Errata

Chapter 1 - page 23

One of the solutions is to create a class that computes the counters and the statistics on demand using, once again, the lazy values:

```
class Stats[T <% Double](private values: DVector[T])
  class _Stats(
    var minValue: Double,
    var maxValue: Double,
    var sum: Double,
    var sumSqr: Double)
  val stats = {
     val _stats = new _Stats(Double.MaxValue,Double.MinValue,0.0, 0.0)
     values.foreach(x => {
        if(x < _stats.minValue) _stats.minValue = x
        if(x > _stats.maxValue) _stats.maxValue = x
        _stats.sum += x
        _stats.sumSqr += x*x
     })
  }
...
```

The same approach is used to compute the multivariate normal distribution:

```
  def gauss: DblVector =
    values.map(x => {
      val y = x - mean
      INV_SQRT_2PI*Math.exp(-0.5*y*y /( stdDev* stdDev))/stdDev
    })
```

Chapter 12 - The Master actor - page 16

The receive message handler processes only two types of messages: Start from the client code and Completed from the workers, as shown in the following code:

```
  override def receive = {
    case s: Start => split
    case msg: Completed => {
      if(aggregator.size >= partitioner.numPartitions-1) {
        aggregate
        workers.foreach( context.stop(_) )
      }
      aggregator.append(msg.xt.toArray)
```

```
    }
    case Terminated(sender) => {
      if(aggregator.size >= partitioner.numPartitions-1) {
        context.stop (self)
        context.system.shutdown
      }
    }
  }
```

Chapter 12 - Master with routing - page 18

```
  override def receive = {
    case msg: Start => split
    case msg: Completed => {
      if(aggregator.size >= partitioner.numPartitions-1) {
        aggregate
        context.stop(router)
      }
      aggregator.append(msg.xt.toArray)
    }
    case Terminated(sender) => {
      if( aggregator.size >= partitioner.numPartitions-1) {
        context.stop(self)
        context.system.shutdown
      }
    }
  }
```