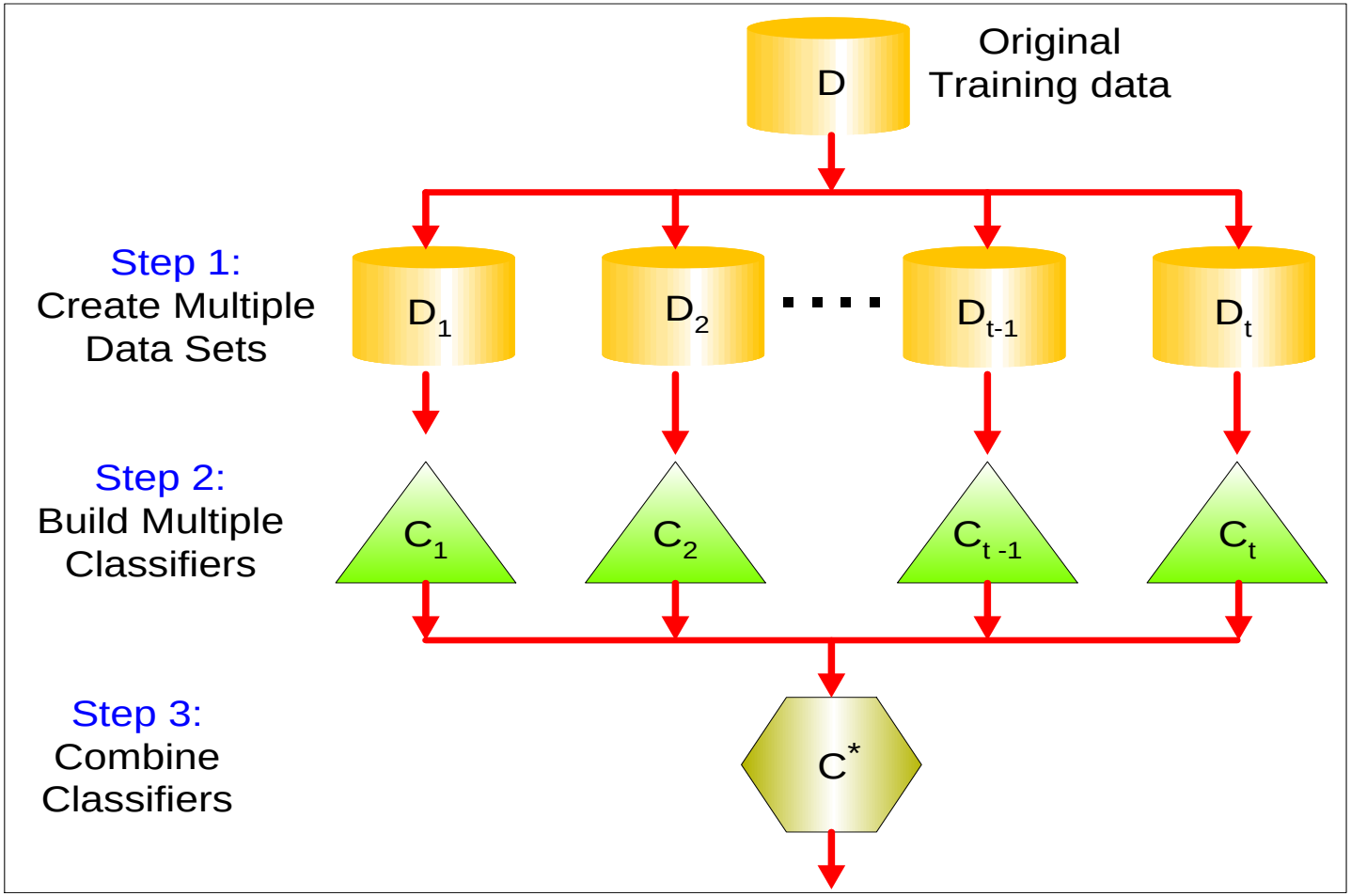
Ensemble Learning: An Introduction

Adapted from Slides by Tan, Steinbach, Kumar

General Idea



Examples of Ensemble Methods

- How to generate an ensemble of classifiers?
  - Bagging
  - Boosting

Bagging

- Sampling with replacement

Training Data

Data ID

Original Data	1	2	3	4	5	6	7	8	9	10
Bagging (Round 1)	7	8	10	8	2	5	10	10	5	9
Bagging (Round 2)	1	4	9	1	2	3	2	7	3	2
Bagging (Round 3)	1	8	5	10	5	5	9	6	3	7

- Build classifier on each bootstrap sample
- Each sample has probability $(1 – 1/n)^n$ of being selected as test data
- Training data = $1- (1 – 1/n)^n$ of the original data

The 0.632 bootstrap

- This method is also called the *0.632 bootstrap*
  - A particular training data has a probability of 1-1/*n* of *not* being picked
  - Thus its probability of ending up in the test data (not selected) is:

$$\left(1-\frac{1}{n}\right)^{n} \approx e^{-1} = 0.368$$

  - This means the training data will contain approximately 63.2% of the instances

Bagging- Summary

- Works well if the base classifiers are unstable (complement each other)
- Increased accuracy because it ***reduces the variance*** of the individual classifier
- Does not focus on any particular instance of the training data
  - Therefore, less susceptible to model over-fitting when applied to noisy data
- What if we want to focus on a particular instances of training data?

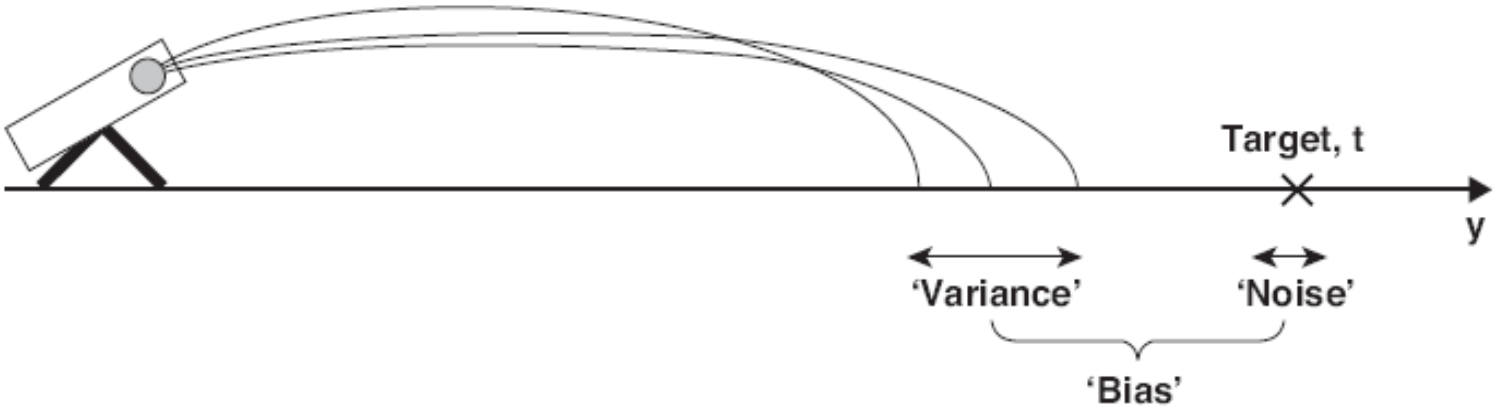


Figure 5.32. Bias-variance decomposition.

In general,
- Bias is contributed to by the training error; a complex model has low bias.
-Variance is caused by future error; a complex model has High variance.
- Bagging reduces the variance in the base classifiers.

Boosting

- An iterative procedure to adaptively change distribution of training data by focusing more on previously misclassified records
  - Initially, all N records are assigned equal weights
  - Unlike bagging, weights may change at the end of a boosting round

Boosting

- Records that are wrongly classified will have their weights increased
- Records that are classified correctly will have their weights decreased

Original Data	1	2	3	4	5	6	7	8	9	10
Boosting (Round 1)	7	3	2	8	7	9	4	10	6	3
Boosting (Round 2)	5	4	9	4	2	5	1	7	4	2
Boosting (Round 3)	4	4	8	10	4	5	4	6	3	4

- Example 4 is hard to classify
- Its weight is increased, therefore it is more likely to be chosen again in subsequent rounds

Boosting

- Equal weights are assigned to each training instance (1/N for round 1) at first round
- After a classifier C_i is learned, the weights are adjusted to allow the subsequent classifier

C_{i+1} to "pay more attention" to data that were misclassified by C_i.

- Final boosted classifier C* combines the votes of each individual classifier
  - Weight of each classifier's vote is a function of its accuracy
- Adaboost – popular boosting algorithm

Adaboost (Adaptive Boost)

- Input:
  - Training set D containing ***N*** instances
  - *T* rounds
  - A classification learning scheme
- Output:
  - A composite model

Adaboost: Training Phase

- Training data D contain N labeled data (X_1,y_1), (X_2,y_2), (X_3,y_3),….(X_N,y_N)
- Initially assign equal weight 1/d to each data
- To generate *T* base classifiers, we need *T* rounds or iterations
- Round i, data from D are sampled with replacement , to form Di (size *N*)
- Each data's chance of being selected in the next rounds depends on its weight
  - Each time the new sample is generated directly from the training data D with different sampling probability according to the weights; these weights are not zero

Adaboost: Training Phase

- Base classifier C_i, is derived from training data of Di
- Error of C_i is tested using Di
- Weights of training data are adjusted depending on how they were classified
  - Correctly classified: Decrease weight
  - Incorrectly classified: Increase weight
- Weight of a data indicates how hard it is to classify it (directly proportional)

Adaboost: Testing Phase

- The lower a classifier error rate, the more accurate it is, and therefore, the higher its weight for voting should be
- Weight of a classifier C_i's vote is

$$\alpha_i = \frac{1}{2}\ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right)$$

- Testing:
  - For each class c, sum the weights of each classifier that assigned class c to X (unseen data)
  - The class with the highest sum is the WINNER!

$$C^*(x_{test}) = \arg\max_y \sum_{i=1}^{T} \alpha_i \delta\left(C_i(x_{test}) = y\right)$$

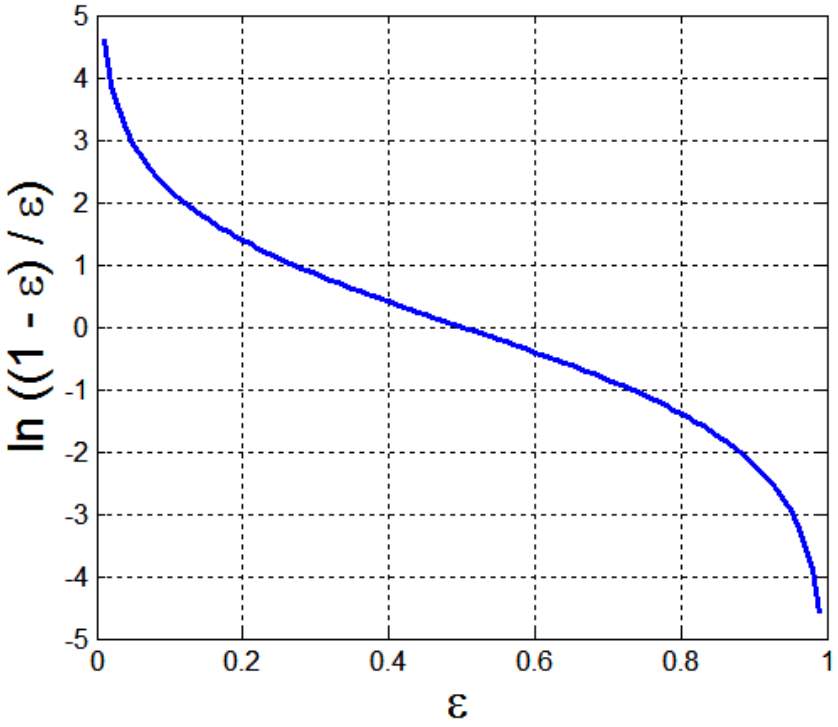
Example: Error and Classifier Weight in AdaBoost

- Base classifiers: C_1, C_2, …, C_T
- Error rate: (*i* = index of classifier, *j*=index of instance)

$$\varepsilon_i = \frac{1}{N}\sum_{j=1}^{N} w_j \delta\left(C_i(x_j) \neq y_j\right)$$

- Importance of a classifier:

$$\alpha_i = \frac{1}{2}\ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right)$$



Example: Data Instance Weight in AdaBoost

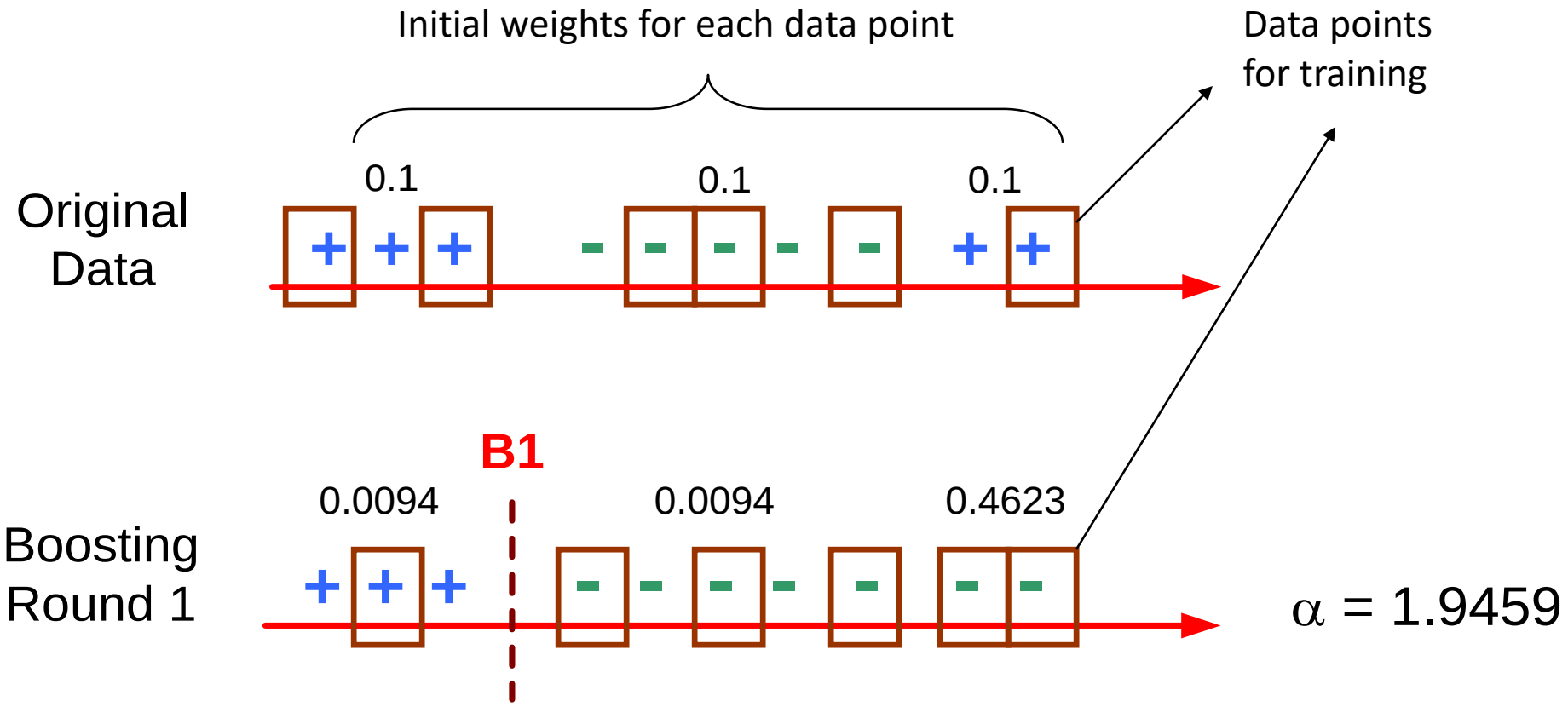
- Assume: *N* training data in D, *T* rounds, (x_j,y_j) are the training data, C_i, a_i are the classifier and weight of the i^{th} round, respectively.
- Weight update on all training data in *D*:

$$w_j^{(i+1)} = \frac{w_j^{(i)}}{Z_i} \begin{cases} \exp^{-\alpha_i} & \text{if } C_i(x_j) = y_j \\ \exp^{\alpha_i} & \text{if } C_i(x_j) \neq y_j \end{cases}$$

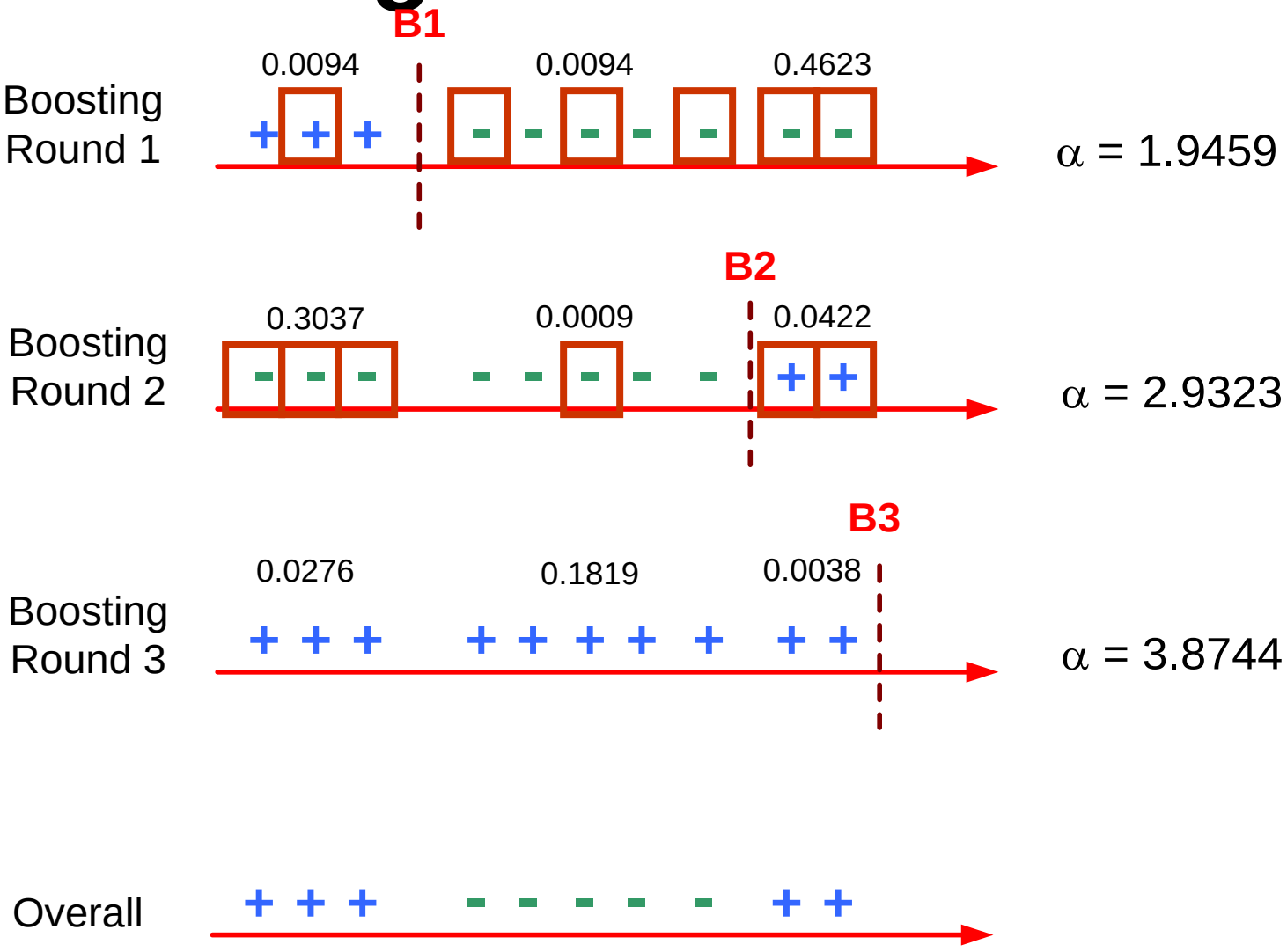
$$\text{where } Z_i \text{ is the normalization factor}$$

$$C^*(x_{test}) = \arg\max_y \sum_{i=1}^{T} \alpha_i \delta\left(C_i(x_{test}) = y\right)$$

Illustrating AdaBoost



Illustrating AdaBoost



Ensemble method for supervised learning Using an explicit loss function

Gradient Boosting

Ricco RAKOTOMALALA

Université Lumière Lyon 2

Outline

1. Preamble
2. Gradient boosting for regression
3. Gradient boosting for classification
4. Regularization (shrinkage, stochastic gradient boosting)
5. Tools and software
6. Conclusion – Pros and cons
7. References

Preamble

Boosting and Gradient Descent

Boosting

BOOSTING is an ensemble method which aggregates classifiers learned sequentially on a sample for which the weights of individuals are adjusted at each step. The classifiers are weighted according to their performance. [RAK, page 28].

Input: B number of models, ALGO learning algorithm, Ω training set, with size = n, y target attribute, X matrix with p predictive attributes.

MODELES = { }
All the instances have the same weight $\omega^1_i = 1/n$ For b = 1 to B Do
Fit the model M_b from $\Omega(\omega^b)$ using ALGO (ω^b weighting system at the step b)

Add M_b into MODELES
Calculate the weighted error rate for M_b : $\varepsilon_b = \sum_{i=1}^{n} \omega_i^b \times I(y_i \neq \hat{y}_i)$
If $\varepsilon_b > 0.5$ or $\varepsilon_b = 0$, STOP the process
Else
Calculate $\alpha_b = \ln \frac{1-\varepsilon_b}{\varepsilon_b}$
The weights are updated $\omega_i^{b+1} = \omega_i^b \times \exp[\alpha_b . I(y_i \neq \hat{y}_i)]$
And normalized so that the sum is equal to 1
End For

A weighted (α_b) vote is used for prediction (this is an additive model)

$$f(x) = sign \sum_{b=1}^{B} \alpha_b \times M_b(x)$$

Gradient descent

Gradient descent is an iterative technique that allows to approach the solution of an optimization problem. In supervised learning, the construction of the model is often to determine the parameters that enable to optimize (max or min) an objective function (ex. Perceptron – Least squares criterion, pages 11 et 12).

$$J(y, f) = \sum_{i=1}^{n} j(y_i, f(x_i))$$

f() is a classifier with some parameters
j() is a cost function comparing the observed value of the target and the prediction of the model for an observation
J() is an overall loss function, additively calculated from all observations
➔ The aim is to minimize **J()** with regard to **f()** i.e. the parameters of f().

$$f_b(x_i) = f_{b-1}(x_i) - \eta \times \nabla j(y_i, f(x_i))$$

$f_b()$ is the version of classifier at step ''b''
η is the learning rate which enables to lead the process
∇ is the gradient i.e. the first order partial derivative of the cost function with regard to the classifier

$$\nabla j(y_i, f(x_i)) = \frac{\partial j(y_i, f(x_i))}{\partial f(x_i)}$$

Boosting = Gradient descent

We can show that ADABOOST consists in to optimize an exponential loss function i.e. each classifier M_t learned from the weighted sample resulting from M_{t-1} allows to minimize an overall loss function [BIS, page 659 ; HAS, page 343]

$$J(f) = \sum_{i=1}^{n} \exp\left(-y_i \times f(x_i)\right)$$

$y \in \{-1, +1\}$

J() is the overall loss function

f() is the aggregate classifier composed of a linear combination of the base classifiers M_b

$$f_b = f_{b-1} + \frac{\alpha_b}{2} \times M_b$$

The aggregate classifier at step ''b'' is corrected with the individual classifier M_b learned from the reweighted sample. M_b is the gradient here i.e. each intermediate model allows to reduce the loss of the global model.

$$\omega_i^b = \omega_i^{b-1} \times \exp\left[\alpha_{b-1}.I\left(y_i \neq M_{b-1}(i)\right)\right]$$

The "gradient" classifier comes from a sample where the weights of individuals depend on the performance of the previous model (idea of iterative corrections)

GRADIENT BOOSTING : generalize the approach with other loss functions

Gradient Boosting for Regression

Gradient Boosting = Gradient Descent + Boosting

Regression

The regression is a supervised learning process which estimates the relationship between a quantitative dependent variable and a set of independent variables.

$$y_i = M_1(x_i) + \varepsilon_{1i}$$

ε is the error term. It represents the inadequacy of the model.

M is any kind of model, we use regression tree.

$$e_{i1} = y_i - M_1(x_i)$$

e is the residual. Estimated value of the error. High value (in absolute value) reflects a bad prediction.

The aim is to model this residual with a second classifier M2 and associate it with the previous one for a better prediction.

$$e_{i1} = M_2(x_i) + \varepsilon_{2i}$$

We can proceed in the same way for the residual e_2, etc.

$$\hat{y_i} = M_1(x_i) + M_2(x_i)$$

The role of M_2 is (additively) compensate the inadequacy of M_1, thereafter we can learn M_3, etc.

Overall loss function

Connection with gradient descent

The sum of the squares of errors is a well-known overall indicator of quality in regression

$$j(y_i, f(x_i)) = \frac{1}{2}(y_i - f(x_i))^2$$

$$J(y, f) = \sum_{i=1}^{n} j(y_i, f(x_i))$$

$$\frac{\partial j(y_i, f(x_i))}{\partial f(x_i)} = \frac{\partial\left[\frac{1}{2}(y_i - f(x_i))^2\right]}{\partial f(x_i)} = f(x_i) - y_i$$

Calculation of the gradient. It is actually equal to the residual, but with an opposite sign i.e. residual = negative gradient

$$\begin{aligned} f_b(x_i) &= f_{b-1}(x_i) + M_b(x_i) \\ &= f_{b-1}(x_i) + (y_i - f_{b-1}(x_i)) \\ &= f_{b-1}(x_i) - 1 \times \frac{\partial j(y_i, f(x_i))}{\partial f(x_i)} \\ &= f_{b-1}(x_i) - \eta \times \nabla j(y_i, f(x_i)) \end{aligned}$$

Thus, we have an iterative process for the construction of the additive model. Modeling the residuals in step "b" (regression tree M_b) corresponds to a gradient. Ultimately, we minimize the overall cost function J()

The learning rate η is equal to 1 here.

Gradient Boosting Algorithm for Regression

We have an iterative process where, at each step, we use the negative value of the gradient: $-\nabla j(y,f)$ [WIK]

Or, more simply, FOR m = 1, …, B (B : parameter of the algorithm)

The trivial tree corresponds to a tree with only the root. The prediction is equal to the mean of the target attribute Y.

Fit the trivial tree $f_0()$ REPEAT

UNTIL CONVERGENCE

Calculate negative gradient $-\nabla j(y,f)$

Fit a regression tree M_b for $-\nabla j(y,f)$

$f_b = f_{b-1} + \gamma_b . M_b$

Must be calculated for all the individuals of the training sample (i = 1, …, n)

j() = square of the error ➔ negative gradient = residual

The depth of the trees is a possible parameter

The models are combined in additive fashion

γ_b is chosen at each step in order to minimize (using a numerical optimization approach)

$$\gamma_b = \arg\min_{\gamma} \sum_{i=1}^{n} j(y_i, f_{b-1}(x_i) + \gamma . M_b(x_i))$$

! The advantage of this generic formulation is that one can use other loss functions and the associated gradients.

Gradient Boosting

Other loss functions

Other loss functions
➔ Other gradient formulation
➔ Other behavior and performance of the aggregate model

Loss function	$-\nabla j(y_i, f(x_i))$	Pros / Cons
$\frac{1}{2}(y_i - f(x_i))^2$	$y_i - f(x_i)$	Sensitivity to small differences, but not robust against the outliers
$\lvert y_i - f(x_i)\rvert$	$signe[y_i - f(x_i)]$	Less sensitive to small differences but robust against outliers
Huber	$y_i - f(x_i)$ si $\lvert y_i - f(x_i)\rvert \leq \delta_b$ $\delta_b.sign[yi - f(xi)]$ si $\lvert y_i - f(x_i)\rvert > \delta_b$ Where δ_b is a quantile of $\{\lvert y_i - f(x_i)\rvert\}$	Combine the benefits of the square error (more sensitive to small values of the gradient) and absolute error (more robust against the outliers)

Gradient Boosting for Classification

Working with the indicator variables (dummy variables)

Loss function and gradient for classification

The categorical target variable is K values {1,..., K}
The algorithm remains identical but: we must define a loss function adapted to the classification process, and calculate the appropriate gradient.

y^k is a dummy (K dummy variables are generated)

$$y_i^k = \begin{cases} 1 \text{ si } Y_i = k \\ 0 \text{ sinon} \end{cases}$$

$$\pi^k(x_i) = \frac{e^{f^k(x_i)}}{\sum_{k=1}^{K} e^{f^k(x_i)}}$$

π_k corresponds to the class membership probability for "k" (value "k" de Y)

$$j(y_i, f(x_i)) = -\sum_{k=1}^{K} y_i^k \times \log \pi^k(x_i)$$

Loss function : MULTINOMIAL DEVIANCE (binomial deviance is a special case for binary target attribute)

$$\nabla j(y_i, f(x_i)) = y_i^k - \pi^k(x_i)$$

Gradient

For the class "k", the gradient is the difference between the associated dummy variable and the class membership probability

We must deal with the dummy variables (y^k), and fit a regression tree on the negative gradient 1 tree for each dummy variable). f^k is the aggregate model for the class "k", needed for the calculation of π^k

Algorithm for classification

The process is not changed compared with the regression. The overall scheme remains the same, except that we use the dummy variables

Y (target) is coded with K dummy variables y^k Fit K trivial trees $f^k_0()$ for each y^k

REPEAT UNTIL CONVERGENCE

Calculate K negative gradients $-\nabla j(y^k, f^k)$ Fit a **regression tree** M^k_b for each $-\nabla j(y^k, f^k)$

$$f^k_b = f^k_{b-1} + \gamma_b . M^k_b$$

We obtain K aggregate models f^k. The class membership probability is calculated with the "softmax" function

$$\pi^k(x_i) = \frac{e^{f^k(x_i)}}{\sum_{k=1}^{K} e^{f^k(x_i)}}$$

Even if we are in the classification context, the internal mechanism is based on a regression tree algorithm. GRADIENT TREE BOOSTING.

!

The assignment rule is $\hat{Y}_i = \arg\max_k \pi^k(x_i)$

Regularization

Approaches to prevent overfitting

Other than the limitation of the depth of the trees

Shrinkage

Include an additional parameter (learning rate) in the update rule

$$f_b = f_{b-1} + \nu.\gamma_b.M_b$$

An additional parameter ($0 < \nu \leq 1$) is used in order to "smooth" the update rule. Empirically, we observe that a low value of ν ($\nu < 0.1$) improves the performance, but the converge is slower (number of needed iterations **B** is higher).

Stochastic Gradient Boosting

Random sampling is introduced. At each step, only a fraction β ($0 < \beta \leq 1$) of the learning sample is used for the construction of the trees M_b [HAS, page 365]

$\beta = 1$, we have the standard algorithm. Typically, $0.5 \leq \beta \leq 0.8$ is suited for a moderate sized dataset [WIK]. Advantages:

1. Reduce the computation time.
2. Prevent overfitting by introducing randomness in the learning process (such as Random forest and Bagging)
3. OOB estimation of the error rate (such as Bagging)

Practice of gradient boosting

Software and packages

Python (scikit-learn)

class `sklearn.ensemble.` **`GradientBoostingClassifier`** (*loss='deviance', learning_rate=0.1, n_estimators=100, subsample=1.0, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, presort='auto'*) [source]

Gradient Boosting for classification.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage `n_classes_` regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

```
#importation of the training set  import p
dtrain = pandas.read_table("ionosphere-train.txt",sep="\t",header=0,decimal=".")
print(dtrain.shape)

y_app = dtrain.as_matrix()[:,32] # target attribute  X_app =
dtrain.as_matrix()[:,0:32] # input attributes  # importation of the
GradientBoostingClassifier class  from sklearn.ensemble import
GradientBoostingClassifier
gb = GradientBoostingClassifier()  # display the parameters  print(gb)
# fit on the training set  gb.fit(X_app,y_app)
# importation of the test set
dtest = pandas.read_table("ionosphere-test.txt",sep="\t",header=0,decimal=".")
print(dtest.shape)
y_test = dtest.as_matrix()[:,32]  X_test = dtest.as_matrix()[:,0:32]  # prediction on the test set
y_pred = gb.predict(X_test)
# evaluation : test error rate = 0.085
from sklearn import metrics
err = 1.0 - metrics.accuracy_score(y_test,y_pred)  print(err)
```

There is also a variable sampling mechanism during the node splitting process, as with Random Forest.

Python (scikit-learn)

Paramétrage

Scikit-learn proposes a tool for determining by cross-validation the "optimal" parameters of a machine learning algorithm.

```
# grid search tool : http://scikit-learn.org/stable/modules/grid_search.html from sklearn.grid_search import GridSearchCV
# Combination of the parameters to evaluate. The tool performs an exhaustive search  # The calculations are intensive in cross-validation
parametres = {"learning_rate":[0.3,0.2,0.1,0.05,0.01],"max_depth":[2,3,4,5,6],"subsample":[1.0,0.8,0.5]}

# The supervised learning algorithm to use: Gradient boosting  gbc = GradientBoostingClassifier()

# Create the objet for searching
grille = GridSearchCV(estimator=gbc,param_grid=parametres,scoring="accuracy")
# Perform the process on the training set
resultats = grille.fit(X_app,y_app)
# best combination of parameters : {'subsample': 0.5, 'learning_rate': 0.2, 'max_depth': 4}  print(resultats.best_params_)
# prediction with the ‘’model’’ identified by cross-validation  ypredc = resultats.predict(X_test)

# performances of the ‘’best’’ model: test error rate = 0.065  err_best = 1.0 - metrics.accuracy_score(y_test,ypredc)
print(err_best)
```

Gradient Boosting

The "gradient boosting" is an ensemble method that generalizes the boosting by providing the opportunity of use other loss functions.

The global frameworks are identical: underlying algorithm = tree, construction in sequential way of models, "variable importance" measurement allows to assess the relevance of the predictors, similar problems for set the right values of parameters.

But unlike boosting, even in the classification context, the underlying algorithm is a regression tree.

Tools/software exist, but we really need to go into the details of the documentation to understand what is behind implementations and the handling of parameters.

Gradient boosting (GBM : Gradient Boosting Machine)

Pros and cons

Especially in classification process which is the main subject of this course.

Advantages

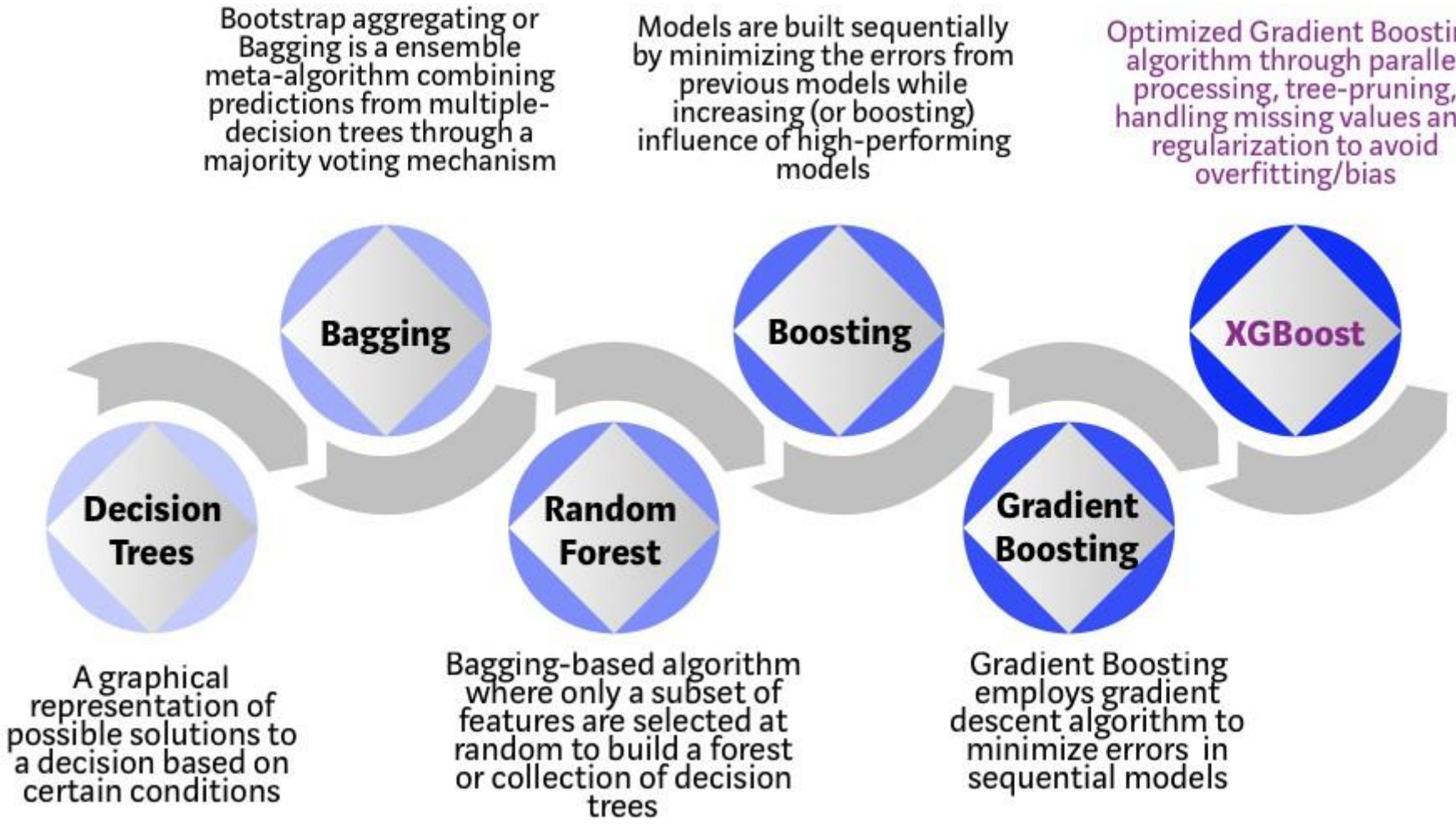
- Compared with to the "usual" boosting, GBM makes the emphasis on the difference $(y-\pi)$ during the construction of the regression trees with the deviance loss function
- Lots of flexibility with the choice of loss functions, adaptable to the characteristics of the studied problems
- GBM has shown its effectiveness in several challenges!

Drawbacks

- Non-explicit model (as for all ensemble methods)
- Many parameters which can interact and influence heavily the behavior of the approach (number of iterations, regularization parameters, etc.)
- Overfitting can occur if values of parameters are not suitable
- Computationally intensive (especially when number of trees is high)
- Memory occupation of the trees

XGBoost

XGBoost is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework.



Model Features

The implementation of the model supports the features of the scikit-learn and R implementations, with new additions like regularization. Three main forms of gradient boosting are supported:

1. **Gradient Boosting** algorithm also called gradient boosting machine including the learning rate.

2. Stochastic Gradient Boosting with sub-sampling at the row, column and column per split levels.

3. Regularized Gradient Boosting with both L1 and L2 regularization.

System Features

The library provides a system for use in a range of computing environments, not least:

1. **Parallelization** of tree construction using all of your CPU cores during training.
2. **Distributed Computing** for training very large models using a cluster of machines.
3. **Out-of-Core Computing** for very large datasets that don't fit into memory.
4. **Cache Optimization** of data structures and algorithm to make best use of hardware.

Algorithm Features

The implementation of the algorithm was engineered for efficiency of compute time and memory resources. A design goal was to make the best use of available resources to train the model. Some key algorithm implementation features include:

1. **Sparse Aware** implementation with automatic handling of missing data values.
2. **Block Structure** to support the parallelization of tree construction.
3. **Continued Training** so that you can further boost an already fitted model on new data