Using Numerical Libraries on Spark

Brian Spector

12th August 2015
How to use existing libraries on Spark

- Call algorithm with data in-memory
  - If data is small enough
- Sample
  - Obtain estimates for parameters
- Reformulate the problem
  - Map/Reduce existing functions across workers
Outline

- NAG Introduction
- Numerical Computing
  - Linear Regression
- Linear Regression Example on Spark
  - Timings
  - Problems Encountered
- MLlib Algorithms
Outline

- NAG Introduction
- Numerical Computing
  - Linear Regression
- Linear Regression Example on Spark
  - Timings
  - Problems Encountered
- MLlib Algorithms
The Numerical Algorithms Group

- Founded in 1970 as a co-operative project out of academia in UK
- Support and Maintain Numerical Library ~1700 Mathematical/Statistical Routines
- NAG’s code is embedded in many vendor libraries (e.g. AMD, Intel)
NAG Library Full Contents

- Root Finding
- Summation of Series
- Quadrature
- Ordinary Differential Equations
- Partial Differential Equations
- Numerical Differentiation
- Integral Equations
- Mesh Generation
- Interpolation
- Curve and Surface Fitting
- Optimization
- Approximations of Special Functions
- Dense Linear Algebra
- Sparse Linear Algebra
- Correlation & Regression Analysis
- Multivariate Methods
- Analysis of Variance
- Random Number Generators
- Univariate Estimation
- Nonparametric Statistics
- Smoothing in Statistics
- Contingency Table Analysis
- Contingency Table Analysis
- Survival Analysis
- Time Series Analysis
- Operations Research
Outline

- NAG Introduction
- Numerical Computing
  - Linear Regression
- Linear Regression Example on Spark
  - Timings
  - Problems Encountered
- MLlib Algorithms
Efficient Algorithms

- At NAG we have 40 years of algorithms
- Linear Algebra, Regression, Optimization
- Data is all in-memory and call a compiled library

```c
void nag_1d_spline_interpolant (Integer m, const double x[],
                             const double y[], Nag_Spline *spline, NagError *fail);
```

- Algorithms take advantage of
  - AVX/AVX2
  - Multi-core
  - LAPACK/BLAS
  - MKL
Efficient Algorithms

- Modern programming languages and compilers (hopefully) take care of some efficiencies for us
  - Python
    - numpy/scipy
  - Compiler flags (-O2)
- Users worry less about efficient computing and more about solving their problem.
...But what happens when it comes to Big Data?

\[ x = (\text{double}*) \text{malloc}(10,000,000 \times 10,000 \times \text{sizeof(double)}) \]

- Past efficient algorithms break down
  - Need different ways of solving same problem

- How do we use our existing functions (libraries) on Spark?
  - We must reformulate the problem
  - An example... (Warning Mathematics Approaching)
Linear Regression Example

- **General Problem**
  - Given a set of input measurements $x_1 x_2 \ldots x_p$ and an outcome measurement $y$, fit a linear model
    \[ y = X\hat{B} + \varepsilon \]

\[
y = \begin{bmatrix} y_1 \\ \vdots \\ y_p \end{bmatrix} \quad X = \begin{bmatrix} x_{1,1} & \cdots & x_{1,p} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,p} \end{bmatrix}
\]

- Three ways of solving the same problem
Linear Regression Example

- Solution 1 (Optimal Solution)

\[ X = \begin{bmatrix}
  x_{1,1} & \cdots & x_{1,p} \\
  \vdots & \ddots & \vdots \\
  x_{n,1} & \cdots & x_{n,p}
\end{bmatrix} = QR^* \text{ where } R^* = \begin{pmatrix}
  R \\
  0
\end{pmatrix} \]

\( R \) is \( p \) by \( p \) Upper Triangular, \( Q \) orthogonal

\( R\hat{B} = c_1 \) where \( c = QTy \) ...

... lots of linear algebra,

but we have an algorithm!
Linear Regression Example

- Solution 2 (Normal Equations)

\[(X^TX)\hat{B} = X^T \ y\]

- If we 3 independent variables

\[
X = \begin{bmatrix}
1 & 2 & 3 \\
\vdots & \vdots & \vdots \\
4 & 5 & 6
\end{bmatrix},
\ y = \begin{bmatrix}
7 \\
\vdots \\
8
\end{bmatrix}
\]
Linear Regression Example

\[(X^TX) = \begin{bmatrix} 1 & \cdots & 4 \\ 2 & \cdots & 5 \\ 3 & \cdots & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ \vdots & \vdots & \vdots \\ 4 & 5 & 6 \end{bmatrix} \]

\[= \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} \\ Z_{3,1} & Z_{3,2} & Z_{3,3} \end{bmatrix} \]

\[\Rightarrow \hat{B} = (X^TX)^{-1}X^Ty\]

- This computation can be mapped out to slave nodes
Solution 3 (Optimization)

\[ \min \sum (y - \hat{X}\hat{B})^2 \]

Iterative over the data
Final answer is an approximate to the solution
Can add constraints to variables (LASSO)
MLlib Algorithms

- Machine Learning/Optimization
- MLlib uses solution #3 for many applications
  - Classification
  - Regression
  - Gaussian Mixture Model
  - Kmeans
- Slave nodes are used for callbacks to access data
  - Map, Reduce, Repeat
MLlib Algorithms

Master/Libraries

Slaves

Numerical Excellence
Outline

- NAG Introduction
- Numerical Computing
  - Linear Regression
- Linear Regression Example on Spark
  - Timings
  - Problems Encountered
- MLlib Algorithms
Example on Spark

- Use Normal Equations (Solution 2) to compute Linear Regression on large dataset.
- NAG Libraries were distributed across Master/Slave Nodes
- Steps:
  1. Read in data
  2. Call routine to compute sum-of-squares matrix on partitions (chunks)
     - Default partition size is 64mb
  3. Call routine to aggregate SSQ matrix together
  4. Call routine to compute optimal coefficients
Linear Regression Example

Master

Libraries Call

Libraries Call

Libraries Call

Libraries Call

Libraries Call
# Linear Regression Example Data

<table>
<thead>
<tr>
<th>Label</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
<th>X4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>2.22</td>
<td>1</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>8.11</td>
<td>0</td>
</tr>
<tr>
<td>68050.9</td>
<td>42.3</td>
<td>12.1</td>
<td>2.32</td>
<td>1</td>
</tr>
<tr>
<td>86565.0</td>
<td>47.3</td>
<td>19.5</td>
<td>7.19</td>
<td>2</td>
</tr>
<tr>
<td>65151.5</td>
<td>47.3</td>
<td>7.4</td>
<td>1.68</td>
<td>0</td>
</tr>
<tr>
<td>78564.1</td>
<td>53.2</td>
<td>11.4</td>
<td>1.74</td>
<td>1</td>
</tr>
<tr>
<td>56556.7</td>
<td>34.9</td>
<td>10.7</td>
<td>6.84</td>
<td>0</td>
</tr>
</tbody>
</table>
# The Data (in Spark)

## Master

<table>
<thead>
<tr>
<th>Label</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
<th>X4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>2.22</td>
<td>1</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>8.11</td>
<td>0</td>
</tr>
<tr>
<td>68050.9</td>
<td>42.3</td>
<td>12.1</td>
<td>2.32</td>
<td>1</td>
</tr>
<tr>
<td>86565.0</td>
<td>47.3</td>
<td>19.5</td>
<td>7.19</td>
<td>2</td>
</tr>
<tr>
<td>65151.5</td>
<td>47.3</td>
<td>7.4</td>
<td>1.68</td>
<td>0</td>
</tr>
<tr>
<td>78564.1</td>
<td>53.2</td>
<td>11.4</td>
<td>1.74</td>
<td>1</td>
</tr>
<tr>
<td>56556.7</td>
<td>34.9</td>
<td>10.7</td>
<td>6.84</td>
<td>0</td>
</tr>
</tbody>
</table>

```python
rdd.parallelize(data).cache()
```

## Slave 1

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>86565.0</td>
<td>47.3</td>
<td>19.5</td>
<td>7.19</td>
<td>2</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>8.11</td>
<td>0</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>2.22</td>
<td>1</td>
</tr>
</tbody>
</table>

## Slave 2

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>65151.5</td>
<td>47.3</td>
<td>7.4</td>
<td>1.68</td>
<td>0</td>
</tr>
<tr>
<td>68050.9</td>
<td>42.3</td>
<td>12.1</td>
<td>2.32</td>
<td>1</td>
</tr>
<tr>
<td>78564.1</td>
<td>53.2</td>
<td>11.4</td>
<td>1.74</td>
<td>1</td>
</tr>
</tbody>
</table>
### The Data (in Spark)

#### Slave 1

<table>
<thead>
<tr>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
<th>Value 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>86565.0</td>
<td>47.3</td>
<td>19.5</td>
<td>7.19</td>
<td>2</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>8.11</td>
<td>0</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>2.22</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Slave 2

<table>
<thead>
<tr>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
<th>Value 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>65151.5</td>
<td>47.3</td>
<td>7.4</td>
<td>1.68</td>
<td>0</td>
</tr>
<tr>
<td>68050.9</td>
<td>42.3</td>
<td>12.1</td>
<td>2.32</td>
<td>1</td>
</tr>
<tr>
<td>78564.1</td>
<td>53.2</td>
<td>11.4</td>
<td>1.74</td>
<td>1</td>
</tr>
</tbody>
</table>

LIBRARY_CALL(data)
The Data (in Spark)

Slave 1

<table>
<thead>
<tr>
<th>86565.0</th>
<th>47.3</th>
<th>19.5</th>
<th>7.19</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>8.11</td>
<td>0</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>2.22</td>
<td>1</td>
</tr>
</tbody>
</table>

Slave 2

<table>
<thead>
<tr>
<th>65151.5</th>
<th>47.3</th>
<th>7.4</th>
<th>1.68</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>68050.9</td>
<td>42.3</td>
<td>12.1</td>
<td>2.32</td>
<td>1</td>
</tr>
<tr>
<td>78564.1</td>
<td>53.2</td>
<td>11.4</td>
<td>1.74</td>
<td>1</td>
</tr>
</tbody>
</table>

Slave 1

<table>
<thead>
<tr>
<th>86565.0</th>
<th>47.3</th>
<th>19.5</th>
<th>7.19</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>8.11</td>
<td>0</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>2.22</td>
<td>1</td>
</tr>
</tbody>
</table>

Slave 2

<table>
<thead>
<tr>
<th>65151.5</th>
<th>47.3</th>
<th>7.4</th>
<th>1.68</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>68050.9</td>
<td>42.3</td>
<td>12.1</td>
<td>2.32</td>
<td>1</td>
</tr>
<tr>
<td>78564.1</td>
<td>53.2</td>
<td>11.4</td>
<td>1.74</td>
<td>1</td>
</tr>
</tbody>
</table>
### Spark Java Transformations:

- **map** *(Function<T,R> f)* - Return a new RDD by applying a function to all elements of this RDD.

- **flatMap** *(FlatMapFunction<T,U> f)* - Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

- **collectPartitions** *(int[] partitionIds)* - Return an array that contains all of the elements in a specific partition of this RDD.

**Getting Data into Contiguous Blocks**

<table>
<thead>
<tr>
<th>Slave 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>65151.5</td>
</tr>
<tr>
<td>68050.9</td>
</tr>
<tr>
<td>78564.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slave 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>65151.5</td>
</tr>
<tr>
<td>68050.9</td>
</tr>
<tr>
<td>78564.1</td>
</tr>
</tbody>
</table>
Getting Data into Contiguous Blocks

- Spark Java Transformations:
  - `foreachPartition(VoidFunction<java.util.Iterator<T>> f)` - Applies a function f to each partition of this RDD.
  - `mapPartitions(FlatMapFunction<java.util.Iterator<T>,U> f)` - Return a new RDD by applying a function to each partition of this RDD.
The Data (in Spark)

Slave 1

<table>
<thead>
<tr>
<th></th>
<th>86565.0</th>
<th>47.3</th>
<th>19.5</th>
<th>7.19</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave 1</td>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>8.11</td>
<td>0</td>
</tr>
<tr>
<td>Slave 1</td>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>2.22</td>
<td>1</td>
</tr>
</tbody>
</table>

Slave 2

<table>
<thead>
<tr>
<th></th>
<th>65151.5</th>
<th>47.3</th>
<th>7.4</th>
<th>1.68</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave 2</td>
<td>68050.9</td>
<td>42.3</td>
<td>12.1</td>
<td>2.32</td>
<td>1</td>
</tr>
<tr>
<td>Slave 2</td>
<td>78564.1</td>
<td>53.2</td>
<td>11.4</td>
<td>1.74</td>
<td>1</td>
</tr>
</tbody>
</table>

mapPartitions

Slave 1

<table>
<thead>
<tr>
<th></th>
<th>86565.0</th>
<th>47.3</th>
<th>19.5</th>
<th>7.19</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave 1</td>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>8.11</td>
<td>0</td>
</tr>
<tr>
<td>Slave 1</td>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>2.22</td>
<td>1</td>
</tr>
</tbody>
</table>

Slave 2

<table>
<thead>
<tr>
<th></th>
<th>65151.5</th>
<th>47.3</th>
<th>7.4</th>
<th>1.68</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave 2</td>
<td>68050.9</td>
<td>42.3</td>
<td>12.1</td>
<td>2.32</td>
<td>1</td>
</tr>
<tr>
<td>Slave 2</td>
<td>78564.1</td>
<td>53.2</td>
<td>11.4</td>
<td>1.74</td>
<td>1</td>
</tr>
</tbody>
</table>
public void LabeledPointCorrelation(JavaRDD<LabeledPoint> datapoints) throws Exception {
    if(dataSize * numVars / datapoints.partitions().size() > 10000000) {
        throw new NAGSparkException("Partitioned Data Further To Use NAG Routine");
    }
    DataClass finalpt = datapoints.mapPartitions(new ComputeSSQ()).reduce(new CombineData());
    ....
    G02BW g02bw = new G02BW();
    g02bw.eval(numVars, finalpt.ssq, ifail);
    if(g02bw.getIFAIL() > 0) {
        System.out.println("Error with NAG (g02bw) IFAIL = " + g02bw.getIFAIL());
        throw new NAGSparkException("Error from g02bw!!");
    }
}
static class ComputeSSQ implements FlatMapFunction<Iterator<LabeledPoint>, DataClass> {
    @Override
    public Iterable<DataClass> call(Iterator<LabeledPoint> iter) throws Exception {
        List<LabeledPoint> mypoints = new ArrayList<LabeledPoint>();
        while(iter.hasNext()) { mypoints.add(iter.next()); }
        int length = mypoints.size();
        int numvars = mypoints.get(0).features().size() + 1;
        double[] x = new double[length*numvars];
        for(int i=0; i<length; i++) {
            for(int j=0; j<numvars-1; j++) {
                x[(j) * length + i] = mypoints.get(i).features().apply(j);
            }
            x[(numvars-1) * length + i] = mypoints.get(i).label();
        }
        G02BU g02bu = new G02BU();
        g02bu.eval("M", "U", length, numvars, x, length, x, 0.0, means, ssq, ifail);
        return Arrays.asList(new DataClass(length,means,ssq));
    }
}
static class CombineData implements Function2<DataClass, DataClass, DataClass> {
    @Override
    public DataClass call(DataClass data1, DataClass data2) throws Exception {
        G02BZ g02bz = new G02BZ();
        g02bz.eval("M", data1.means.length, data1.length, data1.means, data1.ssq, data2.length, data2.means, data2.ssq, IFAIL);
        data1.length = (g02bz.getXSW());
        return data1;
    }
}
The Data (in Spark)

<table>
<thead>
<tr>
<th>Slave 1</th>
<th>Slave 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>86565.0</td>
<td>65151.5</td>
</tr>
<tr>
<td></td>
<td>47.3</td>
</tr>
<tr>
<td></td>
<td>19.5</td>
</tr>
<tr>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
</tr>
<tr>
<td></td>
<td>8.11</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Master</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slave 1</th>
<th>Slave 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>86565.0</td>
<td>65151.5</td>
</tr>
<tr>
<td></td>
<td>47.3</td>
</tr>
<tr>
<td></td>
<td>19.5</td>
</tr>
<tr>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
</tr>
<tr>
<td></td>
<td>8.11</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Master</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slave 1</th>
<th>Slave 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>86565.0</td>
<td>65151.5</td>
</tr>
<tr>
<td></td>
<td>47.3</td>
</tr>
<tr>
<td></td>
<td>19.5</td>
</tr>
<tr>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
</tr>
<tr>
<td></td>
<td>8.11</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Master</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Linear Regression Example

- **Test 1**
  - Data ranges in size from 2 GB – 64 GB on Amazon EC2 Cluster
  - Used an 8 slave xlarge cluster (16 GB RAM)

- **Test 2**
  - Varied the number of slave nodes from 2 - 16
  - Used 16 GB of data to see how algorithms scale
Cluster Utilization
Results of Linear Regression

![Graph showing the relationship between runtime and size of input data. The y-axis represents time in seconds on a log scale, while the x-axis represents size of input data in GB on a log scale. There is a clear linear relationship indicating that as the size of input data increases, the runtime also increases.](image-url)
Results of Scaling

Scaling of NAG Linear Regression Algorithm

Runtime (Seconds) (Log Scale)

Number of Slaves (Log Scale)

NAG Time

Numerical Excellence
NAG vs. MLlib

NAG and MLlib Logistic Regression

Time (Seconds)

Number Of Points (In Millions)

NAG Time

MLLib
Problems Encountered (1)

- **Java Spark Documentation**
  - `return Arrays.asList(new DataClass(length,means,ssq));`

- **Distributing libraries**
  - `spark-ec2/copy-dir`

- **Setting environment variables**
  - `LD_LIBRARY_PATH` - Needs to be set as you submit the job
    
    ```bash
    $ ./spark-submit --jars NAGJava.jar --conf "spark.executor.extraLibraryPath= \${Path-to-Libraries-on-Worker-Nodes}" simpleNAGExample.jar
    ```
  - `NAG_KUSARI_FILE` (NAG license file)
    - Can be set in code via `sc.setExecutorEnv`
Problems Encountered (2)

- The data problem
  - Data locality
  - Inverse of singular matrix

<table>
<thead>
<tr>
<th>Label</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
<th>X4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>2.22</td>
<td>1</td>
</tr>
<tr>
<td>0.0</td>
<td>18.0</td>
<td>0.0</td>
<td>8.11</td>
<td>0</td>
</tr>
<tr>
<td>68050.9</td>
<td>42.3</td>
<td>12.1</td>
<td>2.32</td>
<td>1</td>
</tr>
<tr>
<td>86565.0</td>
<td>47.3</td>
<td>19.5</td>
<td>7.19</td>
<td>2</td>
</tr>
<tr>
<td>65151.5</td>
<td>47.3</td>
<td>7.4</td>
<td>1.68</td>
<td>0</td>
</tr>
<tr>
<td>78564.1</td>
<td>53.2</td>
<td>11.4</td>
<td>1.74</td>
<td>1</td>
</tr>
<tr>
<td>56556.7</td>
<td>34.9</td>
<td>10.7</td>
<td>6.84</td>
<td>0</td>
</tr>
</tbody>
</table>
Problems Encountered (3)

- **Debugging slave nodes**
  - Throw lots of exceptions!

- **Int sizes with Big Data and a numerical library**
  - 32 vs. 64 bit integers

- **Some algorithms don’t work!**
  - `nag_approx_quantiles_fixed (g01anc)` finds approximate quantiles from a data stream of known size using an out-of-core algorithm.
Outline

- NAG Introduction
- Numerical Computing
  - Linear Regression
- Linear Regression Example on Spark
  - Timings
  - Problems Encountered
- MLlib Algorithms
MLlib Algorithms

- **Basic statistics**
  - summary statistics
  - correlations
  - stratified sampling
  - hypothesis testing
  - random data generation

- **Classification and regression**
  - linear models (SVMs, logistic regression, linear regression)
  - naive Bayes
  - decision trees
  - ensembles of trees (Random Forests and Gradient-Boosted Trees)
  - isotonic regression

- **Collaborative filtering**
  - alternating least squares (ALS)

- **Clustering**
  - k-means
  - Gaussian mixture
  - power iteration clustering (PIC)
  - latent Dirichlet allocation (LDA)
  - streaming k-means

- **Dimensionality reduction**
  - singular value decomposition (SVD)
  - principal component analysis (PCA)

- **Feature extraction and transformation**

- **Frequent pattern mining**
  - FP-growth

- **Optimization (developer)**
  - stochastic gradient descent
  - limited-memory BFGS (L-BFGS)
Other Exciting Happenings

- DataFrames
- H2O Sparking Water
- MLlib pipeline
  - Better optimizers
  - Improved APIs
- SparkR
  - Contains all the components for better numerical computations
    - Use existing R code on Spark!
    - Free/Huge open source community
How to use existing libraries on Spark

- Call algorithm with data in-memory
  - If data is small enough
- Sample
  - Obtain estimates for parameters
- Reformulate the problem
  - Map/Reduce existing functions across slaves
NAG and Apache Spark

- Thanks!
- For more information:
  - Email brian.spector@nag.com
  - Check out The NAG Blog

http://blog.nag.com/2015/02/advanced-analytics-on-apache-spark.html