x. For each partition `HyperLogLog` state accumulated in single mutable `Array[Byte]` and at the end when data needs to be transferred somewhere else for merging with another partition, an immutable copy is created.

Some fancy aggregates with DataFrame API

You can write much more complicated aggregation functions, for example, to compute aggregate based on multiple columns. Here is a code sample from our audience analytics project.

```
case class SegmentEstimate(cookieHLL: HyperLogLog, clickHLL:
HyperLogLog)

type SegmentName = String

val dailyEstimates: RDD[(SegmentName, Map[LocalDate,
SegmentEstimate])] =
    segments.groupBy(segment_name).agg(
      segment_name,
      mergeDailySegmentEstimates(
        mkDailySegmentEstimate(      // -- Map[LocalDate,
SegmentEstimate]
          dt,
          mkSegmentEstimate(         // --
SegmentEstimate(cookieHLL, clickHLL)
            cookie_hll,
            click_hll)
        )
      )
    )
```

This code calculates daily audience aggregated by segment. Using Spark `PartialAggregate` function saves a lot of network traffic and minimizes the distributed shuffle size.

This aggregation is possible because of nice properties of `Monoid`

- `HyperLogLog` is a `Monoid` (has `zero` and `plus` operations)
- `SegmentEstimate` is a `Monoid` (tuple of two monoids)
- `Map[K, SegmentEstimate]` is a `Monoid` (map with value monoid value type is monoid itself)

Problems with custom aggregation functions

- Right now, it is a closed API so you need to place all of your code under the `org.apache.spark.sql` package.
- It is not guaranteed that it will work in next Spark release.
- If you want to try, I suggest you to start with `org.apache.spark.sql.catalyst.expressions.sum` as example.

Spark as an in-memory SQL database

We use Spark as an in-memory database that serves SQL (composed with DataFrame API) queries.

People tend to think about Spark with a very batch oriented mindset. Start a Spark cluster in YARN, do the computation, kill the cluster. Submit you application to standalone Spark cluster (Mesos), kill it. The biggest problem with this approach is that after your application is done, the JVM is killed, `SparkContext` is lost, and even if you are running Spark in standalone mode, all data cached by your application is lost.

We use Spark in a totally different way. We start Spark cluster in YARN, load data to it from HDFS, cache it in memory, and **do not shut it down**. We keep JVM running, it holds a reference to `SparkContext` and keeps all the data in memory on worker nodes.

Our backend application is essentially very simple REST/JSON server built with Spray, that holds the `SparkContext` reference, receive requests via URL parameters, runs queries in Spark, and return responses in JSON.

Right now (July 2015) we have data starting from April, and it's around 100g cached in 40 nodes. We need to keep 1-year history, so we don't expect more than 500g. And we are very confident that we can scale horizontally without seriously affecting performance. Right now average request response time is 1-2 seconds which is really good for our use case.

Spark Best practices

Here are configuration options that I found really useful for our specific task. You can find more details about each of them in Spark guide.

```
- spark.scheduler.mode=FAIR
- spark.yarn.executor.memoryOverhead=4000
- spark.sql.autoBroadcastJoinThreshold=300000000 // ~300mb
- spark.serializer=org.apache.spark.serializer.KryoSerializer
- spark.speculation=true
```

Also, I found that it's really important to repartition your dataset if you are going to cache it and use for queries. The optimal number of partitions is around 4-6 for each executor core, with 40 nodes and 6 executor cores we use 1000 partitions for best performance.

If you have too many partitions Spark will spend too much time for coordination, and receiving results from all partitions. If too small, you might have problems with too big block during shuffle that can kill not only performance but all your cluster: SPARK-1476

Other Options

Before starting this project, we were evaluating some other options

Hive

Obviously it's too slow for interactive UI backend, but we found it really useful for batch data processing. We use it to process raw logs and build aggregated tables with `HyperLogLog` inside.

Impala

To get good performance out of Impala, you are required to write C++ user defined functions, and it's was not the task that I wanted to do. Also, I'm not confident that even with custom C++ function Impala can show performance that we need.

Druid

Druid is a really interesting project, and it's used in another project at Collective for a slightly different problem, but it's not in production yet.

- Managing separate Druid cluster – it's not the task that I want to do
- We have batch-oriented process – and druid data ingestion is stream based

- Bad support for some of type of queries that we need – if I need to know intersection of some particular ad with all segments, in case of druid it will be 10k (number of segments) queries, and it will obviously fail to complete in 1-2 seconds
- It was not clear how to get data back from Druid – it's hard to get data back from Druid later, if it will turn out that it doesn't solve out problems well

Conclusion

Spark is Awesome! I didn't have any major issues with it, and it just works! The new DataFrame API is amazing, and we are going to build lots of new cool projects at Collective with Spark MLLib and GraphX, and I'm pretty sure they will all be successful.

Approximate Algorithms in Apache Spark: HyperLogLog and Quantiles

May 19th, 2016 | Tim Hunter, Hossein Falaki and Joseph Bradley


Try the approximate algorithm notebook in Databricks Community Edition

Introduction

Apache Spark is fast, but applications such as preliminary data exploration need to be even faster and are willing to sacrifice some accuracy for a faster result. Since version 1.6, Spark implements *approximate algorithms* for some common tasks: counting the number of distinct elements in a set, finding if an element belongs to a set, computing some basic statistical information for a large set of numbers. Eugene Zhulenev, from Collective, has already blogged in these pages about the use of approximate counting in the advertising business.

The following algorithms have been implemented against DataFrames and Datasets and committed into Apache Spark's branch-2.0, so they will be available in Apache Spark 2.0 for Python, R, and Scala:

- **approxCountDistinct:** returns an estimate of the number of distinct elements
- **approxQuantile:** returns approximate percentiles of numerical data

Researchers have looked at such algorithms for a long time. Spark strives at implementing approximate algorithms that are deterministic (they do not depend on random numbers to work) and that have proven theoretical error bounds: for each algorithm, the user can specify a target error bound, and the result is guaranteed to be within this bound, either exactly (deterministic error bounds) or with very high confidence (probabilistic error bounds). Also, it is important that this algorithm works well for the wealth of use cases seen in the Spark community.

In this blog, we are going to present details on the implementation of **approxCountDistinct** and **approxQuantile** algorithms and showcase its implementation in a Databricks notebook.

Approximate count of distinct elements

In ancient times, imagine Cyrus the Great, emperor of Persia and Babylon, having just completed a census of all his empire, fancied to know how many different first names were used throughout his empire, and he put his vizier to the task. The vizier knew that his lord was impatient and wanted an answer fast, even if just an approximate.

There was an issue, though; some names such as Darius, Atusa or Ardumanish were very popular and appeared often on the census records. Simply counting how many people were living within the empire would give a poor answer, and the emperor would not be fooled.

However, the vizier had some modern and profound knowledge of mathematics. He assembled all the servants of the palace, and said: "Servants, each of you will take a clay tablet from the census record. For each first name that is inscribed on the tablet, you will take the first 3 letters of the name, called l1, l2 and l3, and compute the following number:

```
N = l1 + 31 * l2 + 961 * l3
```

For example, for Darius (D = 3, A = 0, R = 17), you will get N = 16340.

This will give you a number for each name of the tablet. For each number, you will count the number of zeros that end this number. In the case of Hossein (N=17739), this will give you no zero. After each of you does that for each name on his or her tablet, you will convene and you will tell me what is the greatest number of zeros you have observed. Now proceed with great haste and make no calculation mistake, lest you want to endure my wrath!"

At the end of the morning, one servant came back, and said they had found a number with four zeros, and that was the largest they all observed across all the census records. The vizier then announced to his master that he was the master of a population with about 1.3 * 10^4 = 13000 different names. The emperor was highly impressed and he asked the vizier how he had accomplished this feat. To which the vizier uttered one word: "hyper-log-log".

The HyperLogLog algorithm (and its variant HyperLogLog++ implemented in Spark) relies on a clever observation: if the numbers are spread uniformly across a range, then the count of distinct elements can be approximated from the largest number of leading zeros in the binary representation of the numbers. For example, if we observe a number whose digits in binary form are of the form 0…(k times)…01…1, then we can estimate that there are in the order of 2^k elements in the set. This is a very crude estimate but it can be refined to great precision with a sketching algorithm. A thorough explanation of the mechanics behind this algorithm can be found in the original paper.

From the example above with the vizier and his servants, this algorithm does not need to perform shuffling, just map (each servant works on a tablet) and combine (the servants can make pairs and decide which one has the greatest number, until there is only one servant). There is no need move data around, only small statistics about each block of data, which makes it very useful in a large dataset setting such as Spark.

Now, in modern times, how well does this technique work, where datasets are much larger and when servants are replaced with a Spark cluster? We considered a dataset of 25 millions online reviews from an online retail vendor, and we set out to approximate the number of

customers behind these reviews. Since customers write multiple reviews, it is a good fit for approximate distinct counting.

Here is how to get an approximate count of users in PySpark, within 1% of the true value and with high probability:

```
# users: DataFrame[user: string]
users.select(approxCountDistinct("user", rsd =
0.01)).show()
```

This plot (fig. 1) shows how the number of distinct customers varies by the error margin. As expected, the answer becomes more and more precise as the requested error margin decreases.

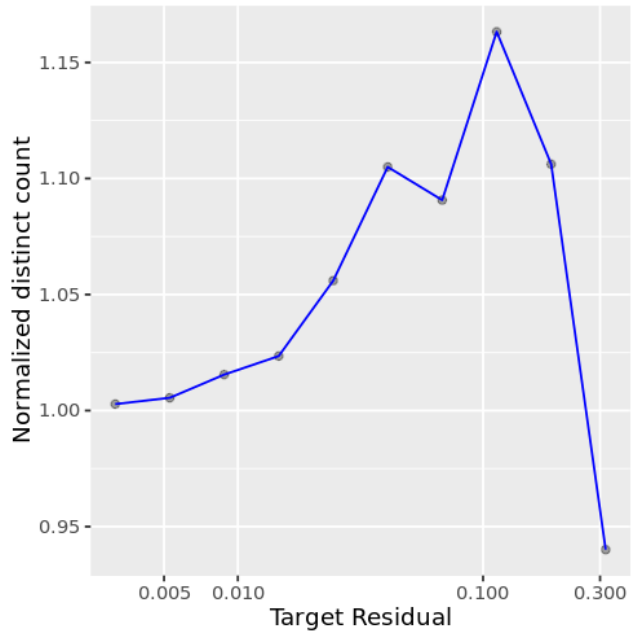


Figure 1

How long does it take to compute? For the analysis on the left, this plot (fig 2.) presents the running time of the approximate counting against the requested precision. For errors above 1%, the running time is just a minute fraction of computing the exact answer. For precise answers, however, the running time increases very fast and it is better to directly compute the exact answer.

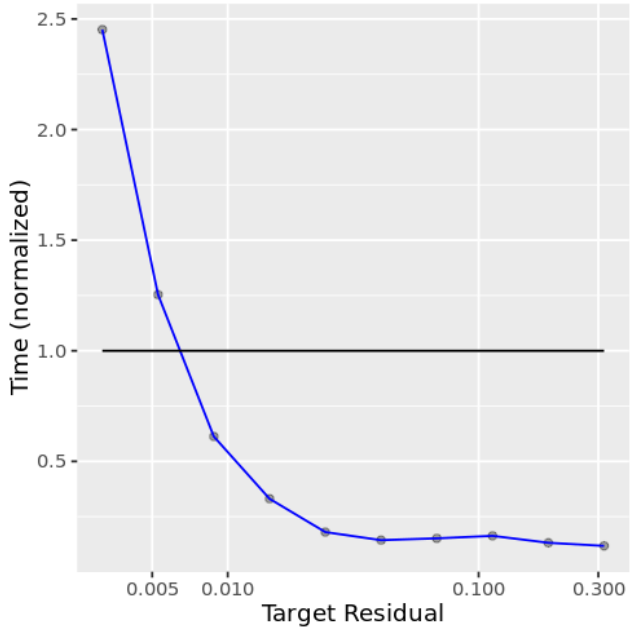


Figure 2

As a conclusion, when using approxCountDistinct, you should keep in mind the following:

- When the requested error on the result is high (> 1%), approximate distinct counting is very fast and returns results for a fraction of the cost of computing the exact result. In fact, the performance is more or less the same for a target error of 20% or 1%.
- For higher precisions, the algorithm hits a wall and starts to take more time than exact counting.

Approximate quantiles

Quantiles (percentiles) are useful in a lot of contexts. For example, when a web service is performing a large number of requests, it is important to have performance insights such as the latency of the requests. More generally, when faced with a large quantity of numbers, one is often interested in some aggregate information such as the mean, the variance, the min, the max, and the percentiles. Also, it is useful to just have the extreme quantiles: the top 1%, 0.1%, 0.01%, and so on.

Spark implements a robust, well-known algorithm that originated in the streaming database community. Like HyperLogLog, it computes some statistics in each node and then aggregates them on the Spark driver. The current algorithm in Spark can be adjusted to trade accuracy against computation time and memory. Based on the same example as before, we look at the length of the text in each review. Most reviewers express their opinions in a few words, but some customers are prolific writers: the longest review in the dataset is more than 1500 words, while there are several thousand 1-word reviews with various degrees of grammatical freedom.

We plot (fig 3.) here the median length of a review (the 50th percentile) as well as more extreme percentiles. This graph shows that there are few very long reviews and that most of them are below 300 characters.

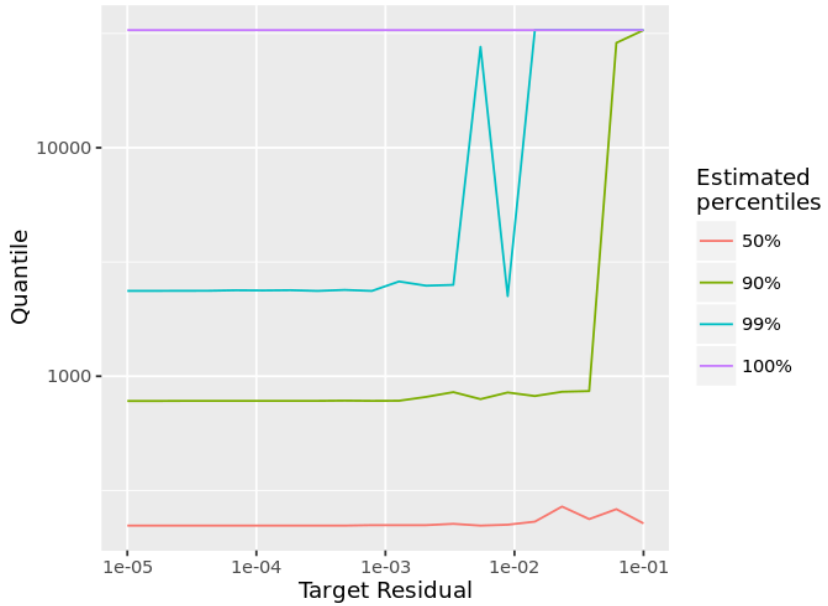


Figure 3

The behavior of approximate quantiles is the same as HyperLogLog: when asking for a rough estimate within a few percent of the exact answer, the algorithm is much faster than an exact computation (fig 4.). For a more precise answer, an exact computation is necessary.

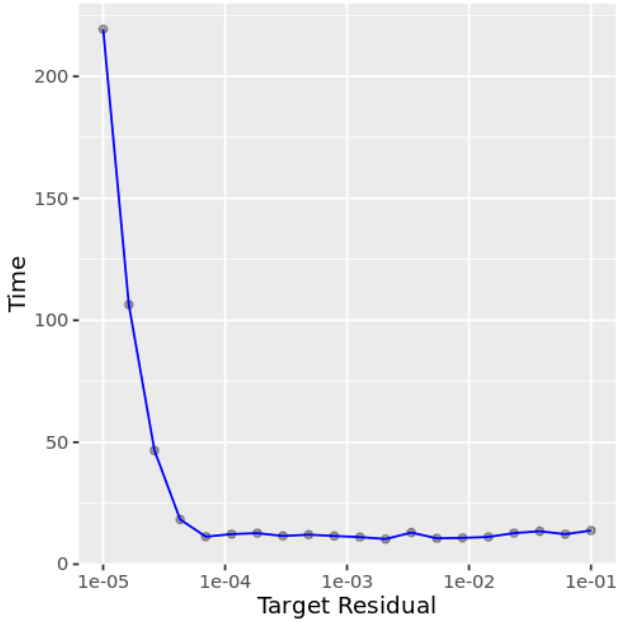


Figure 4

Conclusion

We demonstrated details on the implementation of **approxCountDistinct** and **approxQuantile** algorithms. Though Spark is lighting-fast, sometimes exploratory data applications need even faster results at the expense of sacrificing accuracy. And these two algorithms achieve faster execution.

Apache Spark 2.0 will include some state-of-the art approximation algorithms for even faster results. Users will be able to pick between fast, inexact answers and slower, exact answers. Are there some other approximate algorithms you would like to see? Let us know.

These algorithms are now implemented in a Databricks notebook. To try it out yourself, sign up for an account with Databricks here.

Further Reading

- Interactive Audience Analytics with Spark and HyperLogLog
- HyperLogLog: the analysis of the near-optimal cardinality estimation algorithm
- Approximate Quantiles in Apache Spark notebook

Genome Sequencing in a Nutshell

Part 1 of the Genome Variant Analysis using K-Means, ADAM, and Apache Spark Series

May 24th, 2016 | by Deborah Siegel and Denny Lee

Try the genome variant analysis notebook in Databricks Community Edition

This is a guest post from Deborah Siegel from the Northwest Genome Center and the University of Washington with Denny Lee from Databricks on their collaboration on genome variant analysis with ADAM and Spark.

This is part 1 of the 3 part series Genome Variant Analysis using K-Means, ADAM, and Apache Spark:

1. Genome Sequencing in a Nutshell
2. Parallelizing Genome Variant Analysis
3. Predicting Geographic Population using Genome Variants and K-Means

Introduction

Over the last few years, we have seen a rapid reduction in costs and time of genome sequencing. The potential of understanding the variations in genome sequences range from assisting us in identifying people who are predisposed to common diseases, solving rare diseases, and enabling clinicians to personalize prescription and dosage to the individual.

In this three-part blog, we will provide a primer of genome sequencing and its potential. We will focus on genome variant analysis – that is the differences between genome sequences – and how it can be accelerated by making use of Apache Spark and ADAM (a scalable API and CLI for genome processing) using Databricks Community Edition. Finally, we will execute a k-means clustering algorithm on genomic variant data and build a model that will predict the individual's geographic population of origin based on those variants.

This first post will provide a primer on genome sequencing. You can also skip ahead to the second post Parallelizing Genome Variant Analysis focusing on parallel bioinformatic analysis or the third post on Predicting Geographic Population using Genome Variants and K-Means.

Genome Sequencing

A very simple language analogy

Imagine one long string composed of 3 billion characters and containing roughly 25,000 words interspersed with other characters. Some of the words even make sentences. Changing, adding, or deleting characters or groups of characters could change the structure or meaning of the words and sentences.

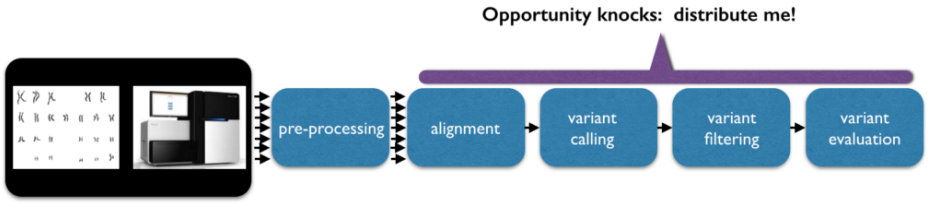
STARTLING-STING
STARTING-STRING

Each long string has very roughly 10-30 million places where such differences may occur. And this makes things interesting. Of course, everything is more complicated. But this has shown itself to be a useful abstraction of genome data.

In the genome, we have been building knowledge about where the words (genes) are located in the string of characters (bases), and we have been discovering the places where they differ (the variants). But we don't know everything. We are still learning about what the effect of the variants are, how the genes are related to each other, and how they may be expressed in different forms and in different quantities under certain circumstances.

Genome Sequencing in a Nutshell

Genome sequencing involves using chemistry and a recording technique to read the characters which code the genome (A,G, C, T) in order (in sequence).



The data is initially read in the form of short strings. For a 30x coverage of a person's genome (30x is a common goal), there may be approximately 600 million short strings of 150 characters each. During data preprocessing, the strings will be mapped/aligned, typically to a reference sequence. There are many different approaches to alignment. Ultimately, this gives every base a defined position. Variant analysis of aligned sequence data finds code differences by comparing the sequence to the reference or to other aligned sequences and assigns genotypes to a person's variants.

Some of the detected variants will be based on noise, and can be filtered with rigid thresholds on parameters such as coverage, quality, and domain-specific biases. Rather than hard filtering, some analysts threshold the variants by fitting a Gaussian mixture model. Even further downstream, analysts quantify and explore the data, try and identify highly significant variants (a small number given the input size), and try to predict what their functional effect might be.

Why sequence?

Genome sequence (and exome sequence, which is a subset) is interesting data from a data science perspective. We can use our knowledge of sequences to gain hints at how and why the code has evolved over long periods of time. Knowledge from genome sequencing studies is becoming more integrated into medicine. Genome sequencing is now used for non-invasive prenatal diagnostics. Genome sequencing will soon be used in clinical screening and diagnostic tests, with much ongoing work to expand genomic medicine.

On the research and discovery side, large cohort and population-scale genome sequencing studies find variants or patterns of variance which may predispose people to common diseases such as autism, heart disease, and specific cancers. Sequencing studies also indicate variants influencing drug metabolism, enabling clinicians to personalize prescriptions and dosage to each individual. In the case of rare heritable diseases, sequencing certain members of a family often leads to finding the causal variants.

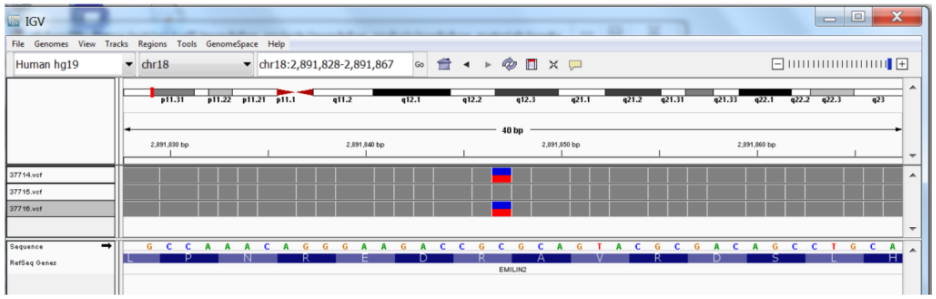
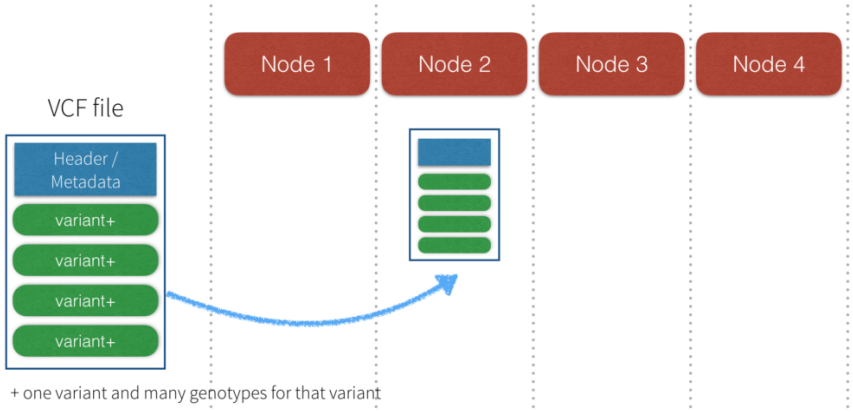
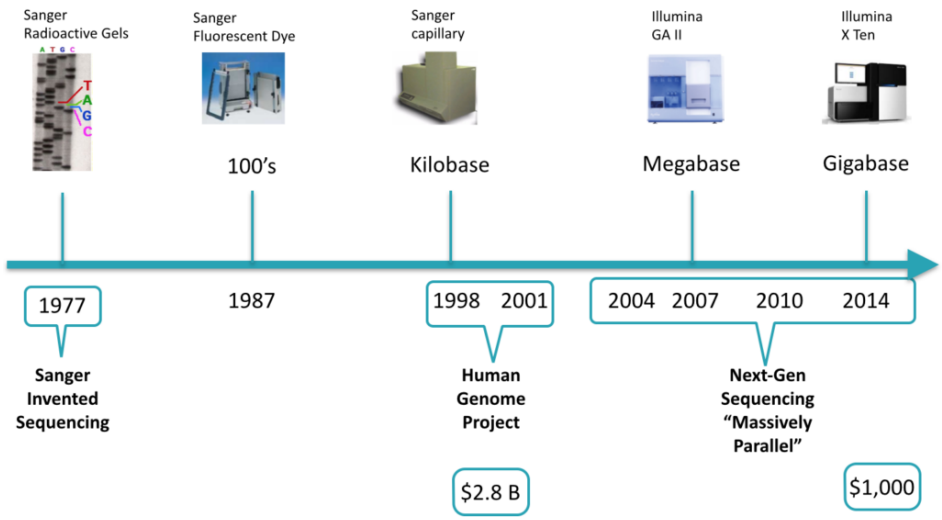


Image Credit: Frederic Reinier, used with permission

In the past five years, sequencing experiments have linked genomic variants to hundreds of rare diseases:

"Individually, a rare disease may affect only a handful of families. Collectively, rare diseases impact 20 to 30 million people in the U.S. alone."





For these reasons, there are resources going towards the reading and analysis of sequences. The National Health Service of the UK has a project to sequence 100,000 genomes of families with members who have rare diseases or cancer by 2017. In the US, The National Human Genome Research Institute (NHGRI) plans to fund common disease research for $240 million and rare disease research for $40 million over the next 4 years. There are also other kinds of sequencing which will benefit from efforts to scale bioinformatics and lower the barrier to applying data science to a large amount of sequence data, such as RNA-seq, microbiome sequencing, and immune system and cancer profile sequencing.

Sequencing technology has been an object of accelerated growth. Between 1998 and 2001, the first human genome was sequenced. It cost $2.8 Billion of 2009 dollars. Today, a genome can be sequenced in 3 days for around $1,000 (for more information, please review National Institutes of Health: National Human Genome Research Institute > DNA Sequencing Costs). During roughly the first 25 years of sequencing experiments, the chemistry allowed only one stretch of DNA to be sequenced at a time, making it laborious, slow, and expensive. The next-generation of sequencing has become massively parallel, enabling sequencing to occur on many stretches of DNA within the same experiment. Also, with molecular indexing, multiple individual's DNA can be sequenced together and the data can be separated out during analysis. It is not implausible to speculate that most people on the planet who opt-in will have their genomes sequenced in the not-so-distant

future. To find out more detail about next-generation sequencing, see Coming of age: ten years of next-generation sequencing technologies

Depending on the application and settings, current sequencing instruments can read ~600 gigabases per day. A medium to large size sequencing center has several such instruments running concurrently. As we will see later on in detail, one of the challenges facing bioinformatics is that downstream software for analyzing variants had been previously optimized for specific, non-extensible file formats, rather than on the data models themselves. The result is that there exist pipeline fragility and obstacles to scalability. Now that we have massively parallel sequencing, many are looking towards parallel bioinformatic analysis.

Public Data

Genome sequence data is generally private. Between 2007 and 2013, The 1000 genomes project was an initial effort for public "population level sequencing". By its final phase, it provided some sequencing coverage data for 2,504 individuals from 26 populations. We used the easily accessible data from this project as a resource to build a notebook in Databricks Community Edition.

Next Steps

In the next blog Parallelizing Genome Variant Analysis we will look into parallel bioinformatic analysis. You can also skip ahead to Predicting Geographic Population using Genome Variants and K-Means.

Attribution

We wanted to give a particular call out to the following resources that helped us create the notebook

- Big Data Genomics ADAM project
- ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing (Berkeley AMPLab)
- Andy Petrella's Lightning Fast Genomics with Spark and ADAM and associated GitHub repo.
- Neil Ferguson Population Stratification Analysis on Genomics Data Using Deep Learning.
- Matthew Conlen Lightning-Viz project.
- Timothy Danford's SlideShare presentations (on Genomics with Spark)
- Centers for Mendelian Genomics uncovering the genomic basis of hundreds of rare conditions
- NIH genome sequencing program targets the genomic bases of common, rare disease
- The 1000 Genomes Project

As well, we'd like to thank for additional contributions and reviews by Anthony Joseph, Xiangrui Meng, Hossein Falaki, and Tim Hunter.

Parallelizing Genome Variant Analysis

Part 2 of the Genome Variant Analysis using K-Means, ADAM, and Apache Spark Series

May 24th, 2016 | by Deborah Siegel and Denny Lee

Try the Genome Variant Analysis Notebook in Databricks Community Edition

This is a guest post from Deborah Siegel from the Northwest Genome Center and the University of Washington with Denny Lee from Databricks on their collaboration on genome variant analysis with ADAM and Spark.

This is part 2 of the 3 part series Genome Variant Analysis using K-Means, ADAM, and Apache Spark:

1. Genome Sequencing in a Nutshell
2. Parallelizing Genome Variant Analysis
3. Predicting Geographic Population using Genome Variants and K-Means

Introduction

Over the last few years, we have seen a rapid reduction in costs and time of genome sequencing. The potential of understanding the variations in genome sequences range from assisting us in identifying people who are predisposed to common diseases, solving rare diseases, and enabling clinicians to personalize prescription and dosage to the individual.

In this three-part blog, we will provide a primer of genome sequencing and its potential. We will focus on genome variant analysis – that is the differences between genome sequences – and how it can be accelerated by making use of Apache Spark and ADAM (a scalable API and CLI for genome processing) using Databricks Community Edition. Finally, we will execute a k-means clustering algorithm on genomic variant data and build a model that will predict the individual's geographic population of origin based on those variants.

This post will focus on Parallelizing Genome Sequence Analysis; for a refresher on genome sequencing, you can review Genome Sequencing in a Nutshell. You can also skip ahead to the third post on Predicting Geographic Population using Genome Variants and K-Means.

Parallelizing Genome Variant Analysis

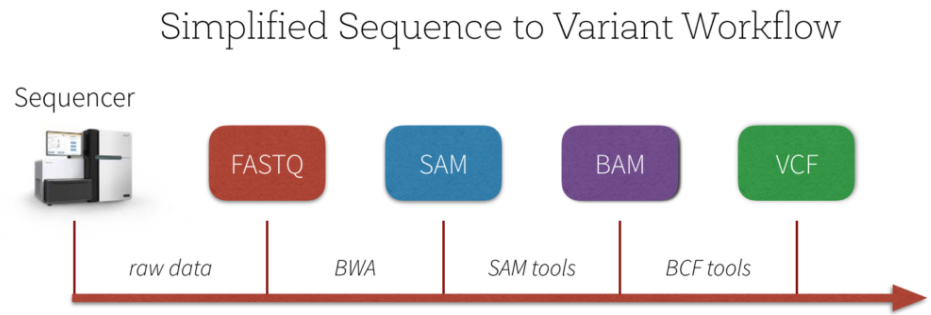
As noted in Genome Sequencing in a Nutshell, there are many steps and stages of the analysis that can be distributed and parallelized in the hopes of significantly improving performance and possibly improving results on very large data Apache Spark is well suited for sequence data because it not only executes many tasks in a distributed parallel fashion, but can do so primarily in-memory with decreased need for intermediate files.

Benchmarks of ADAM + Spark (sorting genome reads and marking duplicate reads for removal) have shown scalable speedups, from 1.5 days on a single node to less than one hour on a commodity cluster (ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing).

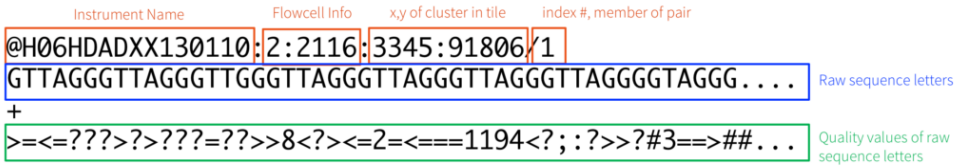
One of the primary issues alluded to when working with current genomic sequence data formats is that they are not easily parallelizable. Fundamentally, the current set of tools and genomic data formats (e.g. SAM, BAM, VCF, etc.) are not well designed for distributed computing environments. To provide context, the next sections provide a simplified background of the workflow from genomic sequence to variant workflow.

Simplified Genome Sequence to Variant Workflow

There are a number of quality control and pre-processing steps that must be initially performed prior to analyzing variants. A simplified workflow can be seen in the image below.



The output of the genome sequencer machine is in FASTQ format – text-based format storing both the short nucleotide sequence reads and their associated quality scores in ASCII format. The image below is an example of the data in FASTQ format.



A typical next step is to use a BWA (Burrows-Wheeler Alignment) sequence alignment tool such as Bowtie to align the large sets of short DNA sequences (reads) to a reference genome and create a SAM file – a sequence alignment map file that stores mapped tags to a genome. The image below (from the Sequence Alignment/Map Format specification) is an example of the SAM format. This specification has a number of terminologies and concepts that are outside the scope of this blog post; for more information, please reference the Sequence Alignment/Map Format specification.

```
@HD VN:1.5 SO:coordinate
@SQ SN:ref LN:45
r001   99 ref  7 30 8M2I4M1D3M = 37  39 TTAGATAAAGGATACTG *
r002    0 ref  9 30 3S6M1P1I4M *  0   0 AAAAGATAAGGATA    *
r003    0 ref  9 30 5S6M       *  0   0 GCCTAAGCTAA       * SA:Z:ref,29,-,6H5M,17,0;
r004    0 ref 16 30 6M14N5M    *  0   0 ATAGCTTCAGC       *
r003 2064 ref 29 17 6H5M       *  0   0 TAGGC             * SA:Z:ref,9,+,5S6M,30,1;
r001  147 ref 37 30 9M         =  7 -39 CAGCGGCAT         * NM:i:1
```

Source: Sequence Alignment/Map Format Specification

The next step is to store the SAM into BAM (a binary version of SAM) typically by using SAMtools (a good reference for this process is Dave Tang's Learning BAM file). Finally, a Variant Call Format (VCF) file is generated by comparing the BAM file to a reference sequence (typically this is done using BCFtools). Note, a great short blog post describing this process is Kaushik Ghose's SAM! BAM! VCF! What?.

Simplified Overview of VCF

With the VCF file, we can finally start performing variant analysis. The VCF itself is a complicated specification so for a more detailed explanation, please reference the 1000 Genomes Project VCF (Variant Call Format) version 4.0 specification.

```
##fileformat=VCFv4.0
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=1000GenomesPilot-NCBI36
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=.,Type=Float,Description="Allele Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">
#CHROM POS     ID        REF ALT    QUAL FILTER INFO                              FORMAT      N
A00001        NA00002        NA00003
20     14370   rs6054257 G      A       29   PASS   NS=3;DP=14;AF=0.5;DB;H2           GT:GQ:DP:
HQ 0|0:48:1:51,51 1|0:48:8:51,51 1/1:43:5:.,.
20     17330   .         T      A       3    q10    NS=3;DP=11;AF=0.017               GT:GQ:DP:
HQ 0|0:49:3:58,50 0|1:3:5:65,3   0/0:41:3
20     1110696 rs6040355 A      G,T     67   PASS   NS=2;DP=10;AF=0.333,0.667;AA=T;DB GT:GQ:DP:
HQ 1|2:21:6:23,27 2|1:2:0:18,2   2/2:35:4
20     1230237 .         T      .       47   PASS   NS=3;DP=13;AA=T                   GT:GQ:DP:
HQ 0|0:54:7:56,60 0|0:48:4:51,51 0/0:61:2
20     1234567 microsat1 GTCT   G,GTACT 50   PASS   NS=3;DP=9;AA=G                    GT:GQ:DP
0/1:35:4       0/2:17:2       1/1:40:3
```

Source: 1000 Genomes Project VCF (Variant Call Format) Cersion 4.0 Specification

While there are various tools which can process and analyze VCFs, they cannot be used in a distributed parallel fashion. A simplified view of a VCF file is that it contains metadata, header, and the data. The metadata is often of interest and should be applied to each genotype. As illustrated in the image below, even if you have four nodes (i.e. Node 1, Node 2, Node 3, Node 4) to process your genotype data, you cannot efficiently distribute the data to all four nodes. With traditional variant analysis tools, the whole file including all of the data, metadata, and header must be sent to a single node. Additionally, the VCF file has more than one observation per row (variants and all their genotypes). This makes it impossible to analyze the genotypes in parallel without reformatting or using special tools.

Another key issue complicating the analysis of VCFs is the level of complexity surrounding the VCF format specification. Referring to the 1000 Genomes Project VCF (Variant Call Format) version 4.0 specification, there are a number of rules surrounding how to interpret the lines within the VCF. Therefore, any data scientist who wants to analyze variant data has to expend a large amount of effort to understand the specific VCFs they are working with and parsing.

Introducing ADAM

Big Data Genomics ADAM project was designed to solve problems around distributing sequence data and parallelizing the processing of sequence data as noted in the technical report ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing. ADAM is comprised of a CLI (tool kit for quickly processing genomics data), numerous APIs (interfaces to transform, analyze, and query genomic data), schemas, and file formats (columnar formats that allow for efficient parallel access to data).

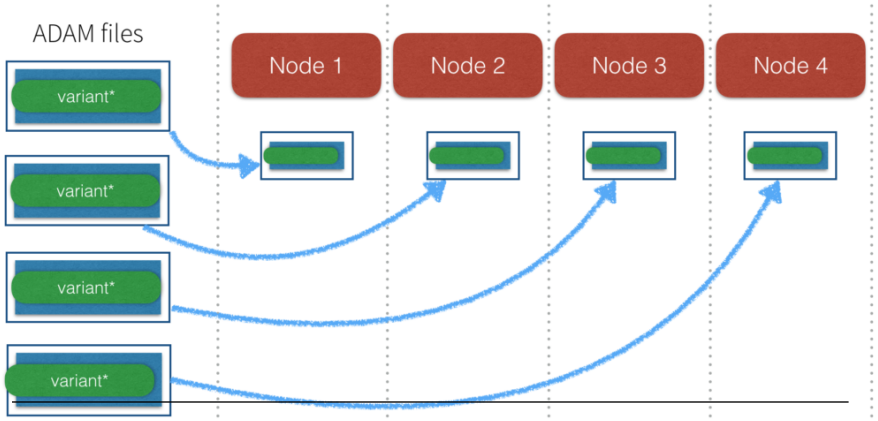
bdg-formats Schemas

To address the complexities of parsing common types of sequence data – such as reads, reference oriented data, variants, genotypes, and assemblies – ADAM utilizes bdg-formats, a set of extensible Apache Avro schemas that are built around data types themselves rather than a file format. In other words, the schemas allows ADAM (or other any tools) to more easily query data instead of building custom code to parse each line of data depending on the file format. These data formats are highly efficient – they are easily serializable, and the information about each particular schema, such as data types, does not have to be sent redundantly with each batch of data. The nodes in your cluster are made aware of what the schema is in an extensible way (data can be added with an extended schema, and analyzed together with data under the old schema).

Parallel distribution via ADAM Parquet

ADAM Parquet files (when compared to binary or text VCF files) enable fast processing because they support parallel distribution of sequence data. In the earlier image of the VCF file, we saw that the entire file must be sent to one node. With ADAM Parquet files, the metadata and header are incorporated into the data elements and schema, and the elements are “tidy” in that there is one observation per element (one genotype of one variant).

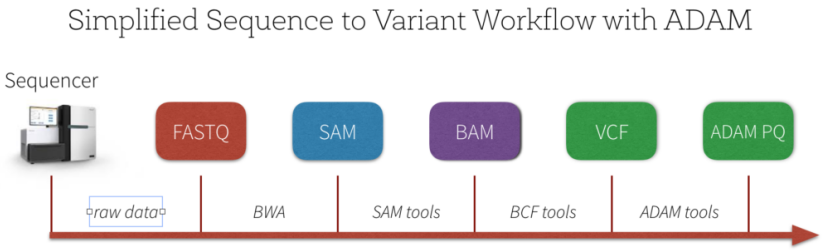
This enables the files to be distributed across multiple nodes. It also makes it simple to filter elements for just the data you want, such as genotypes for a certain panel, without using special tools. ADAM files are stored in the Parquet columnar storage format which is designed for parallel processing. As of GATK4, the Genome Analysis Toolkit is also able to read and write ADAM Parquet formatted data.



*one variant and one genotype for that variant

Updated Simplified Genome Sequence to Variant Workflow

With defined schemas (bdg-format) and ADAM's APIs, data scientists can focus on querying the data instead of parsing the data formats.



Next Steps

In the next blog we will run a parallel bioinformatic analysis example Predicting Geographic Population using Genome Variants and K-Means. You can also review a primer on genome sequencing: Genome Sequencing in a Nutshell.

Attribution

We wanted to give a particular call out to the following resources that helped us create the notebook

- Big Data Genomics ADAM project
- ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing (Berkeley AMPLab)
- Andy Petrella's Lightning Fast Genomics with Spark and ADAM and associated GitHub repo.
- Neil Ferguson Population Stratification Analysis on Genomics Data Using Deep Learning.
- Matthew Conlen Lightning-Viz project.
- Timothy Danford's SlideShare presentations (on Genomics with Spark)
- Centers for Mendelian Genomics uncovering the genomic basis of hundreds of rare conditions

- NIH genome sequencing program targets the genomic bases of common, rare disease
- The 1000 Genomes Project

As well, we'd like to thank for additional contributions and reviews by Anthony Joseph, Xiangrui Meng, Hossein Falaki, and Tim Hunter.

Predicting Geographic Population using Genome Variants and K-Means

Part 3 of the Genome Variant Analysis using K-Means, ADAM, and Apache Spark Series

May 24th, 2016 | by Deborah Siegel and Denny Lee

Try the Genome Variant Analysis Notebook in Databricks Community Edition

This is a guest post from Deborah Siegel from the Northwest Genome Center and the University of Washington with Denny Lee from Databricks on their collaboration on genome variant analysis with ADAM and Spark.

This is part 3 of the 3 part series Genome Variant Analysis using K-Means, ADAM, and Apache Spark:

1. Genome Sequencing in a Nutshell
2. Parallelizing Genome Variant Analysis
3. Predicting Geographic Population using Genome Variants and K-Means

Introduction

Over the last few years, we have seen a rapid reduction in costs and time of genome sequencing. The potential of understanding the variations in genome sequences range from assisting us in identifying people who are predisposed to common diseases, solving rare diseases, and enabling clinicians to personalize prescription and dosage to the individual.

In this three-part blog, we will provide a primer of genome sequencing and its potential. We will focus on genome variant analysis – that is the differences between genome sequences – and how it can be accelerated by making use of Apache Spark and ADAM (a scalable API and CLI for genome processing) using Databricks Community Edition. Finally, we will execute a k-means clustering algorithm on genomic variant data and build a model that will predict the individual's geographic population of origin based on those variants.

This post will focus on predicting geographic population using genome variants and k-means. You can also review the refresher Genome Sequencing in a Nutshell or more details behind Parallelizing Genome Variant Analysis.

Predicting Geographic Population using Genome Variants and K-Means

We will be demonstrating Genome Variant Analysis by performing K-Means on ADAM data using Apache Spark on Databricks Community Edition. The notebook shows how to perform an analysis of public data from the 1000 genomes project using the Big Data Genomics ADAM Project (0.19.0 Release). We attempt k-means clustering to predict which geographic population each person is from and visualize the results.

Preparation

As with most Data Sciences projects, there are a number of preparation tasks that must be completed first. In this example, we will showcase from our example notebook:

- Converting a sample VCF file to ADAM parquet format
- Loading a panel file that describes the data within the sample VCF / ADAM parquet
- Read the ADAM data into RDDs and begin parallel processing of genotypes

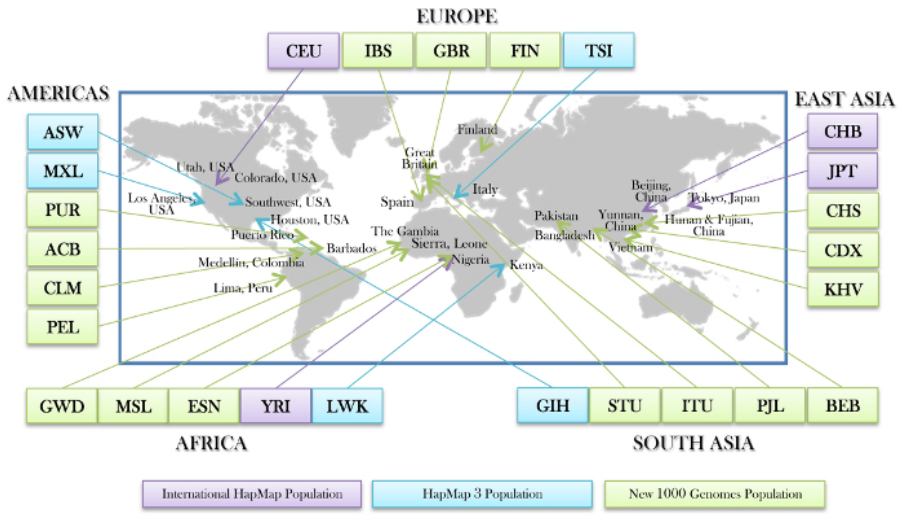
Creating ADAM Parquet Files

To create an ADAM parquet file from VCF, we will first load the VCF file using the ADAM's SparkContext loadGenotypes method. By using the adamParquetSave method, we save the VCF in ADAM parquet format.

```
val gts:RDD[Genotype] = sc.loadGenotypes(vcf_path)
```

Loading the Panel File

While the VCF data contain sample IDs, they do not contain population codes that we will want to predict. Although we are doing an unsupervised algorithm in this analysis, we still need the response variables in order to filter our samples and to estimate our prediction error. We can get the population codes for each sample from the `integrated_call_samples_v3.20130502.ALL.panel` panel file from the 1000 genomes project.



The code snippet below loads panel file using the CSV reader for Spark to create the **`panel`** Spark DataFrame.

Source: 1000-genomes-map_11-6-12-2_750.jpg

```
val panel = sqlContext.read
   .format("com.databricks.spark.csv")
   .option("header", "true")
   .option("inferSchema", "true")
   .option("delimiter", "\\t")
   .load(panel_path)
```

For our k-means clustering algorithms, we will model for 3 clusters, so we will create a filter for 3 populations: British from England and Scotland (GBR), African Ancestry in Southwest US (ASW), and Han Chinese in Bejing, China (CHB). We will do this by create a `filterPanel` DataFrame with only these three populations. As this is a small panel, we are also broadcasting it to all the executors will result in less data shuffling when we do further operations, thus it will be more efficient.

```
// Create filtered panel of the three populations
val filterPanel = panel.select("sample", "pop").where("pop IN
('GBR', 'ASW', 'CHB')")

// Take filtered panel and broadcast it
val fpanel = filterPanel
   .rdd
   .map{x => x(0).toString -> x(1).toString}
   .collectAsMap()
val bPanel = sc.broadcast(fpanel)
```

Parallel processing of genotypes

Using the command below, we will load the genotypes of our three populations. This can be done more efficiently in parallel because the filtered panel is loaded in memory and broadcasted to all the nodes (i.e. bPanel) while the parquet files containing the genotype data enable predicate pushdown to the file level. Thus, only the records we are interested in are loaded from the file.

```
// Create filtered panel of the three populations
val popFilteredGts : RDD[Genotype] =
sc.loadGenotypes(tmp_path).filter(genotype =>
{bPanel.value.contains(genotype.getSampleId)})
```

The notebook contains a number of additional steps including:

- Exploration of the data – our data has a small subset of variants of chromosome 6 covering about half a million base pairs.
- Cleaning and filtering of the data – missing data or if the variant is triallelic.
- Preparation of the data for k-means clustering – create an ML vector for each sample (containing the variants in the exact same order) and then pulling out the features vector to run the model against.

Ultimately, the genotypes for the 805 variants we have left in our data will be the features we use to predict the geographic population. Our next step is to create a features vector and DataFrame to run k-means clustering.

Running KMeans clustering

With the above preparation steps, running k-means clustering against the genome sequence data is similar to the k-means example in the Spark Programming Guide.

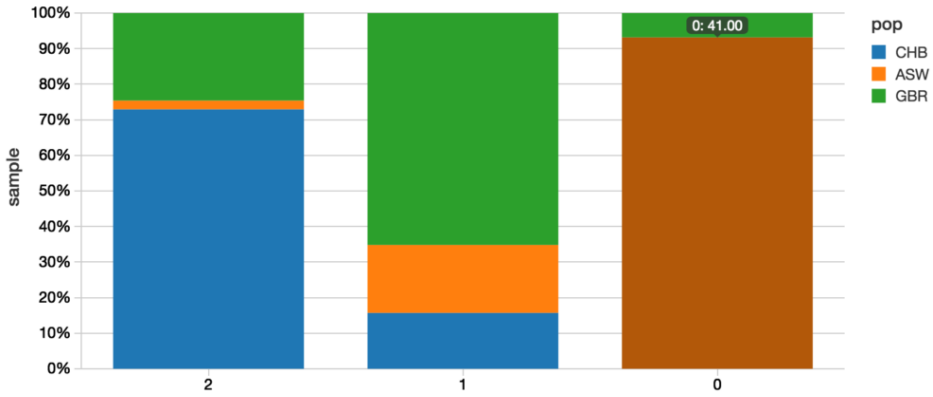
```
import org.apache.spark.mllib.clustering.{KMeans,KMeansModel}

// Cluster the data into three classes using KMeans
val numClusters = 3
val numIterations = 20
val clusters:KMeansModel = KMeans.train(features, numClusters,
numIterations)
```

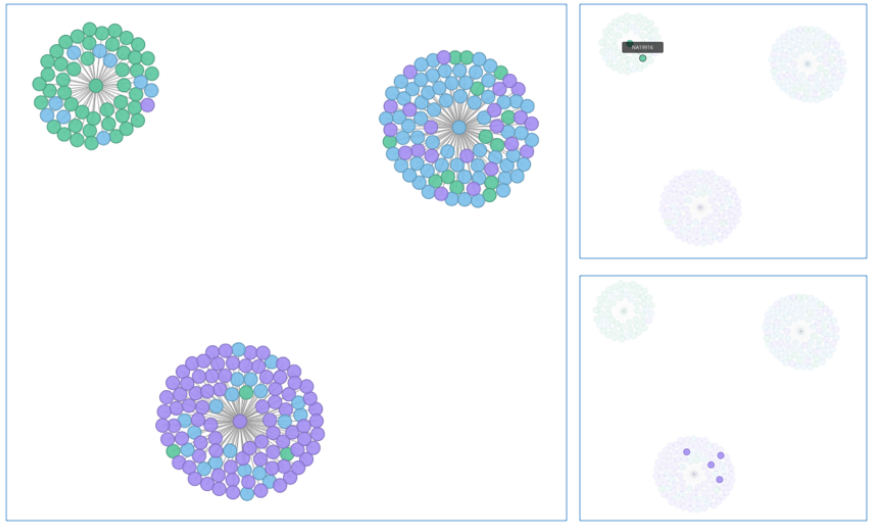
Now that we have our model – clusters – let's predict the populations and compute the confusion matrix. First, we perform the task of creating the **predictionRDD** which contains the original value (i.e. that points to the original geographic population location of CHB, ASW, and GBR) and utilizes **clusters.predict** to output the model's prediction of the geo based on the features (i.e. the genome variants). Next, we convert this into the **predictDF** DataFrame making it easier to query (e.g. using the **display()** command, running R commands in the subsequent cell, etc.). Finally, we join back to the **filterPanel** to obtain the original labels (the actual geographic population location).

```
// Create predictionRDD that utilizes clusters.predict method to
output the model's predicted geo location
val predictionRDD: RDD[(String, Int)] = dataPerSampleId.map(sd =>
{
   (sd._1, clusters.predict(sd._2))
})
```

Below is the graphical representation of the output between the predicted value and actual value.



A quick example of how to calculate the confusion matrix is to use R. While this notebook was primarily written in Scala, we can add a new cell using %r indicating that we are using the R language for our query.



```
%r
resultsRDF <- sql(sqlContext, "SELECT pop, prediction FROM
results_table")
confusion_matrix <- crosstab(resultsRDF, "prediction", "pop")
head(confusion_matrix)
```

With the output being:

```
  prediction_pop CHB GBR ASW
1              2  89  30   3
2              1  14  58  17
3              0   0   3  41
```

Within the notebook, there is also additional SQL code to join the original sample, geographic population, population code, and predicted code so you can map the predictions down to the individual samples.

Visualizing the Clusters with a Force Graph on Lightning-Viz

A fun way to visualize these k-means cluster is to use the force graph via Lightning-Viz. The notebook contains the Python code used to create the lightning visualization. In the animated gif below, you can see the three clusters representing the three populations (top left: 2, top right: 1, bottom: 0). The predicted cluster memberships are the vertices of the cluster while the different colors represent the population. Clicking on the population shows the sampleID, color (actual population), and the predicted population (line to the vertices).

Discussion

In this post, we have provided a primer of genome sequencing (Genome Sequencing in a Nutshell) and the complexities surrounding variant analysis (Parallelizing Genome Variant Analysis). With the introduction of ADAM, we can process variants in a distributed parallelizable manner significantly improving the performance and accuracy of the analysis. This has been demonstrated in the Genome Variant Analysis by performing K-Means on ADAM data using Apache Spark notebook which you can run for yourself in Databricks Community Edition. The promise of genome variant analysis is that we can identify individuals who are predisposed to common diseases, solve rare diseases, and provide personalized treatments. Just as we have seen the massive drop in cost and time with massively parallel sequencing, massively parallel bioinformatic analysis will help us to handle reproducible analysis of the rising deluge of sequence data, and may even contribute to methods of analysis that are currently not available. Ultimately, it will contribute to progress in medicine.

Attribution

We wanted to give a particular call out to the following resources that helped us create the notebook

- Big Data Genomics ADAM project
- ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing (Berkeley AMPLab)

- Andy Petrella's Lightning Fast Genomics with Spark and ADAM and associated GitHub repo.
- Neil Ferguson Population Stratification Analysis on Genomics Data Using Deep Learning.
- Matthew Conlen Lightning-Viz project.
- Timothy Danford's SlideShare presentations (on Genomics with Spark)
- Centers for Mendelian Genomics uncovering the genomic basis of hundreds of rare conditions
- NIH genome sequencing program targets the genomic bases of common, rare disease
- The 1000 Genomes Project

As well, we'd like to thank for additional contributions and reviews by Anthony Joseph, Xiangrui Meng, Hossein Falaki, and Tim Hunter.

Apache Spark 2.0 Preview: Machine Learning Model Persistence

An ability to save and load models across languages

May 31, 2016 | Joseph Bradley

Try the ML Pipeline Persistence Notebook on Databricks Community Edition

Introduction

Consider these Machine Learning (ML) use cases:

- A data scientist produces an ML model and hands it over to an engineering team for deployment in a production environment.
- A data engineer integrates a model training workflow in Python with a model serving workflow in Java.
- A data scientist creates jobs to train many ML models, to be saved and evaluated later.

All of these use cases are easier with model persistence, the ability to save and load models. With the upcoming release of Apache Spark 2.0, Spark's Machine Learning library MLlib will include near-complete support for ML persistence in the DataFrame-based API. This blog post gives an early overview, code examples, and a few details of MLlib's persistence API.

Key features of ML persistence include:

- Support for all language APIs in Spark: Scala, Java, Python & R
- Support for nearly all ML algorithms in the DataFrame-based API
- Support for single models and full Pipelines, both unfitted (a "recipe") and fitted (a result)
- Distributed storage using an exchangeable format

Thanks to all of the community contributors who helped make this big leap forward in MLlib! See the JIRAs for Scala/Java, Python, and R for full lists of contributors.

Learn the API

In Apache Spark 2.0, the DataFrame-based API for MLlib is taking the front seat for ML on Spark. (See this previous blog post for an introduction to this API and the "Pipelines" concept it introduces.) This DataFrame-based API for MLlib provides functionality for saving and loading models that mimics the familiar Spark Data Source API.

We will demonstrate saving and loading models in several languages using the popular MNIST dataset for handwritten digit recognition (LeCun et al., 1998; available from the LibSVM dataset page). This dataset

contains handwritten digits 0–9, plus the ground truth labels. Here are some examples:

Our goal will be to take new images of handwritten digits and identify the digit. See this notebook for the full example code to load this data, fit the models, and save and load them.

Save & load single models

We first show how to save and load single models to share between languages. We will fit a Random Forest Classifier using Python, save it, and then load the same model back using Scala.

```
training = sqlContext.read...  # data: features, label
rf = RandomForestClassifier(numTrees=20)
model = rf.fit(training)
```

We can simply call the `save` method to save this model, and the `load` method to load it right back:

```
model.save("myModelPath")
```

We could also load that same model (which we saved in Python) into a Scala or Java application:

```
// Load the model in Scala
val sameModel =
RandomForestClassificationModel.load("myModelPath")
```

This works for both small, local models such as K-Means models (for clustering) and large, distributed models such as ALS models (for recommendation). The loaded model has the same parameter settings and data, so it will return the same predictions even if loaded on an entirely different Spark deployment.

Save & load full Pipelines

So far, we have only looked at saving and loading a single ML model. In practice, ML workflows consist of many stages, from feature extraction and transformation to model fitting and tuning. MLlib provides Pipelines to help users construct these workflows. (See this notebook for a tutorial on ML Pipelines analyzing a bike sharing dataset.)

MLlib allows users to save and load entire Pipelines. Let's look at how this is done on an example Pipeline with these steps:

- Feature extraction: Binarizer to convert images to black and white
- Model fitting: Random Forest Classifier to take images and predict digits 0–9
- Tuning: Cross-Validation to tune the depth of the trees in the forest

Here is a snippet from our notebook to build this Pipeline:

```
// Construct the Pipeline: Binarizer + Random Forest
val pipeline = new Pipeline().setStages(Array(binarizer, rf))

// Wrap the Pipeline in CrossValidator to do model tuning.
val cv = new CrossValidator().setEstimator(pipeline) ...
```

Before we fit this Pipeline, we will show that we can save entire workflows (before fitting). This workflow could be loaded later to run on another dataset, on another Spark cluster, etc.

```
cv.save("myCVPath")
val sameCV = CrossValidator.load("myCVPath")
```

Finally, we can fit the Pipeline, save it, and load it back later. This saves the feature extraction step, the Random Forest model tuned by Cross-Validation, and the statistics from model tuning.

```
val cvModel = cv.fit(training)
cvModel.save("myCVModelPath")
val sameCVModel = CrossValidatorModel.load("myCVModelPath")
```

Learn the details

Python tuning

The one missing item in Spark 2.0 is Python tuning. Python does not yet support saving and loading CrossValidator and TrainValidationSplit, which are used to tune model hyperparameters; this issue is targeted for Spark 2.1 (SPARK-13786). However, it is still possible to save the results from CrossValidator and TrainValidationSplit from Python. For example, let's use Cross-Validation to tune a Random Forest and then save the best model found during tuning.

```
# Define the workflow
rf = RandomForestClassifier()
cv = CrossValidator(estimator=rf, …)

# Fit the model, running Cross-Validation
cvModel = cv.fit(trainingData)

# Extract the results, i.e., the best Random Forest model
bestModel = cvModel.bestModel

# Save the RandomForest model
bestModel.save("rfModelPath")
```

See the notebook for the full code.

Exchangeable storage format

Internally, we save the model metadata and parameters as JSON and the data as Parquet. These storage formats are exchangeable and can be read using other libraries. Parquet allows us to store both small models (such as Naive Bayes for classification) and large, distributed models (such as ALS for recommendation). The storage path can be any URI supported by Dataset/DataFrame save and load, including paths to S3, local storage, etc.

Language cross-compatibility

Models can be easily saved and loaded across Scala, Java, and Python. R has two limitations. First, not all MLlib models are supported from R, so not all models trained in other languages can be loaded into R. Second, the current R model format stores extra data specific to R, making it a bit hacky to use other languages to load models trained and saved in R. (See the accompanying notebook for the hack.) Better cross-language support for R will be added in the near future.

Conclusion

With the upcoming 2.0 release, the DataFrame-based MLlib API will provide near-complete coverage for persisting models and Pipelines. Persistence is critical for sharing models between teams, creating multi-language ML workflows, and moving models to production. This feature was a final piece in preparing the DataFrame-based MLlib API to become the primary API for Machine Learning in Apache Spark.

What's next?

High-priority items include complete persistence coverage, including Python model tuning algorithms, as well as improved compatibility between R and the other language APIs.

Get started with this tutorial notebook in Scala and Python. You can also just update your current MLlib workflows to use save and load.

Experiment with this API using an Apache Spark branch-2.0 preview in the Databricks Community Edition.

Read More

- Read the notebook with the full code referenced in this blog post.
- Learn about the DataFrame-based API for MLlib & ML Pipelines:
  - Notebook introducing ML Pipelines: tutorial analyzing a bike sharing dataset
  - Original blog post on ML Pipelines

Section 3:
Select Case Studies

Ineractive Optimizes the Mobile Ad Buying Experience at Scale with Machine Learning on Databricks

February 4, 2016 | by Wayne Chan

We are happy to announce that Ineractive chose Databricks as their primary data warehousing and analytics platform — allowing them to ingest and explore data at scale without hampering performance.

You can read the press release here.

Ineractive is a global mobile ad exchange focused on empowering mobile publishers to realize their properties' full potential by providing powerful technologies for the buying and selling of mobile ads programmatically, at scale.

At the heart of the Ineractive platform is a constant flow of advertising traffic increasing upwards to 3-5 million requests per minute and over 240GB of raw data per day. Ineractive quickly realized that their current system was too costly to scale without hampering performance.

Ineractive was also limited in their ability to build, test, and tune machine learning algorithms and models to improve the bidding outcome while satisfying stringent requirements to deliver value to the users of their platform.

Ineractive turned to Databricks to simplify their data ingest and preparation process as their scalability issues became a critical bottleneck for business growth. With Databricks, they are not only able to query the data at scale without issue, but can now leverage the advanced analytics capabilities provided by the platform to build high performing machine learning algorithms and models in a distributed fashion to meet their evolving needs. Furthermore, the simple self-service cluster manager allows Ineractive to provision managed Spark clusters on-demand, simplifying big data infrastructure operations and eliminating disruptive DevOps problems.

As Ineractive's business grows with incoming ad traffic more than tripling annually, they consider Databricks the perfect platform to meet their growing needs while managing costs more efficiently.

Download this case study to learn more about how Ineractive is using Databricks.

Yesware Deploys Production Data Pipeline in Record Time with Databricks

July 23, 2015 | by Kavitha Mariappan and Dave Wang

We are happy to announce that Yesware chose Databricks to build its production data pipeline, completing the project in record time — in just under three weeks.

You can read the press release here.

Yesware, the leading sales acceleration software for sales teams at major enterprise companies such as eBay, New Relic, and IBM, enables sales professionals to have highly effective and successful engagements by providing analytics on their daily interactions with potential customers using billions of data points, including open and reply rate of emails, effectiveness of email templates, engagement rates of e-mails, CTA click-through's, and more.

Yesware encountered many difficulties when it first attempted to build and operate a production data pipeline. It needed to ingest a large volume of data (hundreds of GB per day, and growing rapidly), build highly customizable reports, and must do all of the above with minimum latency. The initial solution was too slow, difficult to maintain, and just not scalable enough. Yesware decided it was time they sourced a better solution.

Apache Spark became the big data technology of choice because of its flexible and easy-to-learn API, fast performance, and native support for crucial capabilities such as SQL and machine learning algorithms. Yesware chose Databricks to implement and operationalize a Spark data pipeline. Why? Because the Databricks platform makes building and running production Spark applications much faster and easier with its cluster manager, interactive workspace, and job scheduler. Databricks also easily integrated with Yesware's existing infrastructure in Amazon Web Services (AWS), further accelerating their adoption and on-ramp processes.

The benefits Yesware gained with Databricks included:

- Deploying a production data pipeline in just under three weeks, compared to over six months with the prior solution.
- Processing over 180 days of historical data in two hours. This was a drastic speedup compared to their prior solution that took 12 hours to process 90 days of data.
- Improving the efficiency of their infrastructure by reducing the cost to just 10% of the prior solution.

- Accelerating the development of new features, enabling their developers and data scientists to collaborate seamlessly, reducing time to prototype new algorithm to hours instead of days.

Download this case study to learn more about how Yesware is using Databricks.

Elsevier Labs Deploys Databricks for Unified Content Analysis

November 12, 2015 | by Kavitha Mariappan and Dave Wang

We are happy to announce that Elsevier Labs has deployed Databricks as its unified content analysis platform, providing significant productivity gains for the entire team and reducing typical project lengths from weeks to just days.

Elsevier Labs is the advanced R&D group within Elsevier – a global provider of scientific information, publishing over 2,500 journals and 33,000 book titles while building web-based information solutions for professionals in science, technology, and medicine.

They needed a fast and scalable analytics platform to develop new methods to extract insights from the published content. Their development process frequently required the application of complex natural language processing (NLP) algorithms to millions of articles and interpretation of the results. Prior to Databricks, Elsevier Labs' productivity was severely hampered because:

- There was substantial manual data movement during the analytics workflow
- The steep learning curve of their legacy analytics platform prevented code reuse
- Presenting findings required significant additional time to build reports and UIs

Databricks enabled Elsevier Labs to effortlessly manage Spark clusters, access their data, collaboratively develop cutting-edge algorithms, and present their findings in a single platform. With the Databricks integrated workspace, the Elsevier Labs team was able to:

- Create, scale, and terminate Spark clusters without specialists with big data DevOps expertise.
- Directly access data in S3 buckets and collaboratively perform analysis in a notebook environment, using Python, Scala, SQL, or even R.
- Present findings to senior management and share results across the entire organization with account-based access control of notebooks.

As a result of deploying Databricks, Elsevier Labs enabled five times more people to contribute to content analysis algorithm development, growing the number of contributors from 3 to 15. Moreover, the people who use Databricks are significantly more productive, reducing typical project lengths from weeks to just days.

Download this case study to learn more about how Elsevier is using Databricks.

Findify's Smart Search Gets Smarter with Spark MLlib and Databricks

February 12, 2016 | by Denny Lee

We are happy to announce that Findify has deployed Databricks as its machine learning and analytics platform, achieving faster time to complete projects, more efficient operations, and improved collaboration.

You can read the press release here.

Findify's mission is to create frictionless online shopping experiences which lead to satisfied, loyal customers and an increase in revenue. By utilizing advanced machine learning techniques, Findify's Smart Search continuously improves its accuracy as users use the service.

Findify needed a data platform to run the machine learning algorithms central to their Smart Search capabilities. Their development process was hampered by difficult-to-maintain custom code and complicated workflows to ensure that their data infrastructure remained operational, such as:

- Devoting substantial DevOps time and resources to cluster and job maintenance.
- Building an additional logic layer for the execution and failure logic of their jobs.
- Difficulty iterating through machine learning models because they could not effectively share information across a globally distributed team.

Databricks enabled Findify to effortlessly manage Spark clusters, access their data, collaboratively develop machine learning algorithms, and present their findings in a single platform. With the Databricks integrated workspace, the Findify team was able to:

- Complete their feature development projects faster and reduce customer frustration in delayed analytics because they could focus on development instead of infrastructure.
- Focus on building innovative features because the managed Spark platform eliminated time spent on DevOps and infrastructure issues.
- Collaborate amongst their globally distributed team more effectively, which enabled them to iterate and visualize their results faster than before.

As a result of deploying Databricks, Findify is able to focus less on DevOps and more on employing machine learning to improve their innovative Smart Search engine.

“Databricks reduced our DevOps time on the Spark platform to almost zero. We don’t worry about failures, restarts, capacity, etc. It just works.”

– Meni Morim, Co-founder and CEO at Findify

Download this case study to learn more about how Findify is using Databricks.

How Sellpoints Launched a New Predictive Analytics Product with Databricks

March 10, 2016 | by Dave Wang

We are excited to announce that Sellpoints, a provider of online advertising solutions dedicated to optimizing return on ad spend (ROAS) for retailers and brands, chose Databricks as their enterprise Spark solution, allowing for the rapid productization of a critical predictive analytics product.

You can read the press release here.

Sellpoints came to Databricks because they wanted to productize an innovative new product based on predictive analytics. Specifically, they wanted to measure the shopping behavior of consumers, glean intelligence from this data (over one billion events per day in this case), and then automatically apply the insights to identify qualified shoppers, run targeted advertising campaigns, and drive prospective shoppers to make a purchase.

The questions Sellpoints faced were common for fast-moving companies that want to build a big data product with Apache Spark: What is the best way to acquire a reliable Spark platform? And what is the most effective way to empower their data teams to become productive? Before choosing Databricks, Sellpoints tried to deploy Apache Spark over Hadoop. But other Spark vendor failed to deliver the performance, reliability, and Spark expertise needed by Sellpoints.

The other solutions did not provide any mechanism to help Sellpoints' business team to leverage the insights from Spark. This meant that Sellpoints had to invest in additional software, and the data science team had to devote time to building dashboards for business users on top of their day-to-day responsibilities.

Sellpoints was able to use Databricks to build its entire data ingest pipeline with Apache Spark in a matter of six weeks. Not only was Databricks able to provide high performance and reliable Spark clusters instantly, it also democratized the access of every Sellpoints team to Spark. The Data scientists used Databricks' integrated workspace to fine-tune models interactively, while the business team took advantage of Spark-powered dashboards to consume insights without any additional BI tools.

With Databricks, Sellpoints gained powerful big data ETL and machine learning capabilities and capture three critical benefits:

- Productized a new predictive analytics offering, improving the ad spend ROI by threefold compared to competitive offerings.
- Reduced the time and effort required to deliver actionable insights to the business team while lowering costs.
- Improved productivity of the engineering and data science team by eliminating the time spent on DevOps and maintaining open source software.

Download this case study to learn more about how Sellpoints is using Databricks.

LendUp Expands Access to Credit with Databricks

December 28, 2015 | by Dave Wang

We are happy to announce a new deployment of Databricks in the financial technology sector with LendUp, a company that builds technology to expand access to credit. LendUp uses Databricks to develop innovative machine learning models that touch all aspects of its lending business. Specifically, it uses Databricks to perform feature engineering at scale and quickly iterate through the model building process. Faster iterations lead to more accurate models, the ability to offer credit to more of the tens of millions of Americans who need it and the ability to establish new products more easily.

Before Databricks, LendUp performed feature extraction on a single machine in AWS. Processing millions of semi-structured documents took multiple days, which delayed updates to critical machine learning models. The productivity of the LendUp team was further impeded by the poor integration of data storage, feature extraction, modeling, and analytics tools, requiring the team to build custom solutions to bring together disparate data sources and analytics tools.

Databricks enables LendUp to speed up feature extraction by replacing the single instance machine with highly tuned Apache Spark clusters. Databricks also provided the critical capabilities to build sophisticated machine learning models in an integrated workspace, replacing a host of disjointed tools and eliminating the need to maintain expensive custom solutions.

As a result of deploying Databricks, LendUp can more readily advance their core mission. LendUp's credit models now include more diverse and larger volumes of semi-structured data, and could be generated within a shorter amount of time.

Download this case study to learn more about how LendUp is using Databricks.

MyFitnessPal Delivers New Feature, Speeds up Pipeline, and Boosts Team Productivity with Databricks

July 2, 2015 | by Kavitha Mariappan and Dave Wang

To learn more about how Databricks helped MyFitnessPal with analytics, check out an earlier article in Wall Street Journal (log-in required) or download the case study.

We are excited to announce that MyFitnessPal (An Under Armour company) uses Databricks to build the production pipeline for its new "Verified Foods" feature, gaining many performance and productivity benefits in the process.

MyFitnessPal aims to build the largest health and fitness community online, by helping people to achieve healthier lifestyles through better diet and more exercise. Health-conscious people can use the MyFitnessPal website or the smartphone app to track their diet and exercise patterns and use the information to reach their fitness goals. MyFitnessPal wanted to further streamline the diet tracking functionality by offering a feature called "Verified Foods", where one can get accurate and up-to-date nutritional information of food items by simply typing the name of the food in the MyFitnessPal application.

To deliver the functionality of "Verified Foods", MyFitnessPal needed to create an accurate food database with a set of sophisticated algorithms. Prior attempts to implement these algorithms without Databricks proved to be not scalable, nor fast enough: They took weeks to run due to the enormous volume of data and their extreme complexity.

MyFitnessPal chose Databricks to implement these algorithms in a production pipeline based on Apache Spark because Databricks delivers the speed and flexibility of Spark in a simple-to-use, zero management platform. Because of the high reliability and fast performance of the data pipeline powered by Databricks, the "Verified Foods" database now includes a comprehensive list of items with readily available and highly accurate nutritional information.

In addition to powering the "Verified Foods" feature, Databricks also delivered a number of key benefits to the Data Engineering & Science team at MyFitnessPal:

- 10X speed improvement, reducing the algorithm run time from weeks to mere hours.

- Dramatically higher team productivity as measured by the number of projects completed in the past quarter.
- Improved team efficiency due to the availability of mature libraries in Spark, and the ability to easily share and re-use code in the Databricks platform.

Download the case study to learn more about Databricks.

How DNV GL Uses Databricks to Build Tomorrow's Energy Grid

April 7, 2016 | Dave Wang

We are proud to announce that DNV GL, a provider of software and independent expert advisory services to the maritime, oil & gas and energy industries, has selected Databricks for large-scale energy data analytics.

You can read the press release here.

DNV GL's expertise and services cover virtually all aspects of the energy industry, including energy efficiency policy, forecasting energy usage, and developing programs for energy conservation. Applying advanced analytics techniques towards energy data is at the core of every DNV GL offering. The data science team at DNV GL analyzes data from smart meters with other climatological, socio- economic, and financial data sources to provide answers to the most challenging questions posed by the top energy and utility companies in the world.

While the proliferation of data from sensors and other sources created opportunities for DNV GL to offer more analytics services to their customers, their analytics team must first overcome the problems associated with processing large volumes of high-frequency sensor data.

In one instance, bottlenecks in legacy analytics platforms delayed time-to-insights by four to five days, preventing DNV GL from meeting their 24-hour turnaround time commitment to customers.

Databricks enabled DNV GL to effortlessly perform advanced analytics on large-scale data from sensors and other data sources with the power of Apache Spark effortlessly. Instead of relying on slow, single-machine analysis, the team was able to spin up Spark clusters in AWS and run machine learning algorithms in minutes. The scalability of the Databricks platform enabled DNV GL to accelerate time-to-value by nearly 100 times than previous methods. In one instance, Databricks improved the processing time of a machine learning batch job from 36 hours to approximately 23 minutes.

As a result of deploying Databricks, DNV GL gained the three broad categories of benefits:

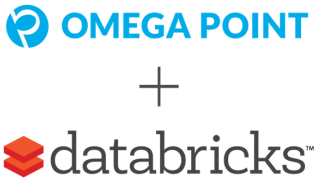
- **Turnkey Spark clusters.** Set up and start provisioning clusters within minutes, compared to several hours with legacy cloud analytics platforms.
- **Faster time to insight.** Speed up data processing by nearly 100 times without incurring additional operational costs.
- **Higher productivity.** Eliminated the need to spend time on DevOps, allowing their data scientists and engineers to focus on solving data problems.

Download this case study to learn more about how DNV GL is using Databricks.

How Omega Point Delivers Portfolio Insights for Financial Services with Databricks

May 3, 2016 | Dave Wang

We are proud to announce that Omega Point, a leading data analytics provider, has selected Databricks for its ability to apply advanced analytics to synthesize insights from disparate large-scale datasets.



You can read the press release here.

Inferring insights from a multitude of large-scale data to inform business-critical decisions is an increasingly common practice across industries. Omega Point is yet another example of this trend in financial services, where a sliver of timely information could have millions of dollars of impact.

Omega Point offers finance and strategy professionals a software platform called *Portfolio Intelligence* that enables them to understand, visualize, and optimize their portfolios. Its offering taps into satellite imagery, web traffic, earnings transcript sentiment, and other large-scale data sets known as “economic exhaust” or “alternative datasets” to uncover portfolios’ exposure to 50+ relevant market factors using cutting edge data science.

Extracting insights from “alternative datasets” requires cutting-edge data processing capabilities to infer valuable information from a plethora of seemingly low-value data. Omega Point selected Apache Spark because of its scalability and flexibility to tackle advanced analytics. However, their initial attempt to deploy Spark by integrating a self-managed PaaS with open source data science tools ran into roadblocks because of reliability and performance problems. As a result, the release of the *Omega Point Portfolio Intelligence Platform* was severely delayed.

Omega Point chose Databricks to power the backbone of its production data pipeline. On a daily basis, Databricks pulls over 120 sources of data directly from Omega Point’s Amazon S3 account and through a sequence of over 80 production Spark jobs, and produces relevant indicators used to assess current economic and financial market trends in clients’ portfolios. Databricks also enabled Omega Point’s data science team to build a systematic learning environment, where Spark’s MLlib and popular open source libraries such as *numpy*, *pandas*, *scipy*, *sklearn*, *matplotlib* could be seamlessly integrated to develop sophisticated models rapidly.

With Databricks, Omega Point gained a high-performance and reliable Spark platform that addressed their reliability and performance bottlenecks. Databricks sped up the release cycle, improved data throughput, and reduced the debugging time for the data engineering

and science teams. Omega Point's deployment and maintenance costs dropped by 75% while its production uptime increased by more than 80%. Ultimately, Databricks enabled Omega Point to accelerate the release of core feature of its flagship product by six months.

Download this case study to learn more about how Omega Point is using Databricks.

Sellpoints Develops Shopper Insights with Databricks

May 11, 2016 | Saman Keshmiri and Christopher Hoshino-Fish

This is a guest blog from our friends at Sellpoints.

Saman is a Data Engineer at Sellpoints, where he responsible for developing, optimizing, and maintaining our backend ETL. Chris is a Data Scientist at Sellpoints, where he is responsible for creating and maintaining analytics pipelines, and developing user behavior models.

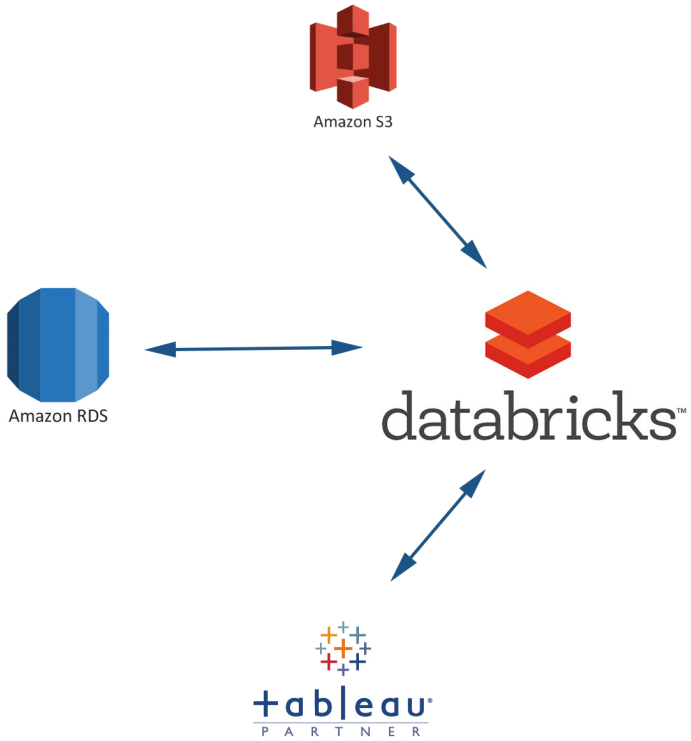
The simplest way to describe Sellpoints is we help brands and retailers convert shoppers into buyers using data analytics. The two primary areas of business for Sellpoints include some of the largest and most complex data sets imaginable; we build and syndicate interactive content for the largest online retailers to help brands increase conversions, and we provide advanced display advertising solutions for our clients. We study behavior trends as users browse retail sites, interact with our widgets, or see targeted display ads. Tracking these events amounts to an average of 5,500 unique data points per second.

The Data Team's primary role is to turn our raw event tracking data into aggregated user behavior information, useful for decision-making by our account managers and clients. We track how people interact with retailer websites prior to purchasing. Our individual log lines represent a single interaction between a customer and some content related to a product, which we then aggregate to answer questions such as "how many visits does a person make prior to purchasing," or "how much of this demonstration video do people watch before closing the window, and are customers more likely to purchase after viewing?" The Data Team here at Sellpoints is responsible for creating user behavior models, producing analytical reports and visualizations for the content we track, and maintaining the visualization infrastructure.

In this blog, we will describe how Sellpoints is able to not only implement our entire backend ETL using the Databricks platform, but unify the entire ETL.

Use Case: Centralized ETL

Sellpoints needed a big data processing platform, one that would be able to replicate our existing ETL, based in Amazon Web Services (AWS), but improve speed and potentially integrate data sources beyond S3, including MySQL and FTP. We also wanted the ability to test MLlib, Apache Spark's machine learning library. We were extremely happy to be accepted as one of Databricks early customers, and outline our goals:



Short Term Goals:

- Replicate existing Hive-based ETL in SparkSQL – processing raw CSV log files into useable tables, optimizing the file size, and partitioning as necessary
- Aggregate data into tables for visualization
- Extract visualization tables with Tableau

Long Term Goals:

- Rewrite ETL in Scala/Spark and optimize for speed
- Integrate MySQL via the JDBC
- Test and productize MLlib algorithms
- Integrate Databricks further into our stack

We have a small Data Team at Sellpoints, consisting of three Data Engineers, one Data Scientist, and a couple Analysts. Because our team is small and we do not have the DevOps resources to maintain a large cluster ourselves, Databricks is the perfect solution. Their platform allows the analysts to spin up clusters with the click of a button, without having to deal with installing packages or specific versions of software. Additionally, Databricks provides a notebook environment for Spark and makes data discovery incredibly intuitive. The ease of use is unparalleled and allows users across our entire Data Team to investigate our raw data.

Architecture and Technical Details

Our immediate goal was to replicate our existing Hive-based ETL in SparkSQL, which turned out to be a breeze. SparkSQL can directly implement Hive libraries and utilizes a virtually identical syntax, so transferring our existing ETL was as simple as copy/paste and uploading the Hive CSV Serde. Soon after, Databricks released their native Spark CSV Serde, and we implemented it without issue. We also needed to extract these tables into Tableau, which Databricks again made simple. Databricks implements a Thrift server by default for JDBC calls, and Tableau has a SparkSQL connector that utilizes the JDBC. We needed to modify one setting here:

```
spark.sql.thriftServer.incrementalCollect = true
```

This tells the Thrift server that data should be sent in small increments rather than collecting all the data into the driver node and then pushing it out, which will cause the driver to run out of space quite quickly if you're extracting a non-trivial amount of data. With that, replicating our ETL in Databricks was complete. We were now leveraging Databricks to process 100GB/day of raw CSV data.

Our next steps involved learning Scala/Spark and the various optimizations that can be made inside Spark, while also starting to integrate other data sources. We had an SQL query that was taking over 2 hours to complete, but the output was being saved to S3 for consumption by Databricks. The query took so long because it involved joining 11 tables together and building a lookup with 10 fields. While optimizations could've been made to the structure of the MySQL db or the query itself, we thought, why not do it in Databricks? We were able to reduce the query time from 2 hours down to less than 2 minutes; and again, since the SparkSQL syntax encompasses all the basic SQL commands, implementing the query was as easy as copy/paste (N.B. we needed to modify our AWS settings in order to get this to work). A simplified version of the code below:

```
import com.mysql.jdbc.Driver

// create database connection
val host = "Internal IP Address of mysql db (10.x.x.x)"
val port = "3306"
val user = "username"
val password = "password"
val dbName = "databaseName"
val dbURL = s"jdbc:mysql://$host:$port/$dbName?user=
$user&password=$password"

//load table as a val
val table =
  sqlContext
    .read
    .format("jdbc")
    .options(
      Map(
        "url" -> s"$dbURL",
        "dbtable" -> "dbName.tableToLoad",
        "driver"  -> "com.mysql.jdbc.Driver"))
    .load()

//register as a temporary SparkSQL table
table.registerTempTable("tableName")

//run the query
val tableData =
```

We also receive and send out files on a daily basis via FTP. We need to download and store copies of these files, so we started downloading them to S3 using Databricks. This allowed us to further centralize our ETL in Databricks . A simplified version of the code below:

```
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPFile;
import org.apache.commons.net.ftp.FTPReply;
import java.io.File;
import java.io.FileOutputStream;

val ftpURL = "ftp.url.com"
val user = "username"
val pswd = "password"
val ftp = new FTPClient()

ftp.connect(ftpURL)
ftp.enterLocalPassiveMode()
ftp.login(user,pswd)

val folderPath = "pathToDownloadFrom"

val files = ftp.listFiles(s"/$folderPath").map(_.getName)

dbutils.fs.mkdirs(s"""/$savePath""")

val outputFile = new File(s"""/dbfs/$savePath/filename.txt""")
val output = new FileOutputStream(outputFile)

ftp.retrieveFile(s"/$folder/filename.txt", output)

output.close()

ftp.logout
```

We switched over from saving our processed data as CSV to Parquet. Parquet is columnar, which means that when you're only using some of the columns in your data, it is able to ignore the other columns. This has massive speed gains when you have trillions of rows, and allows us to decrease time waiting for initial results. We try to keep our individual parquet files around 120MB in size, which is the default block size for Spark and allows the cluster to load data quickly (which we found to be a bottleneck when using small CSV files).

Consolidating our ETL steps into a single location has added benefits when we get to actually analyzing the data. We always try to stay on the cutting edge of machine learning and offline analysis techniques for identifying new user segments, which requires that we maintain a recent set of testable user data, while also being able to compare the output of new techniques to what we're currently using. Being able to consolidate all of our data into a single platform has sped up our iteration process considerably, especially since this single platform includes the MLlib project.

Benefits and Lessons Learned

In the 1.5 years since implementing Databricks, we've been able to:

- Speed up our first-step ETL by ~4x
- Integrate mysql and ftp data sources directly into Databricks, and speed up those portions of the ETL by 100x
- Optimize our processed data for analysis

We have also learned a lot during this process. A few tips and lessons include:

- Store your data as Parquet files
- Leverage the processing power of Spark and implement any RDS ETL in Databricks via the JDBC
- Partition your data as granularly as possible while maintaining larger file sizes – loading data is slow, but processing is fast!

To learn more about how Sellpoints is using Databricks, read the case study.

A special thank you to Ivan Dejanovic for his edits and patience.

Conclusion

Our mission at Databricks is to dramatically simplify big data processing and data science 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 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, check out the previous books in the series and visit the Databricks Blog for more technical tips, best practices, and case studies from the Apache 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 Databricks yourself, start your free trial today!