# Not all ML algorithms go to distributed heaven

Alexey Zinoviev, Java/BigData Trainer,
Apache Ignite Committer

# Follow me

E-mail : zaleslaw.sin@gmail.com

Twitter : @zaleslaw @BigDataRussia

vk.com/big_data_russia **Big Data Russia**

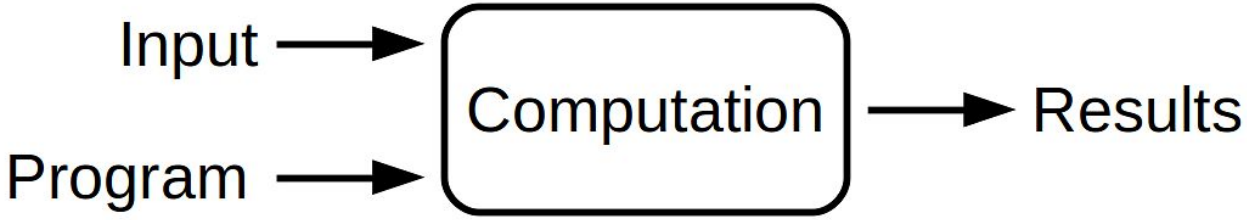
**+ Telegram** @bigdatarussia

vk.com/java_jvm **Java & JVM langs**

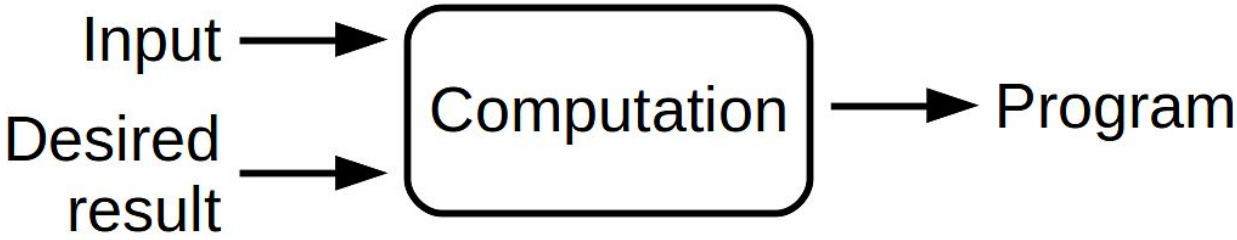
**+ Telegram** @javajvmlangs

# What is Machine Learning?

## Traditional programming




## Machine learning



# ML Task in math form (shortly)

$$y = f(x; w)$$

# ML Task in math form (by Vorontsov)

*X* - objects, *Y* - answers, $f : X \to Y$ arget function

$x_1, ..., x_n \subset X$ training sample

$y_i = y(x_i), i = 1, ..., n$ known answers

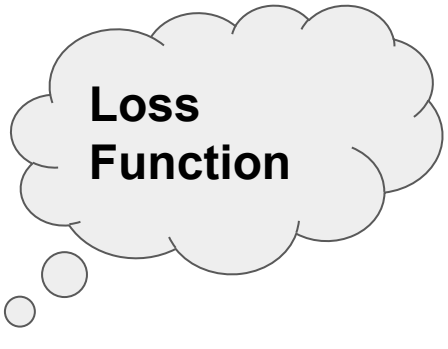
Find decision function $a : X \to Y, a \approx f$

# Model example [Linear Regression]

$$f(x) = \sum_{i=1}^{K} w_i \cdot x_i + w_0$$

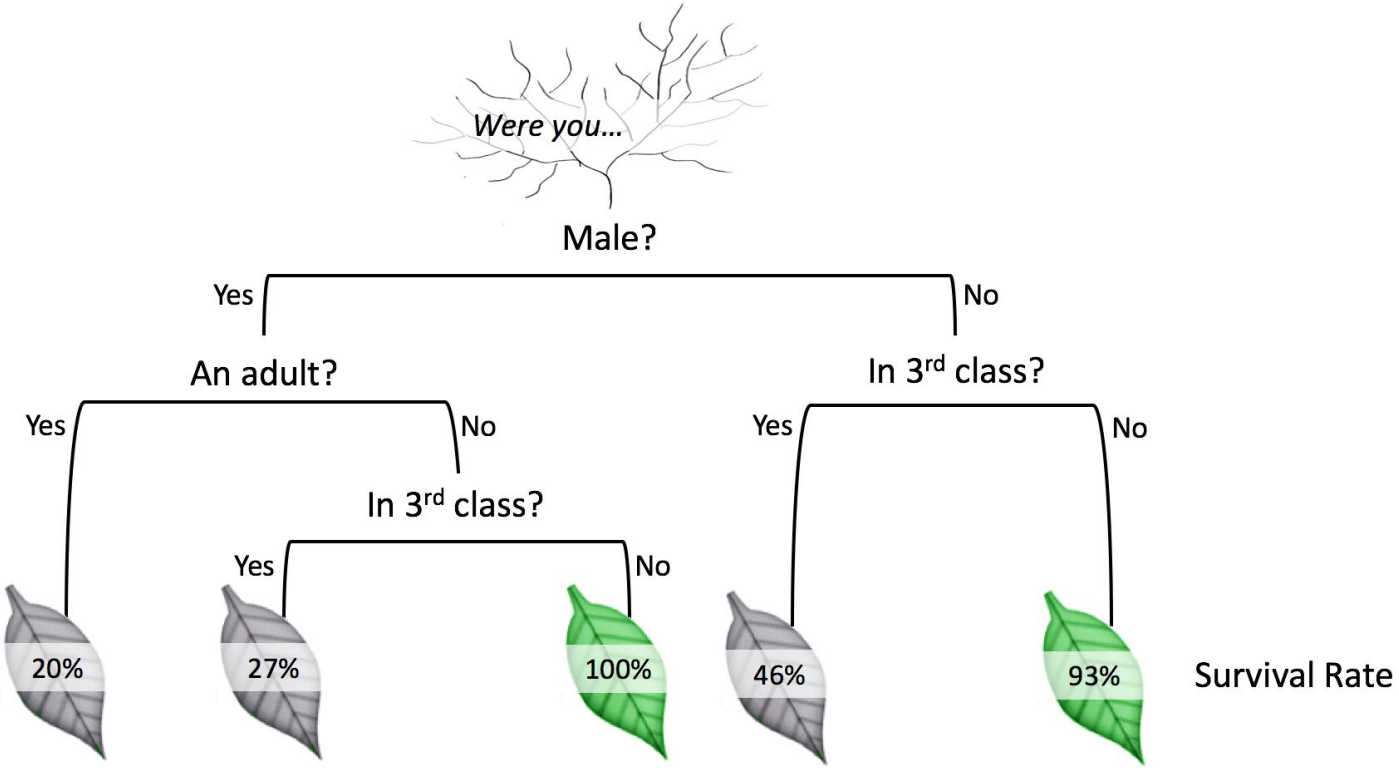
# Model example [Linear Regression]

$$f(x) = \sum_{i=1}^{K} w_i \cdot x_i + w_0$$




$$MSE = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

# Model example [Decision Tree]



# Distributed ML

# scikit-learn


Classification

Regression

Clustering

Neural Networks

Multiclass and multilabel algorithms

Preprocessing

NLP

Dimensionality reduction

Pipelines

Imputation of missing values

Model selection and evaluation

Model persistence

Ensemble methods

Tuning the hyper-parameters

# Take scikit-learn and distribute it!!!

# Take scikit-learn and distribute it!!!

# Take scikit-learn and distribute it!!!

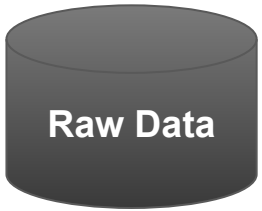


# Main issues of standard implementation

- It designed by scientists and described in papers
- Pseudo-code from papers copied and adopted in Python libs
- Typically, it’s implemented with one global while cycle
- Usually, it uses simple data structures like multi-dimensional arrays
- These data structures are located in shared memory on one computer
- A lot of algorithms has O(n^3) calculation complexity and higher
- As a result all these algorithms could be used for 10^2-10^5 observations effectively

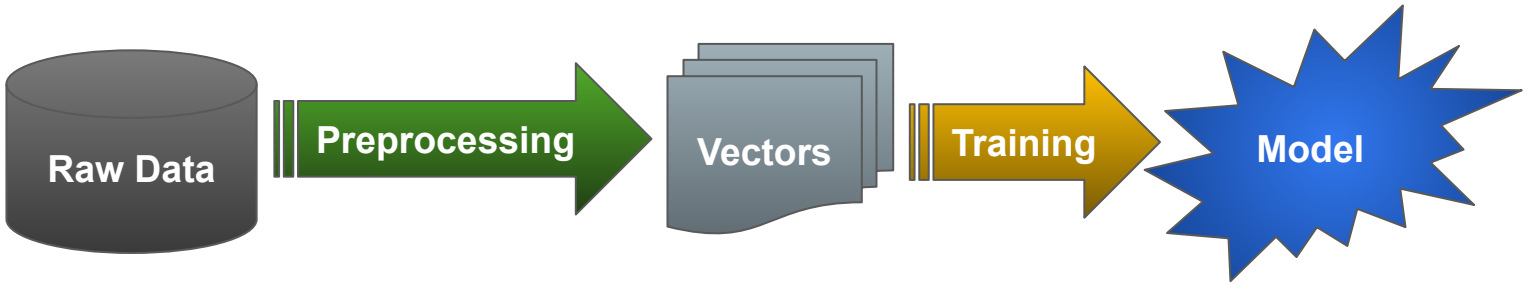
# Distributed Pipeline

# ML Pipeline

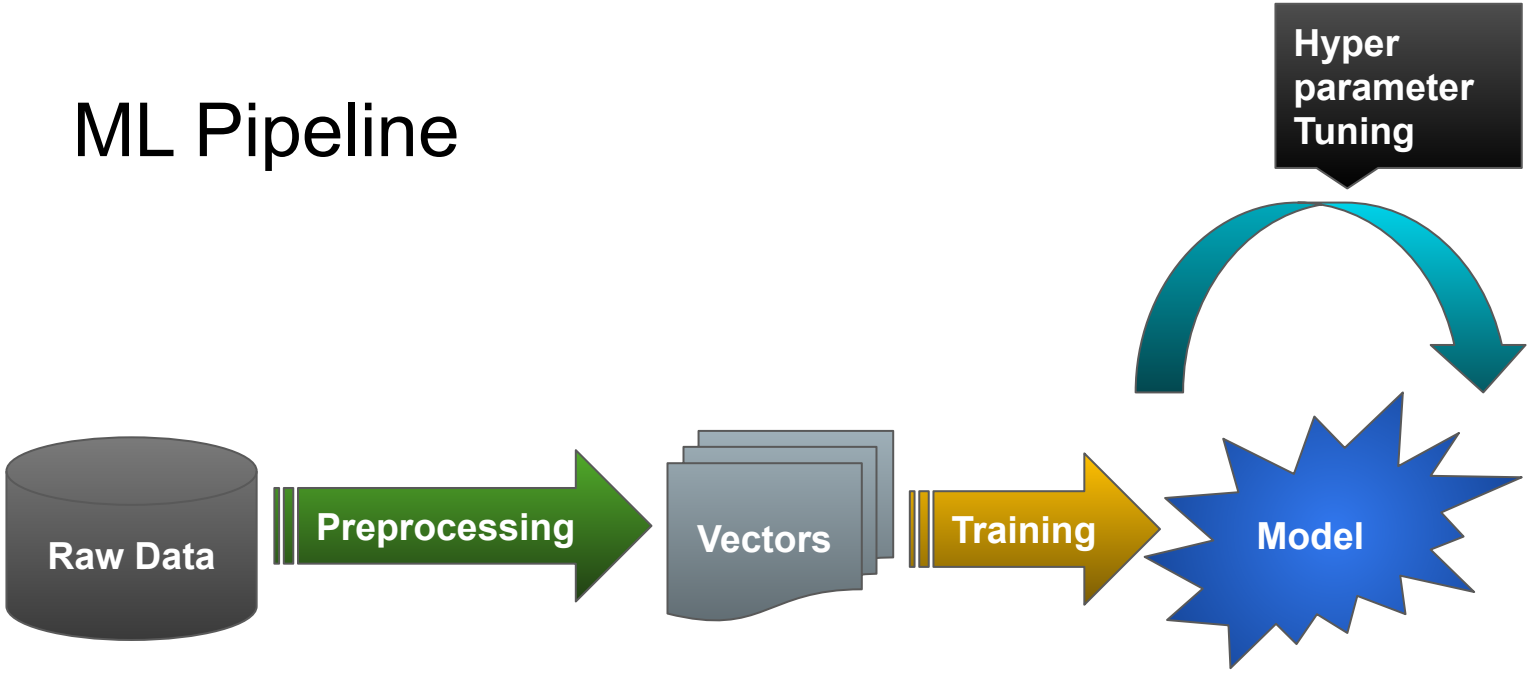


# ML Pipeline

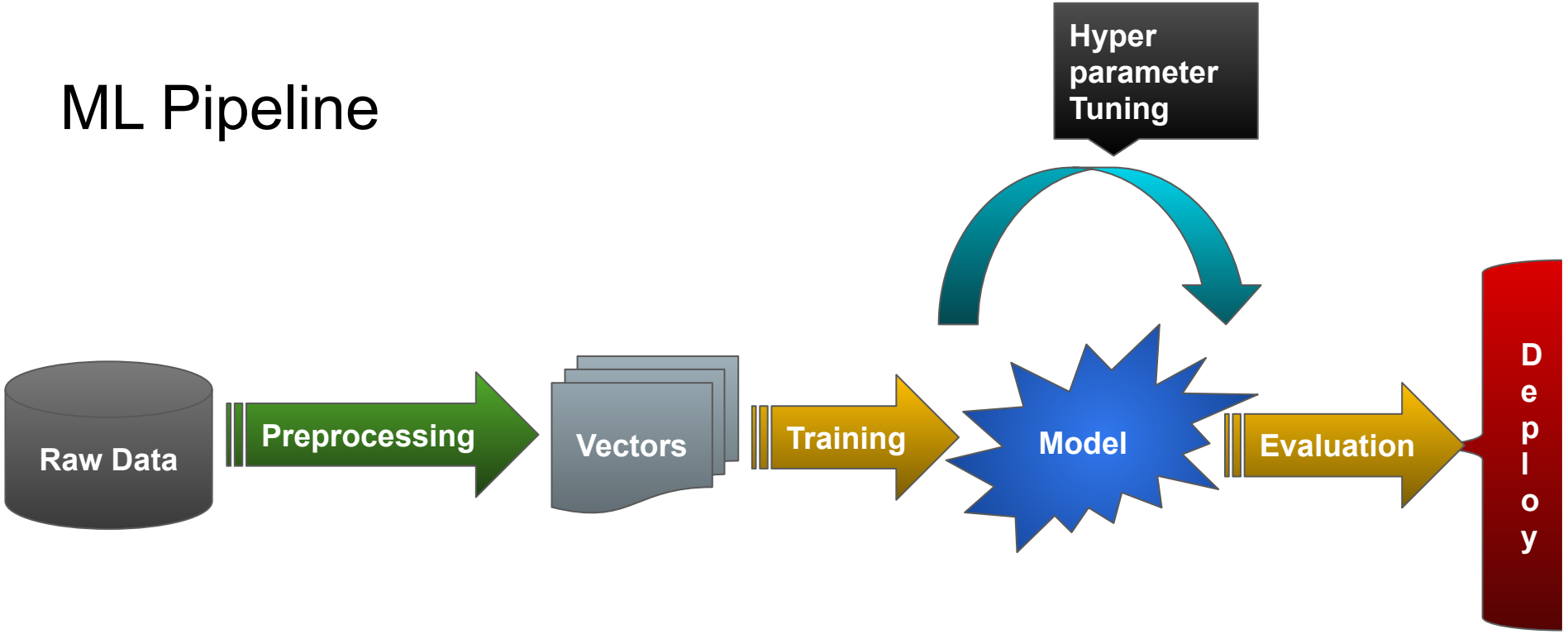
# ML Pipeline



# ML Pipeline


Hyper parameter Tuning
Raw Data
Preprocessing
Vectors
Training
Model

# ML Pipeline


Hyper parameter Tuning
Raw Data
Preprocessing
Vectors
Training
Model
Evaluation
Deploy

# What can be distributed in typical ML Pipeline

- Data primitives (datasets, RDD, dataframes and etc)
- Preprocessing
- Training
- Cross-Validation and another techniques of hyper-parameter tuning
- Prediction (if you need massive prediction, for example)
- Ensembles (like training trees in Random Forest)

# What can be distributed in typical ML Pipeline

| **Step** | **Apache Spark** | **Apache Ignite** |
|---|---|---|
| Dataset | distributed | distributed |
| Preprocessing | distributed | distributed |
| Training | distributed | distributed |
| Prediction | distributed | distributed |
| Evaluation | distributed | distributed (since 2.8) |
| Hyper-parameter tuning | parallel | parallel (since 2.8) |
| Online Learning | distributed in 3 algorithms | distributed |
| Ensembles | for RF* | distributed/parallel |

# Distributed Data Structures

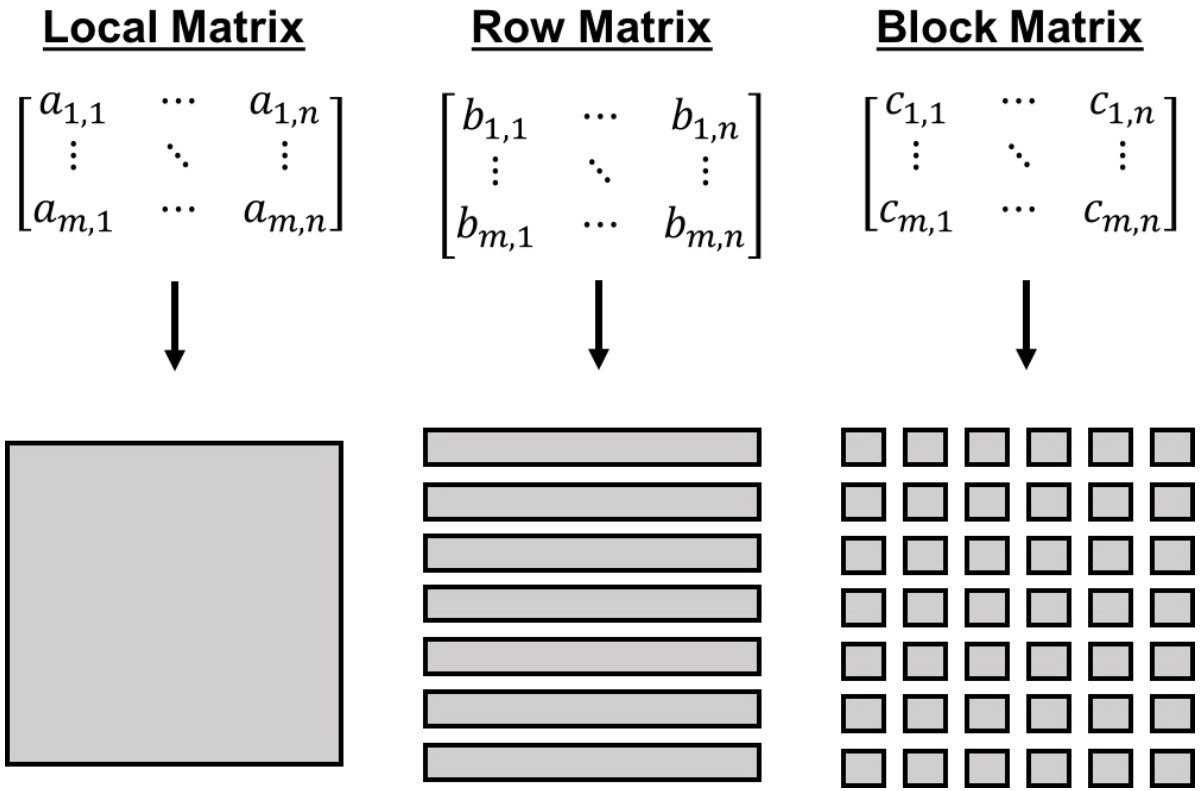
# What can be distributed in typical ML Pipeline

- Horizontal fragmentation wherein subsets of instances are stored at different sites (distributed by rows)
- Vertical fragmentation wherein subsets of attributes of instances are stored at different sites (distributed by columns)
- Cell fragmentation - mixed approach of two above (distributed by row and column ranges)
- Improvement  with data collocation based on some hypothesis (geographic factor, for example)

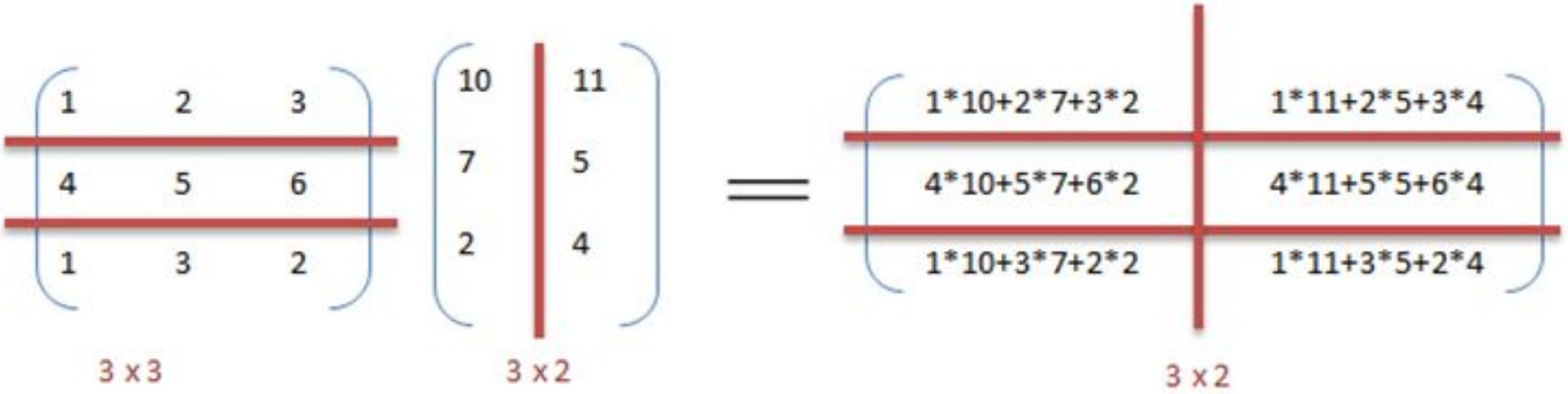
# The main problem with classic ML algorithm

They are designed to learn from a unique data set

# Popular Matrix Representations



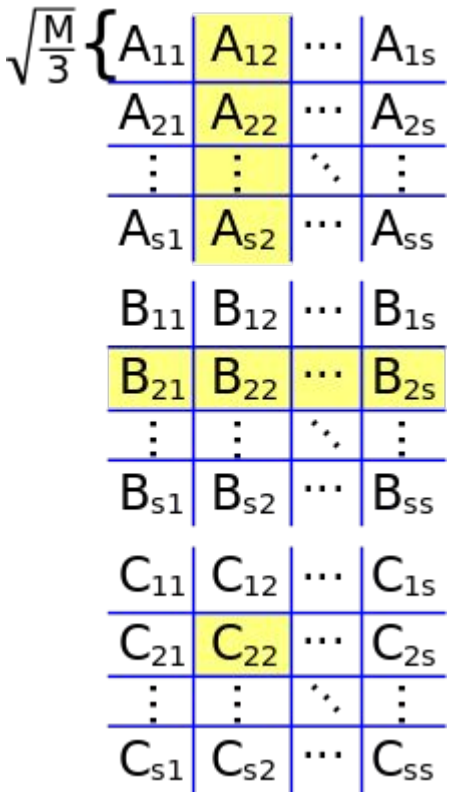
# How to multiply distributed matrices?



# How to multiply distributed matrices?

- Rows * columns (deliver columns to rows in shuffle phase)
- Block * block (Cannon's algorithm)
- SUMMA: Scalable Universal Matrix Multiplication Algorithm
- Dimension Independent Matrix Square using MapReduce (DIMSUM) (Spark PR)
- OverSketch: Approximate Matrix Multiplication for the Cloud
- Polar Coded Distributed Matrix Multiplication

# Block multiplication vs 2.5D LU Decomposition

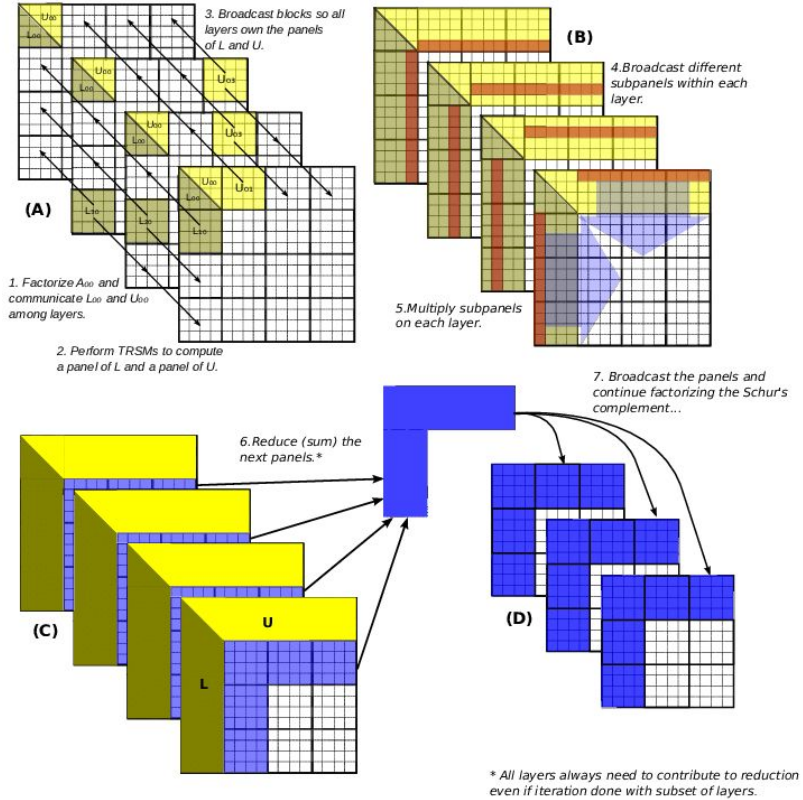


# Block multiplication vs 2.5D LU Decomposition

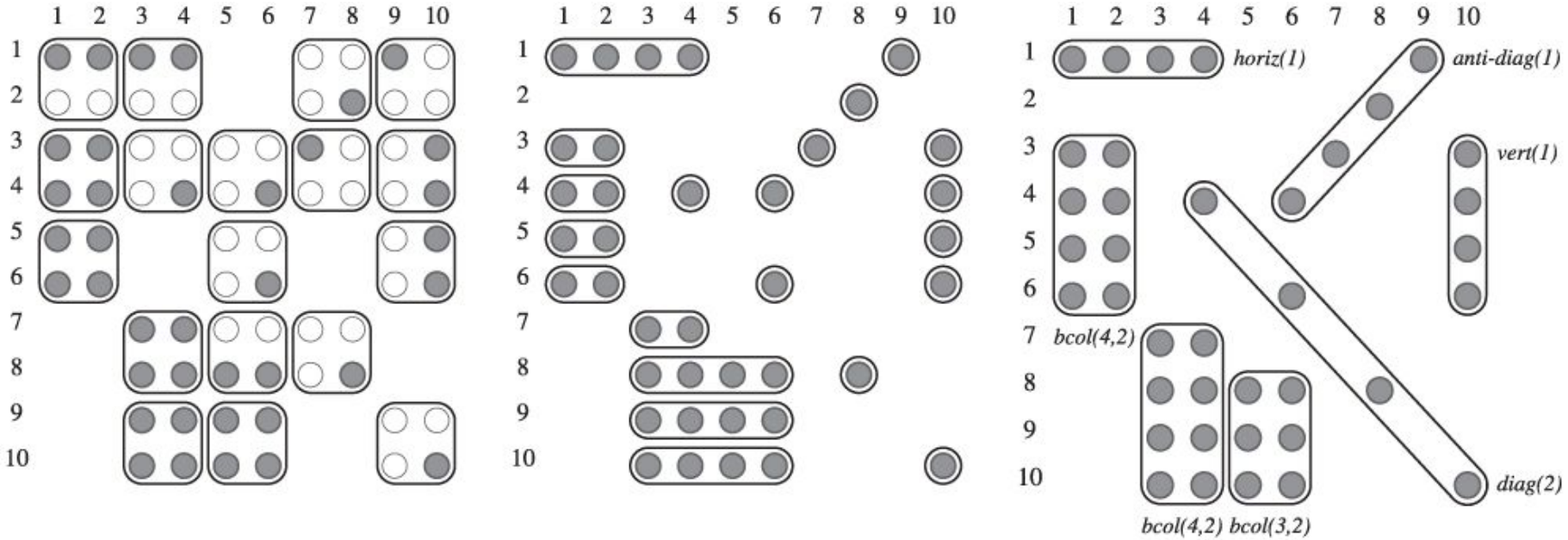
$$\sqrt{\frac{M}{3}}\left\{\begin{array}{|c|c|c|c|}\hline A_{11} & A_{12} & \cdots & A_{1s} \\ \hline A_{21} & A_{22} & \cdots & A_{2s} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{s1} & A_{s2} & \cdots & A_{ss} \\ \hline \end{array}\right.$$

$$\begin{array}{|c|c|c|c|}\hline B_{11} & B_{12} & \cdots & B_{1s} \\ \hline B_{21} & B_{22} & \cdots & B_{2s} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline B_{s1} & B_{s2} & \cdots & B_{ss} \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|c|}\hline C_{11} & C_{12} & \cdots & C_{1s} \\ \hline C_{21} & C_{22} & \cdots & C_{2s} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline C_{s1} & C_{s2} & \cdots & C_{ss} \\ \hline \end{array}$$



# Popular Matrix Representations




(a) BCSR. (b) VBL. (c) CSX.

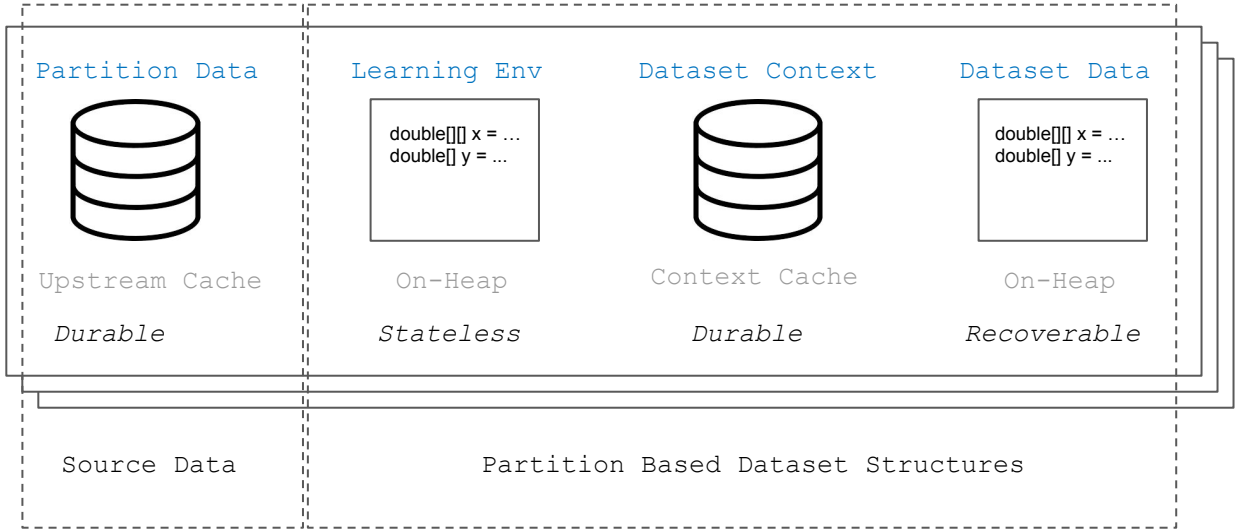
# Reasons to avoid distributed algebra

1. A lot of different Matrix/Vector format
2. Bad performance results for SGD-based algorithms
3. A lot of data are shuffled with Sparse Block Distributed Matrices
4. Extension to algorithms that are not based on Linear Algebra
5. Slow but direct extension of Vector/Matrix distributed operations
6. Illusion that a lot of algorithms could be easily adopted (like DBScan)

# Partition-based dataset

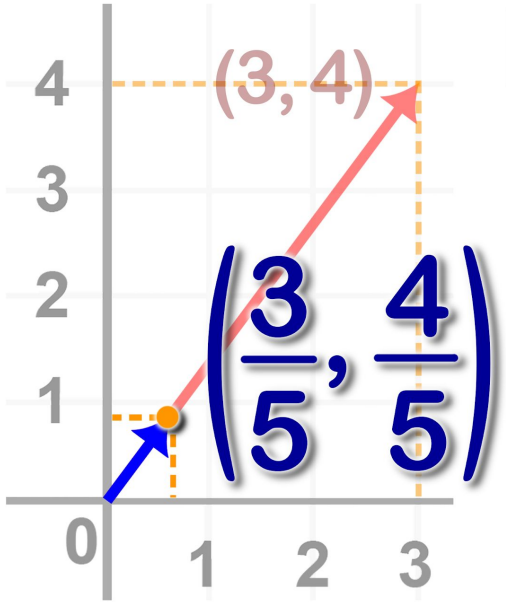
```
Dataset dataset = … // Partition based dataset, internal API

dataset.compute((env, ctx, data) -> map(...), (r1, r2) -> reduce(...))
```



# Preprocessors

# Normalize vector v to L2 norm



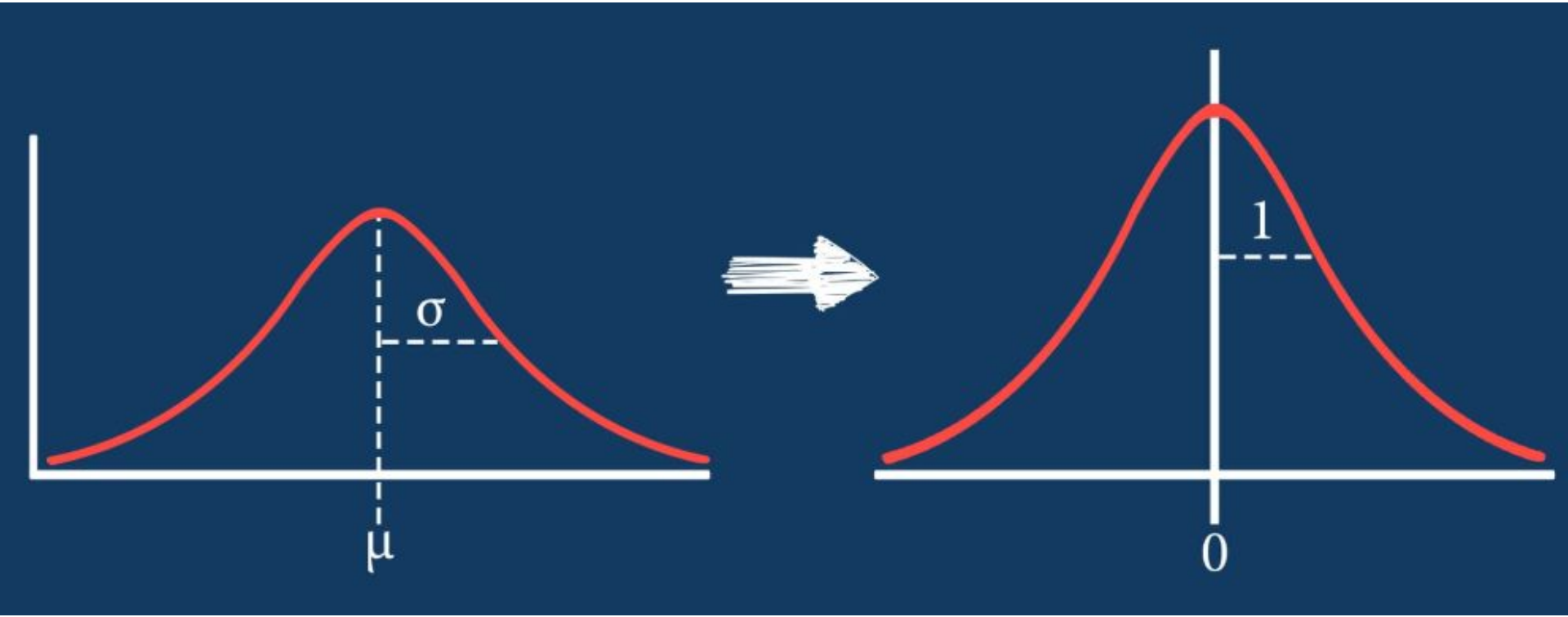

$$|v| = \sqrt{3^2 + 4^2}$$
$$= \sqrt{9 + 16}$$
$$= \sqrt{25}$$
$$= 5$$

$$u = \frac{(3, 4)}{5}$$

# Distributed Vector Normalization

1. Define the p (vector norm) $\|\mathbf{x}\|_p := \left( \sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$
2. Run normalization of each vector on each partition in Map phase

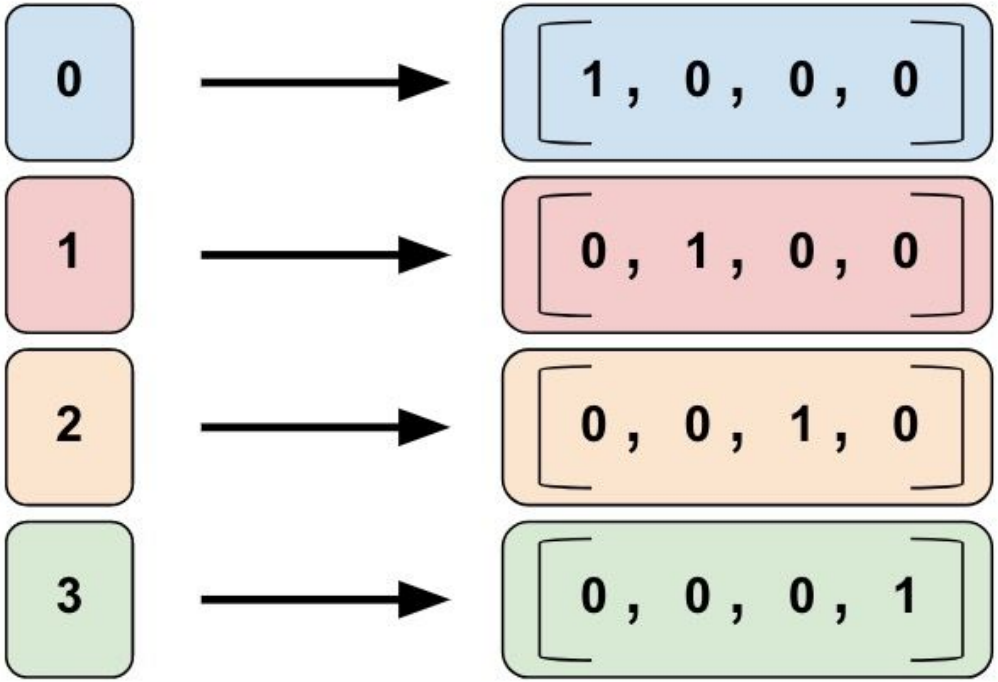
# Standard Scaling



# Distributed Standard Scaling

1. Collect Standard Scaling statistics (mean, variance)
    - one Map-Reduce step to collect
2. Scale each row using statistics (or produced model)
    - one Map step to transform

# One-Hot Encoding



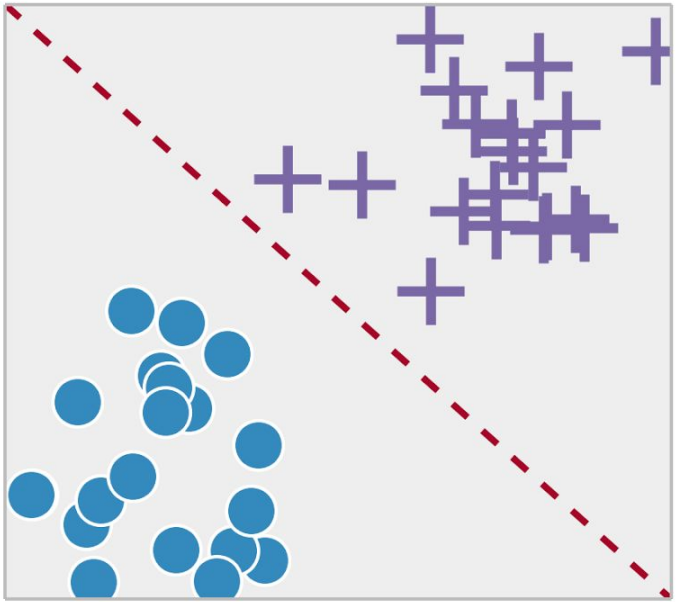
# Distributed Encoding

1. Collect Encoding statistics (Categories frequencies)
    - one Map-Reduce step to collect
2. Transform each row using statistics (or produced model)
    - one Map step to transform
    - NOTE: it adds k-1 new columns for each categorial feature, where k is amount of categories

# ML Algorithms

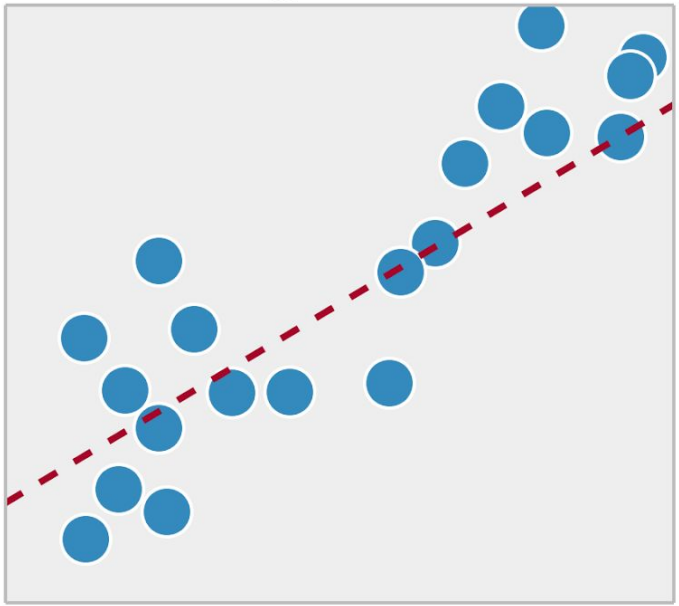
# Classification algorithms

- Logistic Regression
- SVM
- KNN
- ANN
- Decision trees
- Random Forest



# Regression algorithms

- KNN Regression
- Linear Regression
- Decision tree regression
- Random forest regression
- Gradient-boosted tree regression

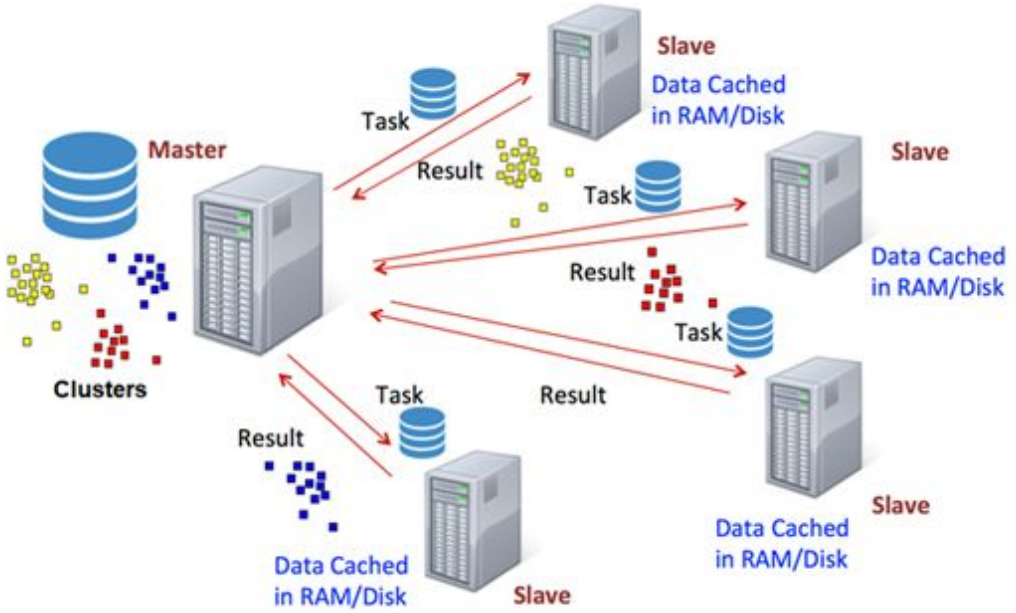


# Distributed approaches to design ML algorithm

1. **Data-Parallelism:** The data is partitioned and distributed onto the different workers. Each worker typically updates all parameters based on its share of the data
2. **Model-Parallelism:** Each worker has access to the entire dataset but only updates a subset of the parameters at a time
3. **Combination** of two above

# The iterative-convergent nature of ML programs

1. Find or prepare something locally
2. Repeat it a few times (`locIterations++`)
3. Reduce results
4. Make next step (`globalIterations++`)
5. Check convergence



# Shortly, Distributed ML Training can be implemented as an ...

Iterative MapReduce algorithm in-memory or on disk

# Distributed approaches to design ML algorithm

1. **Data-Parallelism:** The data is partitioned and distributed onto the different workers. Each worker typically updates all parameters based on its share of the data
2. **Model-Parallelism:** Each worker has access to the entire dataset but only updates a subset of the parameters at a time
3. **Combination** of two above

# Potential acceleration points in Iterative MR

1. Reduce the amount of global iterations
2. Reduce the time of one global iteration
3. Reduce the size of shuffled data pushed through network

# ML algorithms that are easy to scale

1. Linear Regression via SGD
2. Linear Regression via LSQR
3. K-Means
4. Linear SVM
5. KNN
6. Logistic Regression

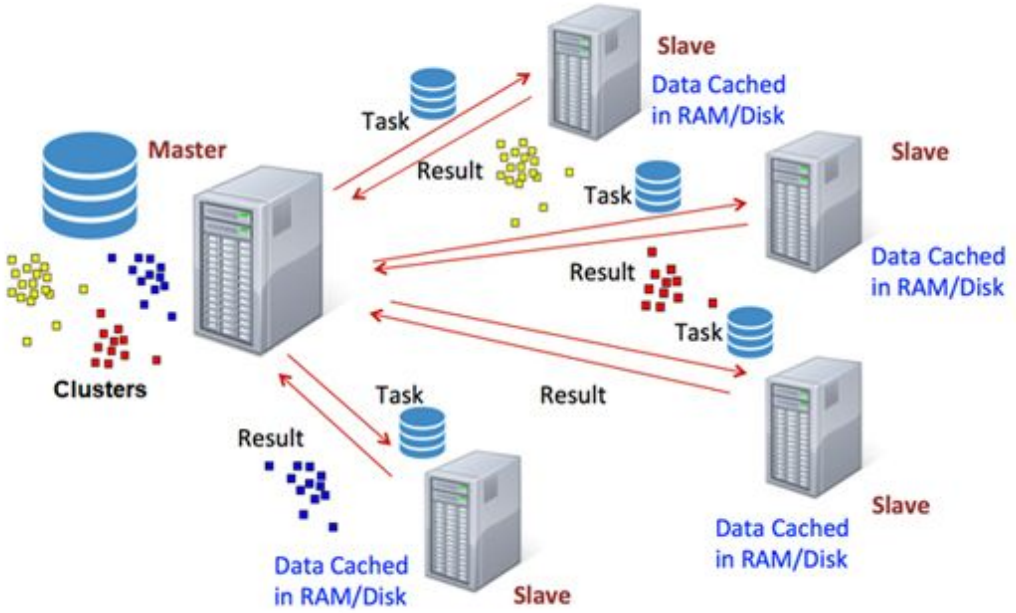
# They are not designed for distributed world

1. PCA (matrix calculations)
2. DBSCAN
3. Topic Modeling (text analysis)

# Linear Regression via LSQR

# The iterative-convergent nature of ML programs

1. Find or prepare something locally
2. Repeat it a few times (`locIterations++`)
3. Reduce results
4. Make next step (`globalIterations++`)
5. Check convergence

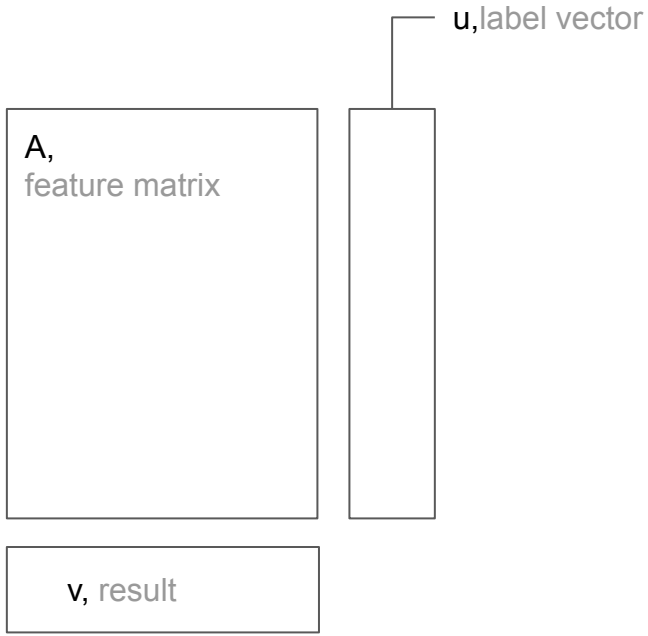


# Linear Regression with MR approach

## Golub-Kahan-Lanczos Bidiagonalization Procedure

core of LSQR linear regression trainer

$$
\begin{aligned}
&for\ k = 1, 2..n \\
&\quad u_k = Av_k - \beta_{k-1}u_{k-1} \\
&\quad a_k = \|u_k\|_2 \\
&\quad u_k = u_k / a_k \\
&\quad v_{k+1} = A^* u_k - a_k v_k \\
&\quad \beta = \|v_{k+1}\|_2 \\
&\quad v_{k+1} = v_{k+1} / \beta_k \\
&end.
\end{aligned}
$$



# Linear Regression with MR approach

## Golub-Kahan-Lanczos Bidiagonalization Procedure

core of LSQR linear regression trainer

$$for\ k = 1, 2..n$$

MapReduce

$$(u_p)_k = (A_p)v_k - \beta_{k-1}(u_p)_{k-1} \quad for\ p = 1, 2..parts$$

$$a_k = \left\| \sum_{p=1}^{parts} \|(u_p)_k\|_2 \right\|_2$$

MapReduce

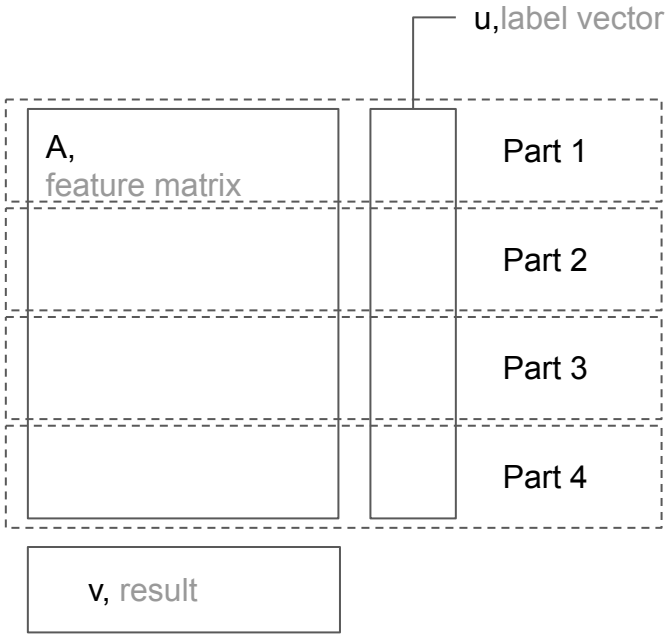
$$(u_p)_k = (u_p)_k / a_k$$

$$v_{k+1} = \sum_{p=1}^{parts} (A_p)^*(u_p)_k - a_k v_k$$

$$\beta = \|v_{k+1}\|_2$$

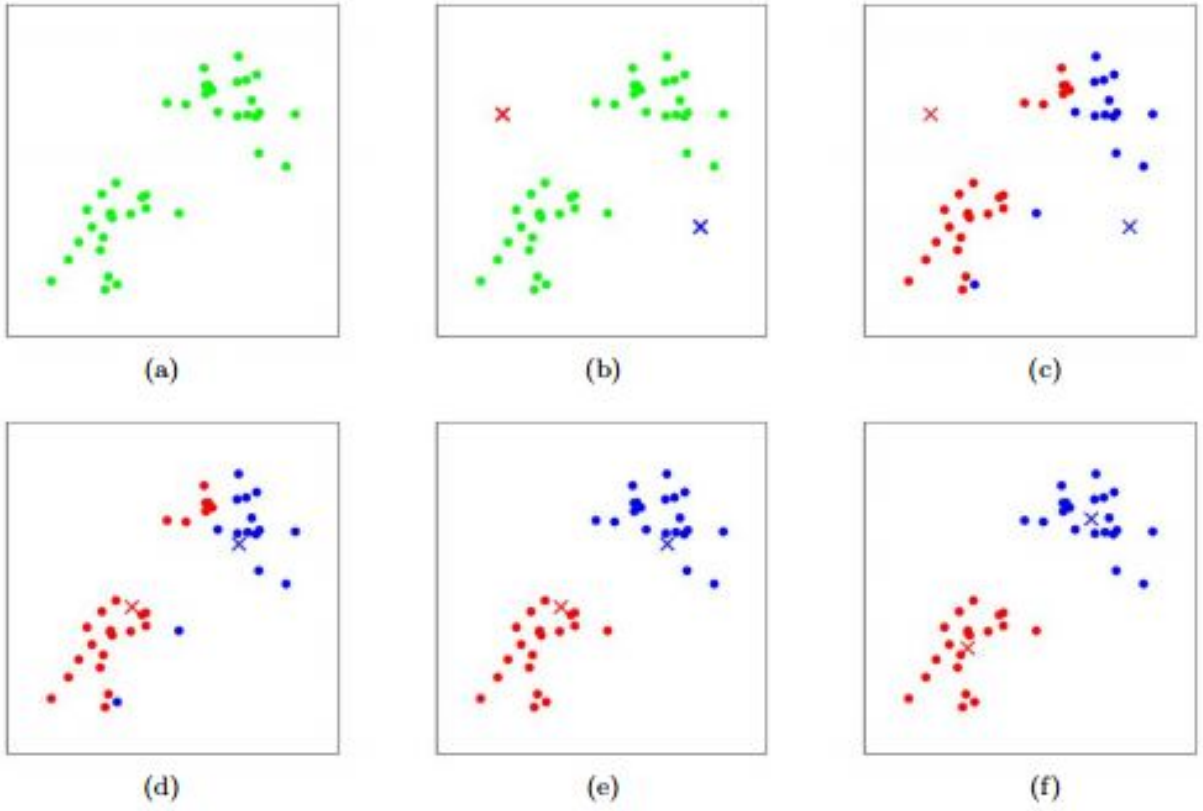
$$v_{k+1} = v_{k+1}/\beta_k$$

$$end.$$



# Clustering (K-Means)

# K-Means in 6 steps



# Distributed K-Means (First version)

1. Fix k
2. Initialize k centers
3. Clusterize points locally on each partition (local K-Means)
4. Push to reducer *{ centroid, amount of points, cluster diameter }*
5. Join on reducer clusters

# Distributed K-Means (Second version)

1. Fix k  & Initialize k cluster centers
2. Spread them among cluster nodes
3. Calculates distances locally on every node
4. Form stat for every cluster center on every node
5. Merge stats on Reducer
6. Recalculate k cluster centers and repeat 3-7 before convergence

# SGD

# Linear Regression Model

$$f(x) = \sum_{i=1}^{K} w_i \cdot x_i + w_0$$

# Target function for Linear Regression

$$f(x) = \sum_{i=1}^{K} w_i \cdot x_i + w_0$$

$$MSE(f, X, Y) = \frac{1}{N} \sum_{i=1}^{N} (f(X_i) - Y(X_i))^2 \to min$$

# Loss Function

$$f(x) = \sum_{i=1}^{K} w_i \cdot x_i + w_0$$

$$MSE(f, X, Y) = \frac{1}{N} \sum_{i=1}^{N} (f(X_i) - Y(X_i))^2 \to min$$

$$\nabla MSE(f, X, Y) = \left[ \frac{\partial MSE(f, X, Y)}{\partial \omega_j} \right]_{j=0...K} = \left[ \frac{2}{N} \sum_{i=1}^{N} x_{ij} (f(x_i) - y(x_i)) \right]_{j=1...K}$$

# Distributed Gradient

$$f(x) = \sum_{i=1}^{K} w_i \cdot x_i + w_0$$

$$MSE(f, X, Y) = \frac{1}{N} \sum_{i=1}^{N} (f(X_i) - Y(X_i))^2 \to min$$

$$\nabla MSE(f, X, Y) = \left[ \frac{\partial MSE(f, X, Y)}{\partial \omega_j} \right]_{j=0...K} = \left[ \frac{2}{N} \sum_{i=1}^{N} x_{ij} (f(x_i) - y(x_i)) \right]_{j=1...K}$$

$$\nabla [\sum_{1}^{n} (y - \hat{y})^2] = \nabla [\sum_{1}^{n_1} (y - \hat{y})^2] + \nabla [\sum_{n_1}^{n_2} (y - \hat{y})^2] + ... + \nabla [\sum_{n_{k-1}}^{n} (y - \hat{y})^2]$$

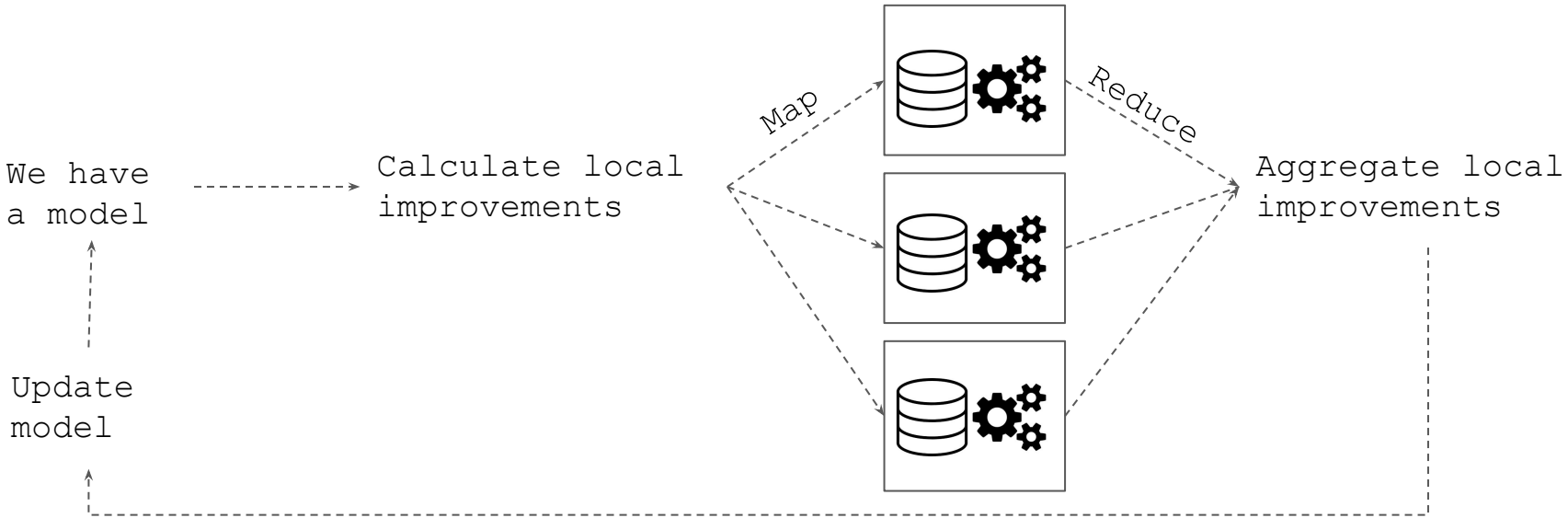
# SGD Pseudocode

```
def SGD(X, Y, Loss, GradLoss, W0, s):
    W = W0
    lastLoss = Double.Inf
     for i = 0 .. maxIterations:
        W = W - s * GradLoss(W, X, Y)
        currentLoss = Loss(Model(W), X, Y)
        if abs(currentLoss - lastLoss) > eps:
            lastLoss = currentLoss
        else:
            break
    return Model(W)
```

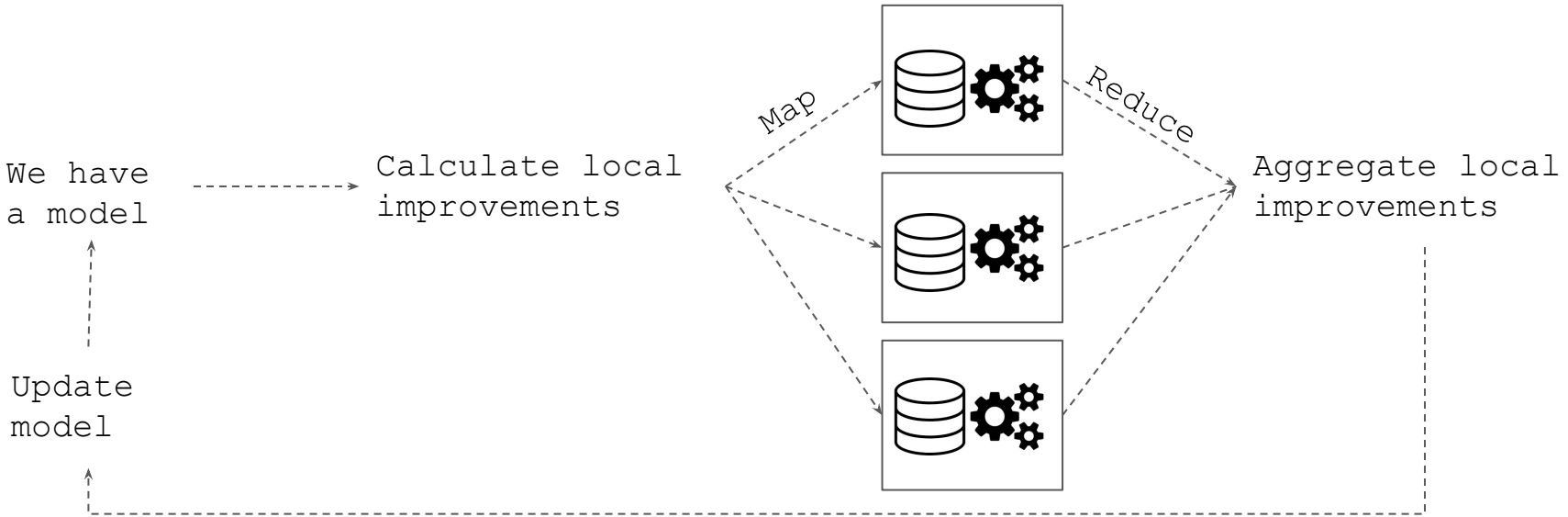
# What can be distributed?

```
def SGD(X, Y, Loss, GradLoss, W0, s):
    W = W0
    lastLoss = Double.Inf
     for i = 0 .. maxIterations:
        W = W - s * GradLoss(W, X, Y)
        currentLoss = Loss(Model(W), X, Y)
        if abs(currentLoss - lastLoss) > eps:
            lastLoss = currentLoss
        else:
            break
    return Model(W)
```

# MapReduce approach



# MapReduce approach




$$\nabla[\sum_{1}^{n}(y-\hat{y})^2] = \nabla[\sum_{1}^{n_1}(y-\hat{y})^2] + \nabla[\sum_{n_1}^{n_2}(y-\hat{y})^2] + \ldots + \nabla[\sum_{n_{k-1}}^{n}(y-\hat{y})^2]$$

# Naive Apache Ignite implementation

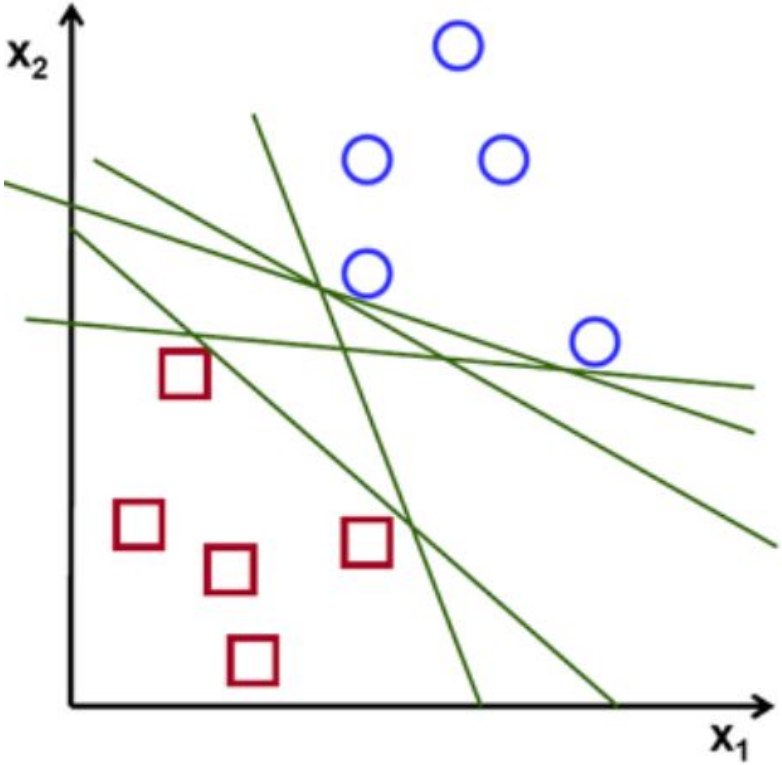
```
try (Dataset<EmptyContext, SimpleLabeledDatasetData> dataset = datasetBuilder.build(
    LearningEnvironmentBuilder.defaultBuilder(),new EmptyContextBuilder<>(),
    new SimpleLabeledDatasetDataBuilder<>(vectorizer)
)) {
    int datasetSize = sizeOf(dataset);
    double error = computeMSE(model, dataset);
    int i = 0;
    while(error > minError && i < maxIterations) {
      Vector grad = dataset.compute(
           data -> computeLocalGrad(model, data), // map phase
           (left, right) -> left.plus(right)       // reduce phase
      );
      grad = grad.times(2.0).divide(datasetSize); // normalize part of grad
      Vector newWeights = model.weights().minus(grad.times(gradStep)); // add anti-gradient
      model.setWeights(newWeights);
      error = computeMSE(model, dataset);
      i++;
    }
}
```

# Distributed Gradient

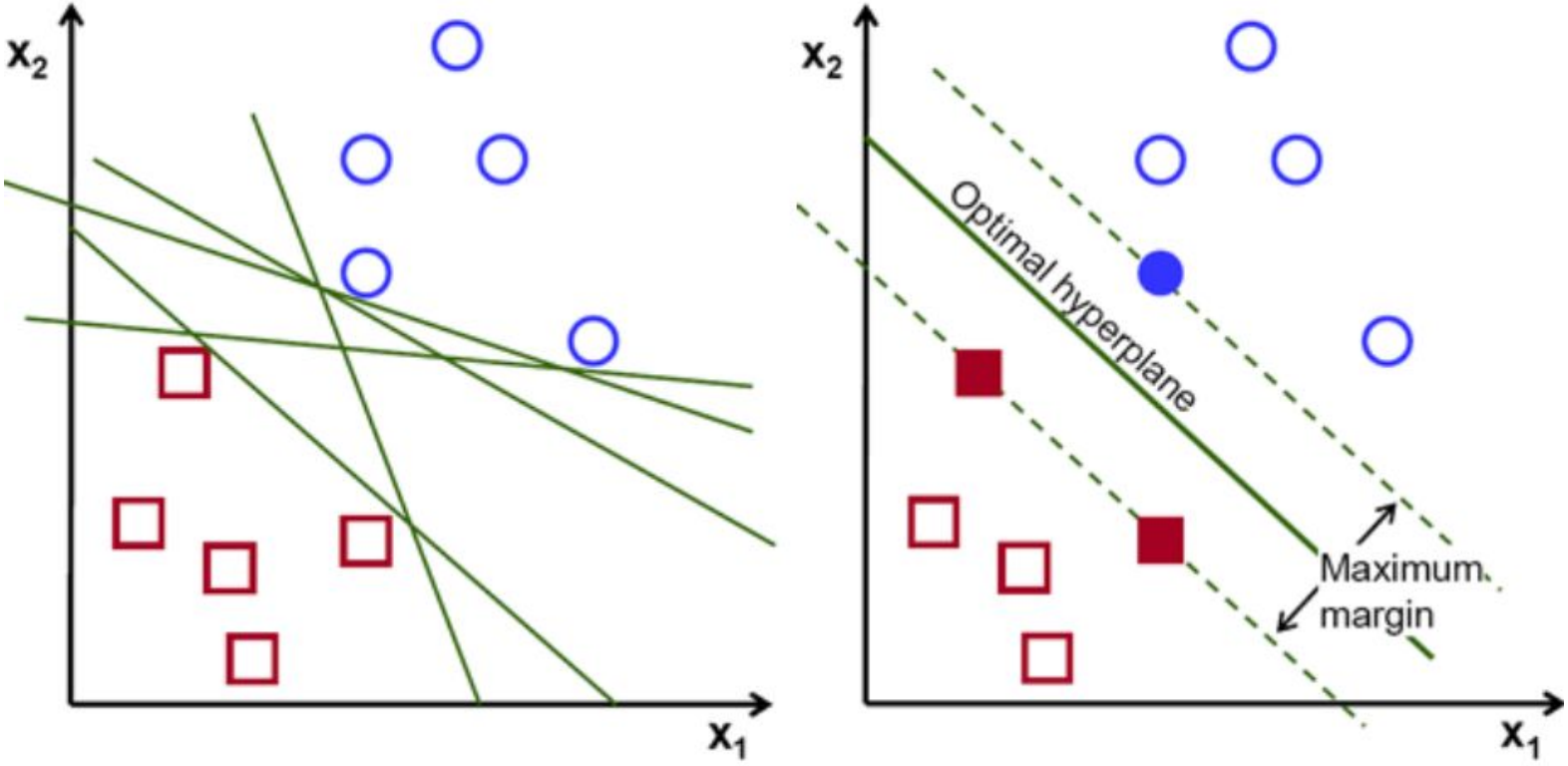
| | loss function $L(\mathbf{w}; \mathbf{x}, y)$ | gradient or sub-gradient |
|---|---|---|
| hinge loss | $\max\{0, 1 - y\mathbf{w}^T\mathbf{x}\}, \quad y \in \{-1, +1\}$ | $\begin{cases} -y \cdot \mathbf{x} & \text{if } y\mathbf{w}^T\mathbf{x} < 1, \\ 0 & \text{otherwise.} \end{cases}$ |
| logistic loss | $\log(1 + \exp(-y\mathbf{w}^T\mathbf{x})), \quad y \in \{-1, +1\}$ | $-y\left(1 - \frac{1}{1+\exp(-y\mathbf{w}^T\mathbf{x})}\right) \cdot \mathbf{x}$ |
| squared loss | $\frac{1}{2}(\mathbf{w}^T\mathbf{x} - y)^2, \quad y \in \mathbb{R}$ | $(\mathbf{w}^T\mathbf{x} - y) \cdot \mathbf{x}$ |

# SVM

# Linear SVM: it’s easy!



# Linear SVM: it's easy!



# Distributed Soft-margin Linear SVM

$$\min_{\mathbf{w}\in\mathbb{R}^d} \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} l_i\left(\mathbf{w}^T\mathbf{x}_i\right)$$

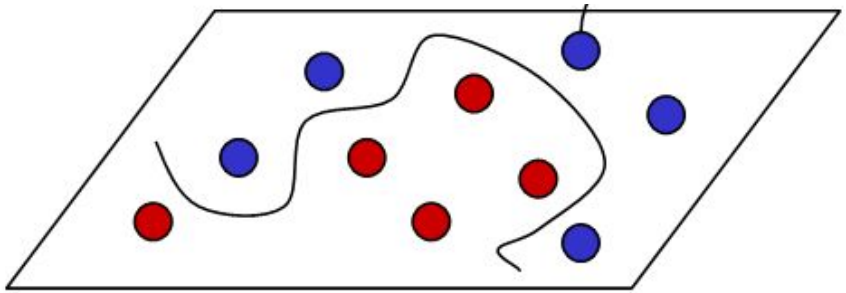
$$l_i = \max\left(0, 1 - y_i\mathbf{w}^T\mathbf{x}_i\right)$$

# Distributed Soft-margin Linear SVM

**Algorithm 1:** CoCoA: Communication-Efficient Distributed Dual Coordinate Ascent

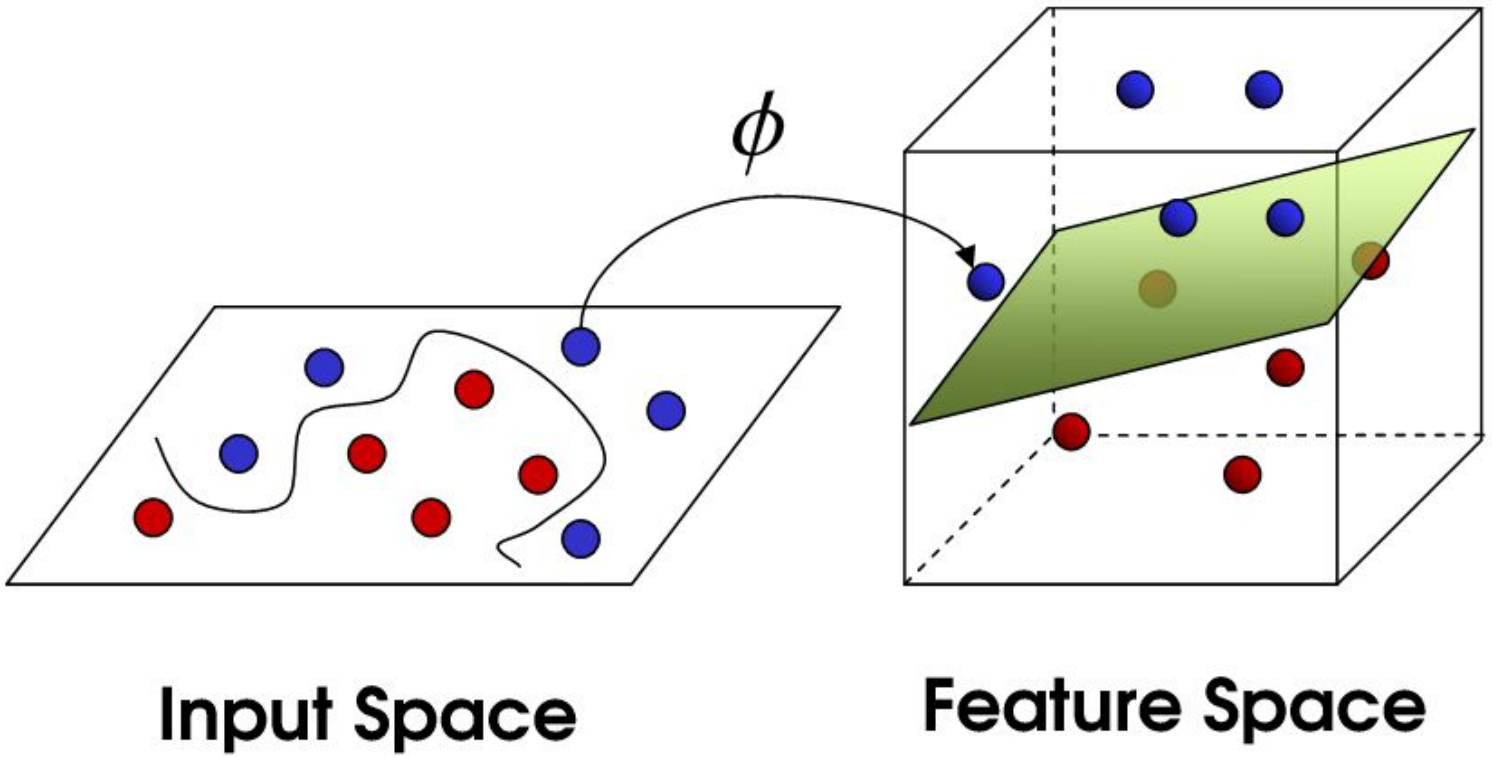
**Input**: $T \geq 1$, scaling parameter $1 \leq \beta_K \leq K$ (default: $\beta_K := 1$).
**Data**: $\{(x_i, y_i)\}_{i=1}^n$ distributed over $K$ machines
**Initialize**: $\alpha_{[k]}^{(0)} \leftarrow 0$ for all machines $k$, and $w^{(0)} \leftarrow 0$
**for** $t = 1, 2, \ldots, T$
  **for** *all machines* $k = 1, 2, \ldots, K$ ***in parallel***
    $(\Delta\alpha_{[k]}, \Delta w_k) \leftarrow \textsc{LocalDualMethod}(\alpha_{[k]}^{(t-1)}, w^{(t-1)})$
    $\alpha_{[k]}^{(t)} \leftarrow \alpha_{[k]}^{(t-1)} + \frac{\beta_K}{K} \Delta\alpha_{[k]}$
  **end**
  *reduce* $w^{(t)} \leftarrow w^{(t-1)} + \frac{\beta_K}{K} \sum_{k=1}^K \Delta w_k$
**end**

# Kernel Trick



Input Space

# Kernel Trick

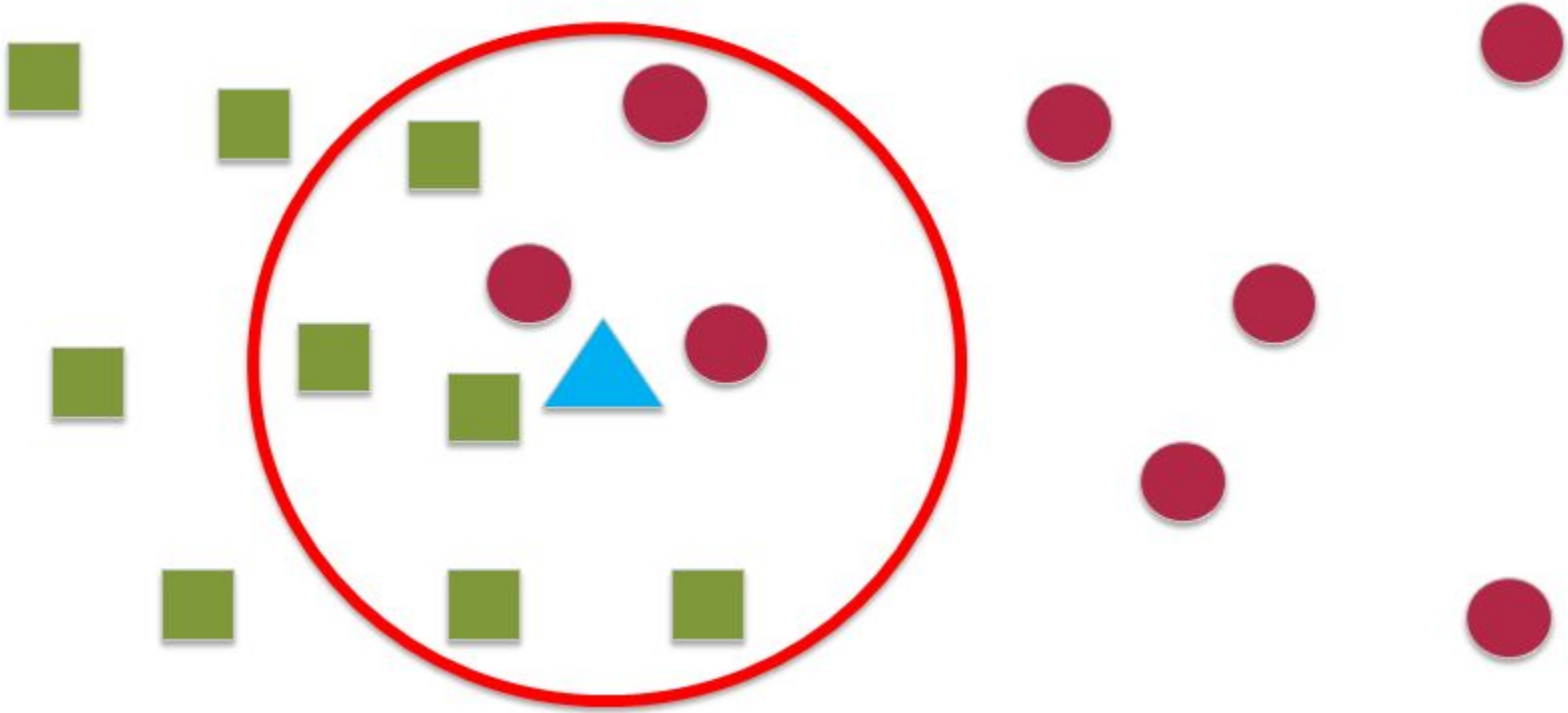


# The main problem with SVM

**No distributed SVM with any kernel except linear**

# KNN

# kNN (k-nearest neighbor)



# Distributed kNN (First version)

1. Compute the cross product between the data we wish to classify and our training data
2. Ship the data evenly across all of our machines
3. Compute the distance between each pair of points locally
4. Reduce for each data point we wish to classify that data point and the K smallest distances, which we then use to predict

# Distributed kNN (Second version)

1. Spread the data on N machines
2. For each predicted point find k nearest neighbour on each node (k * N totally)
3. Collect k * N candidates to Reducer and re-select the k closest neighbours

# The main problem with kNN

**No real training phase**

# Approximate Nearest Neighbours

1. Spread the train data for *N* machines
2. Find limited set of candidates *S* representing all train data with procedure *A*
3. Spread the test data for *M* machines with *S* candidates
4. Classify locally by local kNN based on *S* candidates

# Model Evaluation

# Model Evaluation with K-fold cross validation

Pipeline API

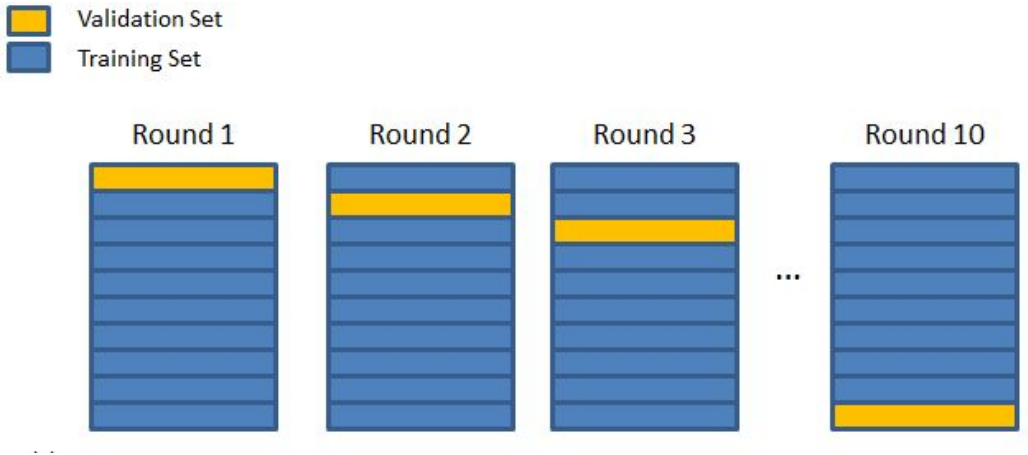
Test-Train Split

Parameter Grid

Binary Evaluator

Binary Classification Metrics

Tuning Hyperparameters



# Distributed Binary Classification Metrics

1. Accuracy is easy
2. Precision & Recall
3. Balanced Accuracy
4. ROC AUC ???? a lot of calculations
5. Iterate among pairs <GroundThruth, Prediction> locally and increment counters locally, after that merge them (MR approach)

# K-fold Cross-Validation

- It could generate K tasks for training and evaluation to run in parallel
- Results could be merged on one node or in distributed data primitive

# Hyper-parameter tuning

# Ensembles in distributed mode

# Empirical rule ##

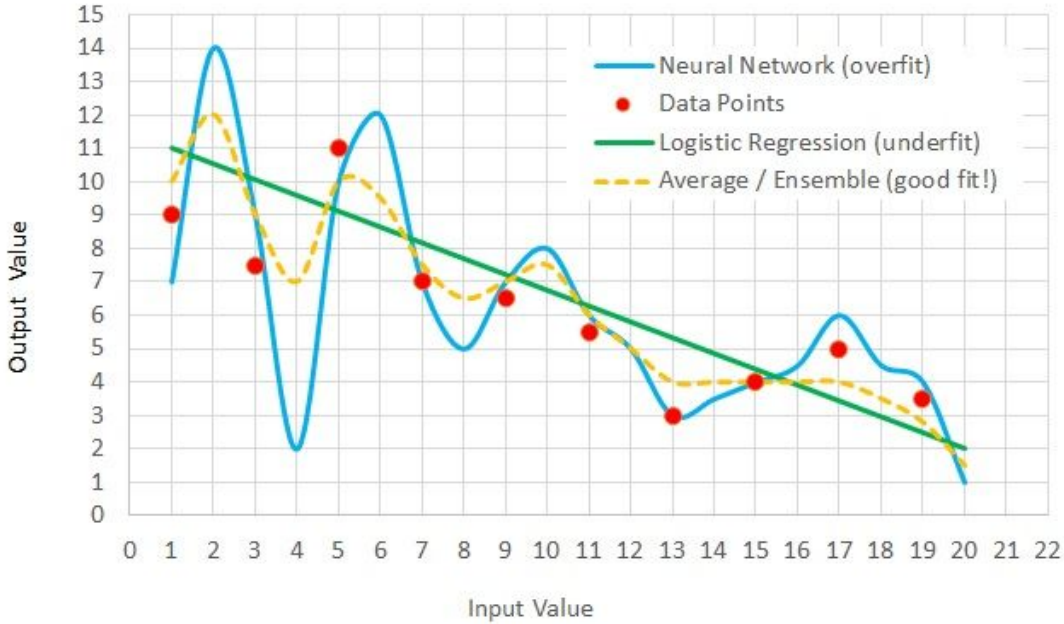
**The computational cost of training several classifiers on subsets of data is lower than training one classifier on the whole data set**

# Machine Learning Ensemble Model Averaging

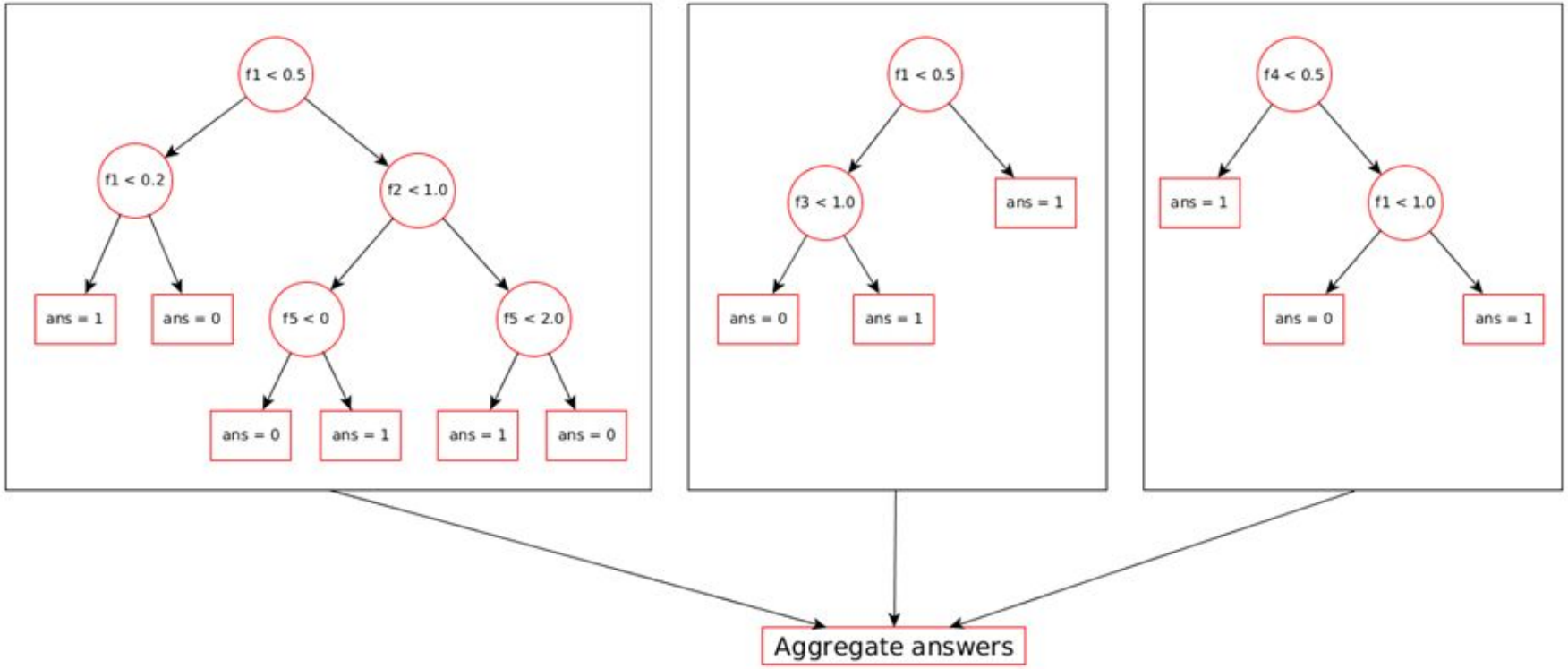
Ensemble as a Mean value of predictions

Majority-based Ensemble

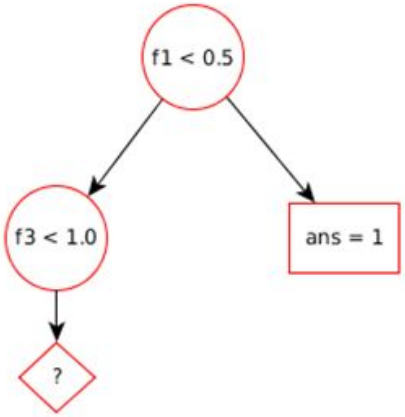
Ensemble as a weighted sum of predictions

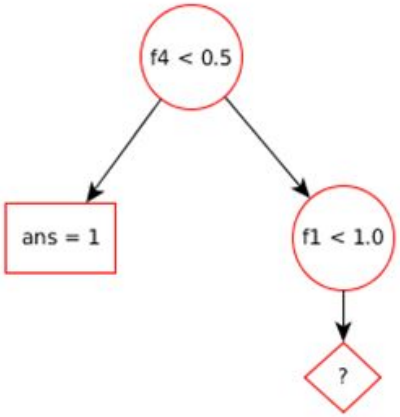


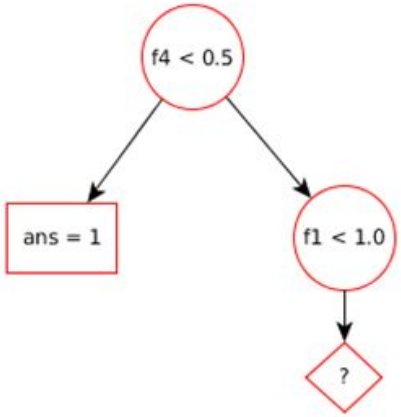
# Random Forest

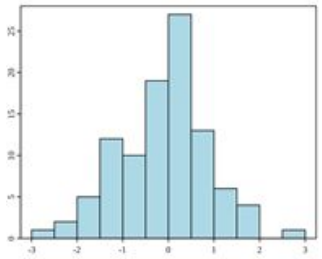


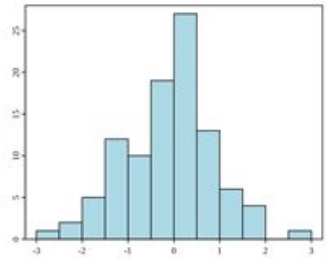
# Distributed Random Forest

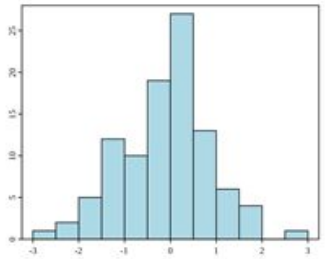


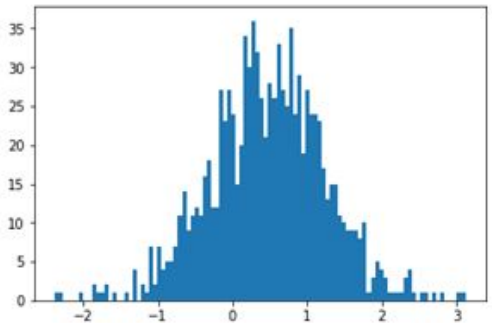



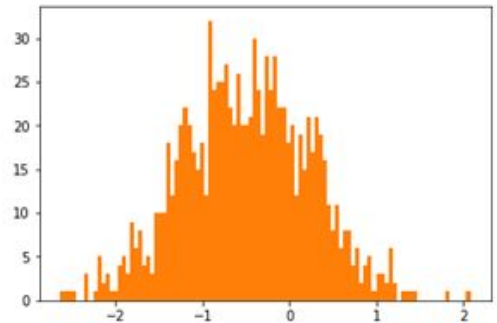




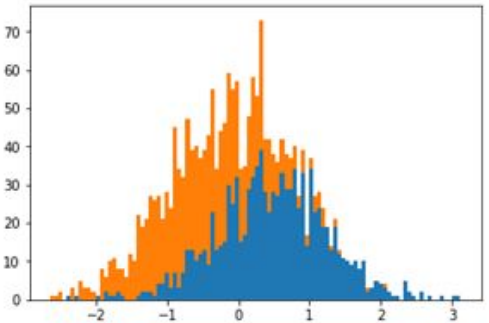
# Distributed Random Forest on Histograms



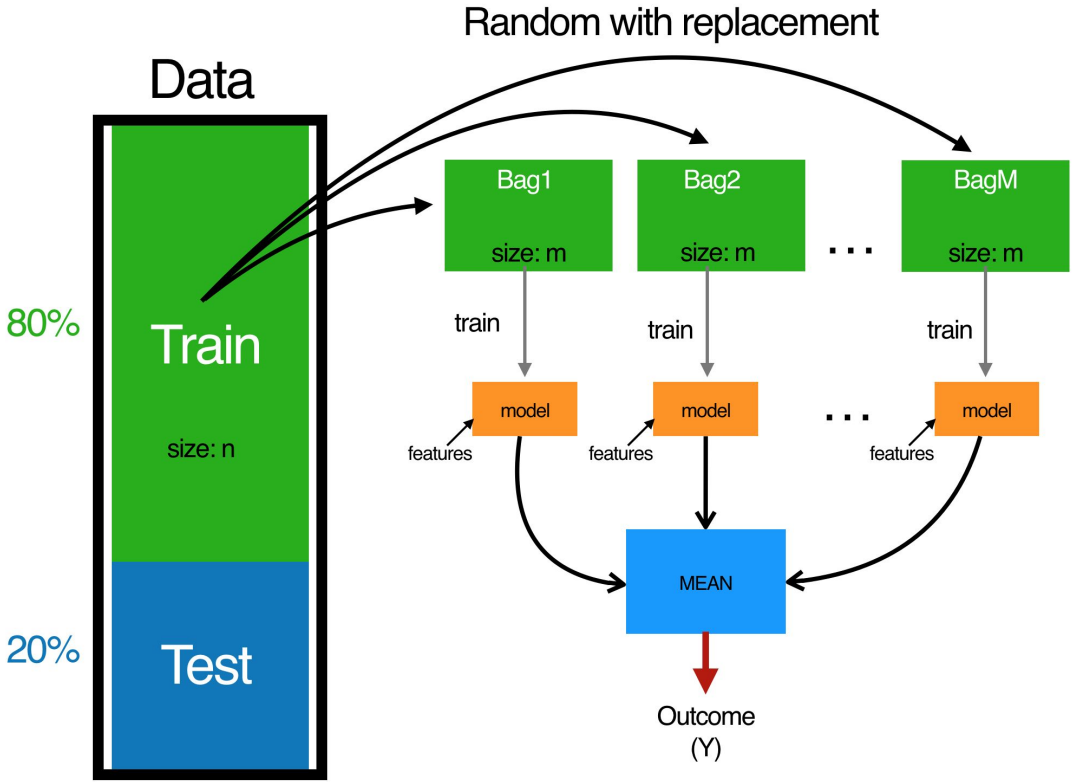

+



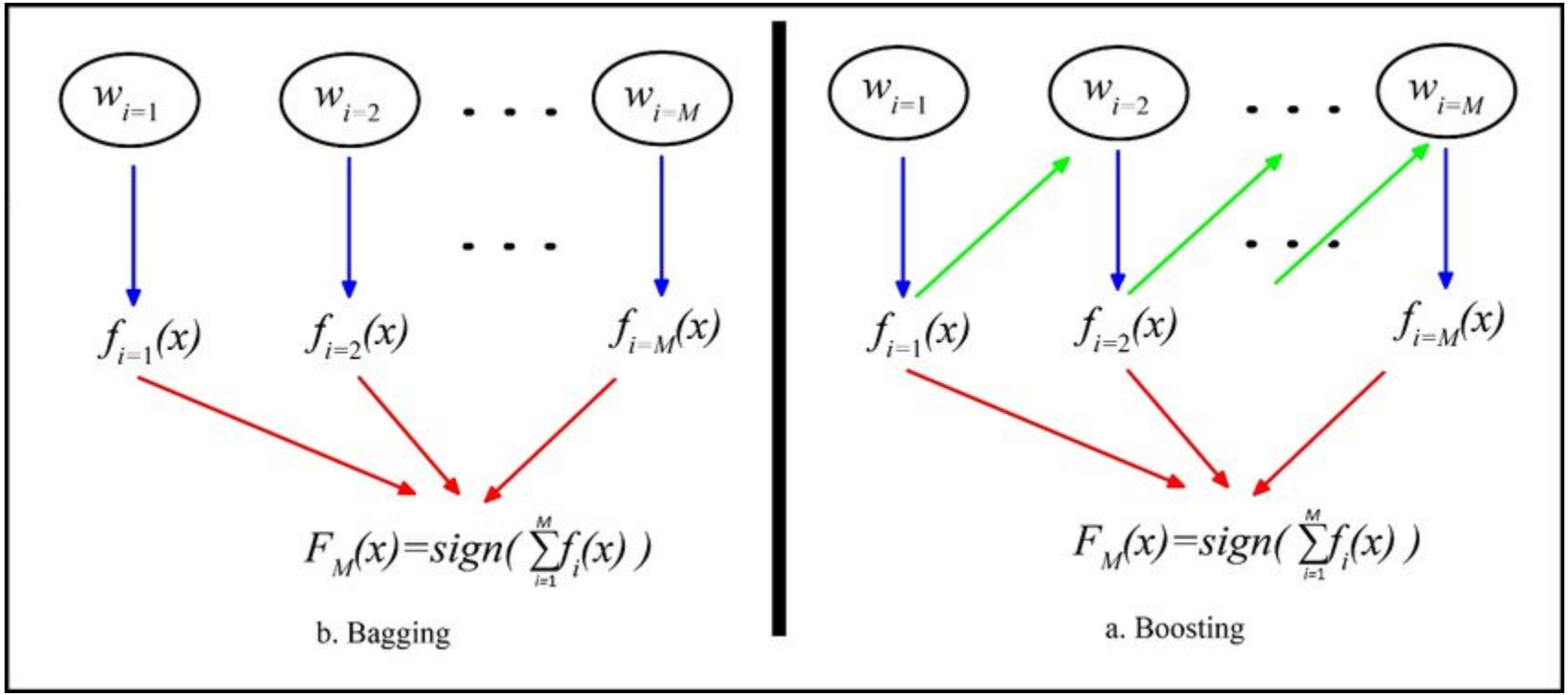

=



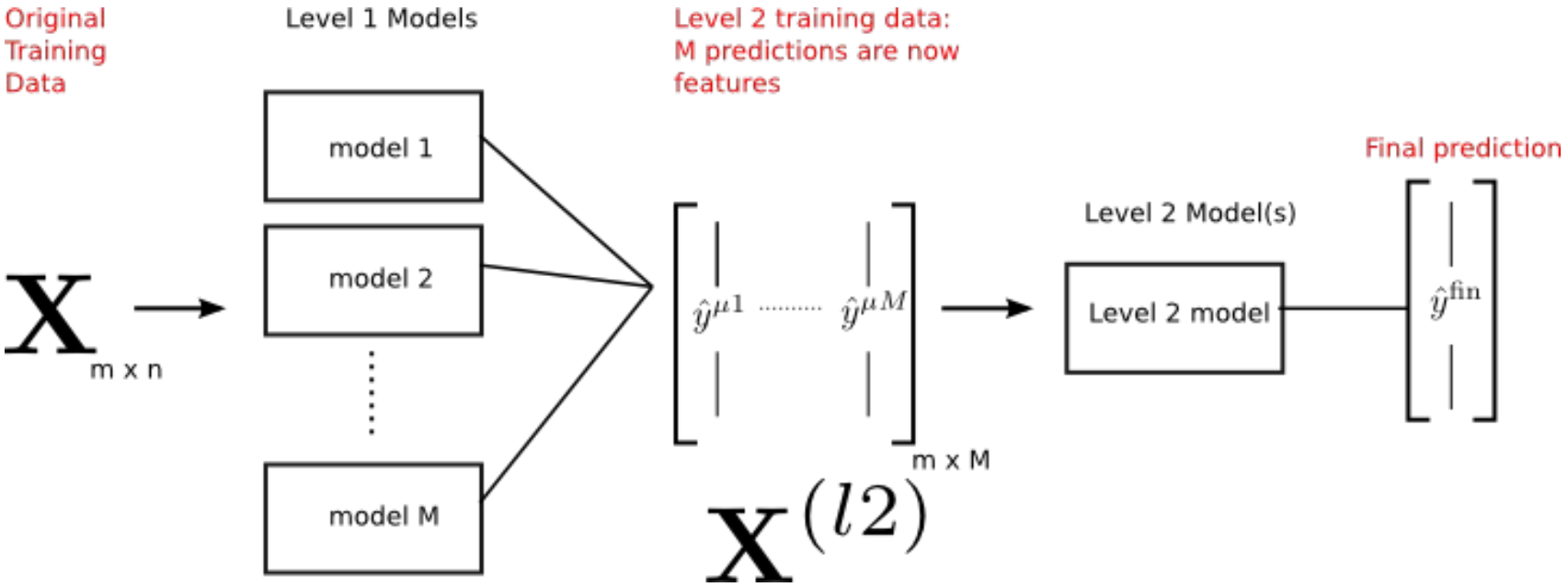
# Bagging


Random with replacement
Data
80%
Train
size: n
20%
Test
Bag1
size: m
Bag2
size: m
BagM
size: m
train
train
train
model
model
model
features
features
features
MEAN
Outcome
(Y)

# Boosting


$w_{i=1}$
$w_{i=2}$
$w_{i=M}$
$f_{i=1}(x)$
$f_{i=2}(x)$
$f_{i=M}(x)$
$F_M(x)=sign(\sum_{i=1}^{M} f_i(x))$
b. Bagging
$w_{i=1}$
$w_{i=2}$
$w_{i=M}$
$f_{i=1}(x)$
$f_{i=2}(x)$
$f_{i=M}(x)$
$F_M(x)=sign(\sum_{i=1}^{M} f_i(x))$
a. Boosting

# Stacking


Original Training Data
Level 1 Models
model 1
model 2
model M
Level 2 training data: M predictions are now features
$\hat{y}^{\mu 1}$ ..... $\hat{y}^{\mu M}$
m x M
$X^{(l2)}$
Level 2 Model(s)
Level 2 model
Final prediction
$\hat{y}^{\text{fin}}$
X
m x n

# How to contribute?

# Apache Ignite Community

> 200 contributors totally

8 contributors to ML module

VK Group

Blog posts

Ignite Documentation

ML Documentation

# Roadmap for Ignite 3.0

NLP (TF-IDF, Word2Vec)

More integration with TF

Clustering: LDA, Bisecting K-Means

Naive Bayes and Statistical package

Dimensionality reduction

… a lot of tasks for beginners:)

# The complexity of ML algorithms

Assume, **n** the sample size and **p** the number of features

| **Algorithm** | **Training complexity** | **Prediction complexity** |
|---|---|---|
| Naive Bayes | O(n*p) | O(p) |
| kNN | O(1) | O(n*p) |
| ANN | O(n*p) + KMeans Complexity | O(p) |
| Decision Tree | O(n^2*p) | O(p) |
| Random Forest | O(n^2*p*amount of trees) | O(p*amount of trees) |
| SVM | O(n^2*p + n^3) | O(amount of sup.vec * p) |
| Multi - SVM | O(O(SVM) * amount of classes) | O(O(SVM) * amount of classes * O(sort(classes))) |

# Papers and links

1. A survey of methods for distributed machine learning
2. Strategies and Principles of Distributed Machine Learning on Big Data
3. Distributed k-means algorithm
4. MapReduce Algorithms for k-means Clustering
5. An Extended Compression Format for the Optimization of Sparse Matrix-Vector Multiplication
6. Communication-Efficient Distributed Dual Coordinate Ascent
7. Distributed K-Nearest Neighbors

# Follow me

E-mail : zaleslaw.sin@gmail.com

Twitter : @zaleslaw @BigDataRussia

vk.com/big_data_russia **Big Data Russia**

**+ Telegram** @bigdatarussia

vk.com/java_jvm **Java & JVM langs**

**+ Telegram** @javajvmlangs