

Errata

Chapter 1 Getting started /

Preprocessing the data set – page 30-31

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: Db1Vector =
  values.map(x => {
    val y = x - mean
    INV_SQRT_2PI*Math.exp(-0.5*y*y /( stdDev* stdDev))/stdDev
  })
```

Chapter 7 Sequential Data Models /

The hidden Markov model / Baum-Welch estimator (EM) – Page 224

The maximum likelihood, **maxLikelihood** is computed as part of the constructor to ensure consistent state of the hidden Markov model.

```
var likelihood = frwrdBckwrDLattice
Range(0, state.maxIters) find( _ => {
  lambda.estimate(state, obs) //1
  val _likelihood = frwrdBckwrDLattice //2
  val diff = likelihood - _likelihood //3
  likelihood = _likelihood
```

```

diff < eps //4
}) match {
  case Some(likelihood) => state.lambda.normalize; likelihood

```

Chapter 7 Sequential Data Models /

The hidden Markov model / Viterbi algorithm - Page 227 - 228

```

class ViterbiPath(_lambda: HMMLambda, _state: HMMState, _obs:
Array[Int]) extends HMMInference(_lambda, _state, _obs) {
  val maxDelta = (recurse(0), state.QStar())
  ...
}

```

The recursive method that implements [M14] and [M15] steps is invoked by the constructor:

```

@scala.annotation.tailrec
def recurse(t: Int): Double = {
  if( t == 0) initial //1
  else {
    Range(0, lambda.getN).foreach( updateMaxDelta(t, _) ) //2
    if( t == obs.size-1) { //6
      val idxMaxDelta = Range(0, lambda.getN).map(i => //[M14]
        (i, state.delta(t, i))).maxBy(_._2)
      state.QStar.update(t+1, idxMaxDelta._1) //7 [M15]
      idxMaxDelta._2
    }
    else
      recurse(t+1)
  }
}

```

Once initialized (line 1) for the first observation, the maximum value of delta and its state index are computed for each of the state (line 2) by the method, **updateMaxDelta**. Next, the index of the column of the transition matrix *A* corresponding to the maximum of delta is computed (line 3). The last step is to update the matrix psi (line 4) (with respect to delta (line 5)). Once the step *t* reaches the maximum number of observation labels (line 6), the optimum sequence of states *q** is computed [M15] (line 7). Ancillary methods are omitted.

```

def updateMaxDelta(t: Int, j: Int): Unit = {
  val idxMaxDelta = Range(0, lambda.getN).map(i => //3
    (i, state.delta(t-1, i)*lambda.A(i, j))).maxBy(_._2)
  state.psi += (t, j, idxDelta._1) //4
  state.delta += (t, j, idxDelta._2) //5
}

```

Chapter 10 Genetic Algorithm /

Crossover / Population- Page 347

The **geneticIndices** method (line 5) computes the relative indices of the crossover bit in the chromosomes and genes:

```

def geneticIndices(prob: Double): GeneticIndices = {
  var idx = (prob*chromosomeSize).floor.toInt
  val chIdx = if(idx == chromosomeSize) chromosomeSize-1 else idx
  idx = (prob*geneSize).floor.toInt
  val gIdx = if(idx == geneSize) geneSize-1 else idx
  GeneticIndices(chIdx, gIdx)
}

```

Chapter 12 Scalable Framework /

Akka / The Master actor - Page 420

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 Scalable Framework /

Akka / Master with routing – Page 422

```

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
    }
  }
}

```

}

}

}