All vs. All Correlation Using Spark/Hadoop

Mahmoud Parsian
Ph.D in Computer Science

Senior Architect @ illumina[1]

July 23, 2015

[1] www.illumina.com

Table of Contents

Outline

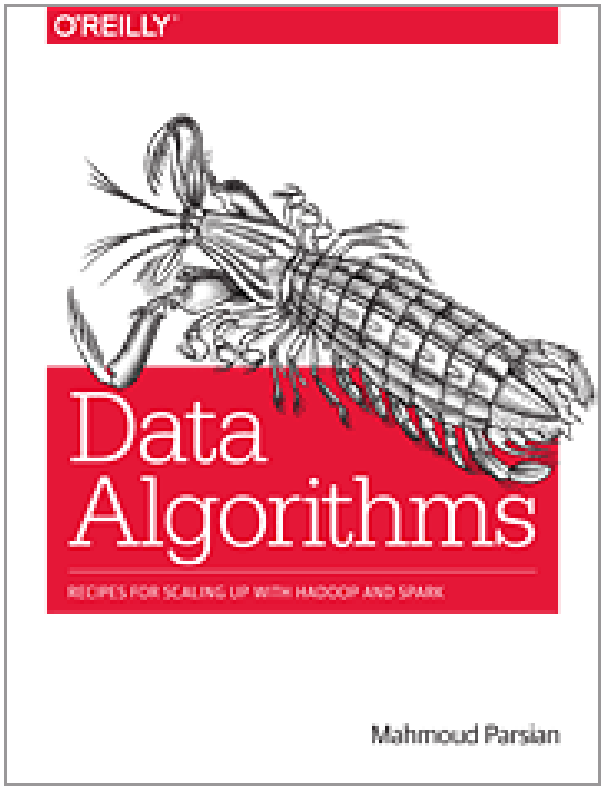
Who am I?

- Name: Mahmoud Parsian
- Education: Ph.D in Computer Science
- Work: Senior Architect @Illumina, Inc
  - Lead Big Data Team @Illumina
  - Develop scalable regression algorithms
  - Develop DNA-Seq and RNA-Seq workflows
  - Use Java/MapReduce/Hadoop/Spark/HBase
- Author: of 3 books
  - Data Algorithms (O'Reilly: http://shop.oreilly.com/product/0636920033950.do/)
  - JDBC Recipies (Apress: http://apress.com/)
  - JDBC MetaData Recipies (Apress: http://apress.com/))

Outline

Problem Statement: Data Algorithms Book

- Details are in Chapter 23:
  http://shop.oreilly.com/product/0636920033950.do
- Source code:
  https://github.com/mahmoudparsian/data-algorithms-book



Problem Statement

- Given thousand of biomarkers for patients, the webcast will show an efficient way of correlating {**"all genes"**} vs. {**"all genes"**}.
- Webcast covers Pearson and Spearman correlations implemented in Spark/Hadoop.
- Magnitude of this data is challenging to store and analyze:
  - several thousands of biomarkers per patient
  - several billions of genes to correlate and analyze
  - Correlate { All Genes } vs. { All Genes }

Problem Statement

- Given thousand of biomarkers for patients, the webcast will show an efficient way of correlating {**"all genes"**} vs. {**"all genes"**}.
- Webcast covers Pearson and Spearman correlations implemented in Spark/Hadoop.
- Magnitude of this data is challenging to store and analyze:
  - several thousands of biomarkers per patient
  - several billions of genes to correlate and analyze
  - Correlate { All Genes } vs. { All Genes }

Problem Statement

- Given thousand of biomarkers for patients, the webcast will show an efficient way of correlating {**"all genes"**} vs. {**"all genes"**}.
- Webcast covers Pearson and Spearman correlations implemented in Spark/Hadoop.
- Magnitude of this data is challenging to store and analyze:
  - several thousands of biomarkers per patient
  - several billions of genes to correlate and analyze
  - Correlate { All Genes } vs. { All Genes }

Magnitude of Data per Analysis

- 1000's of patients
- Each patient may have 100's of 1000's of biomarkers
- Each biomarker may have up to 50,000 genes
- Selected Data = {60,000 biomarkers}
- All-vs-All correlation data = { 60,000 * 50,000 } = 3B records to aggregate
- All-vs-All reduced pairs = { 50,000 } * { 50,000 } = 2.5B pairs
- All-vs-All removing duplicate pairs = { 2.5B / 2 } = 1.25B pairs

Magnitude of Data per Analysis

- 1000's of patients
- Each patient may have 100's of 1000's of biomarkers
- Each biomarker may have up to 50,000 genes
- Selected Data = {60,000 biomarkers}
- All-vs-All correlation data = { 60,000 * 50,000 } = 3B records to aggregate
- All-vs-All reduced pairs = { 50,000 } * { 50,000 } = 2.5B pairs
- All-vs-All removing duplicate pairs = { 2.5B / 2 } = 1.25B pairs

Magnitude of Data per Analysis

- 1000's of patients
- Each patient may have 100's of 1000's of biomarkers
- Each biomarker may have up to 50,000 genes
- Selected Data = {60,000 biomarkers}
- All-vs-All correlation data = { 60,000 * 50,000 } = 3B records to aggregate
- All-vs-All reduced pairs = { 50,000 } * { 50,000 } = 2.5B pairs
- All-vs-All removing duplicate pairs = { 2.5B / 2 } = 1.25B pairs

Magnitude of Data per Analysis

- 1000's of patients
- Each patient may have 100's of 1000's of biomarkers
- Each biomarker may have up to 50,000 genes
- Selected Data = {60,000 biomarkers}
- All-vs-All correlation data = { 60,000 * 50,000 } = 3B records to aggregate
- All-vs-All reduced pairs = { 50,000 } * { 50,000 } = 2.5B pairs
- All-vs-All removing duplicate pairs = { 2.5B / 2 } = 1.25B pairs

Magnitude of Data per Analysis

- 1000's of patients
- Each patient may have 100's of 1000's of biomarkers
- Each biomarker may have up to 50,000 genes
- Selected Data = {60,000 biomarkers}
- All-vs-All correlation data = { 60,000 * 50,000 } = 3B records to aggregate
- All-vs-All reduced pairs = { 50,000 } * { 50,000 } = 2.5B pairs
- All-vs-All removing duplicate pairs = { 2.5B / 2 } = 1.25B pairs

Magnitude of Data per Analysis

- 1000's of patients
- Each patient may have 100's of 1000's of biomarkers
- Each biomarker may have up to 50,000 genes
- Selected Data = {60,000 biomarkers}
- All-vs-All correlation data = { 60,000 * 50,000 } = 3B records to aggregate
- All-vs-All reduced pairs = { 50,000 } * { 50,000 } = 2.5B pairs
- All-vs-All removing duplicate pairs = { 2.5B / 2 } = 1.25B pairs

Outline

Some Basic Definitions

- Biomarker
- Biomarker Record
- Gene

Some Basic Definitions

- Biomarker
- Biomarker Record
- Gene

Some Basic Definitions

- Biomarker
- Biomarker Record
- Gene

What is a Biomarker?

- Individually analyzed data signatures are referred to as "biomarkers".
- Biomarkers encompass data in the form of
  - experimental sample comparisons
  - as well as genotype signatures
- A biomarker most commonly referred to as a "gene signature".
- A sample record of a biomarker will contain a **Gene**, **Reference**, **Patient-ID**, and **Biomarker value**
- A gene is the molecular unit of heredity of a living organism
- Each biomarker has a set of genes

Size of a Biomarker?

- A patient may have any number of biomarkers
- A sample record of a biomarker will contain a Gene, Patient-ID, and Biomarker value
- The number of entries/records for a RNA-Expression can have 50,000 records
- The number of entries/records for a whole genome can have 4.3 million records

Outline

Patient Sample to Biomarkers

Patient Sample $\longrightarrow$ Biomarkers

- Record of a Bioset?
  - Gene-ID (as String)
  - Reference-ID as {r1, r2, r3, r4 }
    - r1: normal
    - r2: disease
    - r3: paired
    - r4: none
  - Patient-ID (as String)
  - Biomarker-Value (as double data type)
- Sample Record of a Bioset:

```
G1234,r2,P1200,0.75
G3456,r3,P1400,0.47
```

Patient Sample to Biomarkers

Patient Sample $\longrightarrow$ Biomarkers

- Record of a Bioset?
  - Gene-ID (as String)
  - Reference-ID as {r1, r2, r3, r4 }
    - r1: normal
    - r2: disease
    - r3: paired
    - r4: none
  - Patient-ID (as String)
  - Biomarker-Value (as double data type)
- Sample Record of a Bioset:

```
G1234,r2,P1200,0.75
G3456,r3,P1400,0.47
```

Patient Sample to Biomarkers

Patient Sample $\longrightarrow$ Biomarkers

- Record of a Bioset?
  - Gene-ID (as String)
  - Reference-ID as {r1, r2, r3, r4 }
    - r1: normal
    - r2: disease
    - r3: paired
    - r4: none
  - Patient-ID (as String)
  - Biomarker-Value (as double data type)
- Sample Record of a Bioset:

```
G1234,r2,P1200,0.75
G3456,r3,P1400,0.47
```

Input Data: Biomarkers

- Data persists in HDFS
- Data structure:

```
/input/<patientID>/<sample1>/biomarker-11.txt
/input/<patientID>/<sample1>/biomarker-12.txt
...
/input/<patientID>/<sample2>/biomarker-21.txt
/input/<patientID>/<sample2>/biomarker-22.txt
...
```

All vs. All Correlation Formulation

What does it mean?

Let Patients $= \{P_1, P_2, ..., P_n\}$

Let Genes $= \{G_1, G_2, ..., G_m\}$

Build a Correlation Matrix				
	P_1	P_2	...	P_n
G_1	value-11	value-12	...	value-1n
G_2	value-21	value-22	...	value-2n
...	...	...	...	...
G_m	value-m1	value-m2	...	value-mn

All vs. All Correlation Formulation

Let Patients $= \{P_1, P_2, ..., P_n\}$
Let Genes $= \{G_1, G_2, ..., G_m\}$

Build a Matrix				
	P_1	P_2	...	P_n
G_1	value-11	value-12	...	value-1n
G_2	value-21	value-22	...	value-2n
...	...	...	...	...
G_m	value-m1	value-m2	...	value-mn

Correlate(G_i, G_j) for $i, j \in \{1, 2, ..., m\}$
where $i < j$

All vs. All Correlation Formulation

Example: Genes $= \{G_1, G_2, G_3, G_4\}$
then the following pairs will be correlated:

$$(G_1, G_2)$$
$$(G_1, G_3)$$
$$(G_1, G_4)$$
$$(G_2, G_3)$$
$$(G_2, G_4)$$
$$(G_3, G_4)$$

Correlate$(G_i,\ G_j)$ for $i, j \in \{1, 2, 3, 4\}$
where $i < j$

Desired Output

(G_i, G_j) (correlation, p-value)
where

- $i < j$
- $-1.00 \leq \text{correlation} \leq 1.00$
- $0.00 \leq \text{p-value} \leq 1.00$

Outline

All-vs-All Algorithm in Spark

Algorithm: High-Level Steps	
Step	**Description**
STEP-1	handle input parameters
STEP-2	create a Spark context object
STEP-3	create list of input files/biomarkers
STEP-4	broadcast reference as global shared object
STEP-5	read all biomarkers from HDFS and create the first RDD
STEP-6	filter biomarkers by reference
STEP-7	create `[(Gene-ID), (Patient-ID, Gene-Value)]` pairs
STEP-8	group biomarkers by geneID
STEP-9	create Cartesian product of all genes
STEP-10	filter redundant pairs of genes
STEP-11	calculate Pearson Correlation and p-value

Pearson Correlation

Listing 1: Pearson Wrapper Class

```
1  import org.apache.commons.math3.distribution.TDistribution;
2  import org.apache.commons.math3.stat.correlation.PearsonsCorrelation;
3  public class Pearson {
4
5     final static PearsonsCorrelation PC = new PearsonsCorrelation();
6
7     public static double getCorrelation(double[] X, double[] Y) {
8          return PC.correlation(X, Y);
9     }
10
11    public static double getPvalue(final double corr, final double n) {
12       double t = Math.abs(corr * Math.sqrt( (n-2.0) / (1.0 - (corr * corr)) ));
13       TDistribution tdist = new TDistribution(n-2);
14       return (2* (1.0 - tdist.cumulativeProbability(t)));
15    }
16 }
```

Spearman Correlation

Listing 2: Spearman Wrapper Class

```
1  import org.apache.commons.math3.distribution.TDistribution;
2  import org.apache.commons.math3.stat.correlation.SpearmansCorrelation;
3  public class Spearman {
4
5      final static SpearmansCorrelation SC = new SpearmansCorrelation();
6
7      public static double getCorrelation(double[] X, double[] Y) {
8          return SC.correlation(X, Y);
9      }
10
11     public static double getPvalue(double corr, double n) {
12        double t = Math.abs(corr * Math.sqrt( (n-2.0) / (1.0 - (corr * corr)) ))
13        TDistribution tdist = new TDistribution(n-2);
14        return (2.0 * (1.0 - tdist.cumulativeProbability(t)));
15     }
16 }
```

STEP-1: handle input parameters

This step reads 2 inputs:

- The first parameter is a reference value, which can be any of
  - r1: normal
  - r2: disease
  - r3: paired
  - r4: none
- The second parameter is an HDFS file, which contains the list of all biomarker files (persisted as HDFS files) required for Pearson correlation.

STEP-1: handle input parameters

Listing 3: STEP-1: handle input parameters

```
1 // STEP-1: handle input parameters
2 if (args.length != 2) {
3     handleError(args);
4 }
5 final String reference = args[0]; // {"r1", "r2", "r3", "r4"}
6 final String biomarkersFileName = args[1];
```

STEP-2: create a Spark context object

```
1  public static JavaSparkContext createJavaSparkContext(boolean useYARN) {
2    JavaSparkContext context;
3    if (useYARN) {
4        context = new JavaSparkContext("yarn-cluster", "MyAnalysis"); // YARN
5    }
6    else {
7        context = new JavaSparkContext();  // Spark cluster
8    }
9    // inject efficiency
10   SparkConf sparkConf = context.getConf();
11   sparkConf.set("spark.kryoserializer.buffer.mb","32");
12   sparkConf.set("spark.shuffle.file.buffer.kb","64");
13   // set a fast serializer
14   sparkConf.set("spark.serializer",
15             "org.apache.spark.serializer.KryoSerializer");
16   sparkConf.set("spark.kryo.registrator",
17             "org.apache.spark.serializer.KryoRegistrator");
18   return context;
19 }
```

STEP-3: create list of input files/biomarkers

This step reads HDFS path, which contains all biomarker files required for Pearson/Spearman correlation. The `biomarkersFileName` is a text HDFS path, which contains all input biomarker files (one biomarker file per line).

Listing 4: STEP-3: create list of input files/biomarkers

```
1       List<String> list = toListOfString(new Path(biomarkersFileName));
```

STEP-4: broadcast "reference" as global shared object

Since "reference" (values are in {"r1", "r2", "r3", "r4"}) value is used for filtering RDD elements, it should be broadcasted to all cluster nodes. In Spark, to broadcast (as a read only shared object) a data structure, we can use the `Broadcast` class. This is how we can use a `Broadcast` class to broadcast a shared data structure:

- To broadcast a shared data structure of type T

```
JavaSparkContext ctx = <instance-of-JavaSparkContext>;
T t = <create-object-of-type-T>;
final Broadcast<T> broadcastT = ctx.broadcast(t);
```

- To read/access a broadcasted shared data structure of type T

```
T t = broadcastT.value();
```

STEP-4: broadcast "reference" as global shared object

MapReduce/Hadoop, you may broadcast a shared data to map() or reduce() functions by using Hadoop's `Configuration` object. You may use `Configuration.set(...)` to brodcast and use `Configuration.get(...)` to read/access broadcasted objects. Spark's API is much richer than Hadoop's API since you may broadcast any type of data structures.

Listing 5: STEP-4: broadcast reference as global shared object

```
1 // broadcast reference which can be accessed from all cluster nodes
2 final Broadcast<String> REF = ctx.broadcast(reference); // "r2"
```

STEP-5: read all biomarkers from HDFS and create the first RDD

This step reads all biomarker files an create a single `JavaRDD<String>`, which can be partitioned further by the following method:

```
JavaRDD<T> coalesce(int numPartitions)
```

Listing 6: STEP-5: read all biomarkers and create the first RDD

```
1 // STEP-5: read all biomarkers from HDFS and create the first RDD
2 JavaRDD<String> biosets = readBiosets(ctx, list);
3 biosets.saveAsTextFile("/output/1");
```

To debug, a sample output of this step is provided:

```
# hadoop fs -cat /output/1/*
g1,r2,p1,1.86
g4,r1,p1,2.44
```

How to Read Data from HDFS

- NOT very efficient for small files

```
JavaRDD<String> records =
    ctx.textFile(inputPath, numberOfPartitions);
```

- Efficient for Small files

```
import org.apache.hadoop.conf.Configuration;
...
hadoopConf = new Configuration();
JavaPairRDD<Text, DoubleWritable> genesAsHadoopRDD =
 ctx.newAPIHadoopFile(
  inputPath,  // String path,
  CustomCombineFileInputFormat.class, // fClass,
  Text.class,  // kClass,
  DoubleWritable.class, // vClass,
  hadoopConf); //
```

CustomCombineFileInputFormat.class

```
1 public class CustomCombineFileInputFormat
2   extends CombineFileInputFormat<Text, DoubleWritable> {
3   final static long MAX_SPLIT_SIZE = 67108864; // 64MB = 64*1024*1024
4   public CustomCombineFileInputFormat() {
5     super();
6     setMaxSplitSize(MAX_SPLIT_SIZE);
7   }
8   @Override
9   public RecordReader<Text, DoubleWritable>
10    createRecordReader(InputSplit split,
11                       TaskAttemptContext context)
12      throws IOException {
13      return new CombineFileRecordReader<Text, DoubleWritable>(
14           (CombineFileSplit)split,
15           context,
16           CustomRecordReader.class);
17  }
18  @Override
19  protected boolean isSplitable(JobContext context, Path file) {
20    return false;
21  }
22 }
```

CustomRecordReader.class

```
1 public class CustomRecordReader
2   extends RecordReader<Text, DoubleWritable> {
3    // define (K,V)
4    private Text key;
5    private DoubleWritable value;
6
7    // define pos and offsets
8    ...
9    public CustomRecordReader(CombineFileSplit split,
10                             TaskAttemptContext context,
11                             Integer index)
12       throws IOException{
13       ...
14   }
15
16   public Text getCurrentKey()  {...}
17   public DoubleWritable getCurrentValue()  {...}
18   public boolean nextKeyValue()  { ...}
19   ...
20   }
21 }
```

STEP-6: filter biomarkers by reference

To filter elements, we just need to implement a filter() function: return `true` for the records you want to keep and return `false` for the records you want to toss out.

Listing 7: STEP-6: filter biomarkers by reference

```
1  // JavaRDD<T> filter(Function<T,Boolean> f)
2  // Return a new RDD containing only the elements that satisfy a predicate.
3  JavaRDD<String> filtered = biosets.filter(new Function<String,Boolean>() {
4      public Boolean call(String record) {
5          String ref = REF.value();
6          String[] tokens = record.split(",");
7          if (ref.equals(tokens[1]))  {
8              return true; // do return these records
9          }
10         else {
11             return false; // do not retrun these records
12         }
13     }
14 });
15 filtered.saveAsTextFile("/output/2");
```

STEP-6: filter biomarkers by reference

Let reference = "r2"
To debug, a sample output of this step is provided: note that only "r2" references will be present in the output (all other references {r1, r3, r4} are dropped from the resulting RDD).

```
# hadoop fs -cat /output/2/*
g1,r2,p1,1.86
g2,r2,p1,0.74
...
```

STEP-7: create (K, V) pairs of ((Gene-ID), (Patient-ID, Biomarker-Value))

This step implements a map() function, which transforms (note that from this point on "reference" is not needed for subsequent steps.)

```
<GeneID><,><reference><,><PatientID><,><BiomarkerValue>
```

into

```
(K, V) pair
where
      K = GeneID
      V = Tuple2<PatientID, BiomarkerValue>
```

STEP-7: create ((Gene), (Patient-ID, Biomarker-Value))

```
1  // STEP-7: create ((Gene-ID), (Patient-ID, Biomarker-Value)) pairs
2  // PairMapFunction<T, K, V>
3  // T => Tuple2<K, V> = Tuple2<gene, Tuple2<patientID, value>>
4  JavaPairRDD<String,Tuple2<String,Double>> pairs =
5      filtered.mapToPair(new PairFunction<
6      String,                    // T
7      String,                    // K = g1234 (as GeneID)
8      Tuple2<String,Double>      // V = <patientID, value>
9      >() {
10     public Tuple2<String,Tuple2<String,Double>> call(String rec) {
11         String[] tokens = rec.split(",");
12         // tokens[0] = 1234
13         // tokens[1] = 2 (this is a ref in {"1", "2", "3", "4"}
14         // tokens[2] = patientID
15         // tokens[3] = value
16         Tuple2<String,Double> V =
17            new Tuple2<String,Double>(tokens[2], Double.valueOf(tokens[3]));
18         return new Tuple2<String,Tuple2<String,Double>>(tokens[0], V);
19     }
20 });
21  pairs.saveAsTextFile("/output/3");
```

Output of STEP-7

To debug, a sample output of this step is provided:

```
# hadoop fs -cat /output/3/*
(g1,(p1,1.86))
(g2,(p1,0.74))
(g3,(p1,1.24))
(g5,(p1,1.69))
(g6,(p1,0.93))
(g7,(p1,1.44))
(g8,(p1,2.11))
(g1,(p2,2.46))
(g2,(p2,3.24))
...
```

STEP-8: group by gene

This step groups data by GeneID. The result of this grouping is a new RDD as:

```
JavaPairRDD<String, Iterable<Tuple2<String,Double>>>

    where
        K = GeneID
        V = Iterable<Tuple2<PatientID, BiomarkerValue>>
```

STEP-8: group by gene

Listing 8: STEP-8: group by gene

```
1 // STEP-8: group by gene
2 JavaPairRDD<String, Iterable<Tuple2<String,Double>>>
3    grouped = pairs.groupByKey();
4 grouped.saveAsTextFile("/output/4");
5 // grouped = (K, V)
6 //    where
7 //          K = gene
8 //          V = Iterable<Tuple2<patientID,value>>
9 grouped.saveAsTextFile("/output/5");
```

Output of STEP-8

debug, a partial sample output of this step is provided: the output is formatted to fit the page.

```
# hadoop fs -cat /output/5/*
(g1,[(p1,1.86), (p1,1.76), (p1,1.16), (p3,1.06),
     (p1,1.86), (p2,1.46), (p2,1.33), (p2,2.46),
     (p2,2.46), (p2,1.33), (p3,2.61), (p1,2.86),
     (p2,2.06), (p2,1.43)])
(g2,[(p2,3.24), (p2,1.24), (p2,2.0), (p3,1.55),
     (p1,1.74), (p2,3.2), (p2,2.5), (p1,0.74),
     (p1,2.84), (p1,1.33), (p3,1.24), (p3,2.1),
     (p1,2.74), (p2,2.24), (p2,2.0)])
...
```

STEP-9: create Cartesian product of all genes

To perform all-genes-vs-all-genes correlation of all genes, we have to create a Cartesian product of all genes. This is accomplished by `JavaPairRDD.cartesian()` function.

Listing 9: STEP-9: create Cartesian product of all genes

```
1 // STEP-9: create Cartesian product of all genes
2 // <U> JavaPairRDD<T,U> cartesian(JavaRDDLike<U,?> other)
3 // Return the Cartesian product of this RDD and another one,
4 // that is, the RDD of all pairs of elements (a, b)
5 // where a is in this and b is in other.
```

STEP-9: create Cartesian product of all genes

Listing 10: STEP-9: create Cartesian product of all genes

```
1  // STEP-9: create Cartesian product of all genes
2  JavaPairRDD<
3              Tuple2<String, Iterable<Tuple2<String,Double>>>,
4              Tuple2<String, Iterable<Tuple2<String,Double>>>
5             > cart = grouped.cartesian(grouped);
6  cart.saveAsTextFile("/output/6");
7  // cart =
8  //         (g1, g1), (g1, g2),  (g1, g3), (g1, g4)
9  //         (g2, g1), (g2, g2),  (g2, g3), (g2, g4)
10 //         (g3, g1), (g3, g2),  (g3, g3), (g3, g4)
11 //         (g4, g1), (g4, g2),  (g4, g3), (g4, g4)
```

STEP-10: filter redundant pairs of genes

Let g1 and g2 be two genes; since Pearson correlation for (g1, g2) is the same as (g2, g1), therefore, to reduce computation time, we will filter duplicate pairs. We only keep the gene pairs of (g1, g2) if and only if g1 < g2.

Listing 11: STEP-10: filter redundant pairs of genes

```
1  // STEP-10: filter redundant pairs of genes
2  // filter it and keep the ones (Ga, Gb)  if and only if (Ga < Gb).
3  // after filtering, we will have:
4  // filtered2 =
5  //         (g1, g2),  (g1, g3), (g1, g4)
6  //         (g2, g3), (g2, g4)
7  //         (g3, g4)
8  //
9  // JavaRDD<T> filter(Function<T,Boolean> f)
10 // Return a new RDD containing only the elements that satisfy a predicate.
```

STEP-10: filter redundant pairs of genes

```
1  // Keep pairs of (g1, g2) if and only if g1 < g2
2  JavaPairRDD<Tuple2<String, Iterable<Tuple2<String,Double>>>,
3              Tuple2<String, Iterable<Tuple2<String,Double>>>> filtered2 =
4      cart.filter(new Function
5          <Tuple2<Tuple2<String, Iterable<Tuple2<String,Double>>>,
6                  Tuple2<String, Iterable<Tuple2<String,Double>>>
7                 >,
8                 Boolean>() {
9      public Boolean call(
10          Tuple2<Tuple2<String, Iterable<Tuple2<String,Double>>>,
11                 Tuple2<String, Iterable<Tuple2<String,Double>>>> pair) {
12         // pair._1 = Tuple2<String, Iterable<Tuple2<String,Double>>>
13         // pair._2 = Tuple2<String, Iterable<Tuple2<String,Double>>>
14         if (smaller(pair._1._1,  pair._2._1))  {
15             return true; // do return these records
16         }
17         else {
18             return false; // do not return these records
19         }
20     }
21 });
22 filtered2.saveAsTextFile("/output/7");
```

Output of STEP-10

To debug this step, partial output of this step is displayed below. Output is formatted to fit the page.

```
# hadoop fs -cat /output/7/*
...
((g1,[(p2,2.46), (p2,2.46), (p2,1.33), (p3,2.61), (p1,2.86)
      (p2,2.06), (p2,1.43), (p1,1.86), (p1,1.76), (p1,1.16)
      (p3,1.06), (p1,1.86), (p2,1.46), (p2,1.33)]),
 (g2,[(p2,3.24), (p2,1.24), (p2,2.0), (p3,1.55), (p1,1.74)
      (p2,3.2), (p1,0.74), (p1,2.84), (p1,1.33), (p3,1.24)
      (p3,2.1), (p1,2.74), (p2,2.24), (p2,2.0), (p2,2.5)])
)
...
((g3,[...],
 (g4,[...])
)
```

STEP-11: calculate Pearson Correlation and p-value

```
1  // STEP-11: calculate Pearson Correlation and p-value
2  // next iterate through all mappedValues
3  // JavaPairRDD<String, List<Tuple2<String,Double>>> mappedvalues
4  // create (K,V), where
5  //      K = Tuple2<String,String>(g1, g2)
6  //      V = Tuple2<Double,Double>(corr, pvalue)
7  //
8  JavaPairRDD<Tuple2<String,String>,Tuple2<Double,Double>> finalresult =
9              filtered2.mapToPair(new PairFunction<
10                 Tuple2<Tuple2<String,Iterable<Tuple2<String,Double>>>,
11                        Tuple2<String,Iterable<Tuple2<String,Double>>>>, // input
12                 Tuple2<String,String>,                                  // K
13                 Tuple2<Double,Double>                                   // V
14             >() {
15       public Tuple2<Tuple2<String,String>,Tuple2<Double,Double>>
16         call(Tuple2<Tuple2<String,Iterable<Tuple2<String,Double>>>,
17                     Tuple2<String,Iterable<Tuple2<String,Double>>>> t) {
18             //// body of call() method
19       }
20 });
21 finalresult.saveAsTextFile("/output/corr");
```

STEP-11: body of call() method

```
1  public Tuple2<Tuple2<String,String>,Tuple2<Double,Double>>
2     call(Tuple2<Tuple2<String,Iterable<Tuple2<String,Double>>>,
3                 Tuple2<String,Iterable<Tuple2<String,Double>>>> t) {
4     Tuple2<String,Iterable<Tuple2<String,Double>>> g1 = t._1;
5     Tuple2<String,Iterable<Tuple2<String,Double>>> g2 = t._2;
6     Map<String, MutableDouble> g1map = toMap(g1._2);
7     Map<String, MutableDouble> g2map = toMap(g2._2);
8     // now perform a correlation(one, other); make sure we order the
9     // values accordingly by patientID, which may have one or more values
10    Tuple2<double[], double[]> XandY = buildXY(g1map, g2map);
11    double[] x = XandY._1;
12    double[] y = XandY._2;
13    // K = pair of genes
14    Tuple2<String,String> K = new Tuple2<String,String>(g1._1,g2._1);
15    if (x.length < 3) {
16       // not enough data to perform correlation
17       return new Tuple2<Tuple2<String,String>,Tuple2<Double,Double>>
18           (K, new Tuple2<Double,Double>(Double.NaN, Double.NaN));
19    }
20    else {
21       double correlation = Pearson.getCorrelation(x, y); // Pearson
22       double pvalue = Pearson.getPvalue(correlation, (double) x.length );
23       return new Tuple2<Tuple2<String,String>,Tuple2<Double,Double>>
24           (K, new Tuple2<Double,Double>(correlation, pvalue));
25    }
26 }
```

STEP-11: buildXY() Method

Listing 12: STEP-11: buildXY() Method

```
1  Tuple2<double[], double[]> buildXY(Map<String, MutableDouble> g1map,
2                                     Map<String, MutableDouble> g2map) {
3     List<Double> x = new ArrayList<Double>();
4     List<Double> y = new ArrayList<Double>();
5     for (Map.Entry<String, MutableDouble> g1Entry : g1map.entrySet()) {
6         String g1PatientID = g1Entry.getKey();
7         MutableDouble g2MD = g2map.get(g1PatientID);
8         if (g2MD != null) {
9            // both one and other for patientID have values
10           x.add(g1Entry.getValue().avg());
11           y.add(g2MD.avg());
12        }
13    }
14    double[] X = toArray(x);
15    double[] Y = toArray(y);
16    return new Tuple2<double[], double[]>(X, Y);
17 }
```

Output of STEP-11

Note that if two genes do have not enough data to correlate, then for a lack of proper value, we have emitted `Double.NaN`.

```
# hadoop fs -cat /output/corr/part*
((g1,g2),(-0.5600331663273436,0.6215767617117369))
((g1,g3),(NaN,NaN))
((g1,g4),(NaN,NaN))
((g1,g5),(-0.02711004213333685,0.9827390963782845))
((g1,g6),(0.19358340989347553,0.8759779754044315))
((g1,g7),(-0.8164277145788058,0.3919024816433061))
((g1,g8),(0.1671231007563918,0.8931045335800389))
((g1,g9),(0.6066217061857698,0.5850485651167254))
((g2,g3),(NaN,NaN))
((g2,g4),(NaN,NaN))
((g2,g5),(-0.8129831655334596,0.3956841419099777))
((g2,g6),(-0.9212118275674606,0.25440121369269475))
((g2,g7),(0.9356247601371344,0.22967428006843038))
((g2,g8),(-0.9104130933946509,0.271527771868302))
((g2,g9),(-0.9983543136507176,0.036528196595012385))
((g3,g4),(NaN,NaN))
...
((g4,g9),(NaN,NaN))
((g5,g6),(0.9754751741958164,0.1412829282172836))
((g5,g7),(-0.5551020210096935,0.625358421978409))
...
((g7,g8),(-0.7057703774748613,0.5012020519367326))
((g7,g9),(-0.9543282445223156,0.19314608347341866))
((g8,g9),(0.885190414128154,0.30805596846331396))
```

Outline

Moral of Story...

- Understand data requirements
- Select proper platform for implementation: Spark
- Partition your RDDs properly
  - number of cluster nodes
  - number of cores per node
  - amount of RAM
- Avoid unnecessary computations (g1, g2), (g2, g1)
- Use filter() often to remove non-needed RDD elements
- Avoid unnecessary RDD.saveAsTextFile(path)
- Run both on YARN and Spark cluster and compare performance
- Verify your test results against R language

Thank you!

Questions?