Lessons Learned while Implementing a Sparse Logistic Regression Algorithm in Spark

Lorand Dali

@lorserker

SPARK SUMMIT

zalando

#EUds9

You don’t have to implement your own optimization algorithm*

*unless you want to play around and learn a lot of new stuff

Use a representation that is suited for distributed implementation

Logistic regression definition

Feature Vector

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}$$

Weights

Prediction

$$\hat{y} = \frac{1}{1 + e^{-(w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_d \cdot x_d)}}$$

Loss

$$J = -\frac{1}{N} \sum_{i=1}^{N} y^{(i)} log \hat{y}^{(i)} + (1 - y^{(i)}) log(1 - \hat{y}^{(i)})$$

Weight update

$$w_k = w_k - \alpha \frac{\partial J}{\partial w_k}$$

Derivative of loss

$$w = w - \alpha \nabla J$$

Gradient

Logistic regression vectorized

FEATURES

WEIGHTS

PREDICTIONS

EXAMPLES

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_d^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \cdots & x_d^{(N)} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix} = \begin{bmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(N)} \end{bmatrix}, \qquad \hat{y} = \begin{bmatrix} \sigma(z^{(1)}) \\ \sigma(z^{(2)}) \\ \vdots \\ \sigma(z^{(N)}) \end{bmatrix}$$

$$w = w - \alpha \nabla J$$

DOT PRODUCTS

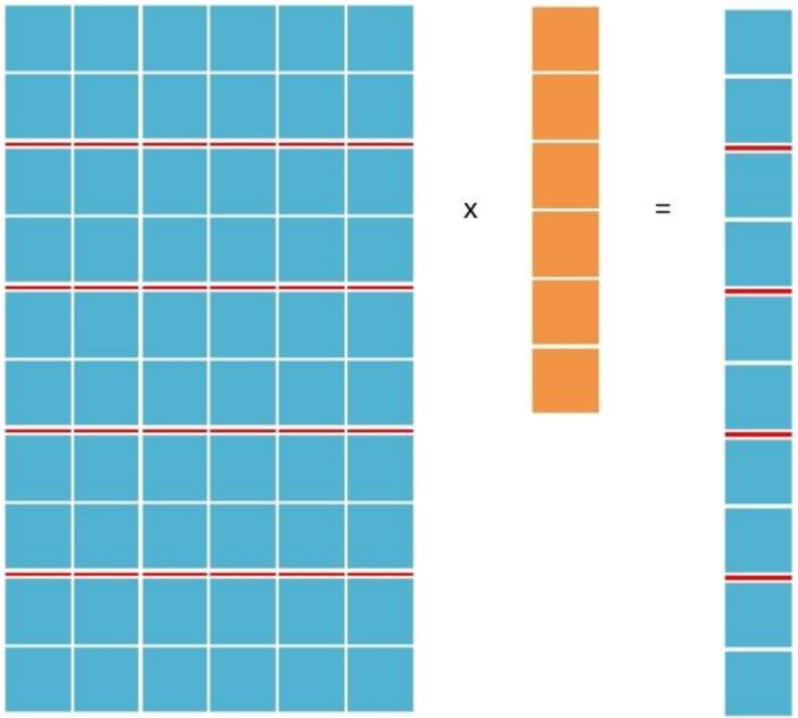
How to compute the gradient vector

$$\nabla J = \frac{1}{N} X^T (\hat{y} - y)$$

$$\nabla J = \frac{1}{N} \cdot \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(N)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(N)} \\ \vdots & \vdots & \ddots & \vdots \\ x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(N)} \end{bmatrix} \cdot \begin{bmatrix} \hat{y}^{(1)} - y^{(1)} \\ \hat{y}^{(2)} - y^{(2)} \\ \vdots \\ \hat{y}^{(N)} - y^{(N)} \end{bmatrix}$$

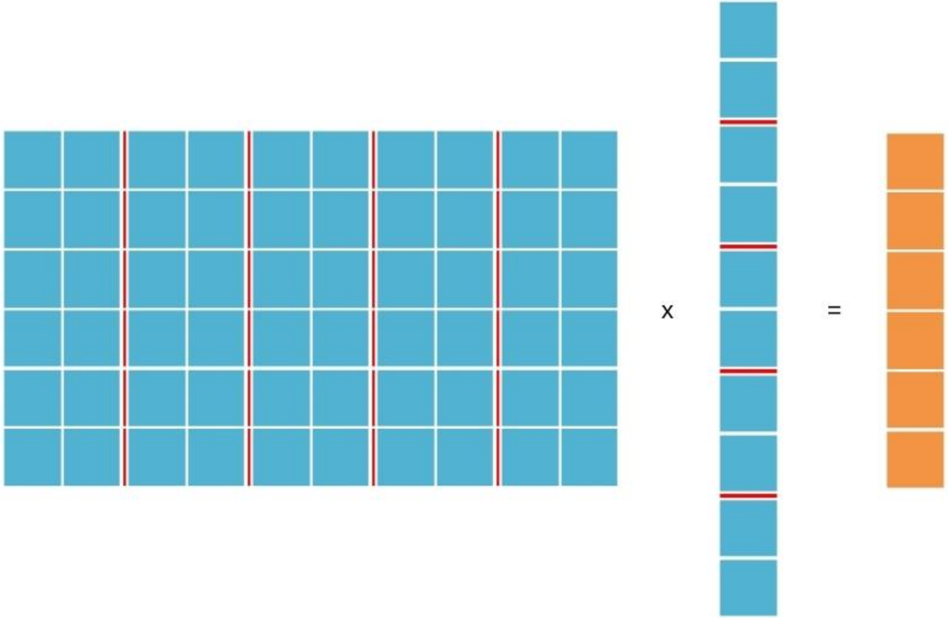
Computing dot products and predictions

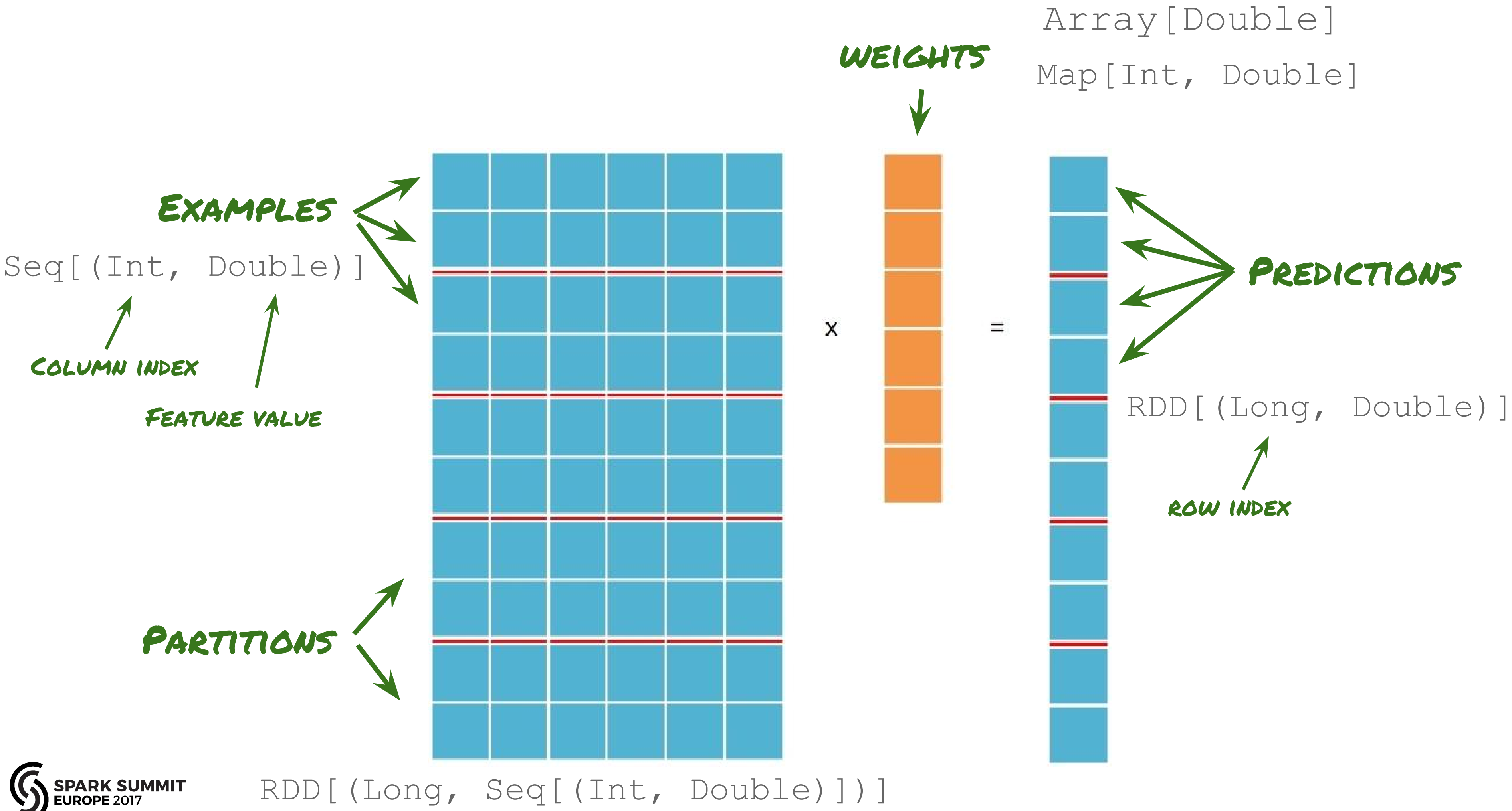
$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_d^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \cdots & x_d^{(N)} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix} = \begin{bmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(N)} \end{bmatrix}$$

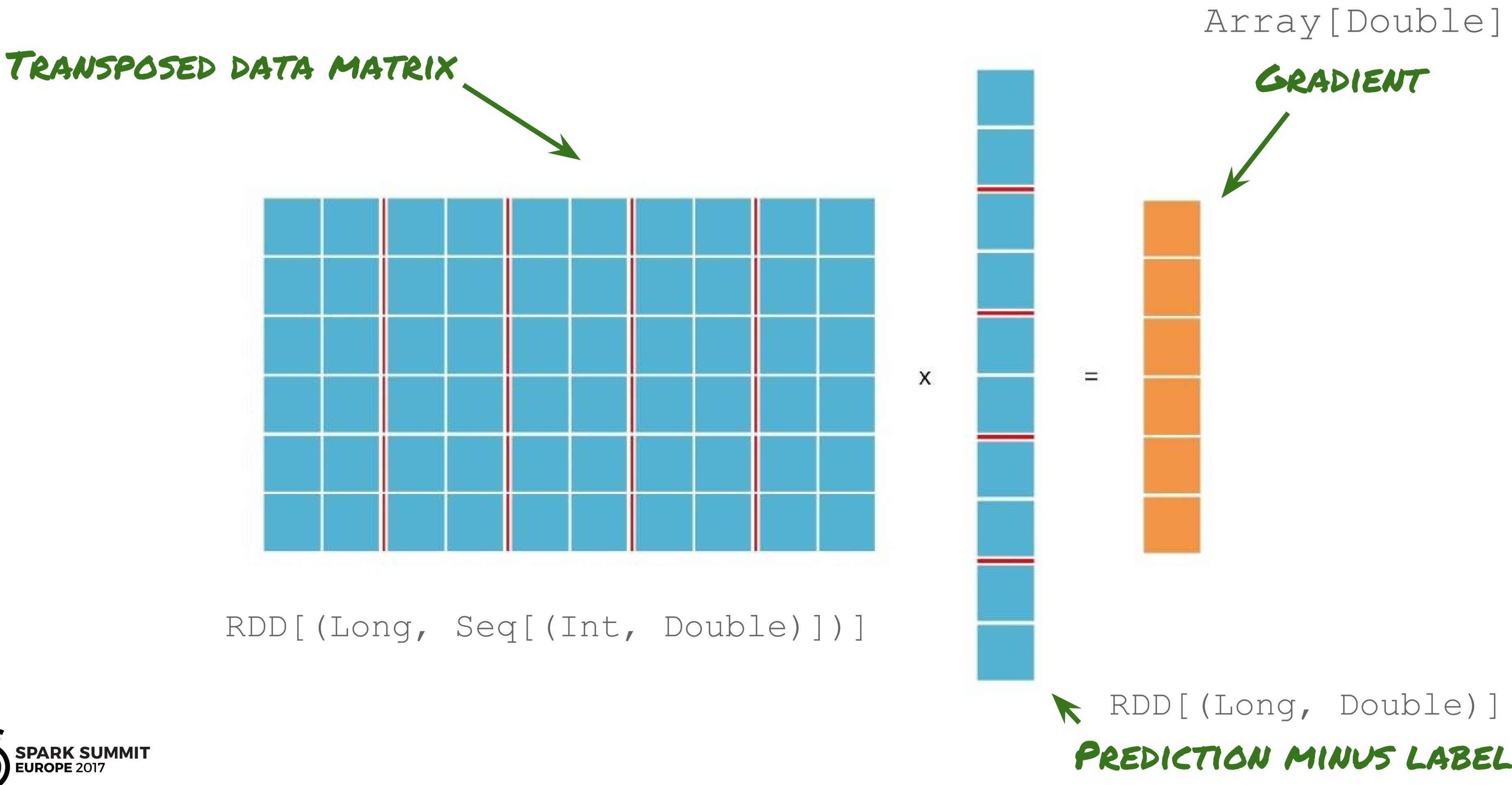


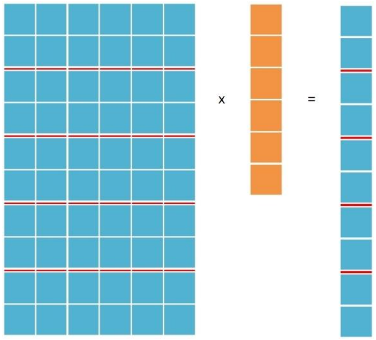
Computing the gradient

$$\begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(N)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(N)} \\ \vdots & \vdots & \ddots & \vdots \\ x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(N)} \end{bmatrix} \cdot \begin{bmatrix} \hat{y}^{(1)} - y^{(1)} \\ \hat{y}^{(2)} - y^{(2)} \\ \vdots \\ \hat{y}^{(N)} - y^{(N)} \end{bmatrix}$$

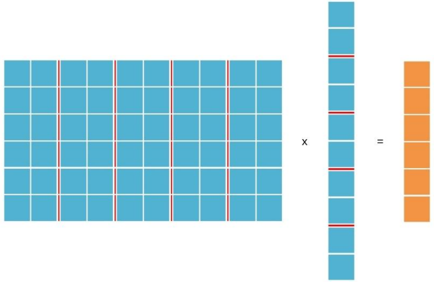



Array[Double]
WEIGHTS
Map[Int, Double]
EXAMPLES
Seq[(Int, Double)]
COLUMN INDEX
FEATURE VALUE
x
=
PREDICTIONS
RDD[(Long, Double)]
ROW INDEX
PARTITIONS
RDD[(Long, Seq[(Int, Double)])]


Array[Double]
TRANSPOSED DATA MATRIX
GRADIENT
x
=
RDD[(Long, Seq[(Int, Double)])]
RDD[(Long, Double)]
PREDICTION MINUS LABEL



```scala
val dotProds: RDD[(Long, Double)] =
  matrix.mapValues(jvals => {
      b + jvals.map{ case (j, x) => x * weights(j) }.sum
  })

val predictions: RDD[(Long, Double)] =
  dotProds.mapValues(z => sigmoid(z))

val deltas: RDD[(Long, Double)] =
  predictions.join(y)
      .mapValues{
          case (predicted, correct) => (predicted - correct)/nRows
      }
```



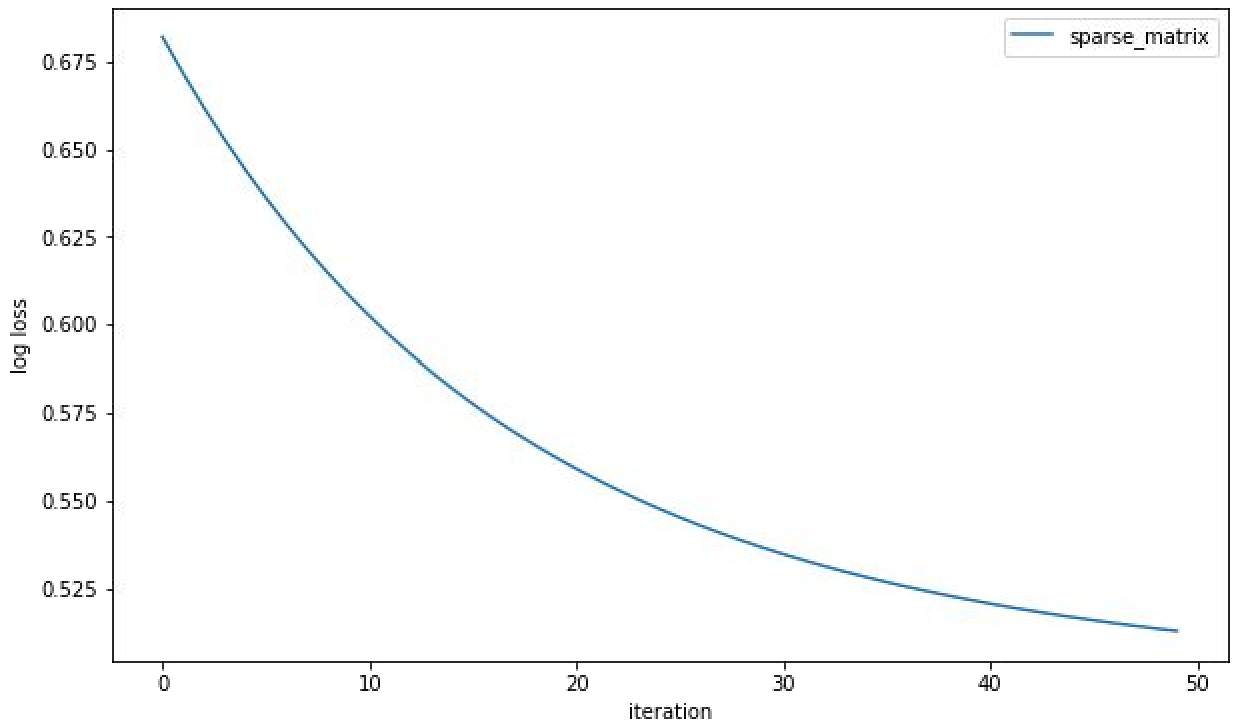
```scala
val gradients: Map[Int, Double] =
  matrix.join(deltas)
    .flatMap{ case (i, (jvals, d)) =>
        jvals.map{ case (j, x) => (j, x * d) } }
    .reduceByKey(_ + _)
    .collect
    .toMap
```

Experimental dataset

- avazu click prediction dataset (sites)
- 20 million examples
- 1 million dimensions
- we just want to try it out

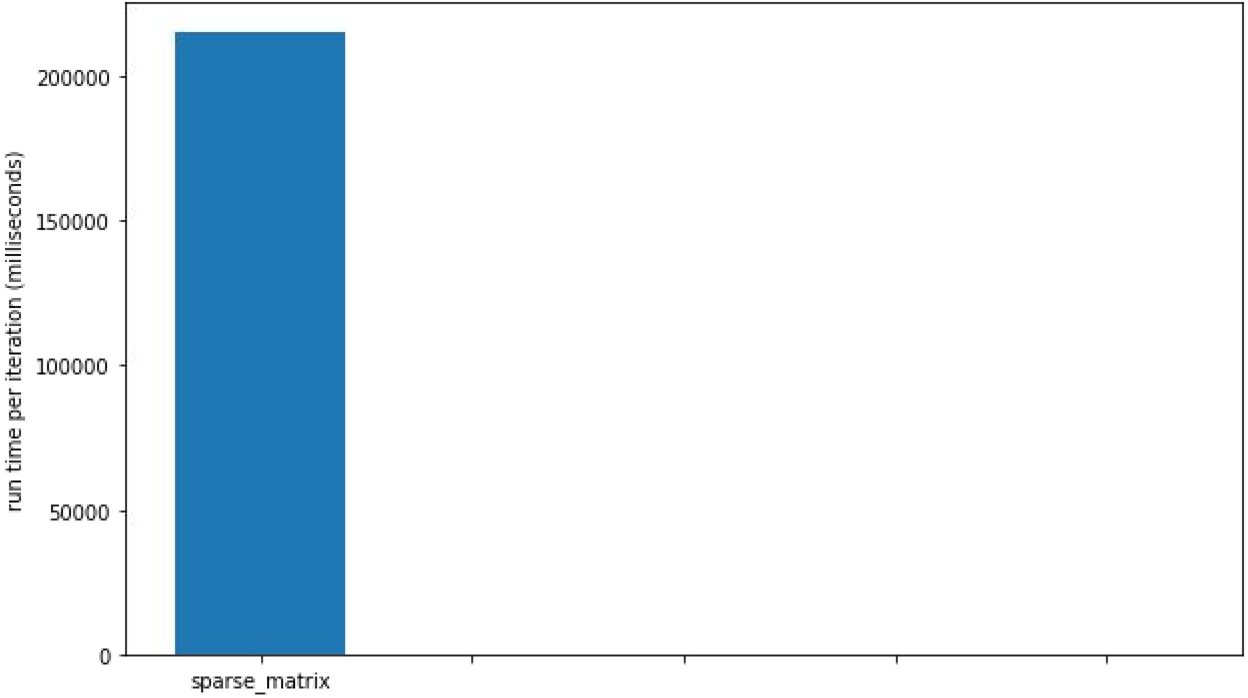
https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#avazu

Learning curve



time per iteration

AWS EMR Cluster
5 nodes of m4.2xlarge

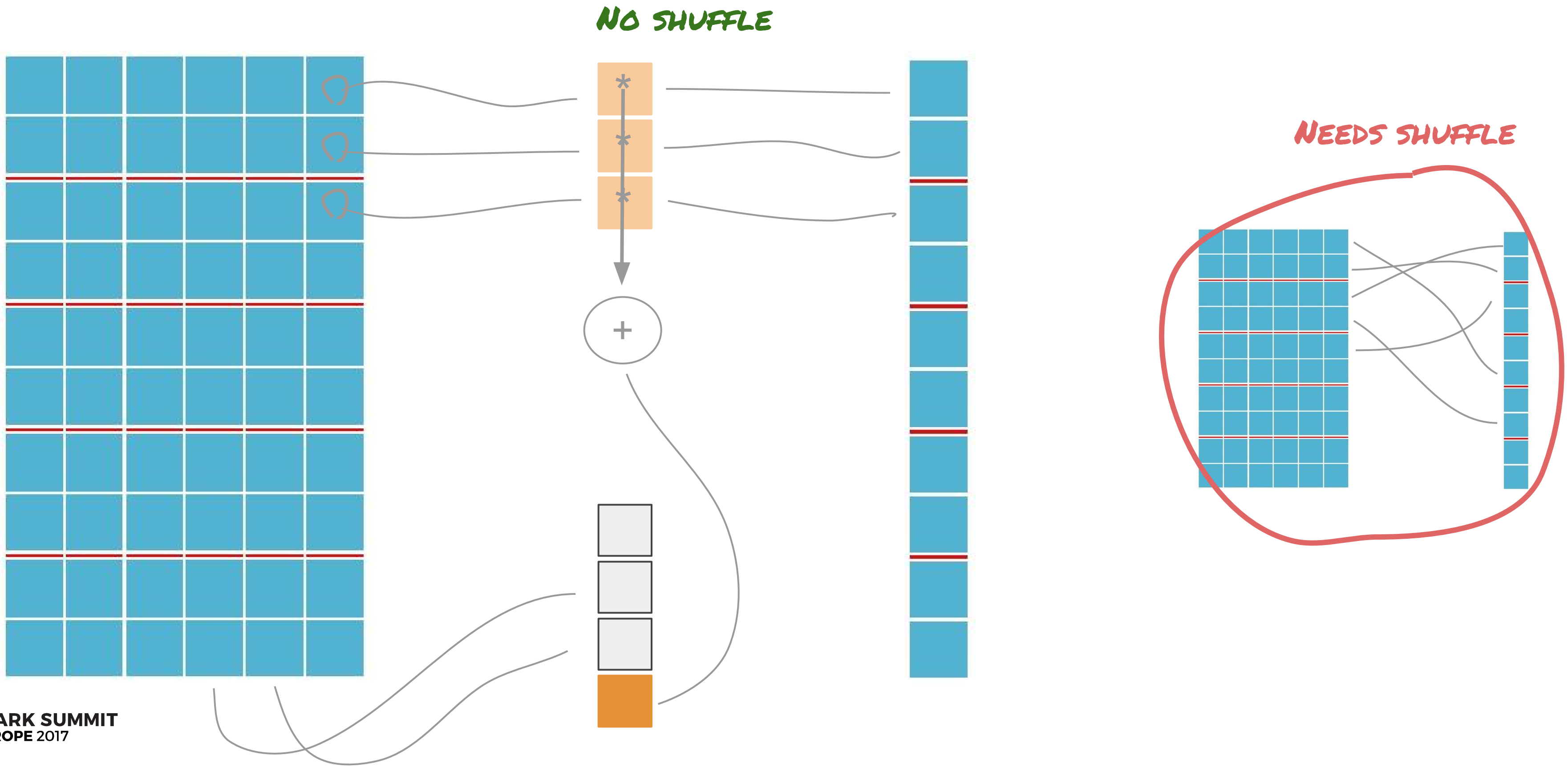


Use a custom partitioner to avoid shuffles

We have two joins in our code

```
val deltas: RDD[(Long, Double)] =
  predictions.join(y)
      .mapValues{
          case (predicted, correct) => (predicted - correct)/nRows
      }

val gradients: Map[Int, Double] =
  matrix.join(deltas)
    .flatMap{ case (i, (jvals, d)) =>
        jvals.map{ case (j, x) => (j, x * d) } }
    .reduceByKey(_ + _)
    .collect
    .toMap
```

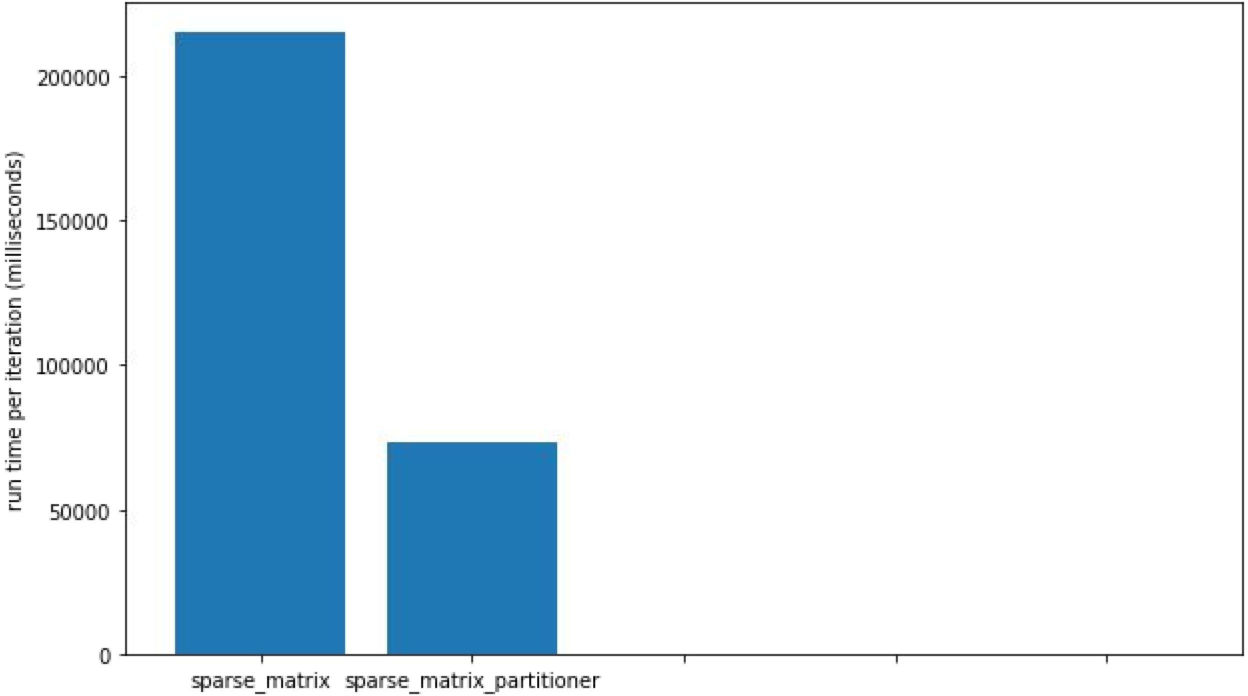
Why is the join expensive



Using a custom partitioner

```scala
val matrix: RDD[(Long, Seq[(Int, Double)])]
val y: RDD[(Long, Double)]

val partitioner = new HashPartitioner(512)

matrix.partitionBy(partitioner).persist()
y.partitionBy(partitioner).persist()
```

time per iteration



Try to avoid joins altogether

Gradient descent without joins

```scala
case class LabeledExample(target: Double, indexes: Array[Int], values: Array[Double])

val data: RDD[LabeledExample]

val gradient = data.flatMap(example => {
    val z = (example.indexes zip example.values).map{ case (i, x) => weights(i) * x}.sum
    val prediction = sigmoid(z)
    (example.indexes zip example.values)
      .map{ case (k, v) => (k, (prediction - example.target) * v / nRows)}
  })
  .reduceByKey(_ + _)
  .collectAsMap()
```

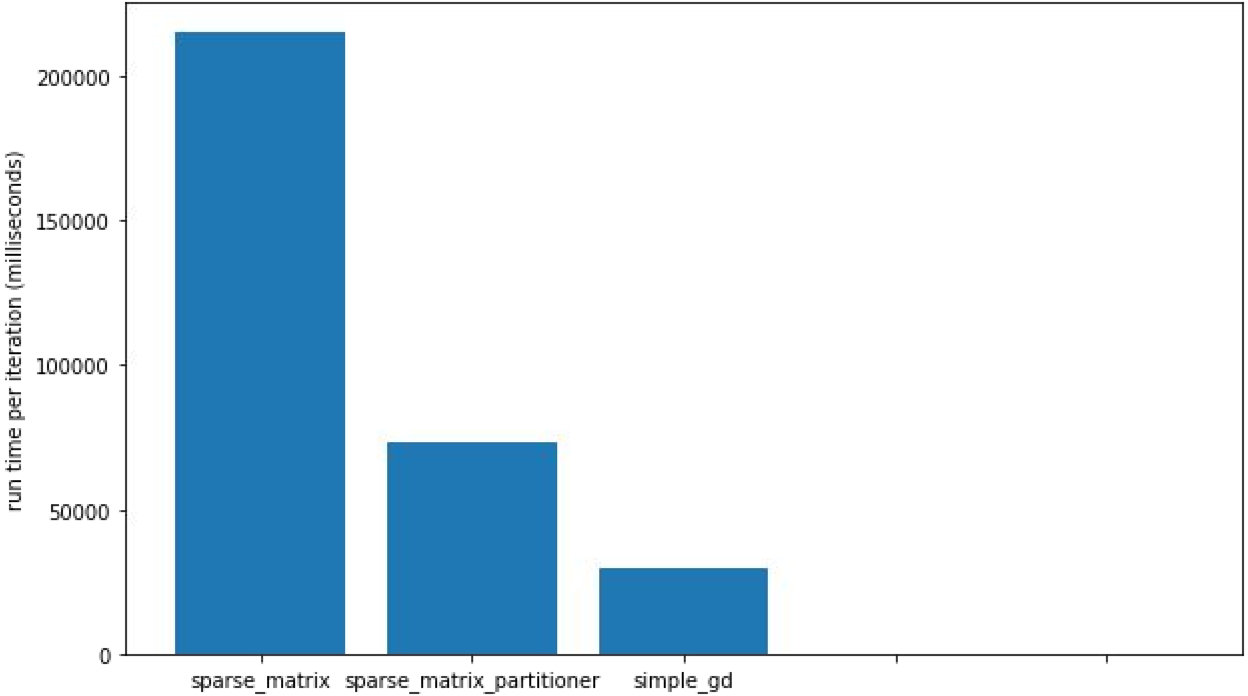
$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_d^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \cdots & x_d^{(N)} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix} = \begin{bmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(N)} \end{bmatrix}$$

DIMENSION

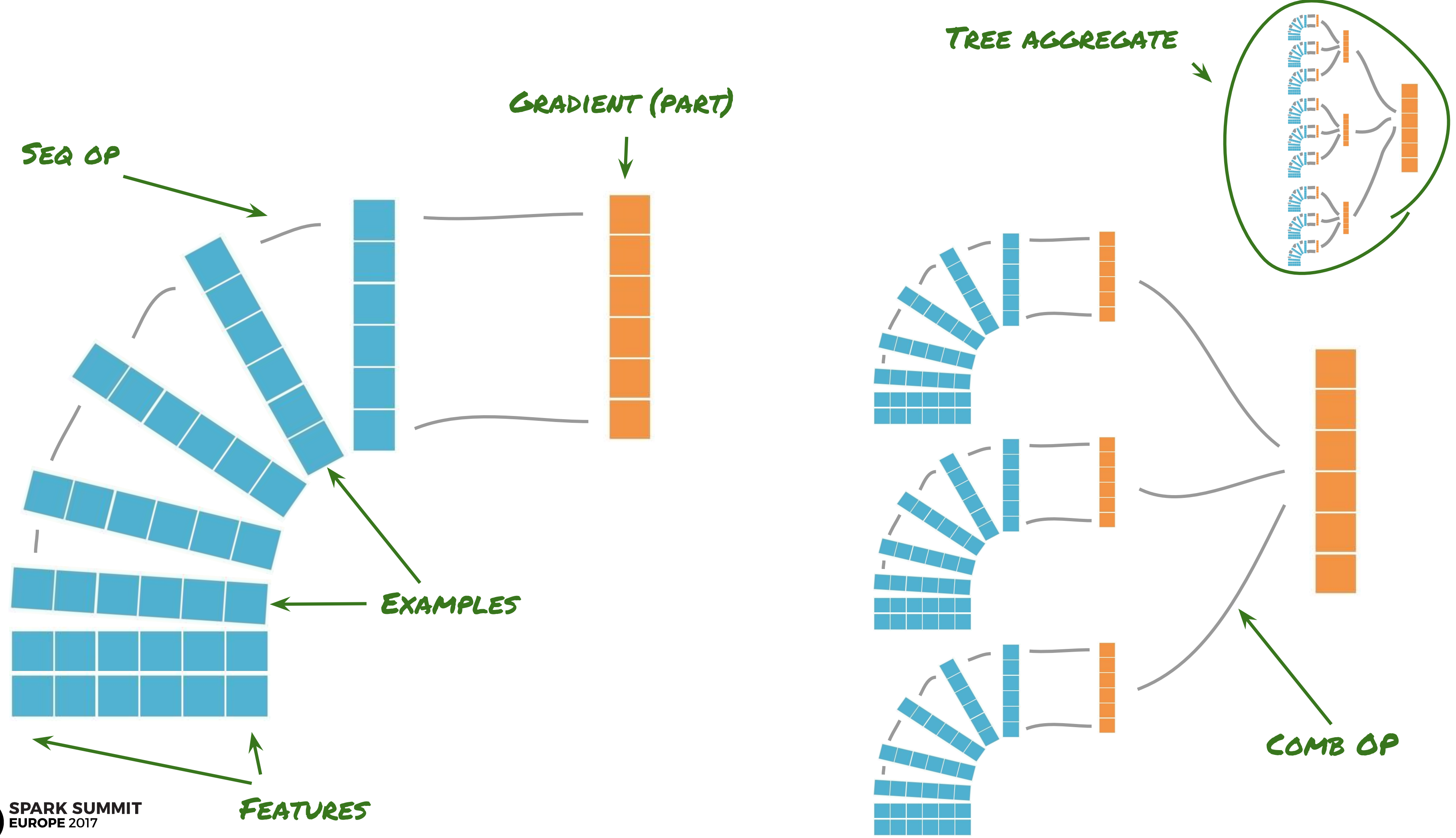
$$\nabla J = \frac{1}{N} X^T (\hat{y} - y)$$

$$\nabla J = \frac{1}{N} \cdot \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(N)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(N)} \\ \vdots & \vdots & \ddots & \vdots \\ x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(N)} \end{bmatrix} \cdot \begin{bmatrix} \hat{y}^{(1)} - y^{(1)} \\ \hat{y}^{(2)} - y^{(2)} \\ \vdots \\ \hat{y}^{(N)} - y^{(N)} \end{bmatrix}$$

time per iteration



Use aggregate and treeAggregate


Tree aggregate
Gradient (part)
Seq op
Examples
Features
Comb op

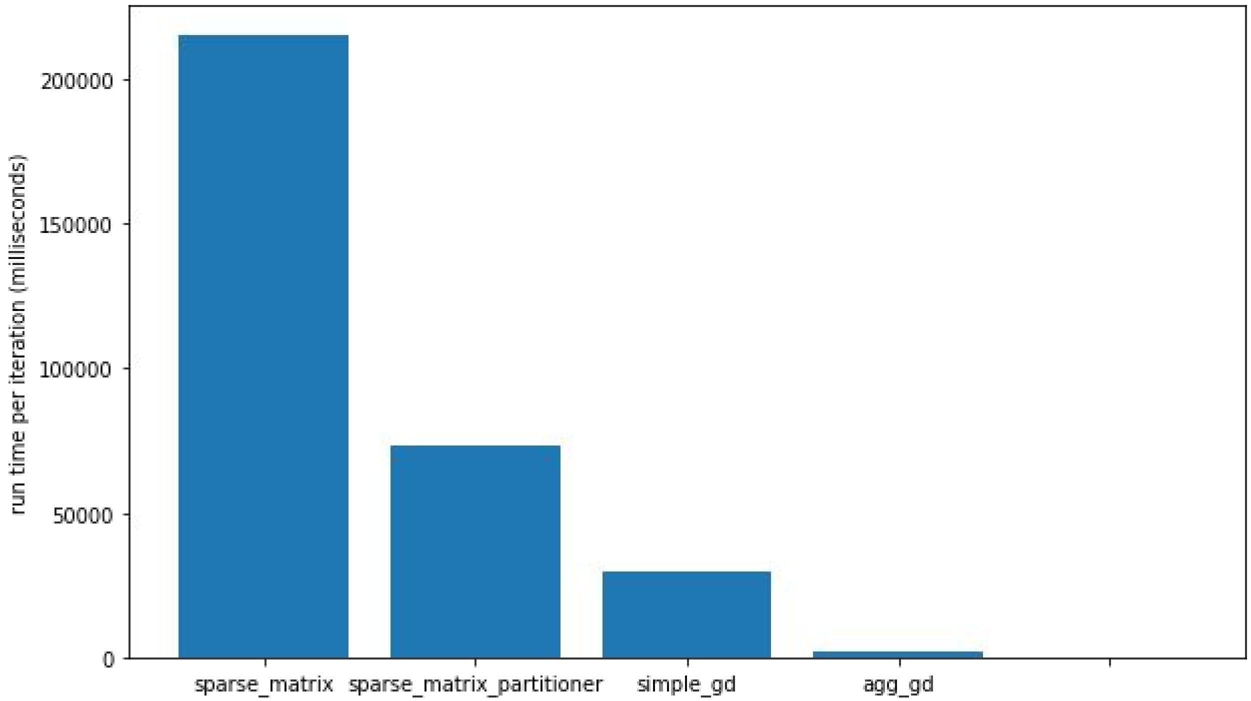
Seq Op

```
class GradientAggregator(weights: Array[Double]) {
  val gradient: Array[Double] = Array.fill(weights.length)(0d)

  def seqOp(example: LabeledExample): this.type = {
    val (target, indexes, values) = (example.target, example.indexes, example.values)
    var (dotProd, k) = (0.0, 0)
    while (k < indexes.length) {
      dotProd += values(k) * weights(indexes(k))
      k += 1
    }
    k = 0
    while (k < indexes.length) {
      gradient(indexes(k)) += (sigmoid(dotProd) - target) * values(k)
      k += 1
    }
    this
  }
```

Comb Op

```
def combOp(other: GradientAggregator): this.type = {
  var k = 0
  while (k < gradient.length) {
    gradient(k) = gradient(k) + other.gradient(k)
    k += 1
  }
  this
}
```

time per iteration

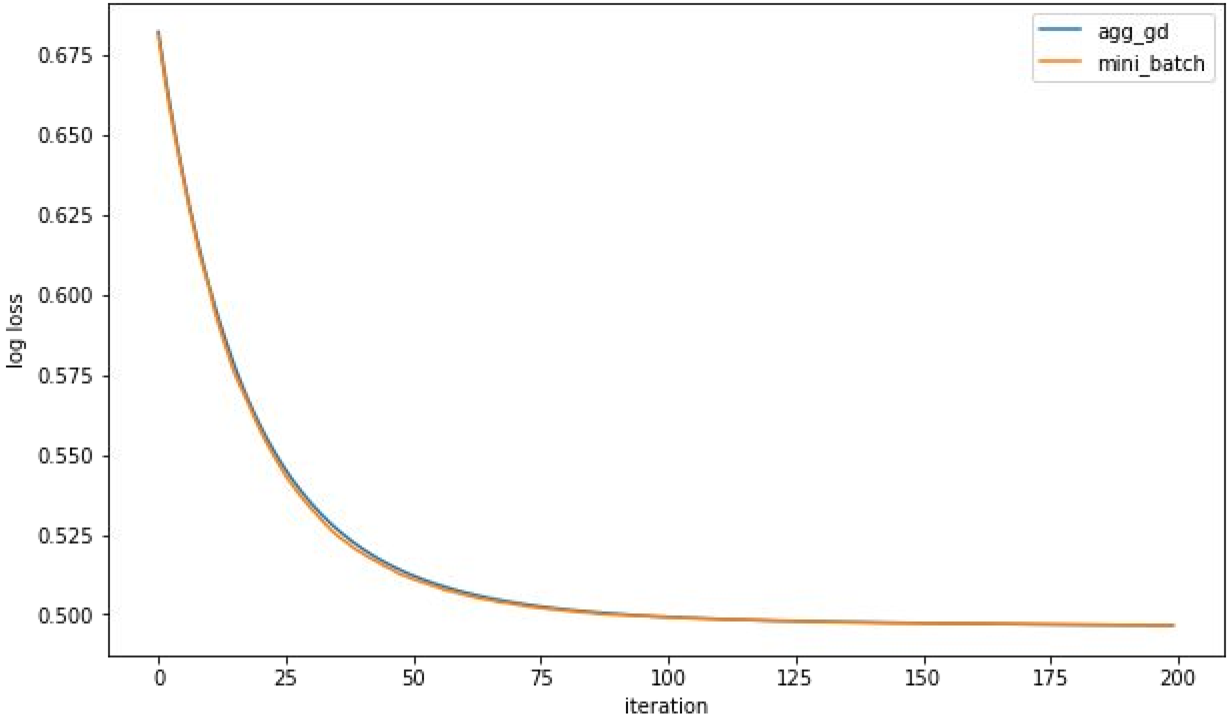


If you can’t decrease the time per iteration, make the iteration smaller

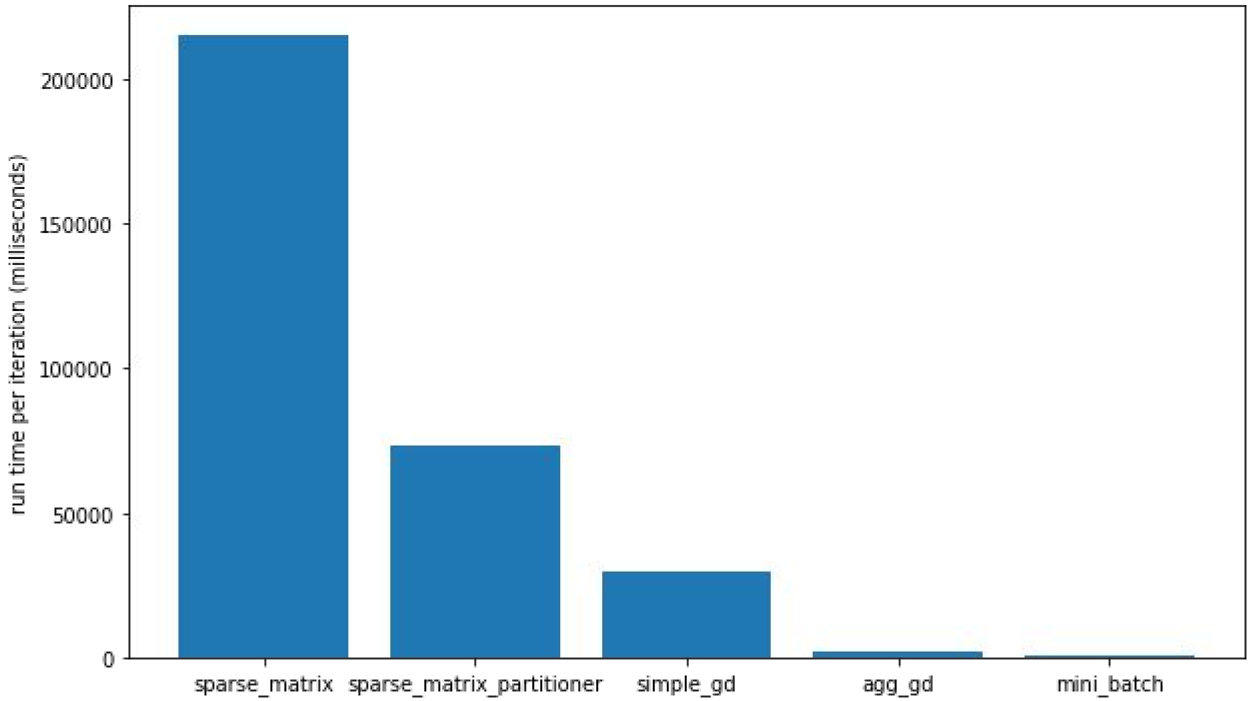
Mini batch gradient descent

```scala
val data: RDD[LabeledExample]

val fraction = batchSize / n
val miniBatch = data.sample(false, fraction)

val aggregator = new GradientAggregator(weights)
val seqOp = (agg: GradientAggregator, example: LabeledExample) => agg.seqOp(example)
val combOp = (agg1: GradientAggregator, agg2: GradientAggregator) => agg1.combOp(agg2)

val result = miniBatch.aggregate(aggregator)(seqOp, combOp)
```

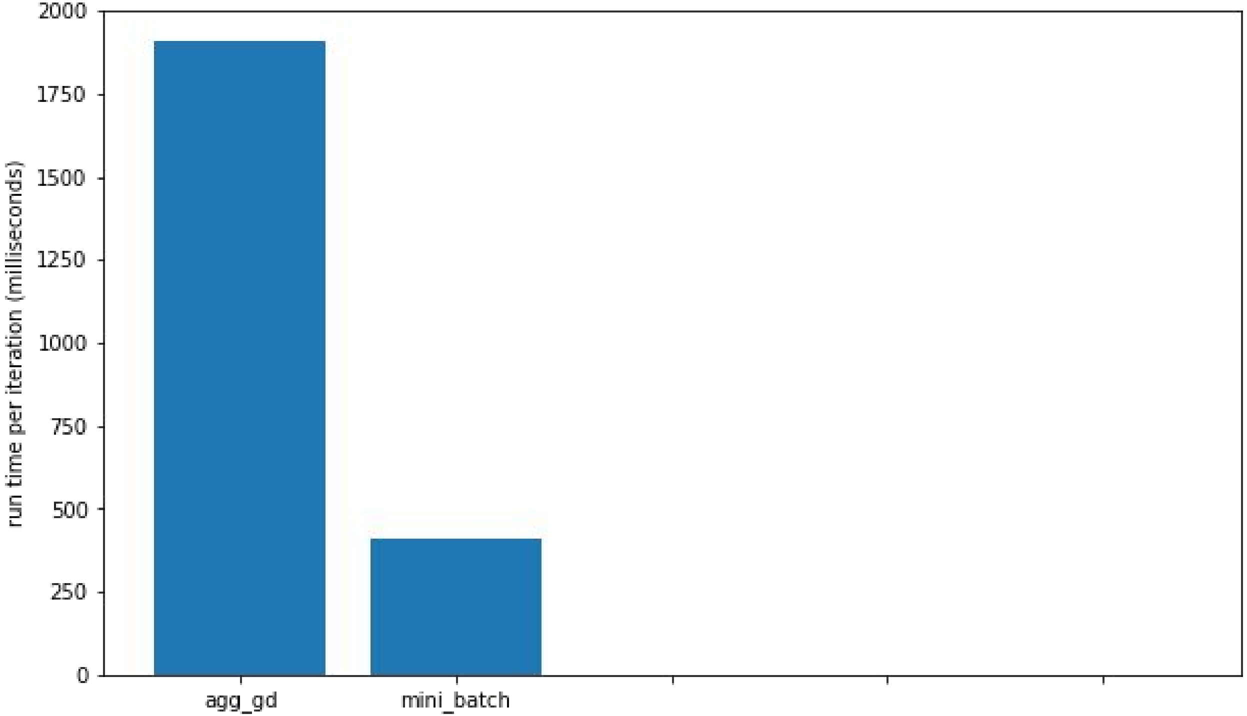
Learning curve still OK



time per iteration



time per iteration



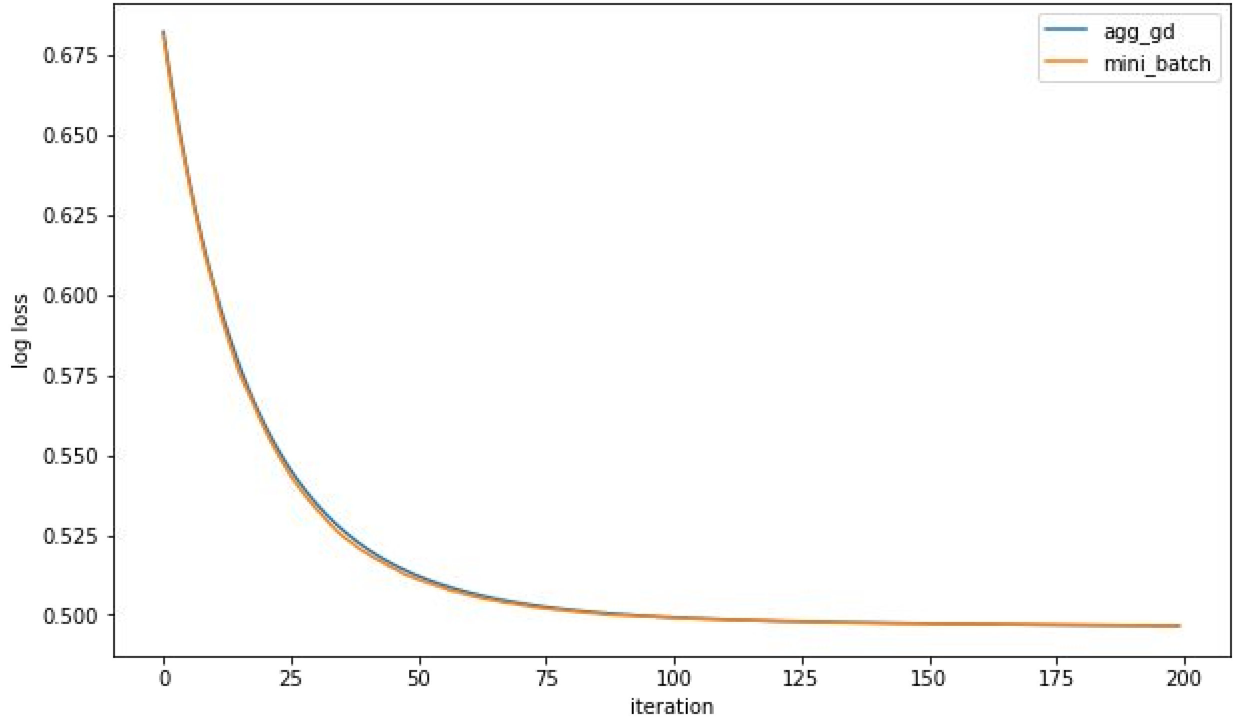
If time per iteration is minimal, try to have fewer iterations

Find a good initialization for the bias

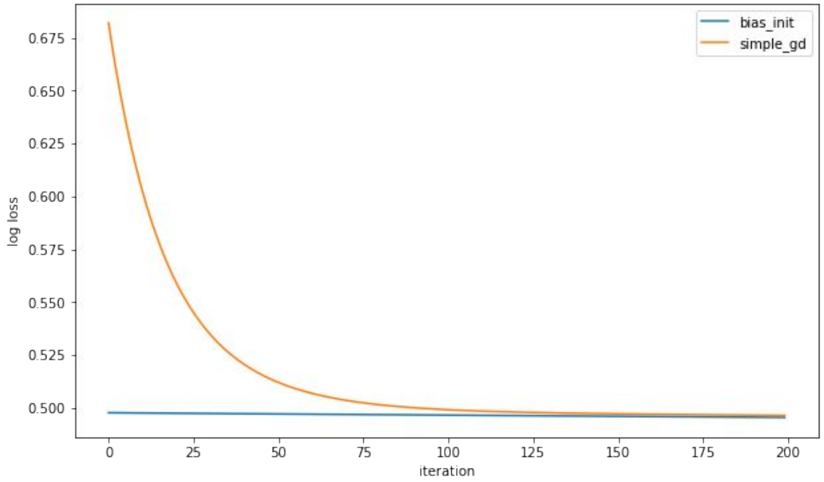
- Usually we initialize weights randomly (or to zero)
- But a careful initialization of the bias can help (especially in very unbalanced datasets)
- We start the gradient descent from a better point and can save several iterations

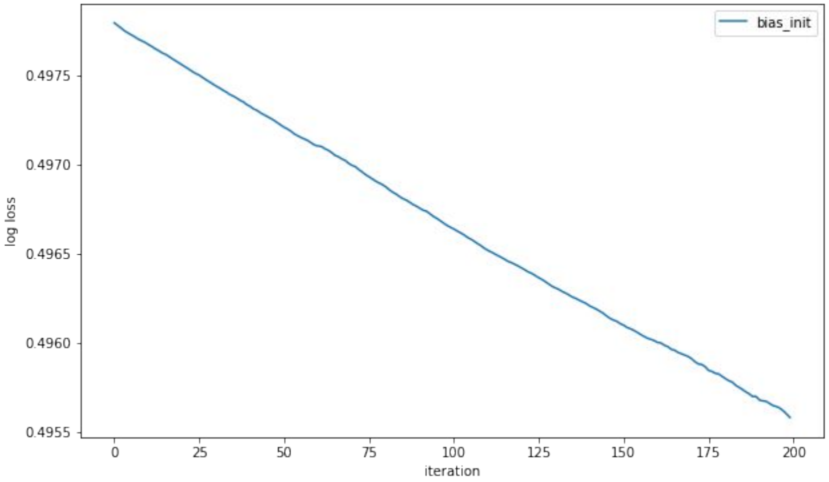
$$b = log(\bar{p}) - log(1 - \bar{p})$$

Learning curve before bias init



Learning curve after bias init



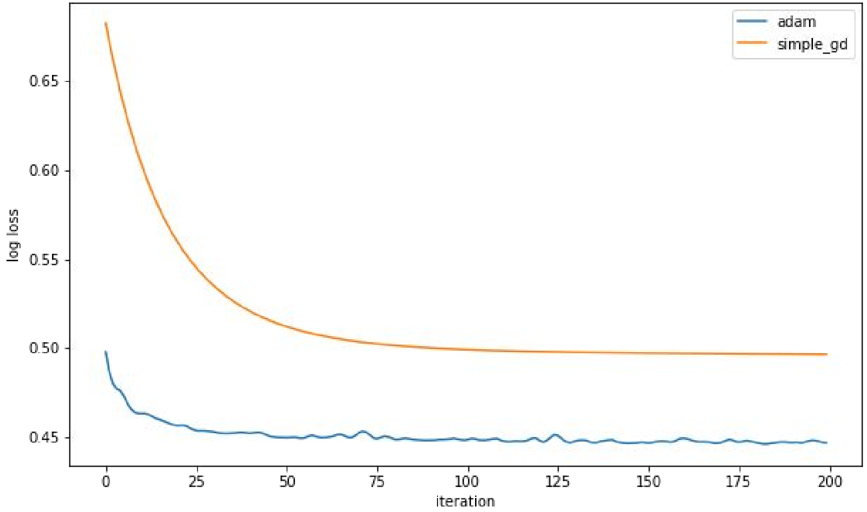


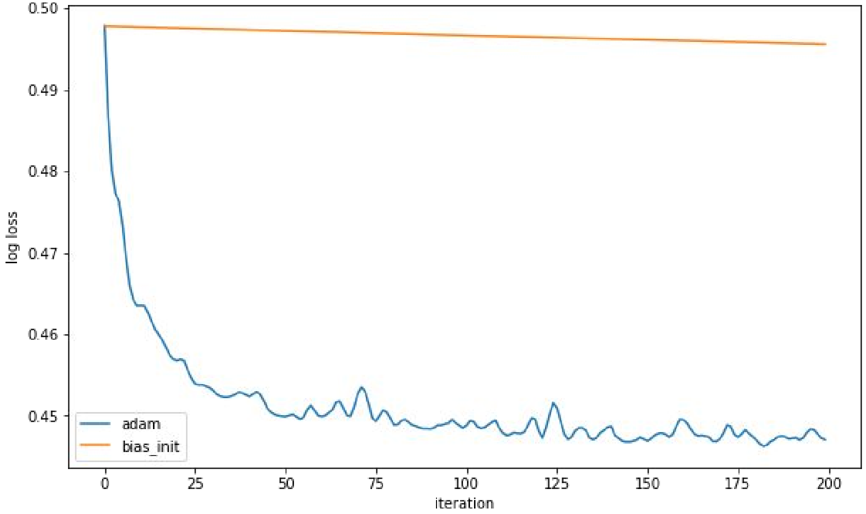
Try a better optimization algorithm to converge faster

ADAM

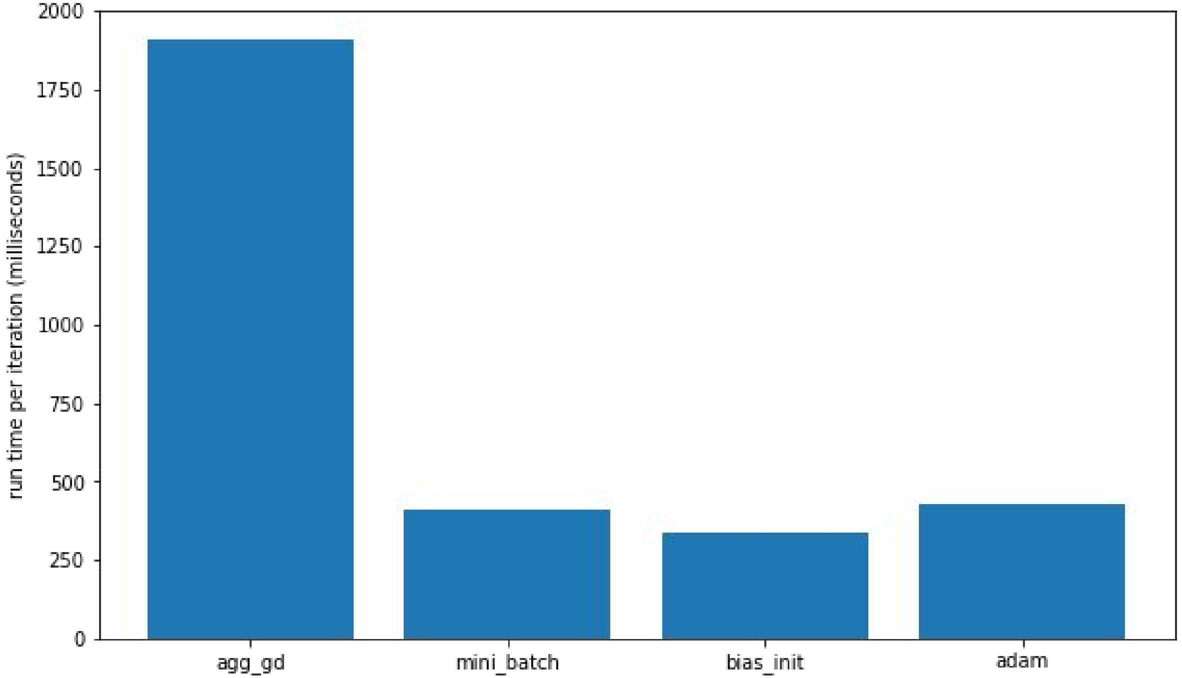
- converges faster
- combines ideas from: gradient descent, momentum and rmsprop
- basically just keeps moving averages and makes larger steps when values are consistent or gradients are small
- useful for making better progress in plateaus

Learning curve ADAM





time per iteration



Conclusion

- we implemented logistic regression from scratch
- the first version was very slow
- but we managed to improve the iteration time 40x
- and also made it converge faster

Thank you!

- Questions, but only simple ones please :)
- Looking forward to discussing offline
- Or write me an email Lorand@Lorand.me
- Play with the code

http://bit.ly/slogreg

- And come work with me at zalando