Feb 2nd, 2015

# Scalable Genomes Clustering With ADAM and Spark

In this post, we will detail how to perform simple scalable population stratification analysis, leveraging ADAM and Spark MLlib, as previously presented at scala.io (http://www.slideshare.net/noootsab/lightning-fast-genomics-with-spark-adam-and-scala).

The data source is the set of genotypes from the 1000genomes (http://1000genomes.org) project, resulting from whole genomes sequencing run on samples taken from about 1000 individuals with a known geographic and ethnic origin.

This dataset is rather large and allows us to test scalability of the methods we present here and gives us the possibility to do interesting machine learning. Based on the data we have, we can for example:

- build models to classify genomes by population
- run unsupervised learning (clustering) to see if populations are reconstructed in the model.
- build models to infer missing genotypes

We've gone the second way (clustering), the line-up being the following:

- Setup the environment
- Collection and extraction of the original data
- Distribute the original data and convert it to the ADAM model
- Collect metadata (samples labels and completeness)
- Filter the data to match our cluster capacity (number of nodes, cpus and mem and wall clock time…)
- Read and prepare the ADAM formatted and distributed genotypes to have them into a separable high-dimensional space (need a metric)
- Apply the KMeans (train/predict)
- Assess performance

## Environment setup

### Cluster

One of the easiest way to setup an environment with flexibility on deployed resources is EC2. Especially because Spark is distributed with scripts to spawn clusters preconfigured on EC2 (see http://spark.apache.org/docs/1.2.0/ec2-scripts.html (http://spark.apache.org/docs/1.2.0/ec2-scripts.html)).

For the case we're discussing here, there are several points worth considering:

- instances flavor: we opted for `m3.xlarge` to give us more memory
- the region: we used `eu-west-1`. Based in Europe, we'd like to have the results nearby
- hadoop 2: this was necessary to deal with the VCFs (use the `--hadoop-major-version="2"` argument)
- **EBS**: since we'll use the result often, we created ESB to have the data persistent even after cluster is stopped (use the `--ebs-vol-size="100"` for `100G` per instance).

A cluster with 4 slaves and 1 master will take about 20 minutes to spawn. When the cluster is stopped, the data in the persistent hdfs (ESB) remains and will be readily available after the following start. They'll be lost only if the cluster is explicitely destroyed.

*Remark*: the spark ec2 scripts install two instances of hdfs, ephemeral and persistent, however only the ephemeral is started. So, you'll need to start the persistent one yourself using:

```
/root/persistent-hdfs/sbin/start-dfs.sh
```

It can also be insteresting to shutdown the ephemeral (to save some memory for instance).

## s3cmd

Since the data we use is available on S3, a client is required, it is worthwhile to install `s3cmd` if some data management is done from the shell.

Luckily, it's very simple, and everything is explained here (http://s3tools.org/s3cmd).

## Spark Notebook

For the operational part, we use the Spark Notebook (http://github.com/andypetrella/spark-notebook). It is our favorite choice because we need something that can rerun our tasks and accomodate easily for changes, in an interactive way.

The easiest is to download the distribution that **matches** both the spark and hadoop versions installed on the cluster. The distributions are available on s3 (https://s3.eu-central-1.amazonaws.com/spark-notebook/index.html) or docker (https://registry.hub.docker.com/u/andypetrella/spark-notebook/), here is the zip (https://s3.eu-central-1.amazonaws.com/spark-notebook/zip/spark-notebook-0.2.1-spark-1.2.0-hadoop-2.0.0-cdh4.2.0.zip) for spark 1.2.0 and hadoop 2.0.0 cdh4.2.0.

Before starting the notebook, you have to make sure to load the spark environment variables (`/root/spark/conf/spark-env.sh`). And to use s3, the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables must be set as well.

The spark-notebook server can then be launched from the root of its installation, using for example port 8999 (because the default port 9000 is used by hadoop):

```
bin/spark-notebook -Dhttp.port=8999
```

You can access the UI in your browser on localhost:8999 by opening an ssh tunnel, for exemple from you local machine issuing:

```
ssh -L 8999:localhost:8999 <spark-master>
```

It might also be required to open the 8999 port on the ec2 console.

In the distribution, a notebook called `Clustering Genomes using Adam and MLLib` contains the code this blog post is illustrating.

# Data collection

The 1000 genomes project genotypes are available in VCF format from ftp servers (http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase1/analysis_results/integrated_call_sets/) (ncbi and ebi) and also in s3 (http://aws.amazon.com/1000genomes/).

While repositories with such datasets converted in ADAM format are under development (f.i. eggo (https://github.com/bigdatagenomics/eggo)), most datasets have to be collected from traditional (e.g ftp servers) sources and distributed/converted for scalable processing.

The master node EBS disk is used as a buffer space to get the gzipped vcf files (one per chromosome), decompress them and send them to hdfs. Below, you'll find the flow for chomosome 1.

## Get the VCF for chromosome 1

With the EBS disk mounted on `/vol0` :

```
cd /vol0/data
s3cmd get s3://1000genomes/phase1/analysis_results/integrated_call_sets/ALL.chr1.integrated_phase1_v3.20101123.snps_indels_svs.genotypes.vcf.gz
```

## Decompress

As seen above, the files are gizzed, hence we need to decompress them. However, it takes quite a while, so launch the following command and grab a beer!

```
gunzip ALL.chr1.integrated_phase1_v3.20101123.snps_indels_svs.genotypes.vcf.gz
```

This task takes around **one hour**, as we'll see later on, it explains why ADAM is so important when dealing with such data.

## Put VCF in persistent HDFS

The unzipped vcf file then has to be copied to hdfs in order to be readable with ADAM. This is optional but then, the convertion has to be done from the driver (where the VCF resides) rather than on the cluster.

```
/root/persistent-hdfs/bin/hadoop fs -put ALL.chr1.integrated_phase1_v3.20101123.snps_indels_svs.genotypes.vcf /data/ALL.chr1.vcf
```

## Free some space on disk

Delete VCF from disk! Along the same line, after having converted the VCF in ADAM and saved either on the hdfs or in s3, it can be good to remove the VCF from hdfs and save space.

# Notebook

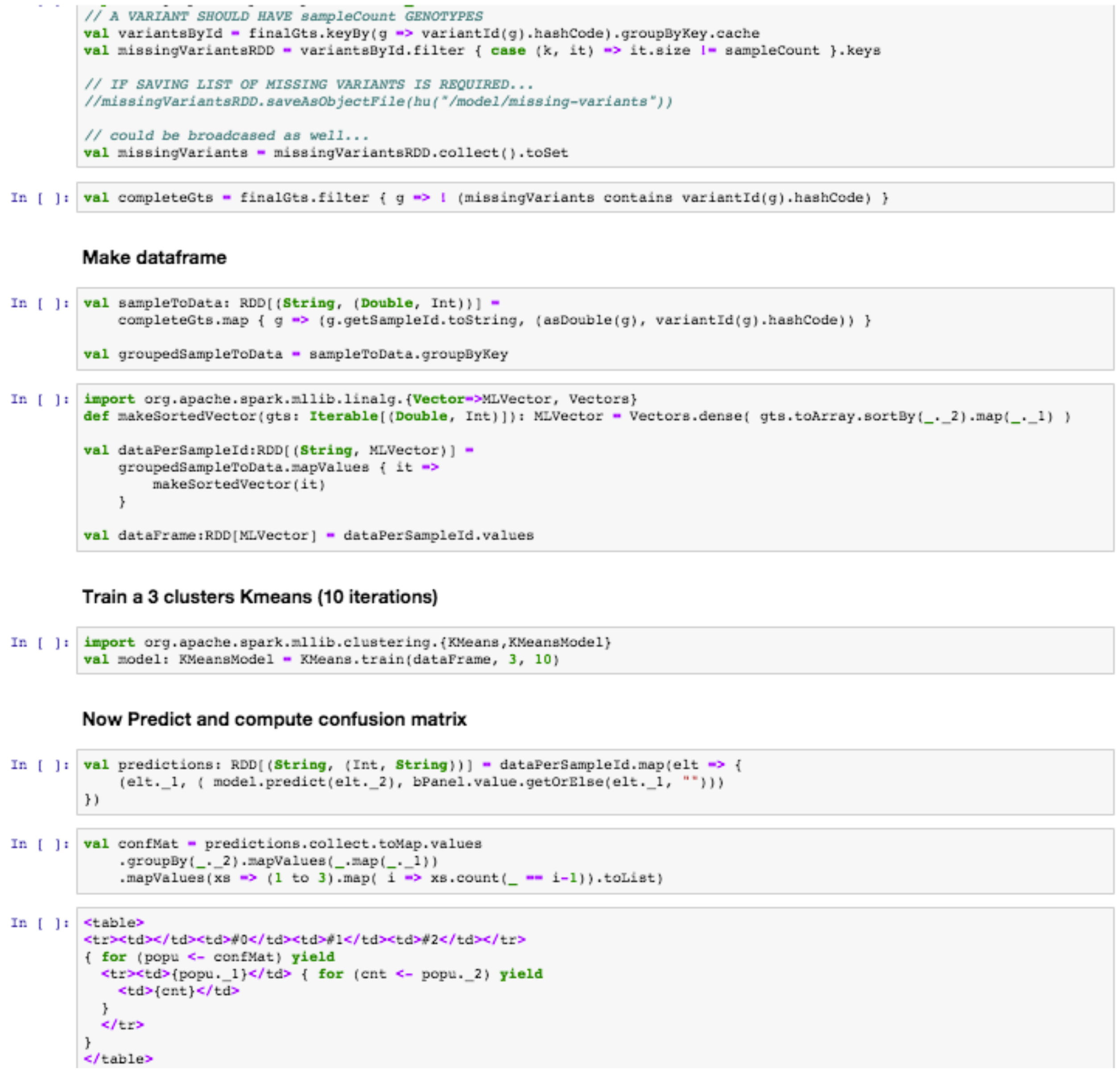
In the next section we'll cover the nitty gritty details of our exploration and results.

Although some code excerpts are presented, yet seeing them running can improve satisfaction or reduce perplexity.

That's why we created some notebooks for you! To use them, launch the Spark Notebook as descrived above, you'll see them in the default list:

- Convert ADAM
- Read 1000Genomes dataset (chr-N)
- Clustering Genomics Data using Adam and MLLib

Here is a screenshot of the clustering analysis notebook:



```
// A VARIANT SHOULD HAVE sampleCount GENOTYPES
val variantsById = finalGts.keyBy(g => variantId(g).hashCode).groupByKey.cache
val missingVariantsRDD = variantsById.filter { case (k, it) => it.size != sampleCount }.keys

// IF SAVING LIST OF MISSING VARIANTS IS REQUIRED...
//missingVariantsRDD.saveAsObjectFile(hu("/model/missing-variants"))

// could be broadcased as well...
val missingVariants = missingVariantsRDD.collect().toSet
```

```
In [ ]: val completeGts = finalGts.filter { g => ! (missingVariants contains variantId(g).hashCode) }
```

**Make dataframe**

```
In [ ]: val sampleToData: RDD[(String, (Double, Int))] =
            completeGts.map { g => (g.getSampleId.toString, (asDouble(g), variantId(g).hashCode)) }

        val groupedSampleToData = sampleToData.groupByKey
```

```
In [ ]: import org.apache.spark.mllib.linalg.{Vector=>MLVector, Vectors}
        def makeSortedVector(gts: Iterable[(Double, Int)]): MLVector = Vectors.dense( gts.toArray.sortBy(_._2).map(_._1) )

        val dataPerSampleId:RDD[(String, MLVector)] =
            groupedSampleToData.mapValues { it =>
                makeSortedVector(it)
            }

        val dataFrame:RDD[MLVector] = dataPerSampleId.values
```

**Train a 3 clusters Kmeans (10 iterations)**

```
In [ ]: import org.apache.spark.mllib.clustering.{KMeans,KMeansModel}
        val model: KMeansModel = KMeans.train(dataFrame, 3, 10)
```

**Now Predict and compute confusion matrix**

```
In [ ]: val predictions: RDD[(String, (Int, String))] = dataPerSampleId.map(elt => {
            (elt._1, ( model.predict(elt._2), bPanel.value.getOrElse(elt._1, "")))
        })
```

```
In [ ]: val confMat = predictions.collect.toMap.values
            .groupBy(_._2).mapValues(_.map(_._1))
            .mapValues(xs => (1 to 3).map( i => xs.count(_ == i-1)).toList)
```

```
In [ ]: <table>
        <tr><td></td><td>#0</td><td>#1</td><td>#2</td></tr>
        { for (popu <- confMat) yield
          <tr><td>{popu._1}</td> { for (cnt <- popu._2) yield
            <td>{cnt}</td>
          }
          </tr>
        }
        </table>
```

# Data Analysis

## Data preparation (Convert VCF to ADAM)

Now that the VCF file is in HDFS, we can use ADAM and our cluster to convert it to the ADAM format, which undr the hood is a parquet (optimized) version based on the bdg-formats (https://github.com/bigdatagenomics/bdg-formats) schema (in avro). The resulting data consists of partitions saved as gz files (each of size 7MB), either on the cluster hdfs or on s3. In our case, we saved on both, a local copy for performance and a s3 copy reusable on other clusters.

The code to do this is pretty trivial:

```
// UTIL FUNCTION TO MAKE HDFS URLS
def hu(s:String) = s"hdfs://$master:9010/data/$s"

// INPUT AND OUTPUT FILES ON HDFS
val vcfFile = hu("/data/ALL.chr1.vcf")
val outputFile = vcfFile+".adam"

// READ-CONVERT-SAVE
val variantContext: RDD[VariantContext] = sparkContext.adamVCFLoad(vcfFile, dict = None)
val genotypes = variantContext.flatMap(p => p.genotypes)
gts.adamParquetSave(outputFile)
```

## Samples: location filter and population labels

For practical reasons (available resources), we will not train the k-means model on all variants. We select a pretty arbitrary slice of a chromosome to limit ourselves to a dataset size that is processed in a few minutes.

For example, selecting genotypes for variants located on chromosome1 between position 1 and 1,000,000 is done with a simple filter:

```
val start = 1
val end = 1000000
val sampledGts = genotypes.filter(g => (g.getVariant.getStart >= start && g.getVariant.getEnd <= end) )
```

Our protocol consists in measuring how the processing a fixed sample will scale with the cluster size. We also check how performance scales with dataset size by varying the number of variants.

Also, we do not include all populations, the reason is that populations relationships are best represented by hierarchical clustering, using simple K-means will not work well if we do not flatten the structure. So we select only 3 populations and train the K-means with 3 clusters. This really aims at targeting the purpose of evaluating the technologies, not discovering something original in the data.

The samples populations are available from the 1000genomes data repository and are converted into a map with samples IDs as keys and populations labels as value. This map is then broadcasted in the cluster to avoid shipping it in every closure:

```
# IN THE SHELL...
wget ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/integrated_call_samples_v3.20130502.ALL.panel -O /vol0/data/ALL.panel
```

```
// IN THE NOTEBOOK
val panelFile = "/vol0/data/ALL.panel"

// populations to select
val pops = Set("GBR", "ASW", "CHB")

// TRANSFORM THE panelFile Content in the sampleID -> population map
// containing the populations of interest (pops)
val panel: Map[String, String] = ...

// broadcast the panel
val bPanel = sparkContext.broadcast(panel)
```

And we can filter the genotypes for hte selected populations:

```
genotypes.filter(g =>  bPanel.value.contains(g.getSampleId))
```

To understand if the k-means extracted population structure, we will compare the clusters assignments with the populations labels of the samples, i.e. in a confusion matrix.

## Missing data

Some data is missing, a few genotypes are not present in the Sample x Variant matrix. As we have plenty of variants to play with (up to ~ 30,000,000), removing the ones for which some genotypes are missing across the 1000 samples does not hurt.

First, we must identify all such incomplete variants and optionally save the list on disk, this can come handy for the prediction phase. For convenience (later runs), the list of complete list of variants is saved as well:

```
// NUMBER OF SAMPLES
val sampleCount = genotypes.map(_.getSampleId.toString.hashCode).distinct.count

// A VARIANT SHOULD HAVE sampleCount GENOTYPES
// variantId returns string identifier for a variant (see notebook ref...)
val variantsById = gts.keyBy(g => variantId(g).hashCode).groupByKey
val missingVariantsRDD = variantsById.filter { case (k, it) => it.size != sampleCoun
t }.keys
missingVariantsRDD.saveAsObjectFile("/tmp/model/missing-variants")

// could be broadcased as well...
val missingVariants = missingVariantsRDD.collect().toSet
```

Then, we remove all these incomplete variants from the dataset:

```
genotypes.filter { g => ! (missingVariants contains variantId(g).hashCode) }
```

## Features extraction

Before running the clustering algorithm (K-Means), we need to transform the data from a flat representation (RDD of genotypes) to a more structured one, matching the input requirements of MLLib training methods.

Each sample must be represented by a vector of features in a space with a defined metric. MLLib relies on the breeze library for linear algebra and the euclidian metric is the one provided.

Usually a Mahanatan distance is used in genetics, with genotypes encoded as 0, 1 or 2 (1 being the heterozygote). We have used this encoding albeit with breeze provides only the euclidian distance. A `asDouble(Genotype)` function does the genotype encoding.

The rdd tranformations to obtain encoded genotypes, grouped by sampleId are:

```
val sampleToData: RDD[(String, (Double, Int))] =
    genotypes.map { g => (g.getSampleId.toString, (asDouble(g), variantId(g).hashCode
)) }

val groupedSampleToData = sampleToData.groupByKey
```

And for each sample, we sort the genotypes by variant (i.e. variant name hash) so that each sample vector has its features consistently ordered (Vector is the MLLib Vector class):

```
def makeSortedVector(gts: Iterable[(Double, Int)]): Vector = Vectors.dense( gts.toArr
ay.sortBy(_._2).map(_._1) )

val dataPerSampleId:RDD[(String, MLVector)] =
    groupedSampleToData.mapValues { it =>
      makeSortedVector(it)
    }

val dataFrame:RDD[MLVector] = dataPerSampleId.values
```

At this stage, we have a dataset ready for training with MLLib!

## Training and Predictions with K-Means

Training the model is achieved very easily, in this case with 3 clusters and 10 iterations…

```
val model: KMeansModel = KMeans.train(dataFrame, 3, 10)
```

In order to check whether the samples clusters match the samples populations, we used the model to predict the cluster of each sample and compared these with the population label of the sample.

There is one prediction for each sample (the key of the predictions RDD), as value we keep the predicted class (the cluster number as Int) and the population label:

```
val predictions: RDD[(String, (Int, String))] = dataPerSampleId.map(elt => {
    (elt._1, ( model.predict(elt._2), bPanel.value.get(elt._1)))
})
```

We can extract and display the confusion matrix, clearly showing that the clustering actually matches pretty well the population:

```
    #0   #1   #2
GBR  0    0   89
ASW 54    0    7
CHB  0   97    0
```

## Performance

We have taken a few metrics to get an idea of how the ADAM and MLLib scale with available resources and dataset size. We ran the notebook on 2 clusters (2 and 20 slaves). We processed 3 datasets, one is a very limited sample (2,168 variants) the next is a medium one (121,023 variants). We also processed the entire chromosome 22 but only on the 20 nodes cluster (491,222 variants).

Note that we processed 114 partitions, which in the case of the 20 nodes (80 cores) cluster leads to a penalty because on average, 114/80 tasks are assigned to a core while 2 to 3 minimum are required to evenly distribute cores utilization. We systematically lose a factor 1.5 in performance on the 20 nodes cluster.

```
                                    2 NODES       20 NODES

Cluster launch:                      10 min         30 min

Count chr22 genotypes (from S3):      6 min        1.1 min
Save chr22 from s3 to HDFS:          26 min        3.5 min
Count chr22 genotypes (from HDFS):   10 min        1.4 min

2168 Variants
Missing data (collect):               7 sec          3 sec
Train (10 iterations):               20 sec         30 sec
Predict (collect):                  0.5 sec        0.3 sec

121,023 Variants
Missing data (collect):             7.8 min         33 sec
Train (10 iterations):              2.1 min         28 sec
Predict (collect):                    8 sec          2 sec

491,222 Variants
Missing data (collect):                            3.7 min
Train (10 iterations):                             1.6 min
Predict (collect):                                  25 sec
```

We have not gathered here other metrics like memory utilization, amount of data shuffled etc, but this gives already a good idea on the scalability of the processing with ADAM and MLLib.

## Conclusions

We have shown a flow to manipulate genetic data at scale with ADAM and MLLib. With the help of the spark notebook, it is pretty easy to develop such scalable genomes processing on top of ADAM and Spark. The cluster size is very transparent for the development phase, and the system proves to scale well with dataset size and number of node.

All in all, it becomes really fun and efficient to engage into distributed computing with such good APIs (ADAM, Spark), underlying data formats (parquet, avro), infrastructure (EC2 and the like), machine learning implementations

(MLLib) and interactive development/execution environments (Spark-notebook).

Posted by Big Data Genomics Feb 2nd, 2015 ADAM (/blog/categories/adam/), talks (/blog/categories/talks/)

Tweet 0 Like Share Sign Up to see what your friends like.

« ADAM 0.15.0 Released (/blog/2014/11/26/adam-0-dot-15-dot-0-released/)

ADAM 0.16.0 Released » (/blog/2015/02/18/adam-0-dot-16-dot-0-released/)

## Comments

**2 Comments** bigdatagenomics Login

Recommend Share Sort by Best

Join the discussion…

**Deborah Siegel** · a month ago

Hi, thanks for the great post. Did spark on ec2 have access to the adam operations, or did you scp the adam jar to ec2? (as described here -> https://github.com/bigdatageno... )

· Reply · Share ›

**zenfractal** Mod Deborah Siegel · a month ago

It's easiest to scp the ADAM jar to ec2 and launch the job from there. The Spark driver coordinates with the workers to run the ADAM job. Launching the job from your machine would require modifying the firewall (or setting up a proxy) to allow connections between the driver and workers.

Drop into our Gitter channel if you have any questions: https://gitter.im/bigdatagenom...

· Reply · Share ›

ALSO ON **BIGDATAGENOMICS** WHAT'S THIS?

**An example RFC**

1 comment • a year ago

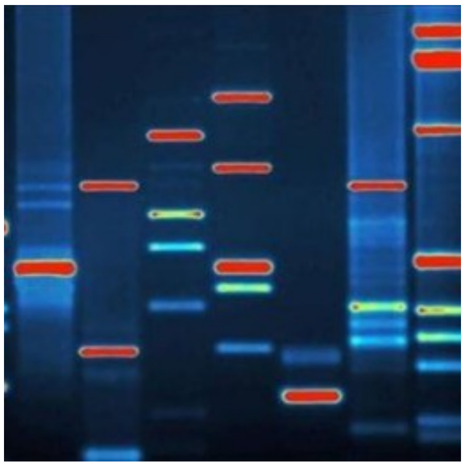
Avatarzenfractal — All RFCs are open to comment by the general community.

**ADAM 0.14.0 Released**

1 comment • 6 months ago

Avatarjing lin — hello, I see in the doc says JAR is available in adam/adam-cli/target/adam-0.6.1.jar,but i do not find it in three。 why

Subscribe Add Disqus to your site Privacy

 (/)

---

*Chat with us...* GITTER JOIN CHAT → (https://gitter.im/bigdatagenomics/adam?utm_source=badge&utm_medium=badge&utm_campaign=pr-badge)

*If you're interested in contributing, take a look at the open "pick me up!" issues (https://github.com/bigdatagenomics/adam/issues?labels=pick+me+up%21&page=1&state=open).*

---

This project is supported in part by NIH BD2K Award 1-U54HG007990-01 and NIH Cancer Cloud Pilot Award HHSN261201400006C

---

GitHub Repos

### eggo

Ready-to-go Parquet-formatted public 'omics datasets

(https://github.com/bigdatagenomics/eggo)

### adam

A genomics processing engine and specialized file format built using Apache Avro, Apache Spark and Parquet. Apache 2 licensed.

(https://github.com/bigdatagenomics/adam)

### avocado

A Variant Caller, Distributed. Apache 2 licensed.

(https://github.com/bigdatagenomics/avocado)

### PacMin

Assembler for PacBio reads. Apache 2 licensed.

(https://github.com/bigdatagenomics/PacMin)

### recipes

Recipes using BDG projects. Apache 2 licensed.

(https://github.com/bigdatagenomics/recipes)

### bigdatagenomics.github.io
Web Site for the Big Data Genomics Group

(https://github.com/bigdatagenomics/bigdatagenomics.github.io)

### RNAdam
An RNA pipeline built on top of ADAM. Apache 2 licensed.

(https://github.com/bigdatagenomics/RNAdam)

### mango
Visualization tools for genomic data. Apache 2 licensed.

(https://github.com/bigdatagenomics/mango)

### ContEst
Spark implementation of the ContEST contamination estimation algorithm.

(https://github.com/bigdatagenomics/ContEst)

### qc-metrics
Read and variant metrics, useable for pipeline quality control purposes. Apache 2 licensed.

(https://github.com/bigdatagenomics/qc-metrics)

### corretto
Read error correction utilities.

(https://github.com/bigdatagenomics/corretto)

### bdg-utils
General (non-omics) code used across BDG products. Apache 2 licensed.

(https://github.com/bigdatagenomics/bdg-utils)

### bdg-formats
Open source formats for scalable genomic processing systems using Avro. Apache 2 licensed.

(https://github.com/bigdatagenomics/bdg-formats)

### bdg-services
Utility classes for wrapping services or other interfaces around a Spark/ADAM cluster. Apache 2 licensed.

(https://github.com/bigdatagenomics/bdg-services)

@bigdatagenomics (https://github.com/bigdatagenomics) on GitHub

## Latest Tweets

**Tweets** Follow

**Big Data Genomics** @bigdatagenomics 18 Feb

ADAM 0.16.0 is out! BQSR 3.5x faster, multiline FASTQ, better visualization, RegionJoin impl, region coverage calc bdgenomics.org/blog/2015/02/1…

Expand

**Big Data Genomics** @bigdatagenomics 10 Feb

Step-by-step guide: using ML for population stratification, leveraging ADAM, Spark MLlib and Spark Notebook bdgenomics.org/blog/2015/02/0…

Expand

**Spark Notebook** @SparkNotebook 2 Jan

Soon new version to be published. Meanwhile, if you wanna play @bigdatagenomics & ADAM → github.com/andypetrella/s… High 5 @xtordoir @noootsab

Retweeted by Big Data Genomics

Expand

**Big Data Genomics** @bigdatagenomics 26 Nov

ADAM 0.15.0 released. Memory and performance improvements, improved

Tweet to @bigdatagenomics

### Recent Posts

ADAM 0.16.0 Released (/blog/2015/02/18/adam-0-dot-16-dot-0-released/)

Scalable Genomes Clustering With ADAM and Spark (/blog/2015/02/02/scalable-genomes-clustering-with-adam-and-spark/)

ADAM 0.15.0 Released (/blog/2014/11/26/adam-0-dot-15-dot-0-released/)

Lightning Fast Genomics (/blog/2014/11/03/lightning-fast-genomics/)

ADAM 0.14.0 Released (/blog/2014/09/17/adam-0-dot-14-dot-0-released/)

*YourKit is supporting the Big Data Genomics open source project with its full-featured Java*

*Profiler. YourKit, LLC is the creator of innovative and intelligent tools for profiling Java and .NET applications. Take a look at YourKit's leading software products: YourKit Java Profiler (http://www.yourkit.com/java/profiler/index.jsp) and YourKit .NET Profiler (http://www.yourkit.com/.net/profiler/index.jsp).*