

AIM3 – Scalable Data Analysis and Data Mining

Classification

Christoph Boden, Sebastian Schelter, Juan Soto, Volker Markl

Based on Material from Jure Leskovec, Jeff Ullman (Stanford),
Michael Jordan, Dan Klein (UC Berkeley) and
Jimmy Lin (University of Maryland / twitter)

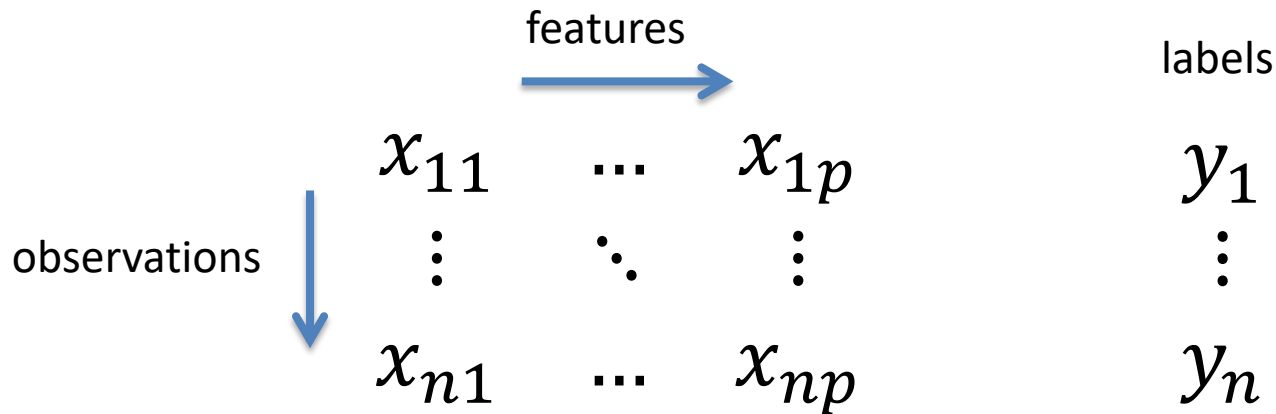


Fachgebiet Datenbanksysteme und Informationsmanagement
Technische Universität Berlin

<http://www.dima.tu-berlin.de/>

Classification

- A rather a loose confederation of „themes“ in statistical inference and decision theory with a focus on computational methodology and empirical evaluation
- Data Matrix: Object \times Attributes (Continuous, Categorical, ...)



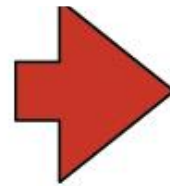
- supervised vs. unsupervised learning

- In classification problems, each entity in some domain can be placed in one of a discrete set of categories: yes/no, friend/foe, good/bad/indifferent, blue/red/green, etc.
- Given a *training set* of labeled entities, develop a rule for assigning labels to entities in a test set (supervised learning)
- Many variations on this theme:
 - binary classification
 - multi-category classification
 - non-exclusive categories
 - Ranking
- Many criteria to assess rules and their predictions
 - overall errors
 - costs associated with different kinds of errors
 - operating points

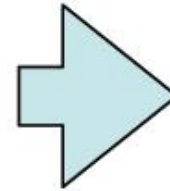
input data



features

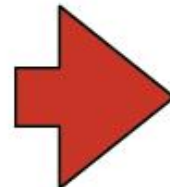


$$\begin{bmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,n} \end{bmatrix}$$

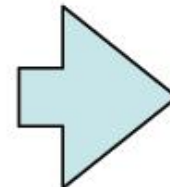


output

“Danger”



$$\begin{bmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,n} \end{bmatrix}$$



Cat

- Input: email
- Output: spam/ham
- Setup:
 - Get a large collection of example emails, each labeled "spam" or "ham"
 - Note: someone has to hand label all this data
 - Want to learn to predict labels of new, future emails
- Features: The attributes used to make the ham / spam decision
 - Words: FREE!
 - Text Patterns: \$dd, CAPS
 - Non-text: SenderInContacts
 - ...



Dear Sir.

First, I must solicit your confidence in this transaction, this is by virtue of its nature as being utterly confidential and top secret. ...

TO BE REMOVED FROM FUTURE MAILINGS, SIMPLY REPLY TO THIS MESSAGE AND PUT "REMOVE" IN THE SUBJECT.

99 MILLION EMAIL ADDRESSES FOR ONLY \$99

Ok, I know this is blatantly OT but I'm beginning to go insane. Had an old Dell Dimension XPS sitting in the corner and decided to put it to use, I know it was working pre being stuck in the corner, but when I plugged it in, hit the power nothing happened.

- Input: images / pixel grids
- Output: a digit 0-9
- Setup:
 - Get a large collection of example images, each labeled with a digit
 - Note: someone has to hand label all this data
 - Want to learn to predict labels of new, future digit images
- Features: The attributes used to make the digit decision
 - Pixels: (6,8)=ON
 - Shape Patterns: NumComponents, AspectRatio, NumLoops
 - ...
- Current state-of-the-art: Human-level performance



0



1



2



1



??

■ Fraud detection

- input: account activity,
- classes: fraud / no fraud

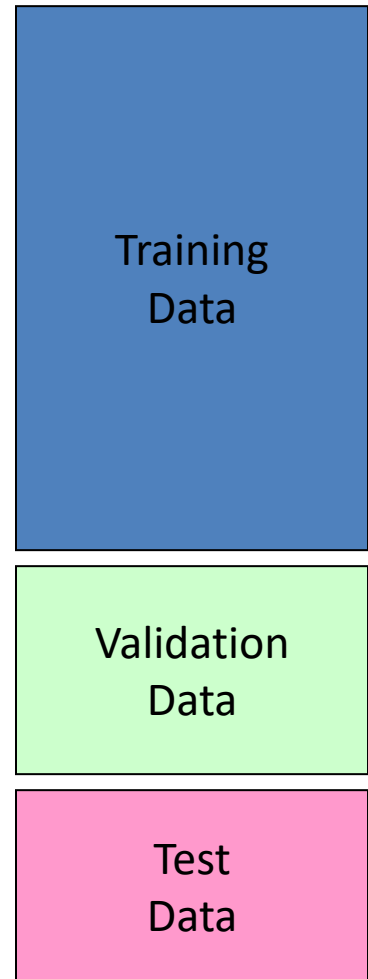
■ Web page spam detection

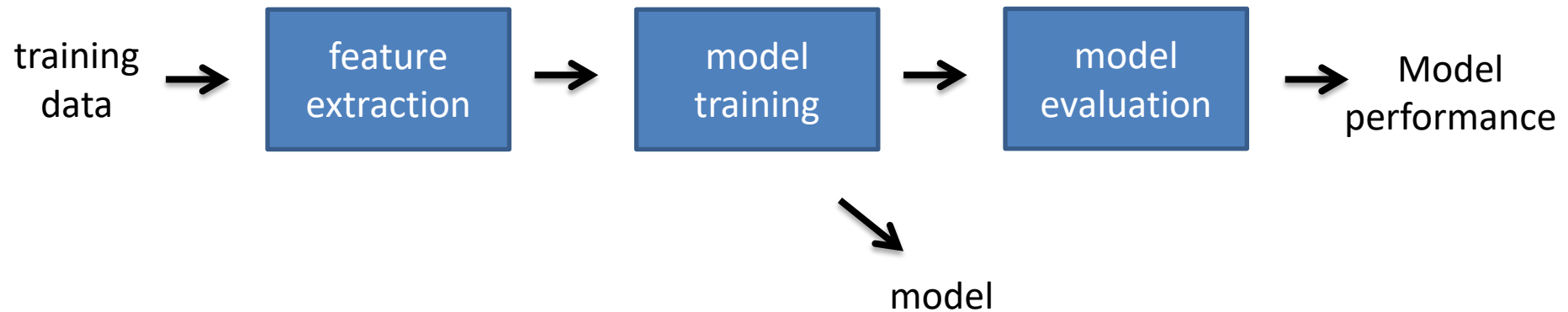
- input: HTML/rendered page,
- classes: spam / ham

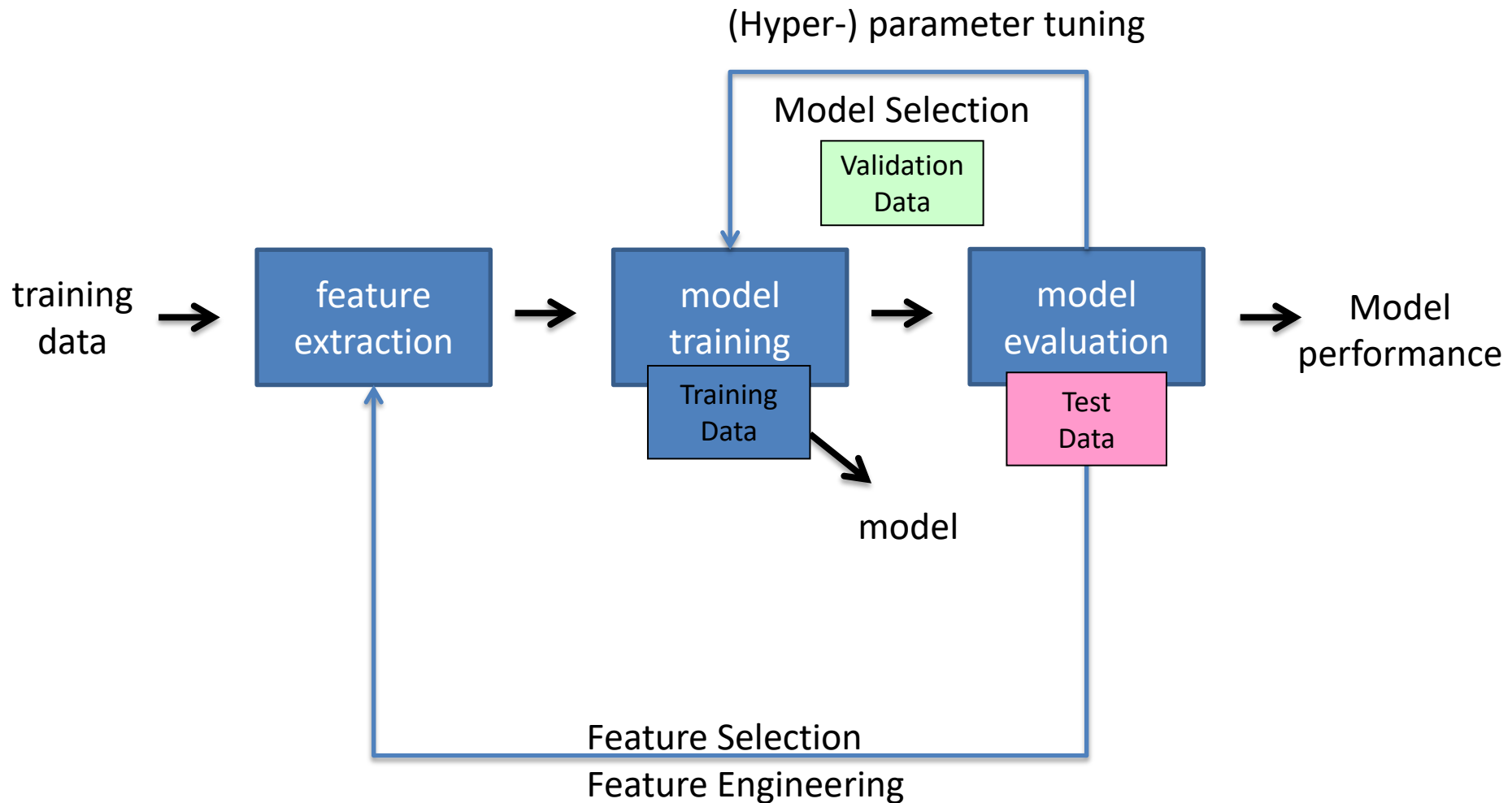
■ sponsored search: predict the bounce rate of an ad

- input: clickstreams, ad, keywords ...
- classes: true bounce rate
(the fraction of users who click on the ad but almost immediately move on to other tasks)

- Data: labeled instances, e.g. emails marked spam/ham
 - Training set
 - Validation set
 - Test set
- Training
 - Estimate parameters on training set
 - Tune hyperparameters on validation set
 - Report results on test set
 - Anything short of this yields over-optimistic claims
- Evaluation
 - Many different metrics
 - Ideally, the criteria used to train the classifier should be closely related to those used to evaluate the classifier
- Statistical issues
 - Want a classifier which does well on *test* data
 - Overfitting: fitting the training data very closely, but not generalizing well
 - Error bars: want realistic (conservative) estimates of accuracy







- learn a function $f: X \rightarrow Y$ that maps the input domain of data X onto the output domain Y which minimizes prediction error („loss function“) $l: Y \times Y \rightarrow \mathbb{R}$ on the provided training data
- Assume training and test data is sampled from some (unknown) joint distribution $P(X, Y)$ over $X \times Y$
- Objective is to find function f which minimizes the expected loss: $E_{(x,y) \sim P} l(f(x), y)$ – such that it generalizes to previously unseen data
- *Learning*: train the model
- *Inference*: predict on unseen data points

- Success or failure of a machine learning classifier often depends on choosing good descriptions of objects
 - the choice of description can also be viewed as a learning problem
 - but good human intuitions are often needed here
- Strategy: overshoot and regularize
 - Come up with lots of features: better to include irrelevant features than to miss important features
- Feature selection: find a subset of features that produces the “best” model $f_w(x)$ with respect to the data set
- Occam's razor
 - the simplest explanation that explains the data is the best
- Also: explore combinations of features in linear models

- a fast and space-efficient way of vectorizing features
- *Hash* high dimensional feature vectors into a *lower* dimensional feature space
- use hash values as indices in feature vector directly
 - (+) don't need to know dimensionality of features in advance (otherwise additional pass over the data)
 - (+) automatically reduces the dimensionality of the featurespace
 - (+) obviously more memory-efficient
 - (+) faster (no dictionary / index lookup)
 - (-) one way transform – can't inspect features
- Use signed hash function:
 - collisions are likely to cancel out rather than accumulate error
 - the expected mean of any output feature's value is zero

Reasons for scaling up machine learning:

- large number of data instances
- high input dimensionality
- model and algorithm complexity
- inference time constraints
- model selection and parameter sweeps („model selection management“)*

*Interesting Vision Paper: Model Selection Management Systems: The Next Frontier of Advanced Analytics (linked on ISIS)

Naïve Bayes

- Learning and classification methods based on probability theory.
 - Bayes theorem plays a critical role in probabilistic learning and classification.
- Build a *generative model* that approximates how data is produced.
- Uses *prior* probability of each category given no information about an item.
- Categorization produces a *posterior* probability distribution over the possible categories given a description of an item (and prior probabilities).

- Given a set of training data:
 - $(class, feature_1, \dots, feature_n) = (C, W_1, \dots, W_n)$
- Want to train a probabilistic classifier, that picks the most probable class label (C) given the observed features ($\{W\}$)
 - Thus, pick the label c^* that maximizes the conditional probability

$$c^* = \operatorname{argmax}_C P(C \mid W_1, \dots, W_n)$$

Bayes Rule

$$= \operatorname{argmax}_C \frac{P(W_1, \dots, W_n \mid C)P(C)}{P(W_1, \dots, W_n)}$$

$$= \operatorname{argmax}_C P(W_1, \dots, W_n \mid C)P(C)$$

(naive) conditional
independence
assumption

$$= \operatorname{argmax}_C P(C) \prod_i P(W_i \mid C)$$

$$W_j \perp W_i \mid C$$

■ Bag-of-Words Naïve Bayes:

- Predict unknown class label (spam vs. ham)
- Assume evidence features (e.g. the words) are independent
- Warning: subtly different assumptions than before!

■ Generative model

$$P(C, W_1 \dots W_n) = P(C) \prod_i P(W_i | C)$$

*Word at position
i, not ith word in
the dictionary!*

■ Tied distributions and bag-of-words

- Usually, each variable gets its own conditional probability distribution $P(F|Y)$
- In a bag-of-words model
 - Each position is identically distributed
 - All positions share the same conditional probs $P(W|C)$
 - Why make this assumption?

- Model: $P(C, W_1 \dots W_n) = P(C) \prod_i P(W_i|C)$
- What are the parameters?

$P(C)$

ham	: 0.66
spam	: 0.33

$P(W|\text{spam})$

the	:	0.0156
to	:	0.0153
and	:	0.0115
of	:	0.0095
you	:	0.0093
a	:	0.0086
with	:	0.0080
from	:	0.0075
...		

$P(W|\text{ham})$

the	:	0.0210
to	:	0.0133
of	:	0.0119
2002	:	0.0110
with	:	0.0108
from	:	0.0107
and	:	0.0105
a	:	0.0100
...		

- Where do these tables come from?

- Posterior determined by *relative* probabilities (odds ratios):

$$\frac{P(W|\text{ham})}{P(W|\text{spam})}$$

```

south-west : inf
nation      : inf
morally     : inf
nicely      : inf
extent      : inf
seriously   : inf
...
    
```

$$\frac{P(W|\text{spam})}{P(W|\text{ham})}$$

```

screens     : inf
minute      : inf
guaranteed  : inf
$205.00     : inf
delivery    : inf
signature    : inf
...
    
```

What went wrong here?

- Laplace's estimate (extended):

- Pretend you saw every outcome k extra times

$$P_{LAP,k}(x) = \frac{c(x) + k}{N + k|X|}$$

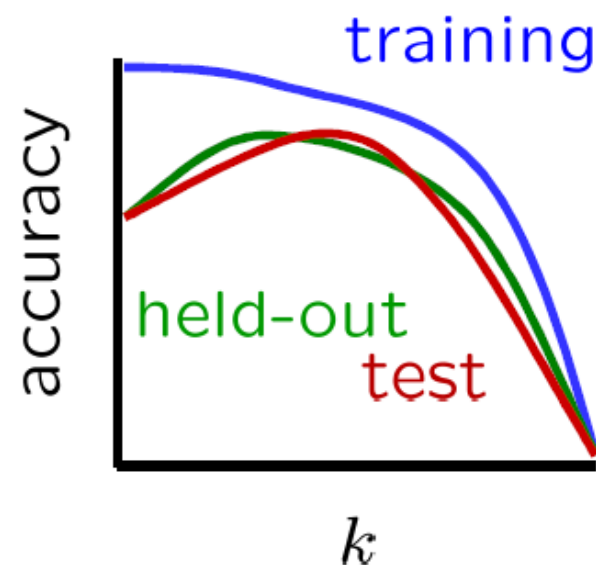
- What's Laplace with $k = 0$?
 - k is the **strength** of the prior

- Laplace for conditionals:

- Smooth each condition independently:

$$P_{LAP,k}(x|y) = \frac{c(x, y) + k}{c(y) + k|X|}$$

- Now we've got two kinds of unknowns
 - Parameters: the probabilities $P(X|Y)$, $P(Y)$
 - Hyperparameters, like the amount of smoothing to do: k , α
- Where to learn?
 - Learn parameters from training data
 - Tune hyperparameters on different data
 - Why?
 - For each value of the hyperparameters, train and test on the held-out data
 - Choose the best value and do a final test on the test data



- Need more features— words aren't enough!
 - Have you emailed the sender before?
 - Have 1K other people just gotten the same email?
 - Is the sending information consistent?
 - Is the email in ALL CAPS?
 - Do inline URLs point where they say they point?
 - Does the email address you by (your) name?

- Can add these information sources as new variables in the NB model

- In practice, Laplace often performs poorly for $P(X|Y)$:
 - When $|X|$ is very large
 - When $|Y|$ is very large
- Another option: linear interpolation
 - Also get $P(X)$ from the data
 - Make sure the estimate of $P(X|Y)$ isn't too different from $P(X)$

$$P_{LIN}(x|y) = \alpha \hat{P}(x|y) + (1.0 - \alpha) \hat{P}(x)$$

- What if α is 0? 1?

- For real classification problems, smoothing is critical
- New odds ratios:

$$\frac{P(W|\text{ham})}{P(W|\text{spam})}$$

helvetica	:	11.4
seems	:	10.8
group	:	10.2
ago	:	8.4
areas	:	8.3
...		

$$\frac{P(W|\text{spam})}{P(W|\text{ham})}$$

verdana	:	28.8
Credit	:	28.4
ORDER	:	27.2
	:	26.9
money	:	26.5
...		

Do these make more sense?

- Classification results of naïve Bayes (the class with maximum posterior probability) are usually fairly accurate.
- However, due to the inadequacy of the conditional independence assumption, the actual posterior-probability numerical estimates are not.
 - Output probabilities are generally very close to 0 or 1.

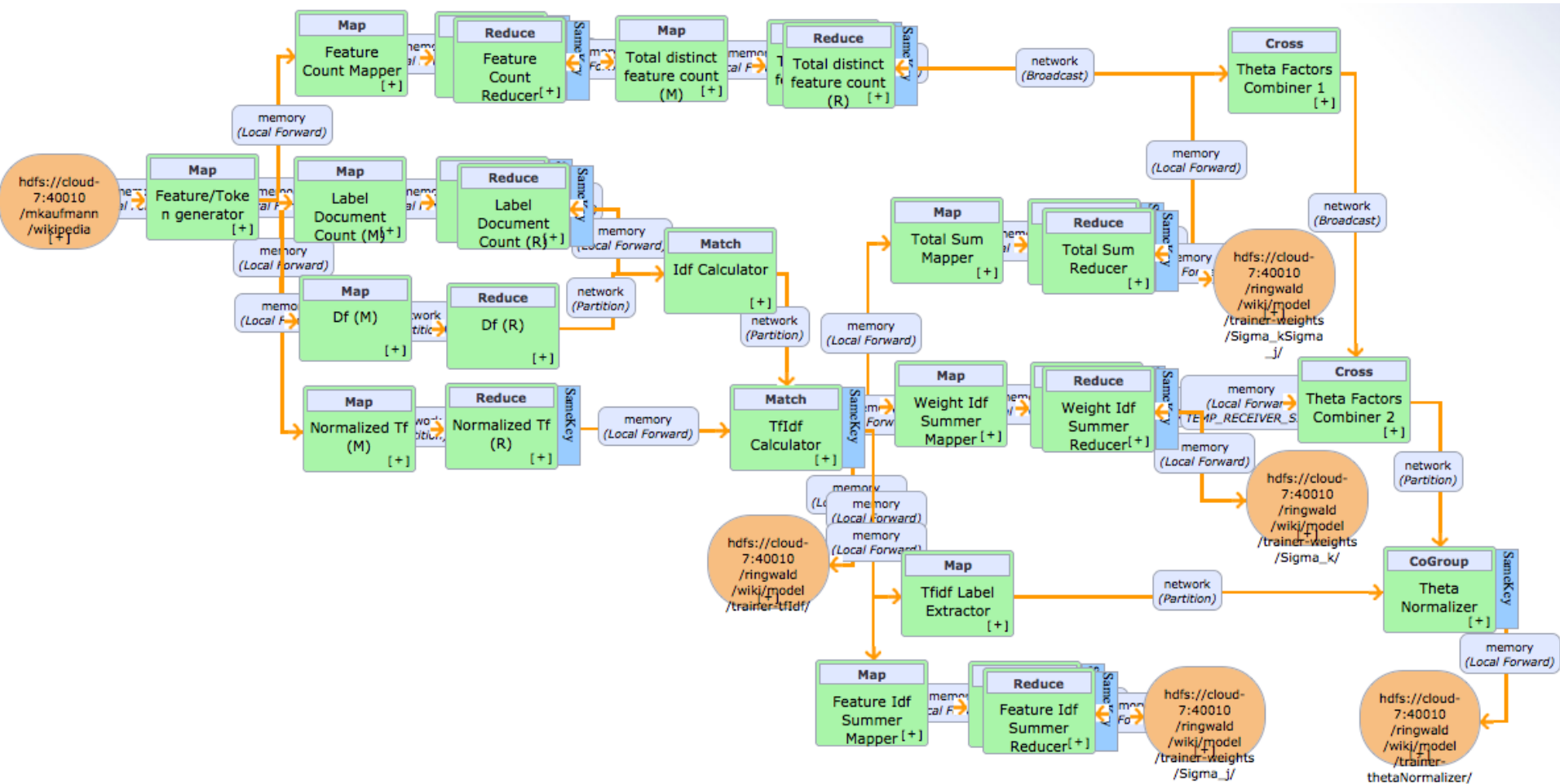
- Usual feature representation for documents:
 - Term-Frequency

$$tf = \frac{\text{frequency}(\text{term}, \text{document})}{|\{w \in \text{document}\}|}$$

- Inverse Document Frequency

$$idf = \log \frac{\text{number of documents}}{|\{d \in \text{Documents} : \text{term} \in d\}|}$$

$$tf - idf = tf \times idf$$



Discriminative Classifiers

$X \in \mathbb{R}^n$ - real valued random input vector

$Y \in \mathbb{R}$ - real valued random output variable

with joint distribution $P(X, Y)$

- supervised learning = search for a function $f_w(X)$ for predicting Y
- Need to define a loss (or cost) function $l(Y, f_w(X))$
e.g. squared loss: $(Y - f(x))^2 \rightarrow$ regression

- **Observation:** many common data analytics tasks can be framed as an optimization problem:

$$\min_w f(w)$$

objective function

$$s. t. \quad h_i(w) = 0, \quad i = 1 \dots p$$

equality constraints

$$g_i(w) \geq 0, \quad i = 1 \dots m$$

inequality constraints

$$w = \operatorname{argmin}_w \left(\sum_i^M l(f_w(x_i), y_i) + \lambda \Omega(w) \right)$$

$f_w(x)$ – *model*

$l(f_w(x), y)$ – *loss function*

$\Omega(w)$ – *regularization*

- Support Vector Machines (max. margin)

$$\min_w \frac{1}{2} \|w\|^2$$

$$\text{s.t.} \quad (w \cdot x + b) \geq 1, \quad \forall x \text{ where } y = 1$$

$$(w \cdot x + b) \leq -1, \quad \forall x \text{ where } y = 0$$

- Logistic Regression:

$$\min_w - \sum_{i=1}^M y_i \cdot \log f_w(x_i) + (1 - y_i) \cdot \log(1 - f_w(x_i))$$

$$f_w(x) = \frac{1}{1 + e^{w^T x}}$$

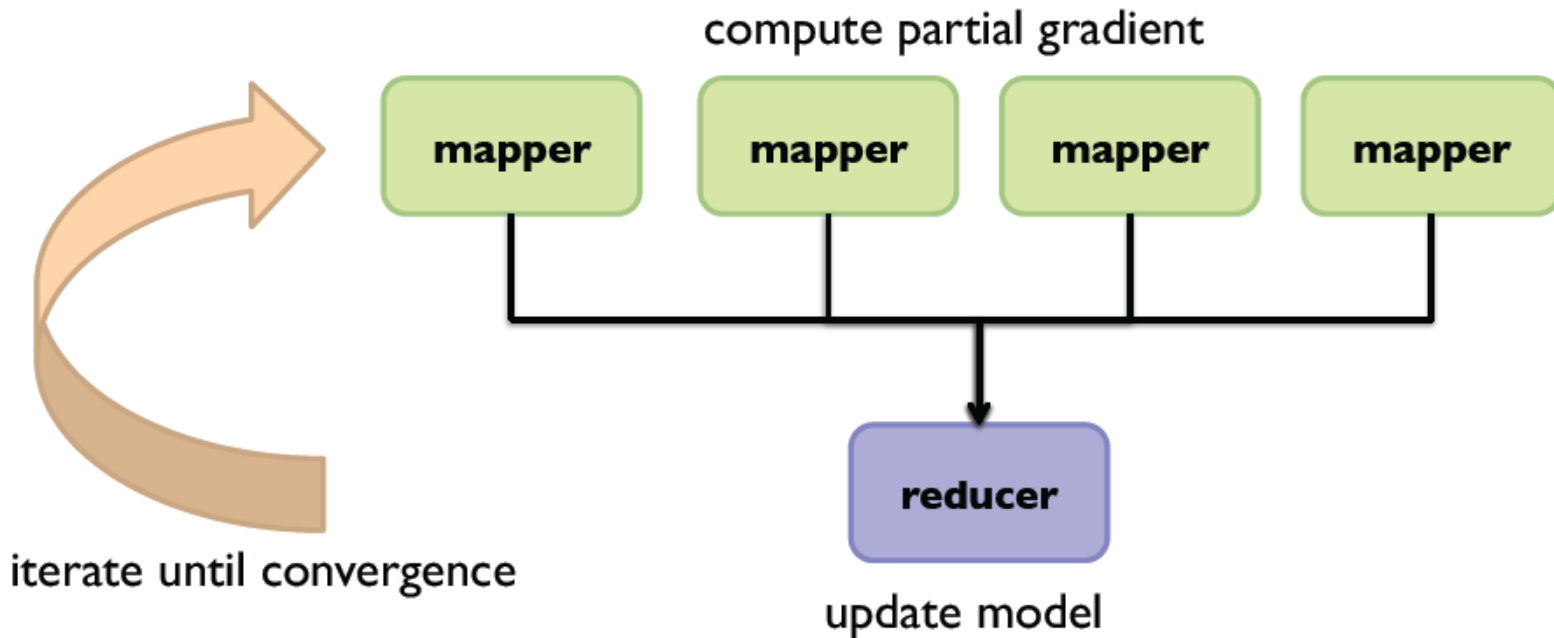
- If loss function **differentiable** and **convex**
 - solve with gradient decent

$$w' = w - \left(\lambda \partial_w \Omega(w) + \sum_i^M \partial_w l(f_w(x), y) \right)$$

- Iterate until convergence criterion
- Gradient-descent algorithms are quite robust:
 - can tolerate noise in gradient estimates
 - node failures
 - even receiving out of order gradient information

while providing statistical guarantees

$$w' = w - \underbrace{\left(\lambda \partial_w \Omega(w) + \underbrace{\sum_i^M \partial_w l(f_w(x), y)}_{\text{mappers}} \right)}_{\text{single reducer}}$$



- Hadoop is bad at iterative algorithms
 - High job startup costs
 - Awkward to retain state across iterations
- High sensitivity to skew
 - Iteration speed bounded by slowest task
- Potentially poor cluster utilization
 - Must shuffle all data to a single reducer

- Update model after each data point (in random order):

$$w' = w - \left(\lambda \partial_w \Omega(w) + \partial_w l(f_w(x), y) \right)$$

- Solves the iteration Problem!
- What about the single reducer problem?

Ensembles




- Learn multiple models
 - Simplest possible technique: majority voting
 - Simple weighted voting:

$$y = \operatorname{argmax}_y \sum_{k=1}^n \alpha_k \cdot f_k(y, w)$$

- Why does it work?
 - If errors uncorrelated, multiple classifiers being wrong is less likely
 - Reduces the variance component of error
- Embarrassingly parallel ensemble learning:
 - Train each classifier on partitioned input
 - Contrast with boosting: more difficult to parallelize

When you only have a hammer...
... get rid of everything that's not a nail!



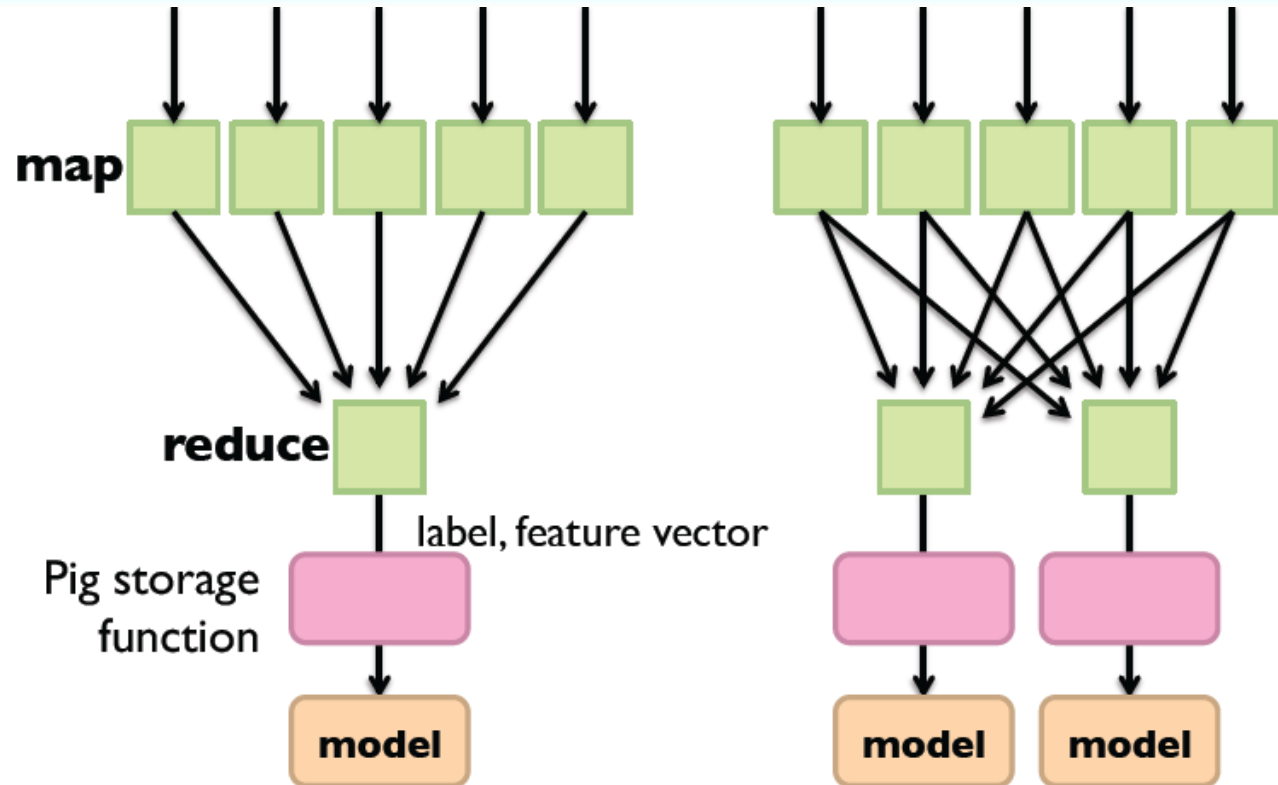
Stochastic gradient descent
Ensemble methods

... good fit for acyclic dataflows

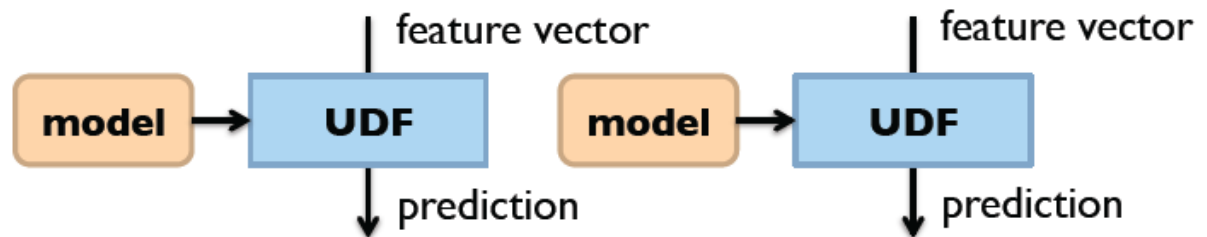
	AVG	SGD
initialize	sum = 0 count = 0	initialize weights
update	add to sum increment count	$w' = w - (\lambda \partial_w \Omega(w) + \partial_w l(f_w(x), y))$
terminate	return sum / count	return weights

Machine learning is basically a user-defined aggregate function!

Classifier Training



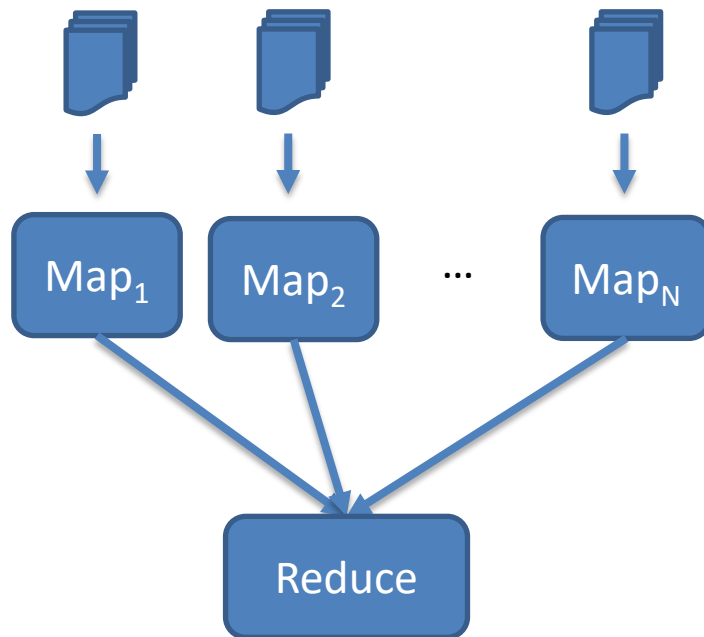
Making Predictions



Just like any other parallel Pig dataflow

Map-Reduce Implementation

$$w' = w - \left(\lambda \frac{\partial}{\partial w} \Omega(w) + \sum_{(x,y) \in (X,Y)} \frac{\partial}{\partial w} l(f_w(x), y) \right)$$

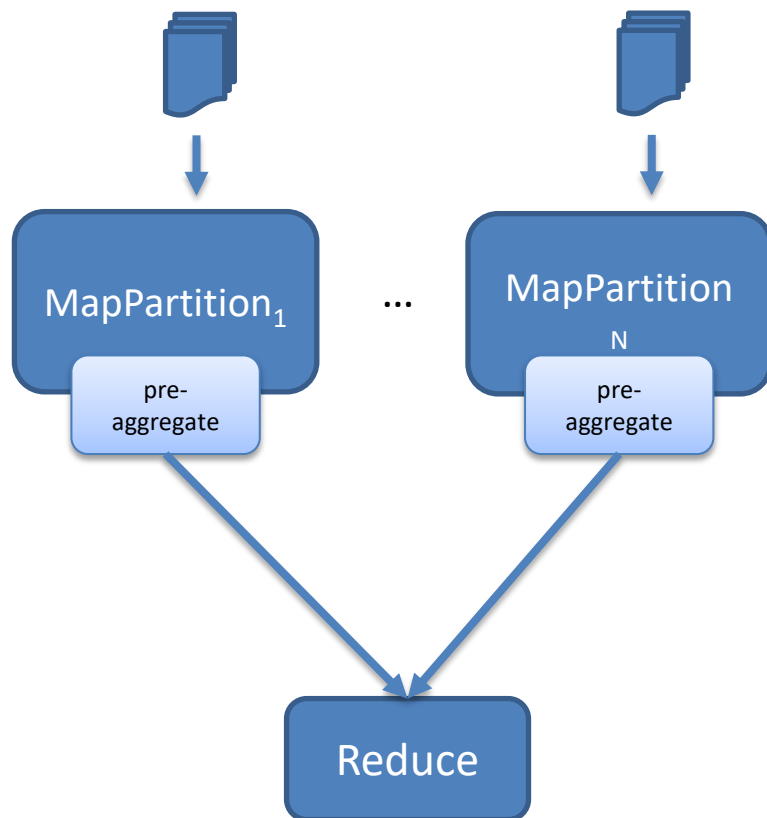


compute gradient per data point

sum up partial gradients

Map-Partition Implementation

$$w' = w - \left(\lambda \frac{\partial}{\partial w} \Omega(w) + \sum_{(x,y) \in (X,Y)} \frac{\partial}{\partial w} l(f_w(x), y) \right)$$

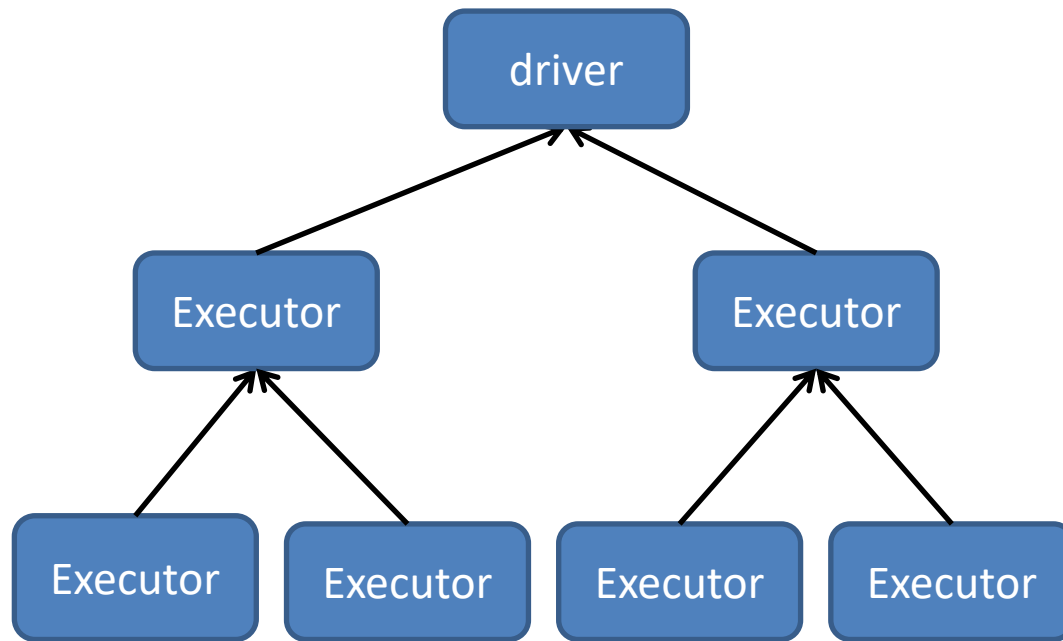


compute gradient per data point (per partition)

locally sum up partial gradients (in udf)

aggregate pre-aggregated partial sums

Tree-Aggregate (Spark)



```
val data: RDD[LabeledPoint] = utils.readData()
data.persist(StorageLevel.MEMORY_AND_DISK_SER)

var w: DenseVector[Double] = DenseVector.fill(numDimensions) { 0.5 }

for (i <- 1 to params.iterations) {

  gamma = params.stepSize / sqrt(i)
  val bcWeights = data.context.broadcast(w)

  val (gradientSum) = data.treeAggregate((DenseVector.zeros[Double](numDimensions)))(
    seqOp = (partSum, dp) => {
      partSum += loss.subgradient(bcWeights.value, dp.features, dp.label)
      partSum
    },
    combOp = (partSum1, partSum2) => {
      (partSum1 + partSum2)
    })

  val reg_gradient: DenseVector[Double] = regularizer.subgradient(w)
  w -= gamma * (gradientSum + regularizer.lambda * reg_gradient)
}
```



```
val train: DataSet[LabeledPoint] = Utils.libSVMDataToBreezeLabeledPoints(env, parser.trainDir,
    parser.numDimensions)
```

```
val weightVector: DataSet[DenseVector[Double]] =
    env.fromElements(DenseVector.fill[Double](sgdparams.numDimensions, 0.5))
```

```
val resultingWeights = initialWeightVector.iterate(numIterations) {

    weights => {
        val update : DataSet[DenseVector[Double]] =
            train
                .mapPartition(new computeGradientMapPartitionFunction(lossFunction, regFunction))
                .withBroadcastSet(weights, WEIGHT_VECTOR)
                .withParameters(iterationParameters)
                .reduce{(a,b) => a += b}

        val finalWeights: DataSet[DenseVector[Double]] = weights.union(update).reduce{ _ + _ }

        finalWeights
    }
}
```



```

//
class computeGradientMapPartitionFunction(lossFunction: LossFunction, regFunction: Regularizer) extends RichMapPartitionFunction[LabeledPoint, DenseVector[Double]] {

  var originalW: DenseVector[Double] = _
  var fraction: Double = 0.0
  var seed: Int = 0
  var numDimensions = 0
  var stepSize: Double = 0.0
  var lambda: Double = 0.0
  var loss: Functions.LossFunction = lossFunction
  var regularizer: Functions.Regularizer = regFunction

  // retrieve broadcasted weight vector on local mapper
  override def open(parameters: Configuration): Unit = {
    originalW = getRuntimeContext.getBroadcastVariable(WEIGHT_VECTOR).get(0)

    val defaultLossFunction = loss
    val defaultRegularizer = regularizer

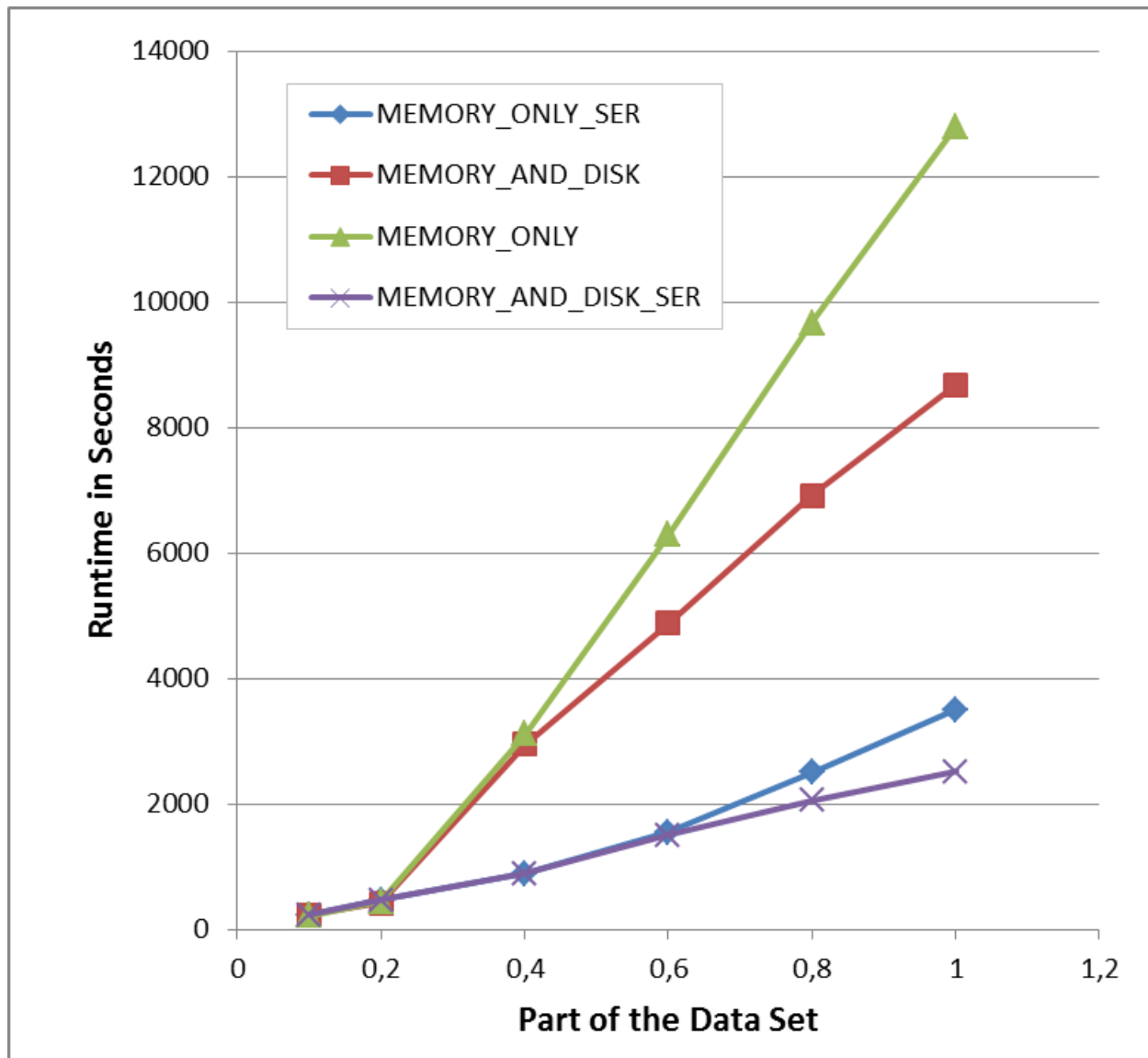
    fraction = parameters.getDouble("fraction", 1.0)
    seed = parameters.getInteger("seed", 42)
    stepSize = parameters.getDouble("stepSize", 0.1)
    lambda = parameters.getDouble("lambda", 0.1)
    numDimensions = parameters.getInteger("dimensions", 2000)
    loss = parameters.getClass(
      "LossFunction",
      defaultLossFunction.getClass,
      defaultLossFunction.getClass.getClassLoader).newInstance()
    regularizer = parameters.getClass(
      "Regularizer",
      defaultRegularizer.getClass,
      defaultRegularizer.getClass.getClassLoader).newInstance()
  }

  // map partition function override
  override def mapPartition(values: Iterable[LabeledPoint], out: Collector[DenseVector[Double]]): Unit = {
    var partialSum: DenseVector[Double] = DenseVector.zeros[Double](numDimensions)
    val learning_rate: Double = stepSize / sqrt(getIterationRuntimeContext.getSuperstepNumber.toDouble)
    val numPartitions = getIterationRuntimeContext.getNumberOfParallelSubtasks()

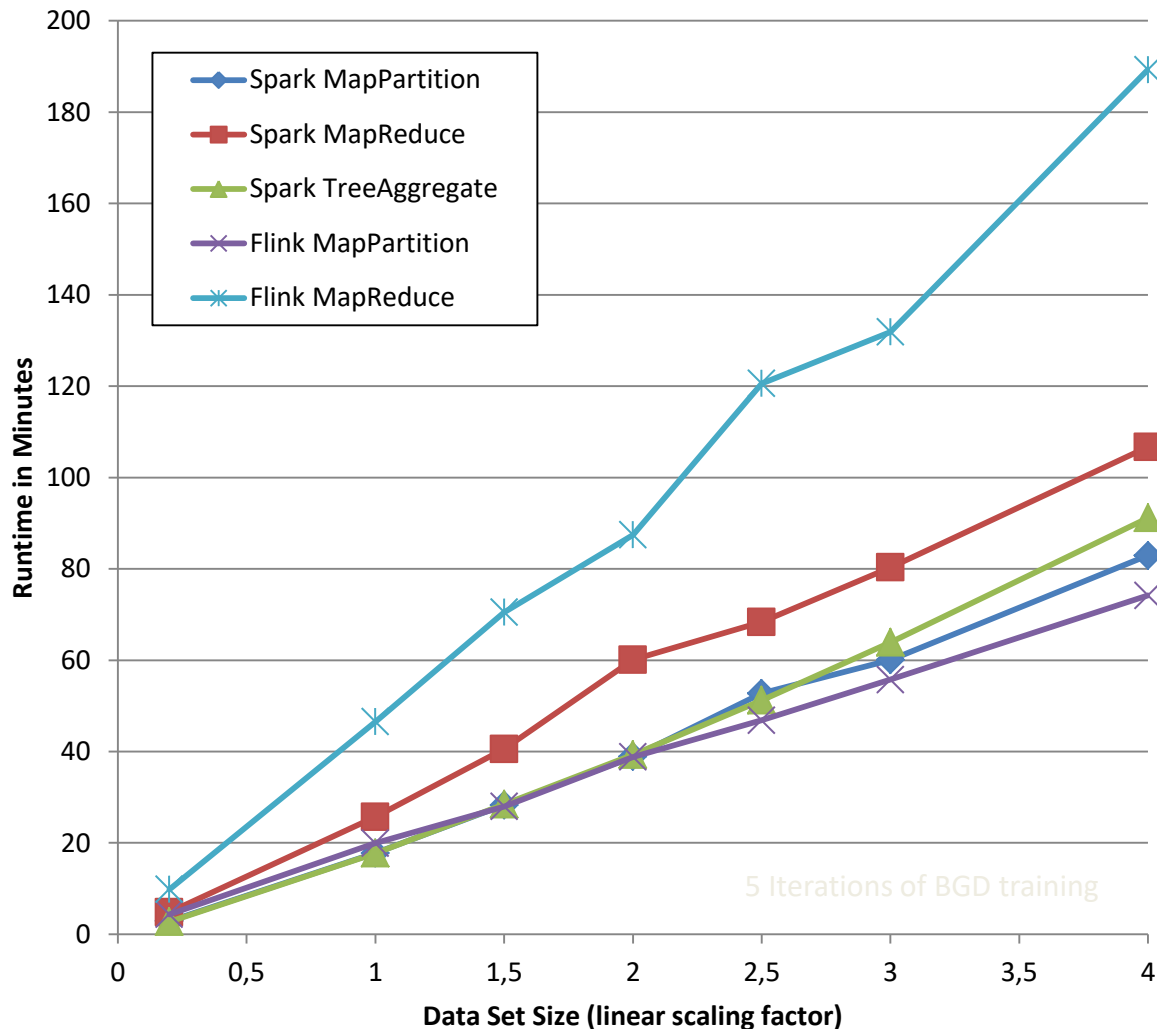
    println("[ " + Calendar.getInstance().getTime() + " ] --> starting MapPartition for iteration #" + getIterationRuntimeContext.getSuperstepNumber.toDouble)
    println("numDimensions = " + numDimensions)
    // sum up all gradients into partial sum
    val iterator = values.iterator()
    while(iterator.hasNext){
      val currDataPoint = iterator.next()
      partialSum += loss.subgradient(originalW, currDataPoint.features, currDataPoint.label)
    }

    out.collect((-1.0) * learning_rate * (partialSum.toDenseVector + (lambda / numPartitions * regularizer.subgradient(originalW))))
  }
}

```

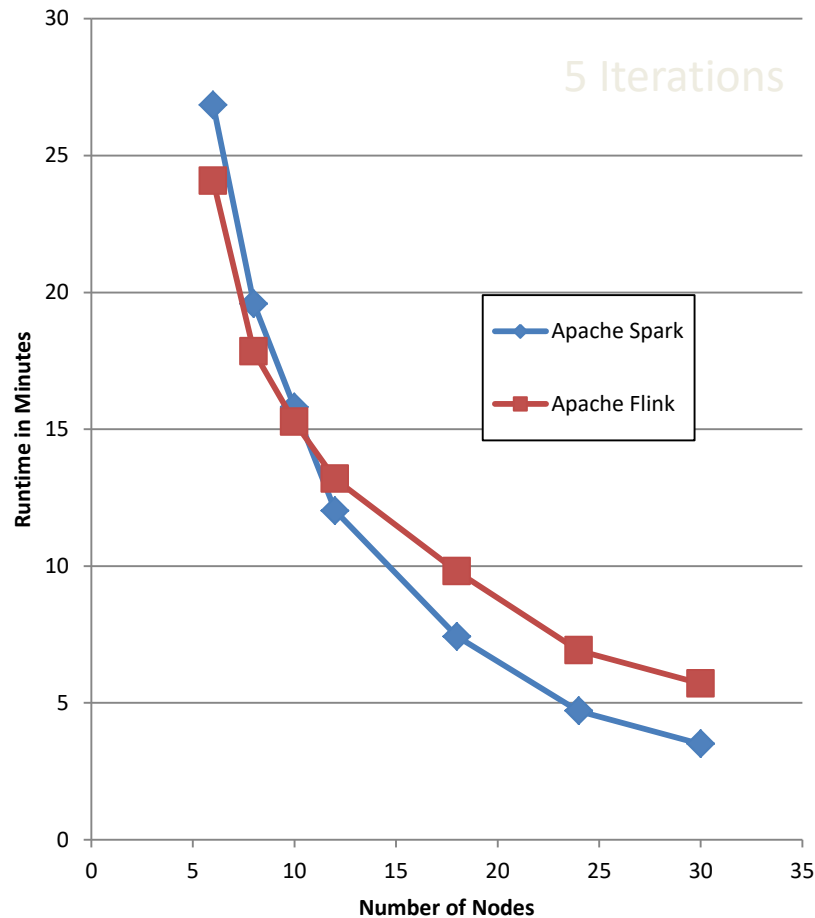


Production Scaling: Implementation Strategies

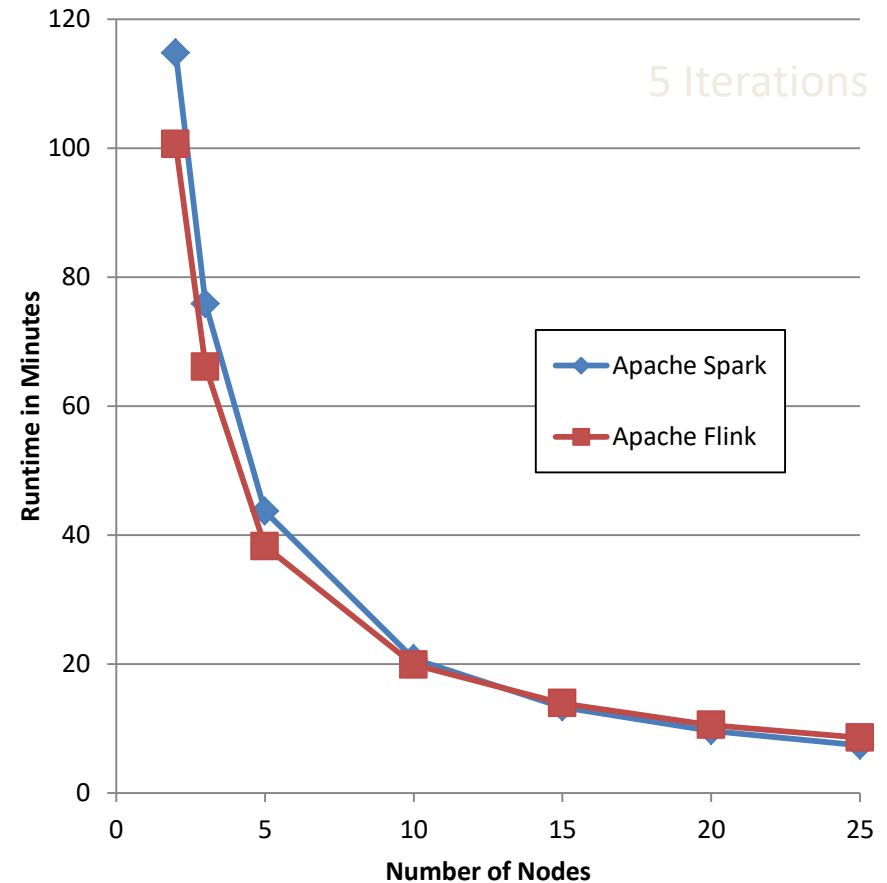


- choice of implementation strategy matters!
- all implementation scale gracefully out-of-core
- Spark's MapPartition slightly faster than TreeAggregate, but not robust
- unfortunate kryo serialization bug penalizing Flink's MapReduce implementation

Strong Scaling Experiments

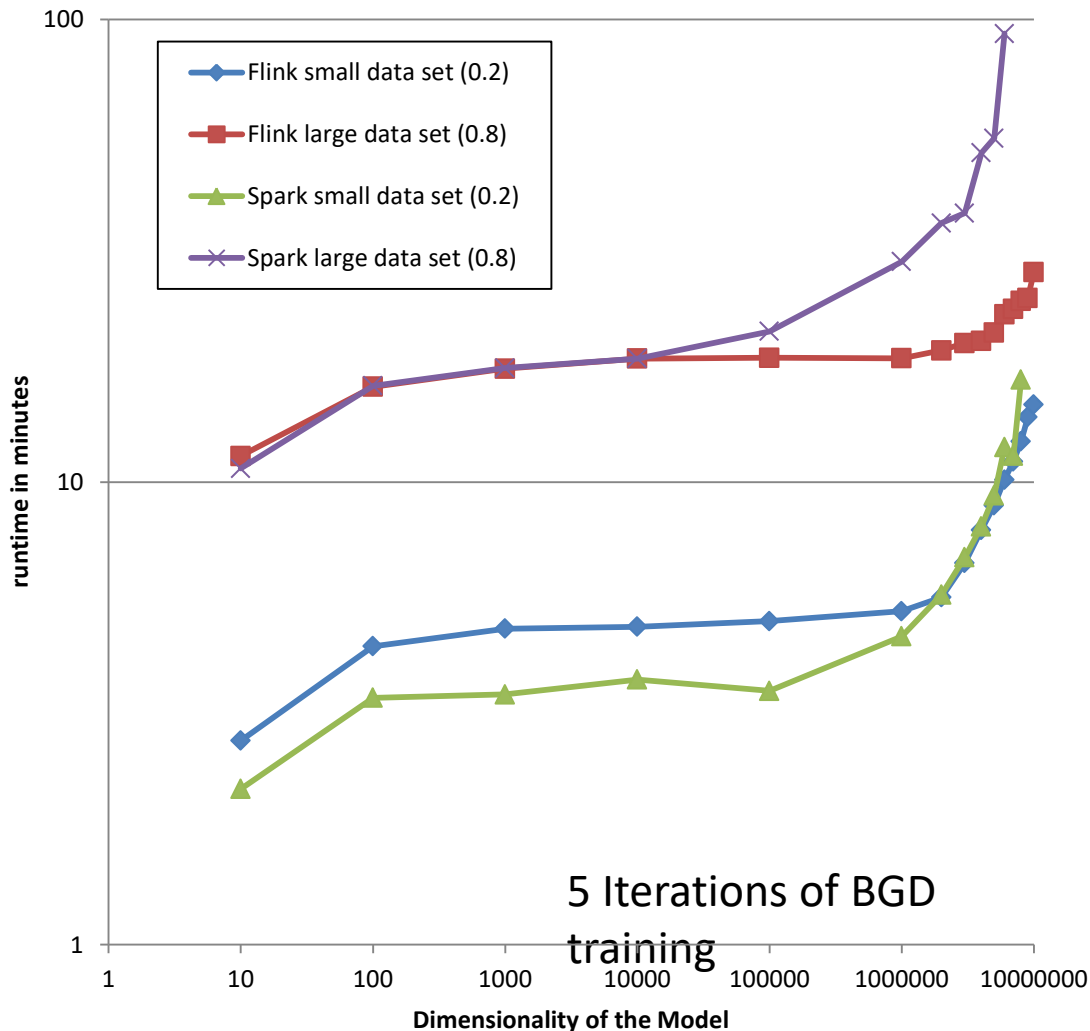


K-Means Clustering



Batch Gradient Descent

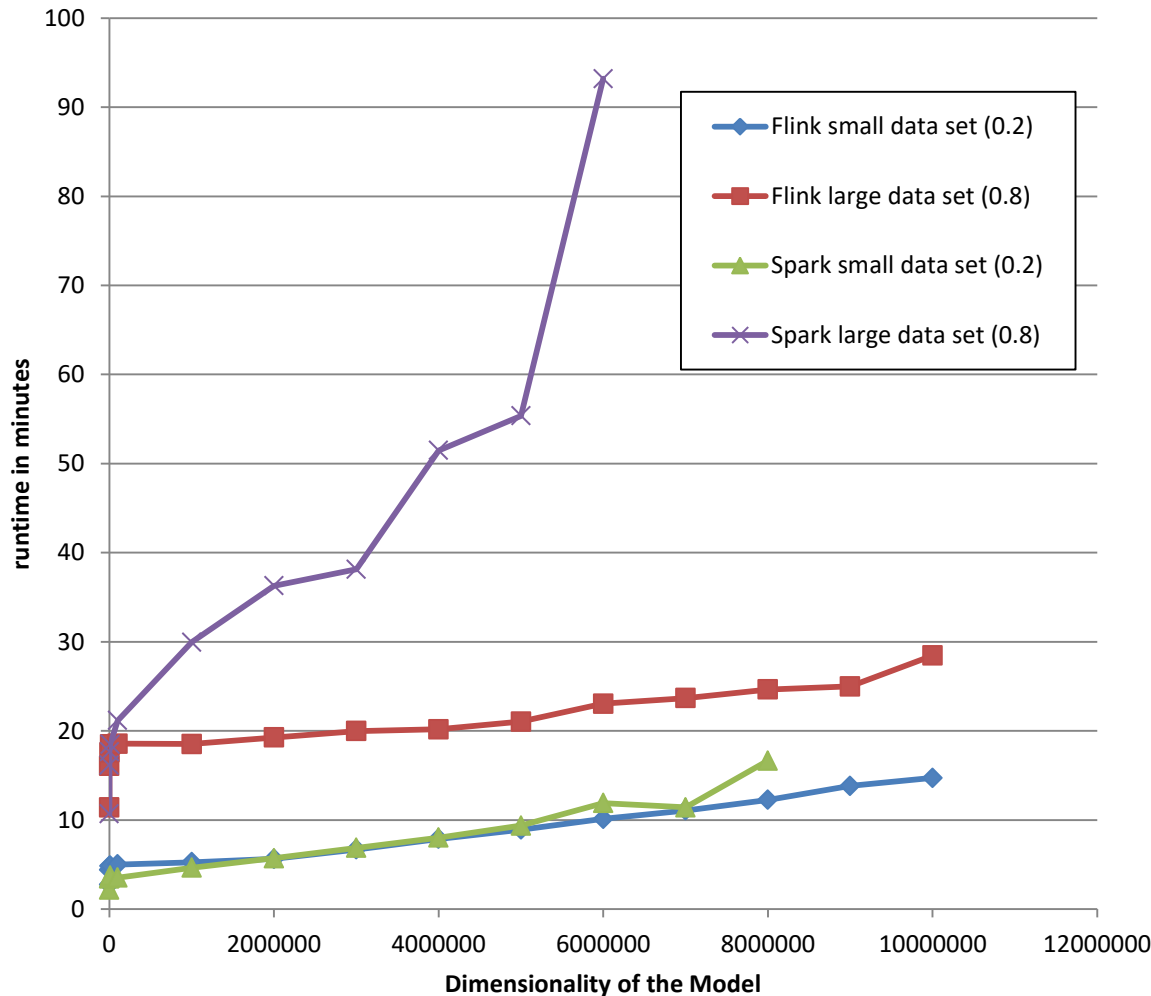
Dimensionality Scaling (log-log)



two data sets:

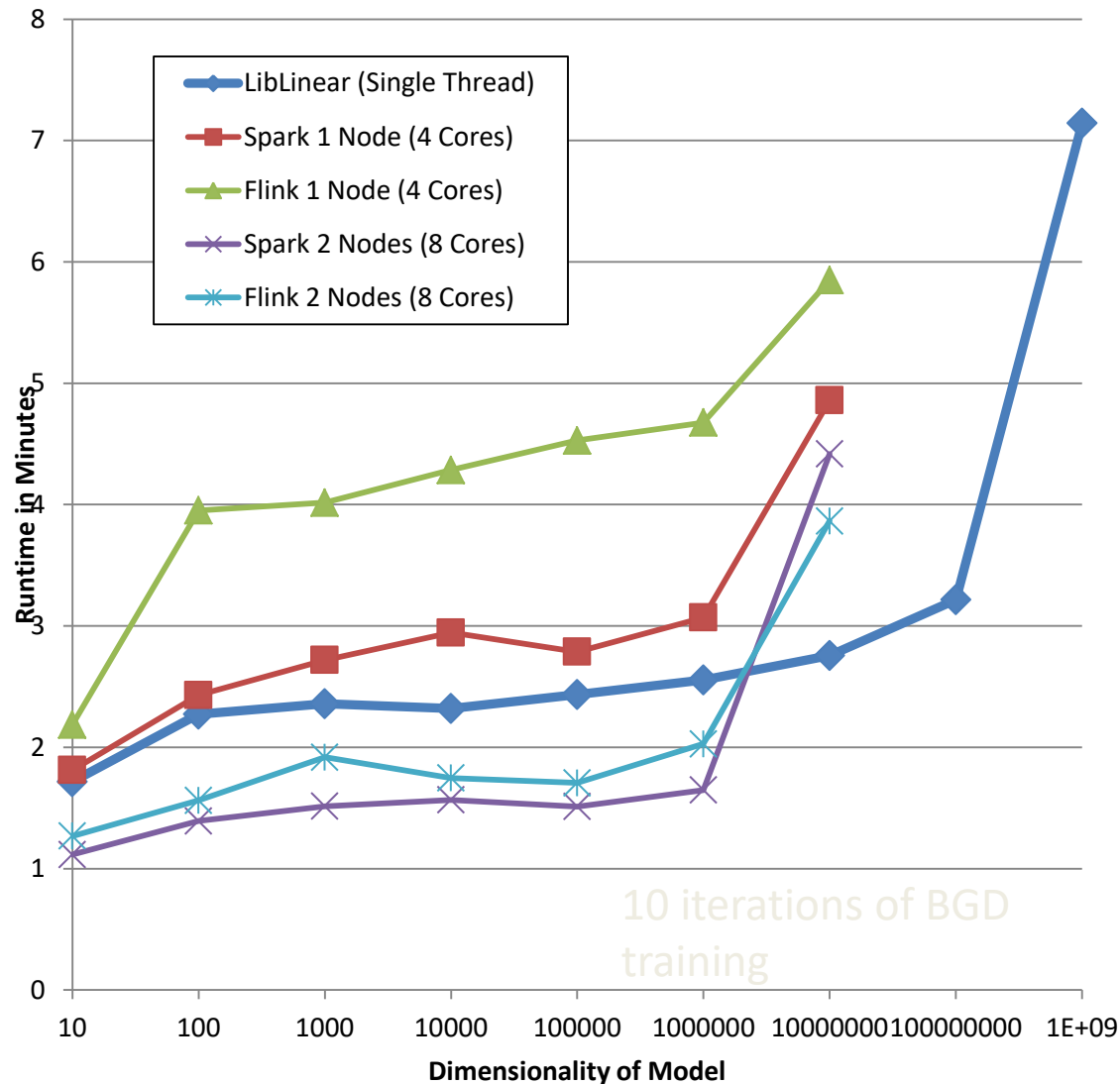
- 0.2 = size of combined main memory
- 0.8 = bigger than combined main memory
- Spark performance comparable or better than flink for small dimensions

Dimensionality Scaling



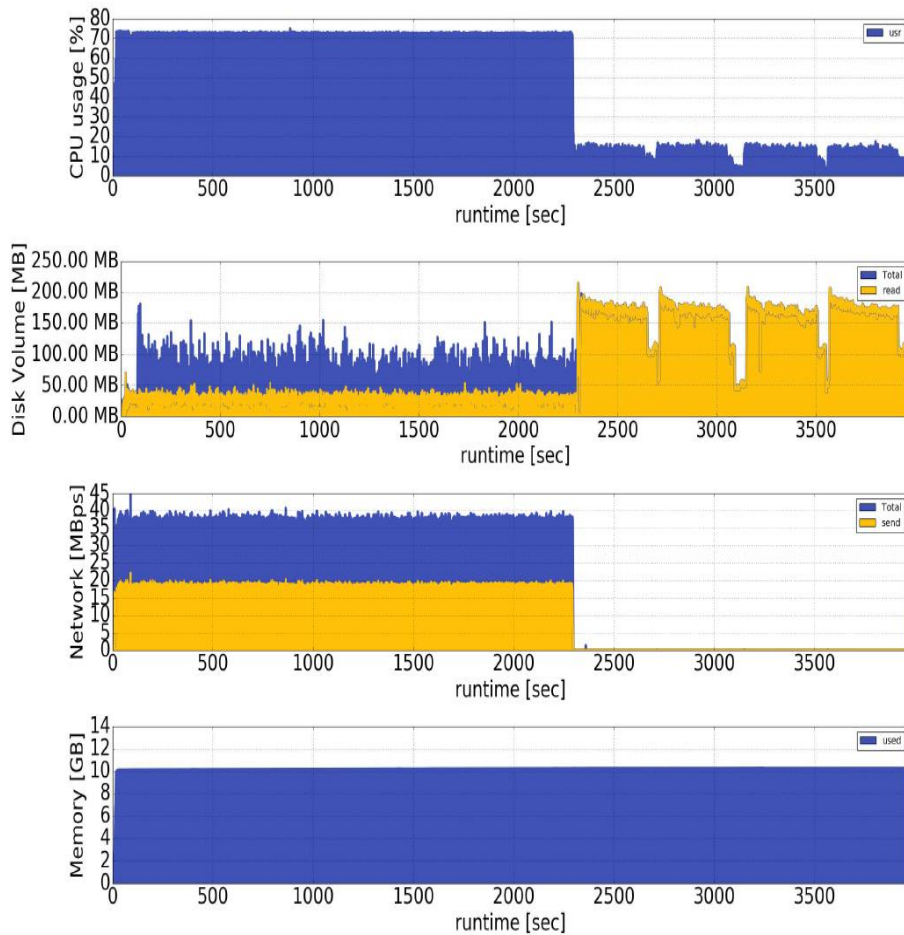
- spark fails to train models beyond 6m dimensions on 0.8 data set
- spark fails to train models beyond 8m dimensions on 0.2 data set
- flink robustly scales to 10m dimensions for both data sets
- flink fails to train models greater than 10m dimensions

COST: vs. Single Threaded Implementation

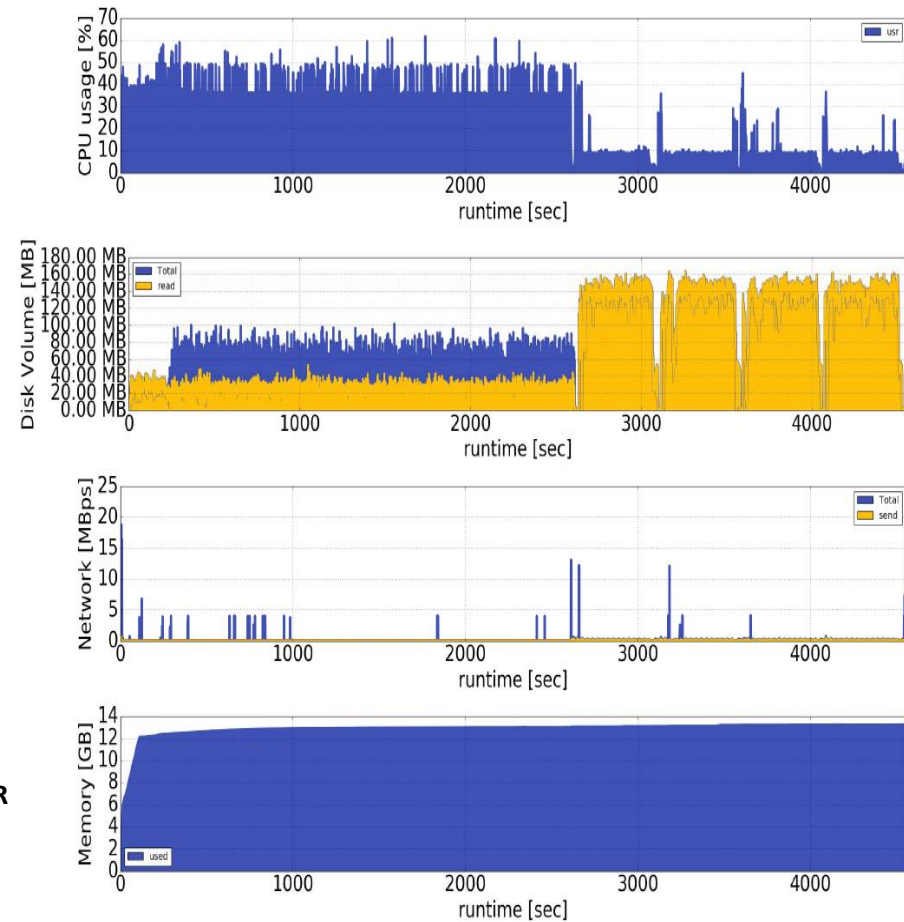


- 4GB subsample of criteo data set
- 2 machines (8 cores) sufficient to outperform single threaded impl.
- both Flink and Spark fail to train with 100m dimensions or beyond

Batch Gradient Descent on 4 Nodes

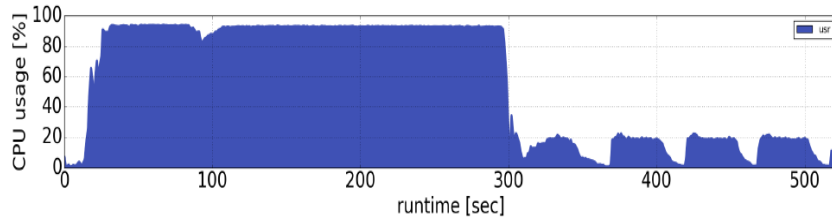


Apache Flink

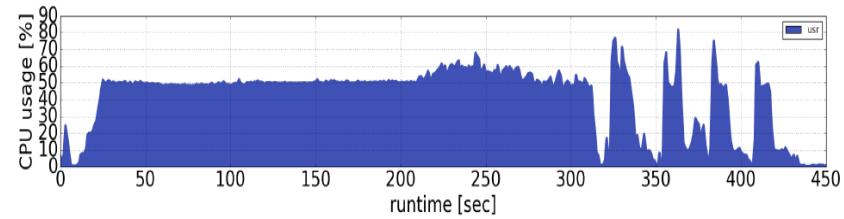


Apache Spark

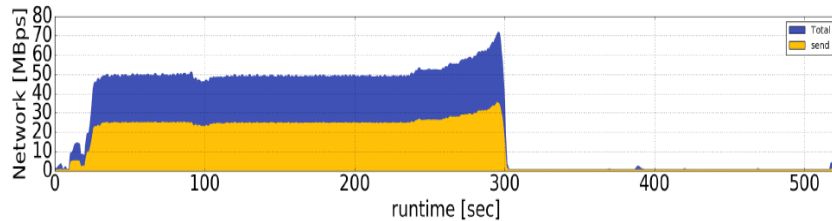
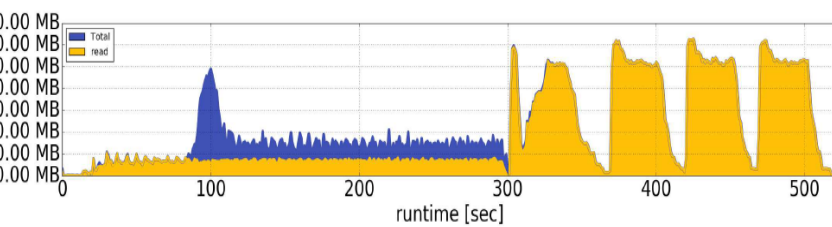
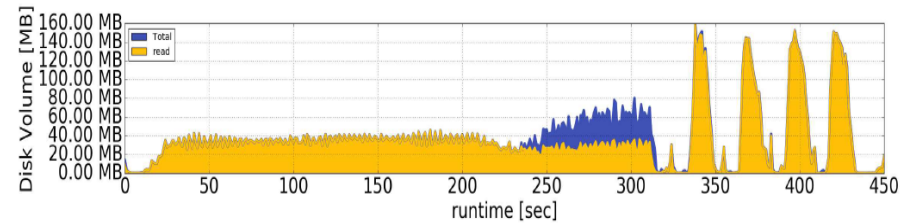
Batch Gradient Descent on 25 Nodes



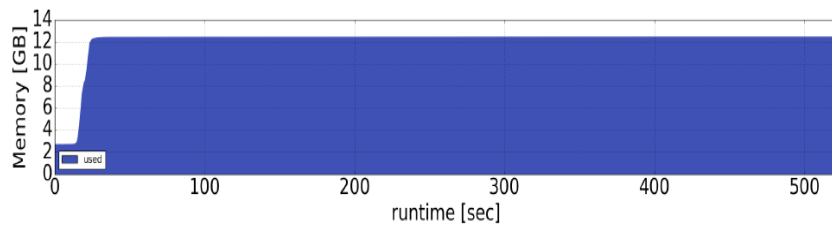
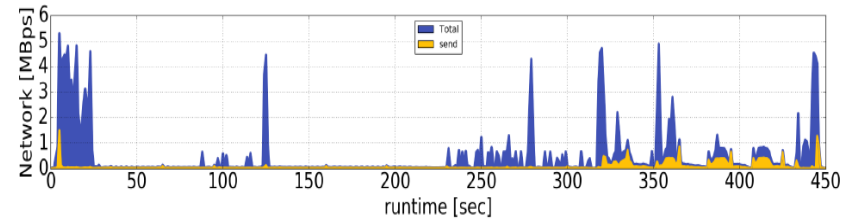
CPU



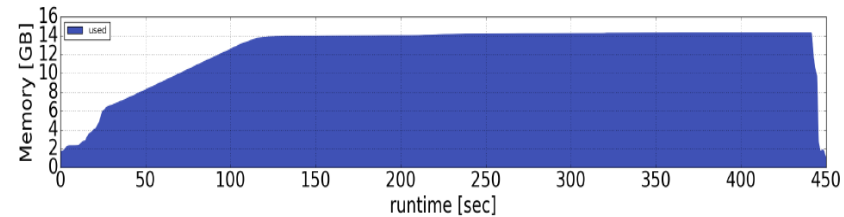
DSK



NET



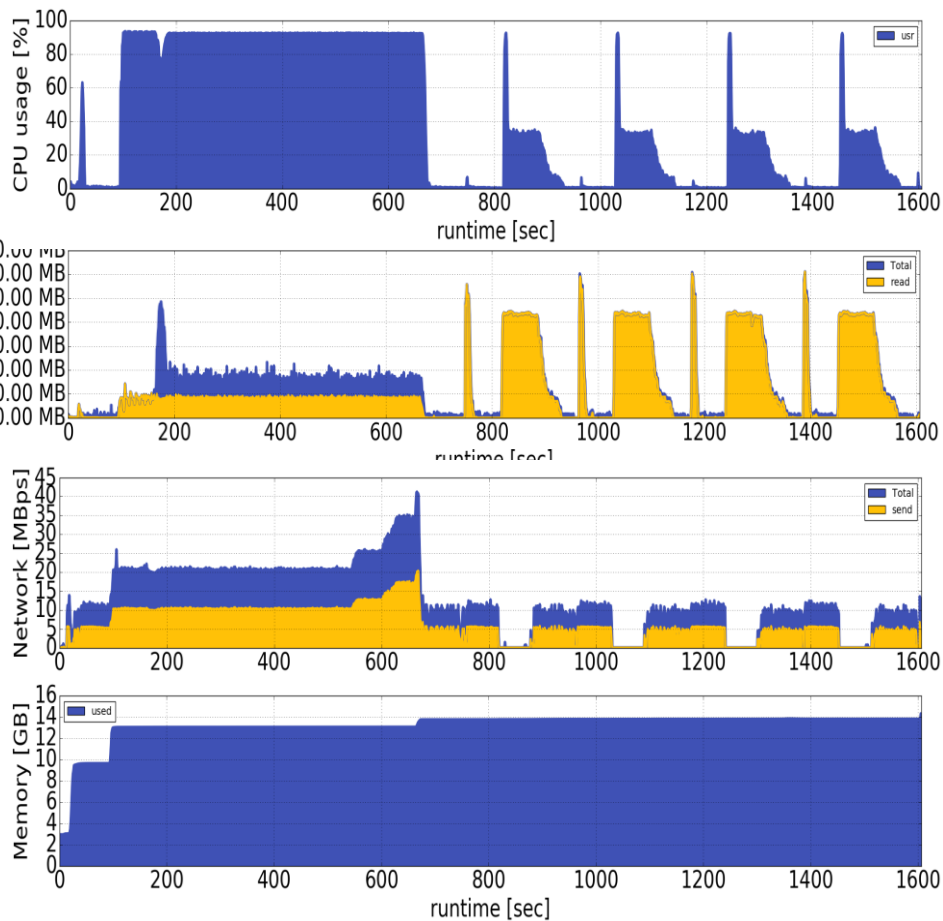
MMR



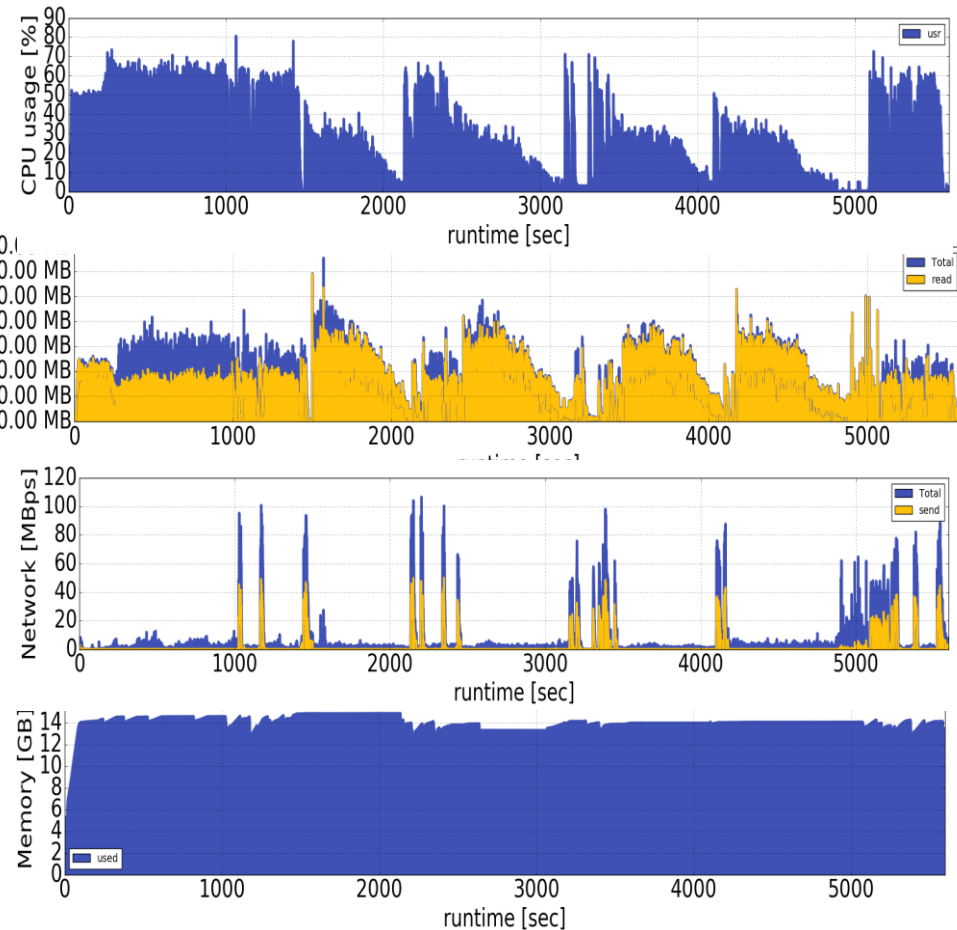
Apache Flink

Apache Spark

BGD – 0.8 Data Set - 6 Million Dimensions



Apache Flink



Apache Spark

- logistic function (also sigmoid function):

$$f_w(x) = \frac{1}{1 + e^{-w^T x}}$$

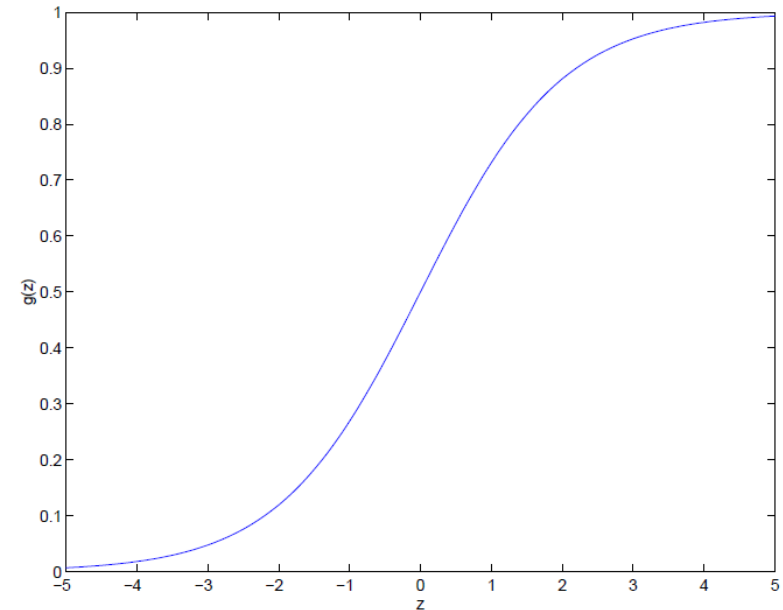
Now assume:

$$P(y = 0 | x ; w) = 1 - f_w(x)$$

$$P(y = 1 | x ; w) = f_w(x)$$

Or more compactly:

$$P(y | x ; w) = (f_w(x))^y (1 - f_w(x))^{1-y}$$



$$L(w) = P(y|x; w) = \prod_{i=1}^N P(y_i | x_i; w) = \prod_{i=1}^N (f_w(x_i))^{y_i} (1 - f_w(x_i))^{1-y_i}$$

$$l(w) = \log L(w) = \sum_{i=1}^N y_i \log f_w(x_i) + (1 - y_i) \log(1 - f_w(x_i)) + \lambda \|w\|^2$$

Batch Gradient Decent:

$$\frac{\partial}{\partial w_j} l(w) = \alpha \sum_{i=1}^N (y - f_w(x)) x_j + \frac{\lambda}{N} w_j$$

Stochastic Gradient Decent:

$$\frac{\partial}{\partial w_j} l(w) = \alpha (y - f_w(x)) x_j + \frac{\lambda}{N} w_j$$

- Need to choose learning rate α :

$$\frac{\partial}{\partial w_j} l(w) = \frac{\alpha}{t+t_0} (y - f_w(x))x_j + \frac{\lambda}{N} w_j$$

- Choose t_0 so that expected initial updates are comparable to the expected size of the weights
- Choose α :
 - Select a small subsample
 - Try various rates (e.g. 10, 1, 0.1, 0.01 ...)
 - Pick the α one that most reduces the objective function
 - Use α on full data set

Stopping criteria: How many iterations of SGD?

■ Early stopping with cross validation

- create validation set
- monitor cost function on validation set
- Stop when loss stops decreasing

■ Early stopping

- Extract two disjoint subsamples **A** and **B** of training data
- Train on **A**, stop by validating on **B**
- Number of epochs is an estimate of **k**
- Train for **k** epochs on the full data set

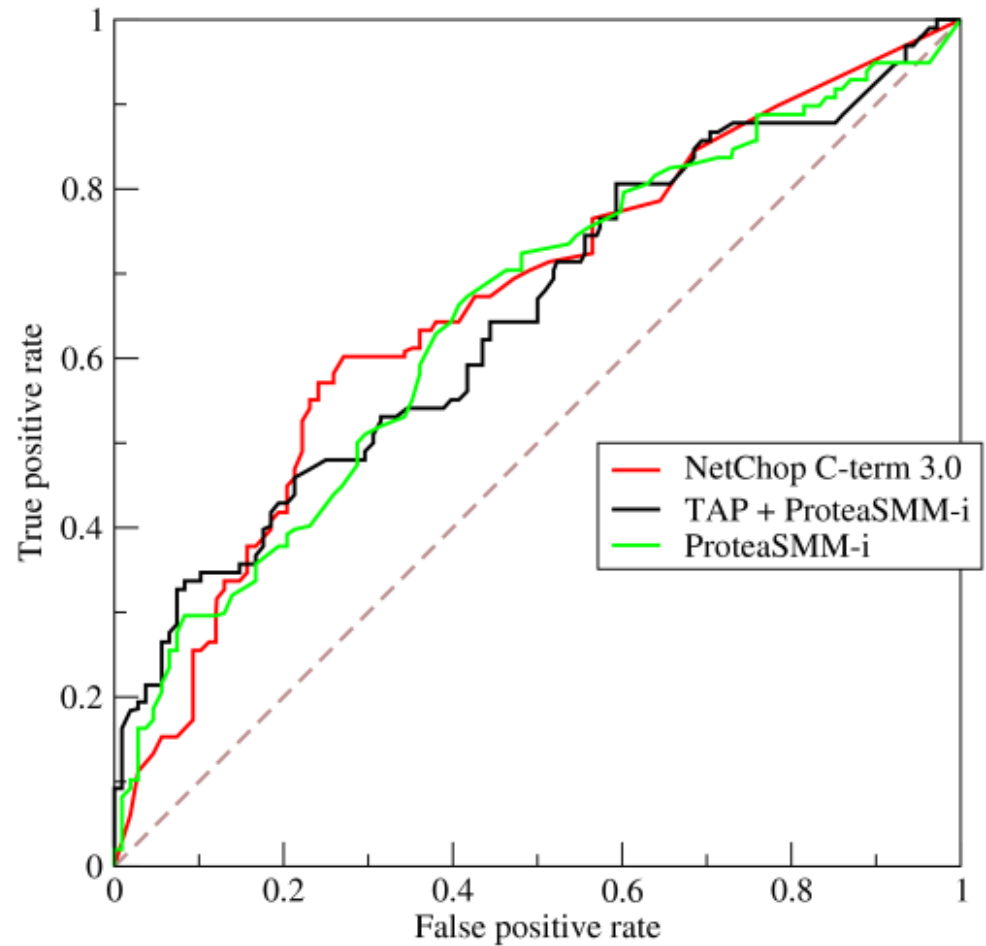
■ Accuracy: $TP / (P + N)$

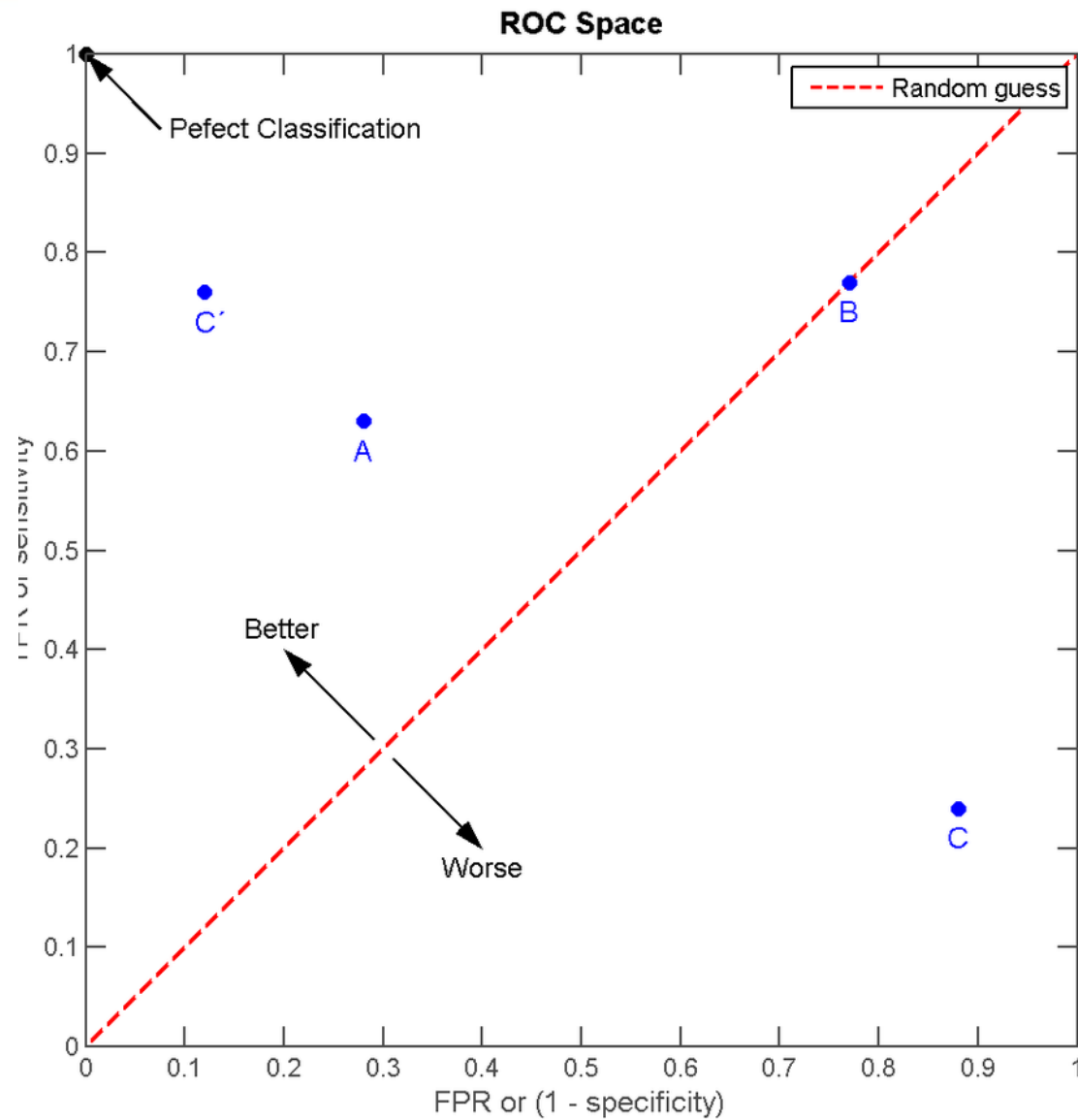
- Why could this be misleading?

	Y = 1	Y = 0
$f_w(x) = 1$	True positive	False positive (Type I error)
$f_w(x) = 0$	False negative (Type II error)	True negative
	Sensitivity = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$	Specificity = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$

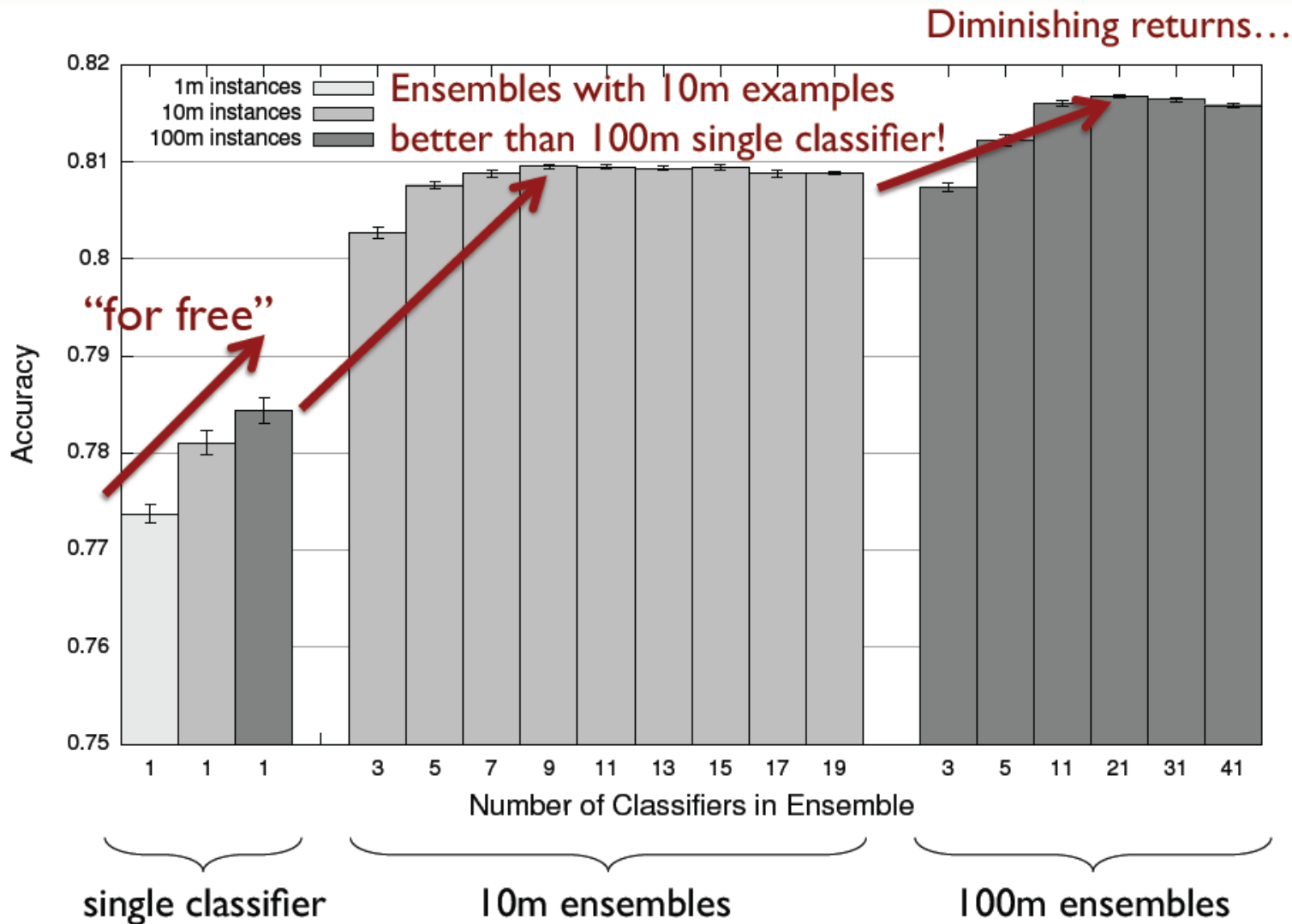
□ $TPR = TP / P$

□ $FPR = FP / N$



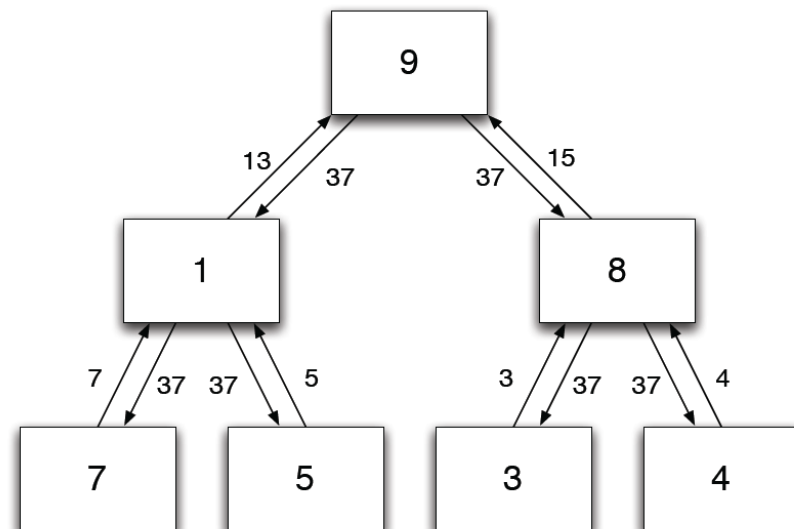


- Binary polarity classification: {positive, negative} sentiment
 - Independently interesting task
 - Illustrates end-to-end flow
 - Use the “emoticon trick” to gather data
- Data
 - Test: 500k positive/500k negative tweets from 9/1/2011
 - Training: {1m, 10m, 100m} instances from before (50/50 split)
- Features: Sliding window byte-4grams
- Models:
 - Logistic regression with SGD (L2 regularization)
 - Ensembles of various sizes (simple weighted voting)



* Large-Scale Machine Learning at Twitter Jimmy Lin, Alek Kocicz, SIGMOD 2012

- efficiently accumulates and broadcasts values across all nodes of a computation
 - every node starts with a number and ends up with the sum of the numbers across all the nodes
 - provides straightforward parallelization of gradient-based optimization algorithms such as gradient descent or L-BFGS
 - gradients are accumulated locally, and the global gradient is obtained by AllReduce.



Algorithm 2 Sketch of the proposed learning architecture

Require: Data split across nodes

for all nodes k **do**

\mathbf{w}^k = result of stochastic gradient descent on the data of node k

Compute the weighted average $\bar{\mathbf{w}}$ as in (2) using AllReduce.

Start a preconditioned L-BFGS optimization from $\bar{\mathbf{w}}$.

for $t = 1, \dots, T$ **do**

 Compute \mathbf{g}^k the (local batch) gradient of examples on node k .

 Compute $\mathbf{g} = \sum_{k=1}^m \mathbf{g}^k$ using AllReduce.

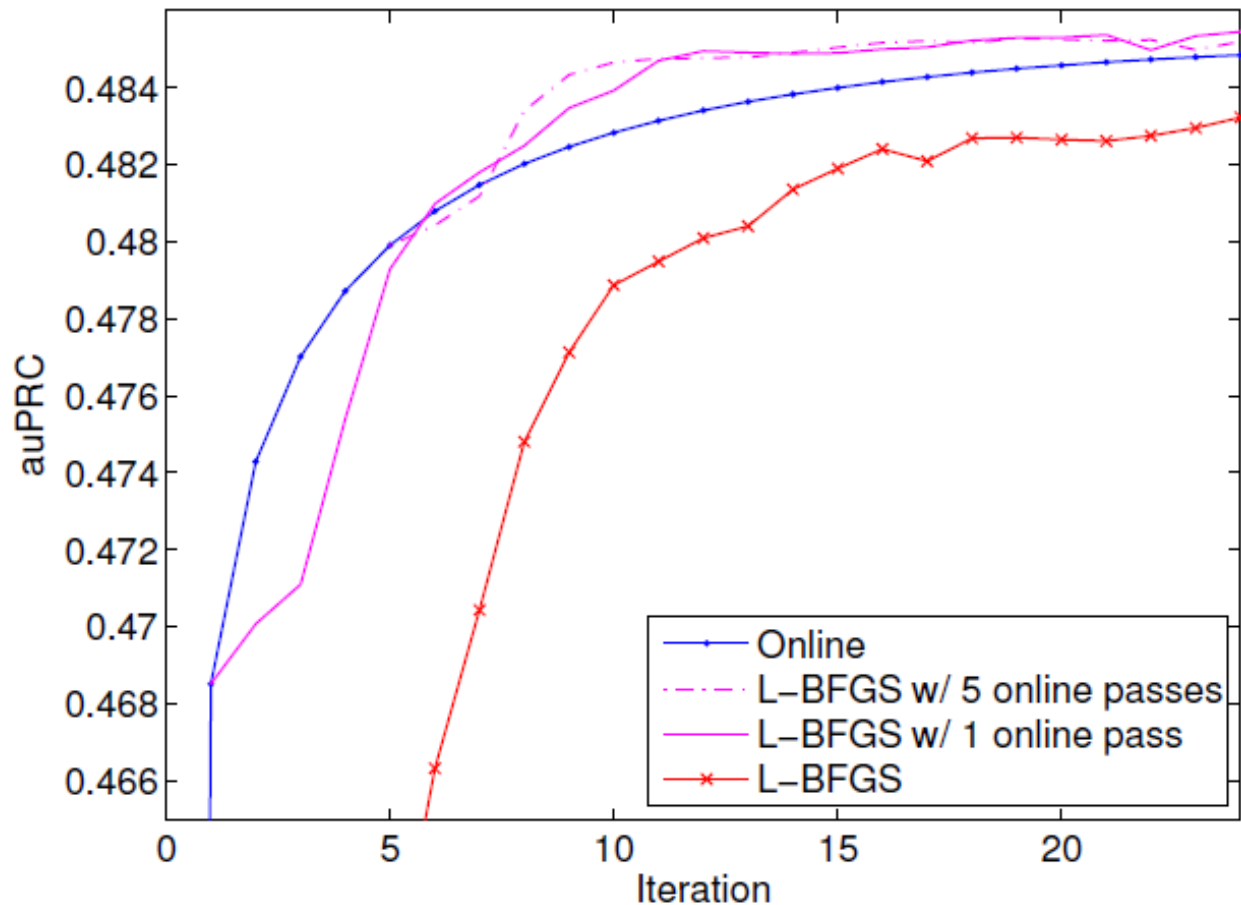
 Add the regularization part in the gradient.

 Take an L-BFGS step.

end for

end for

Display advertising



* A Reliable Effective Terascale Linear Learning System

Alekh Agarwal, Olivier Chapelle, Miroslav Dudik, John Langford

- easiest way to deal with a very large training set is to reduce it by subsampling
- Sometimes similar test errors can be achieved with smaller training sets and there is no need for large-scale learning.

	1%	10%	100%
auROC	0.8178	0.8301	0.8344
auPRC	0.4505	0.4753	0.4856
NLL	0.2654	0.2582	0.2554

- even if the drop does not appear large at a first sight, it can cause a substantial loss of revenue

- Deep Learning
 - learning representations
 - making use of unlabeled data
 - can be useful to do feature extraction
- Approximative Solutions
 - asynchronous algorithms (e.g HogWild)
 - parameter servers
- Decision Trees, Random Forests etc.
- Modern Hardware (GPUs, TPUs, ...)