GRADIENT DESCENT

DEFINING OBJECTIVE

- Once we have define our loss function, the ML task to solve can simply be expressed as minimizing it over some parameters:
  - E.g., for Linear Regression:
    - Detailed way: $\underset{\beta_0,\beta_1\ldots,\beta_n}{\arg\min} \sum_i^n (y^i - (\beta_0 + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)} + \ldots + \beta_n x_n^{(i)}))^2$
    - Simplified way: $\underset{\beta}{\arg\min} \sum_i^n (y^i - f(\beta, x))^2$
    - Generic way: $\underset{\theta}{\arg\min}\, L_2(f(\theta, x))$

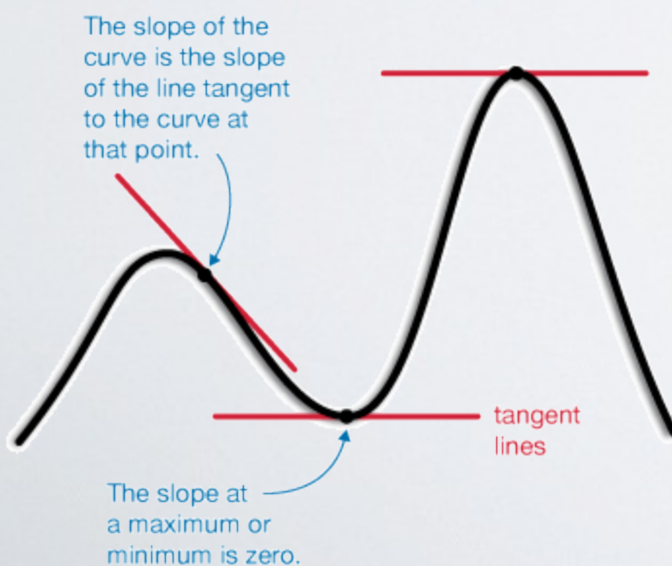
SOLVING MINIMIZATION

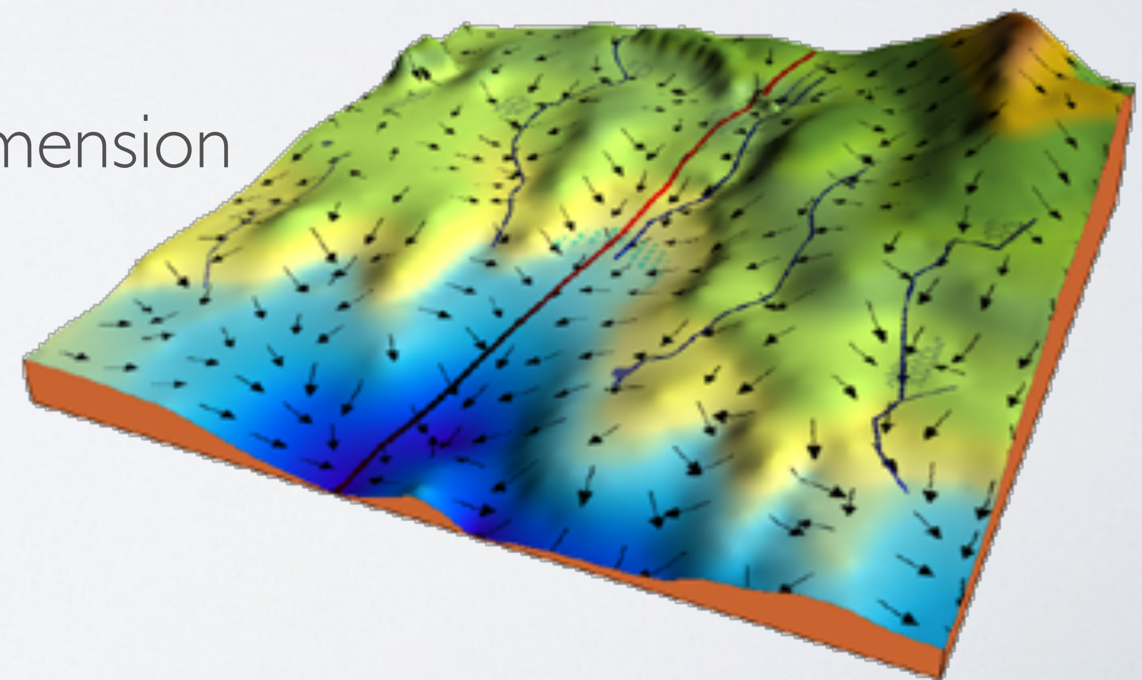
- Once we have expressed our task as a minimization problem, we just have to explore the parameter space to find a good solution
  - Exhaustive search is usually impossible
  - In the general case, no close form solution
  - One could use any optimization method
    - Genetic Algorithm
    - Simulated Annealing
    - EM…
  - Most used in general case: Gradient descent

GRADIENT DESCENT

- Greedy approach
  - Start from arbitrary point
  - Search for the nearest local minimum
    - If the problem is convex, find the global minimum, i.e., best possible solution
    - => linear regression
    - Else, find one local minimum, without guarantee.

- What is a gradient?
  - Generalization of a derivative to multiple dimension

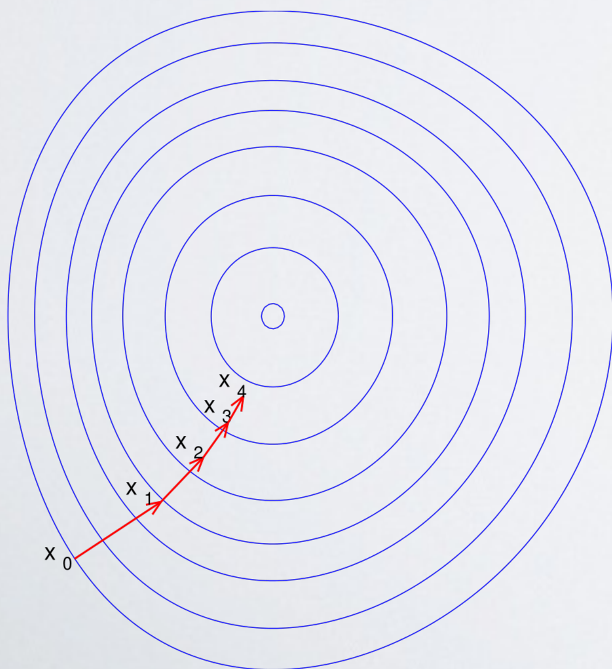


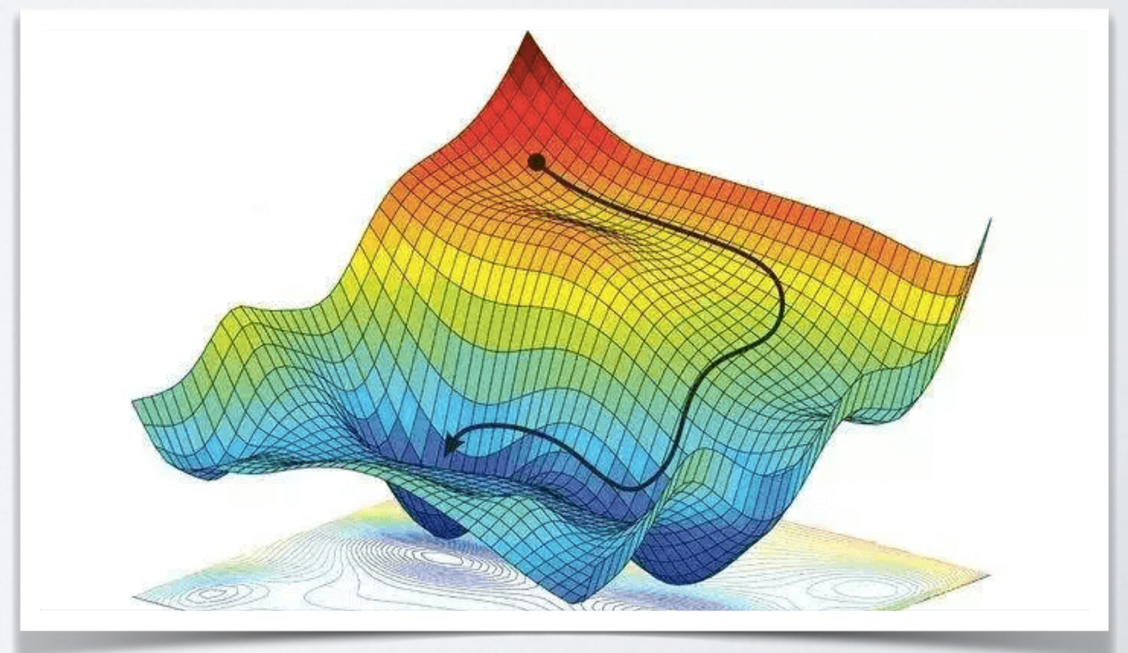


https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

GRADIENT DESCENT

- The objective of gradient descent is to follow the gradient/ derivative in order to find a minimal point
  - Example: linear regression.
    - The "altitude" is given by the loss function
    - Each of the β is a "direction" in which we can move
    - Gradient descent answers the question: "in which direction and by how much should I change the β so as to go "down" in the loss function, optimally





GRADIENT DESCENT

- In practice:
  - Update the parameters $\mathbf{a}$ of function F, by subtracting its gradient ∇F at point $\mathbf{a}_n$, multiplied by a parameter γ to control the speed
    - $\mathbf{a}_{n+1} = \mathbf{a}_n - \gamma \nabla F(\mathbf{a}_n)$
    - Subtract because we want to *descend* the gradient, i.e. minimize the function.

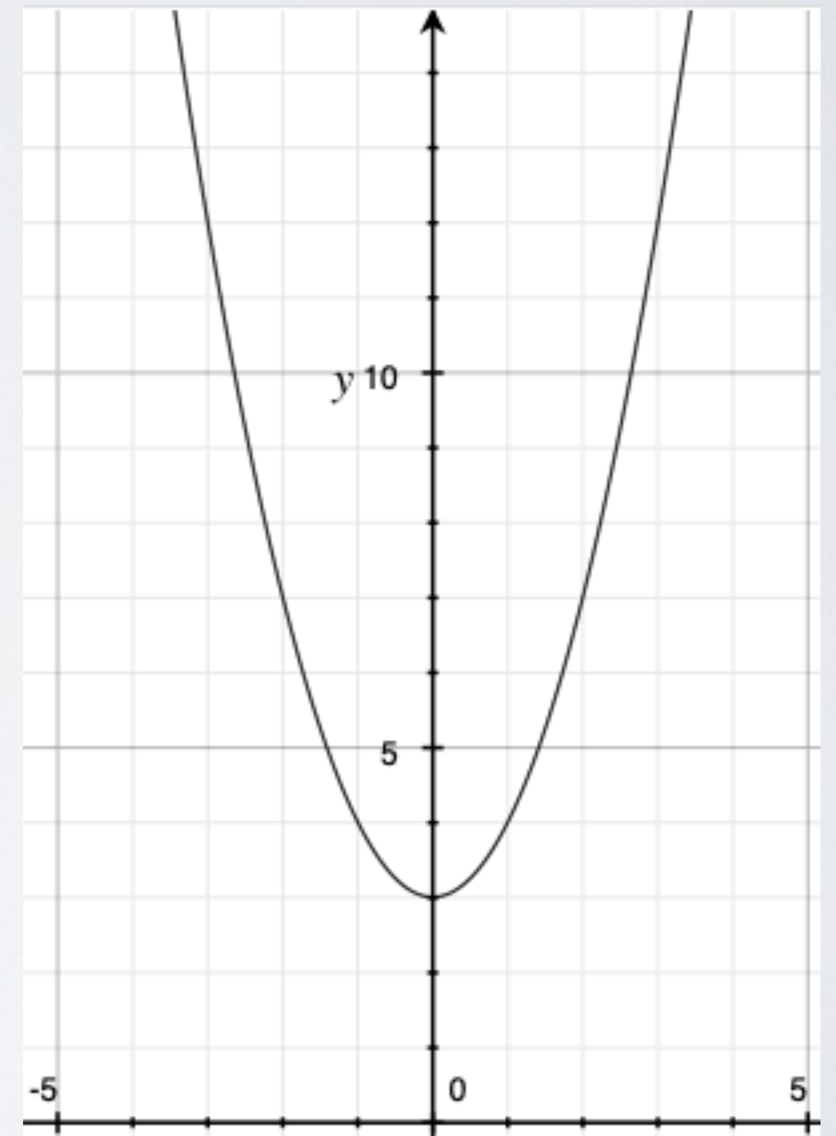
- The gradient has a value for each parameter
  - =>Compute the *partial derivative*, for each parameter

GRADIENT DESCENT

- Reminder: Common derivative
  - c'=0
  - $x' = 1, (ax)' = a$
  - $(x^a)' = ax^{a-1}$ => $(x^2)' = 2x$
  - …
- Common derivative rules
  - Mult by constant: $(cf)' = cf'$
  - Sum rule: $(f + g)' = f' + g'$
  - Chain rule: $f(g(x))' = f'(g(x))g'(x)$ <=> $(f \circ g)' = (f' \circ g)g'$
  - …
- Boring? Use a solver like Wolfram alpha…

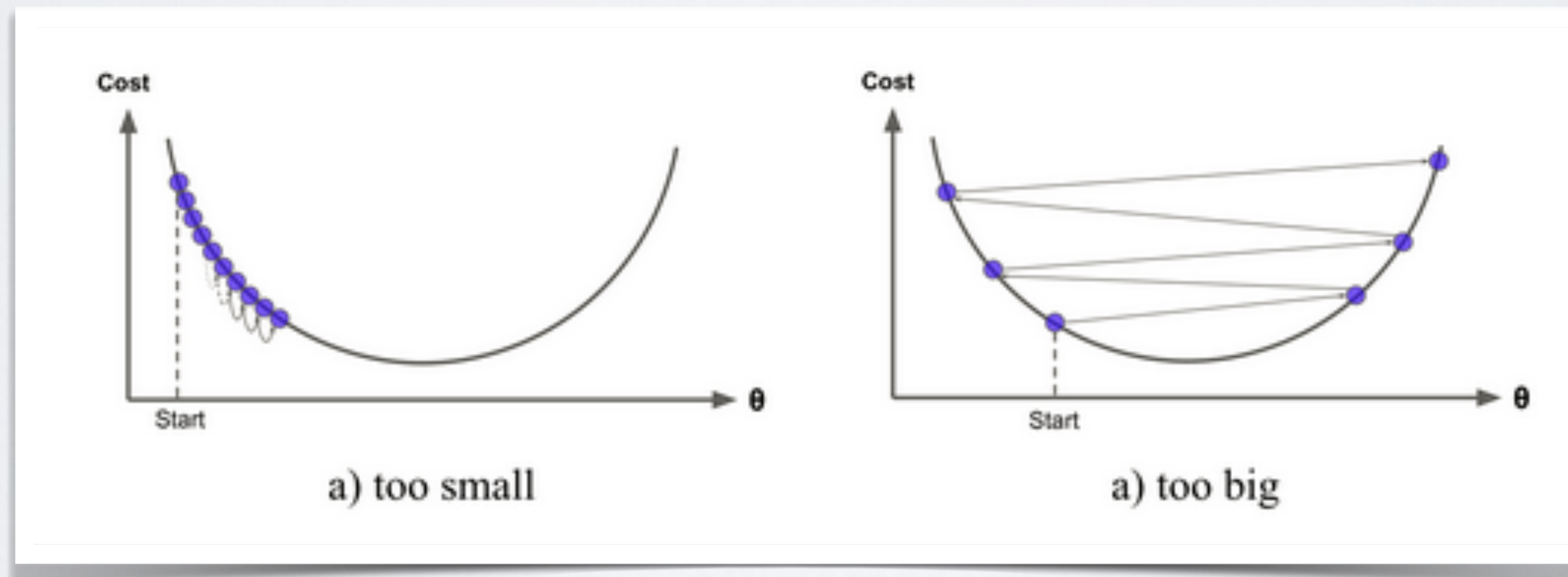
GRADIENT DESCENT

- Consider a function $f(x) = x^2 + 3$
  - $f'(x) = 2x$
  - x=4: Gradient= 8
  - x=2: Gradient=4
  - x=-1: Gradient=-2
  - x=-0.01: Gradient = -0.02
  - x=0: Gradient=0
- Start at random, x=4, $\alpha = 0.25$
  - x=4-(0.25*8)=2
  - x=2-1=1
  - x=1-0.25=0.75
  - x=0.75-0.18=0.56….
  - => Converge to x=0



GRADIENT DESCENT

- Of course, the choice of α will affect the learning
  - Strategies exist to adapt α dynamically
- We stop when reaching a fix point

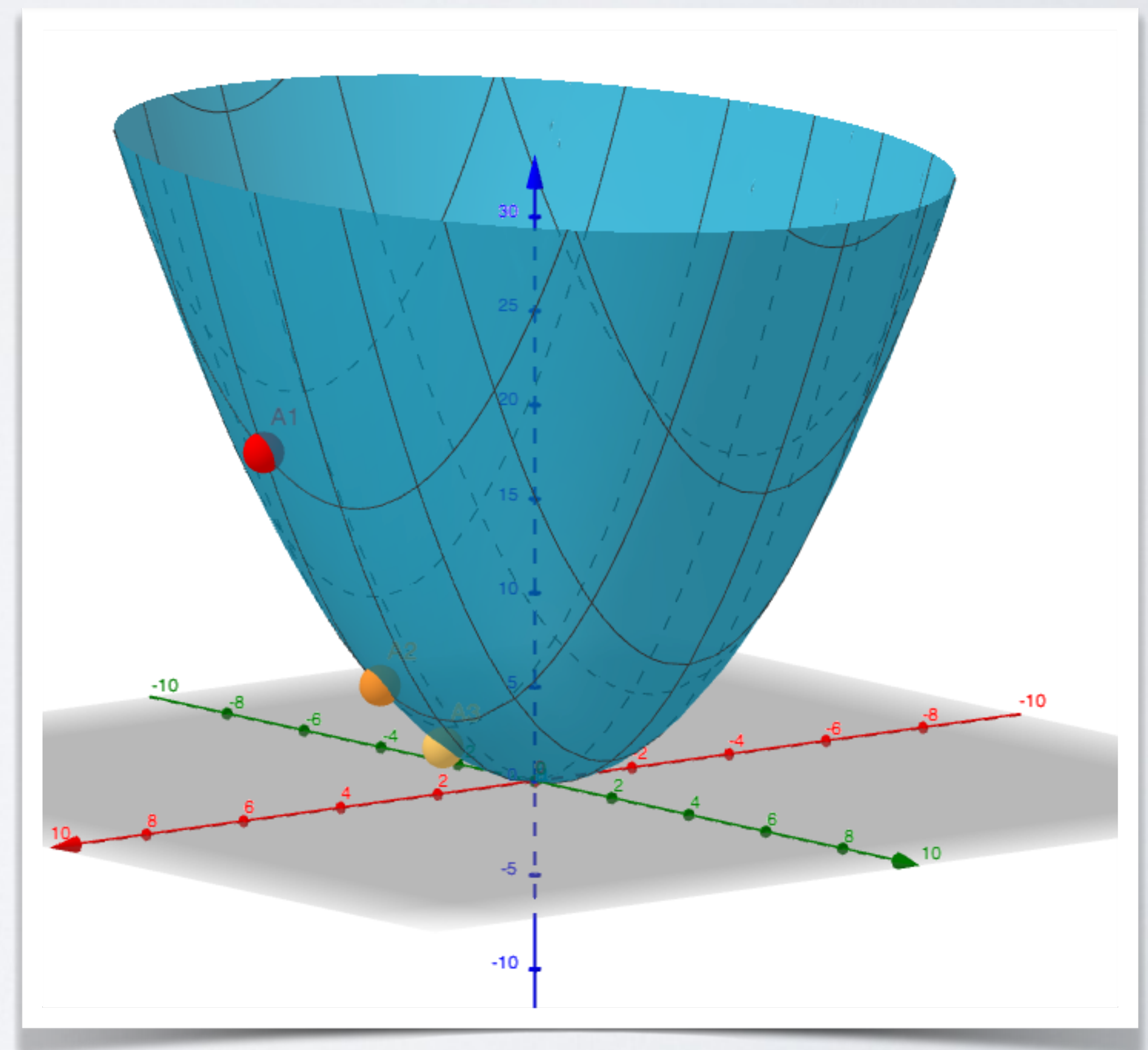


https://medium.com/analytics-vidhya/neural-networks-part-3-understanding-back-propagation-learning-rates-3482a981a2f0

GRADIENT DESCENT

- Consider a function $f(x, y) = x^2 + 0.5y^2$
  - $\frac{\partial f}{\partial x} = 2x + 0$
  - $\frac{\partial f}{\partial y} = 0.5(2y) = y$

- x=4,y=-2, $\alpha = 0.25$
  - x=4-2=2,   y=-2-(-0.5)=-1.5
  - x=2-1=1,   y=-1.5-(-0.375)=-1.125
  -

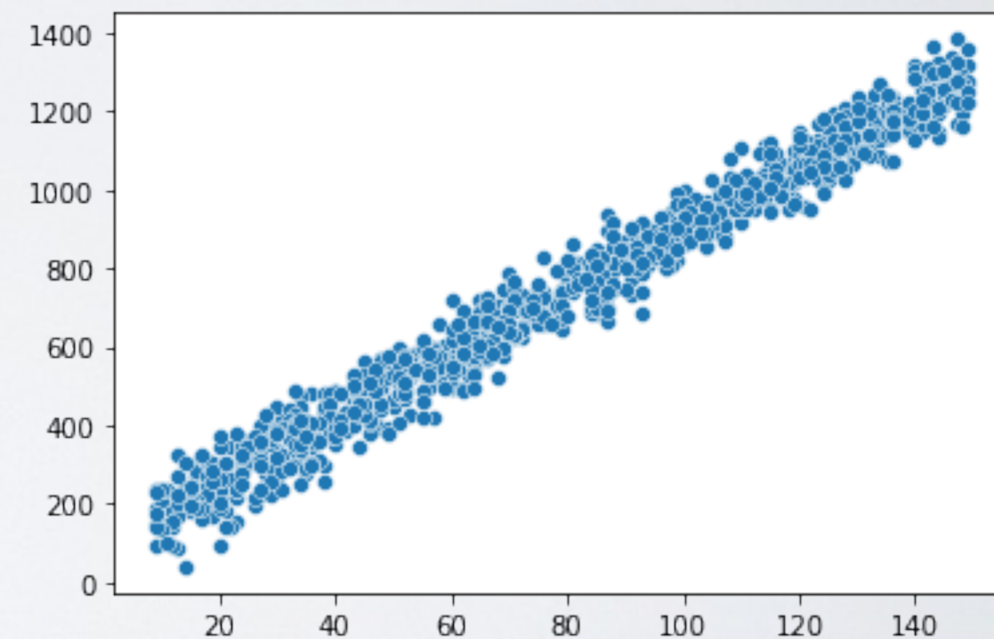


GRADIENT DESCENT

- Practical example for parameter exploration
  - Apartments defined by surface
  - Price =

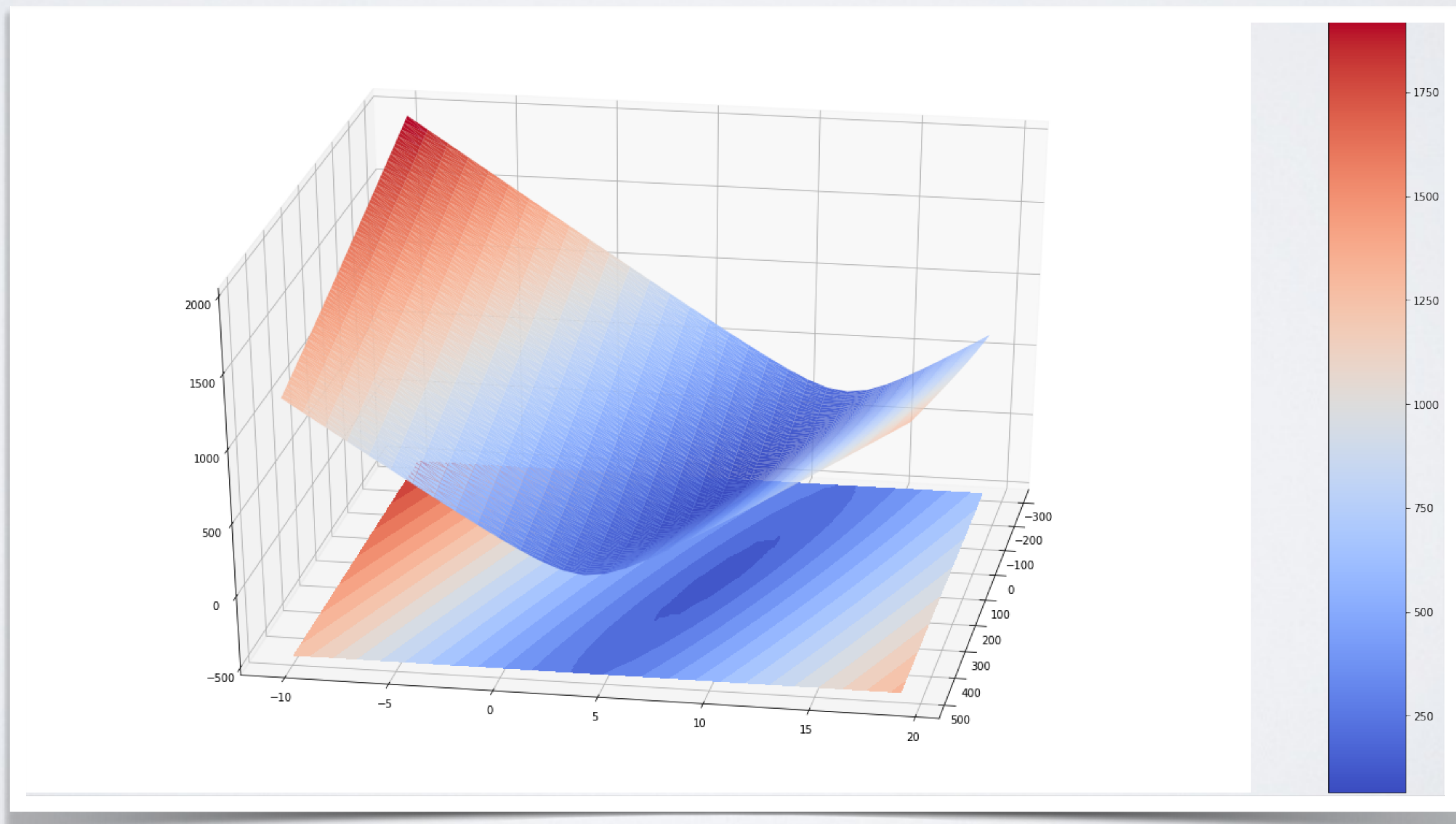
```
prices = 100+surface*8+np.random.normal(0, 50, len(surface))
```

  - Define linear regression:
  - $y^i = \beta_0 + \beta_1 x_i$
  - $\underset{\beta_0,\beta_1}{\arg\min} \frac{1}{N} \sum_i^N (y_i - (\beta_0 + \beta_1 x_i))^2$



GRADIENT DESCENT

- Plotting the solution space:
  - $x : \beta_0, y : \beta_1, z = \frac{1}{N}\sum_{i}^{N}(y_i - (\beta_0 + \beta_1 x_i))^2$
  - (Here, exhaustive search: very costly)



GRADIENT DESCENT

- Computing gradients for linear regression with 2 parameters
  - $\ell(\beta_0, \beta_1) = \frac{1}{N} \sum_i^N (y_i - (\beta_0 + \beta_1 x_i))^2$
  - $\frac{\partial \ell}{\partial \beta_0} = \frac{1}{N} \sum_i^N 2(y_i - (\beta_0 + \beta_1 x_i))(-1)$
  - $\frac{\partial \ell}{\partial \beta_0} = \frac{1}{N} \sum_i^N - 2(y_i - (\beta_0 + \beta_1 x_i))$
  - $\frac{\partial \ell}{\partial \beta_0} = \frac{-2}{N} \sum_i^N (y_i - \hat{y}_i)$
  - Prediction is too low=> Increase β_0 (proportionally to error).
  - Too high=> Decrease β_0 if, decrease if too high

GRADIENT DESCENT

- Computing gradients for linear regression with 2 parameters
  - $\ell(\beta_0, \beta_1) = \frac{1}{N} \sum_{i}^{N} (y_i - (\beta_0 + \beta_1 x_i))^2$
  - $\frac{\partial \ell}{\partial \beta_1} = \frac{1}{N} \sum_{i}^{N} 2(y_i - (\beta_0 + \beta_1 x_i))(-x_i)$
  - $\frac{\partial \ell}{\partial \beta_1} = \frac{1}{N} \sum_{i}^{N} - 2x_i(y_i - (\beta_0 + \beta_1 x_i))$
  - $\frac{\partial \ell}{\partial \beta_1} = \frac{-2}{N} \sum_{i}^{N} x_i(y_i - \hat{y}_i)$
- x_i positive=>Lower if too high, increase if too low.
- x_i negative=>Increase if too high, decrease if too low.
- If two items with equal absolute error of opposite sign, different $|x_i|$=> gradient correct largest $|x_i|$ (increase the slope)
- If x_i =0, the coefficient have no effect anyway

GRADIENT DESCENT

- Generic case: more than a single variable
  - $\ell(b, w) = \frac{1}{N} \sum_{i}^{N} (y_i - (b + wx_i))^2$
    - Vecteur form: w: vector of weights, x_i: vector of features
  - Same derivation:
    - $\frac{\partial \ell}{\partial w} = \frac{-2}{N} \sum_{i}^{N} x_i(y_i - \hat{y}_i)$
- Partial gradient for each feature (for each observation) is proportional to the feature value for this observation
  - A same error for an observation can contribute differently for each coefficient:
    - Increase or decrease (sign of the feature)
    - Strong or weak effect (magnitude of the feature)

REGULARIZATION

REGULARIZATION

- We have seen that a drawback of ML methods is that they can overfit

- When the ML objective can be clearly expressed, there is a generic way to limit overfitting: regularization
    - Two types of regularization:
        - L1 or Lasso regularization
        - L2 or Ridge regularization

L2 REGULARIZATION

- L2 or Ridge Regularization
  - $\ell(b, w) = \frac{1}{N} \sum_{i}^{N} (y_i - (b + \sum_{j}^{p} (w_j x_{ij})))^2 + \lambda \sum_{j}^{p} w_j^2$
  - $\ell(b, w) = \frac{1}{N} \sum_{i}^{N} (y_i - \hat{y}_i)^2 + \lambda \sum_{j}^{p} w_j^2$
  - Notation: $\sum_{j}^{p} w_j^2 = \|w\|_2^2$

L2 REGULARIZATION

- Expressed as a general principle
  - $$\ell(b,w) = \frac{1}{N}\sum_{i}^{N} f(y_i, \hat{y}_i, b, w) + \lambda \sum_{j}^{p} w_j^2$$
    - Some parameters are regularized, and some others might not be (intercept…)

- Intuition: we force coefficients to be small.
  - If λ=0, normal regression
  - If λ->∞, all coefficients tends towards 0
  - /!\ The magnitude of coefficients depends on the magnitude of variables!
    - Important to normalize the variables, else you will constraint more the variables of lower amplitude

L1 REGULARIZATION

- L1 or Lasso Regularization
  - Lasso: Least Absolute Shrinkage and Selection Operator
  - $\ell(b, w) = \frac{1}{N} \sum_{i}^{N} (y_i - (b + \sum_{j}^{p} (w_j x_{ij})))^2 + \lambda \sum_{j}^{p} |w_j|$
  - $\ell(b, w) = \frac{1}{N} \sum_{i}^{N} (y_i - \hat{y}_i)^2 + \lambda \sum_{j}^{p} |w_j|$
  - Notation: $\sum_{j}^{p} |w_j| = \|w\|_1$

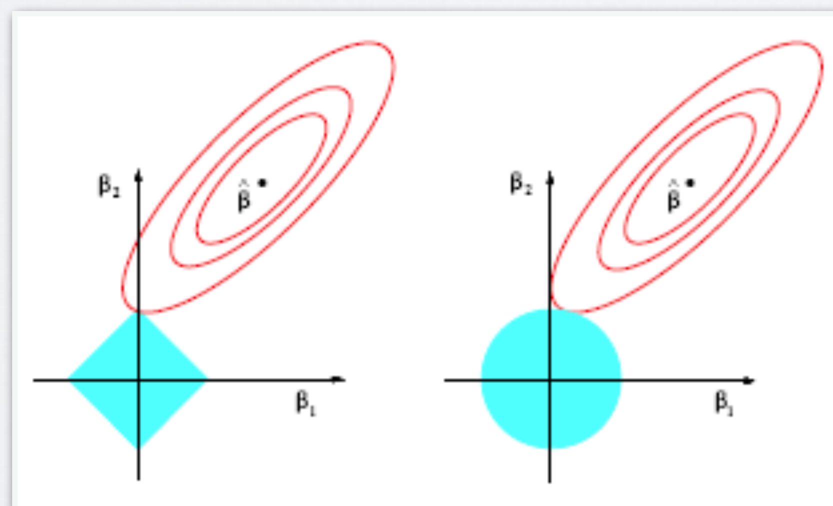
REGULARIZATION

- Similar methods, different results:
  - L1 regularization tends to force some values to be 0
  - L2 regularization tends not to attribute 0
- L1 regularization thus performs **variable selection**
  - Variables for which the coefficient is 0 can be discarded

REGULARIZATION

- Why different behaviors ?
  - We minimize the sum of error+constraints
  - Red lines represent error (every point of a circle have same error)
  - Similarly for blue.
  - Intersection is the optimal solution (for that error, minimize constraint)

- => For a same error, L1 favors 0



https://online.stat.psu.edu/stat508/book/export/html/749

REGULARIZATION

- Bayesian interpretation
  - Different priors on the coefficients

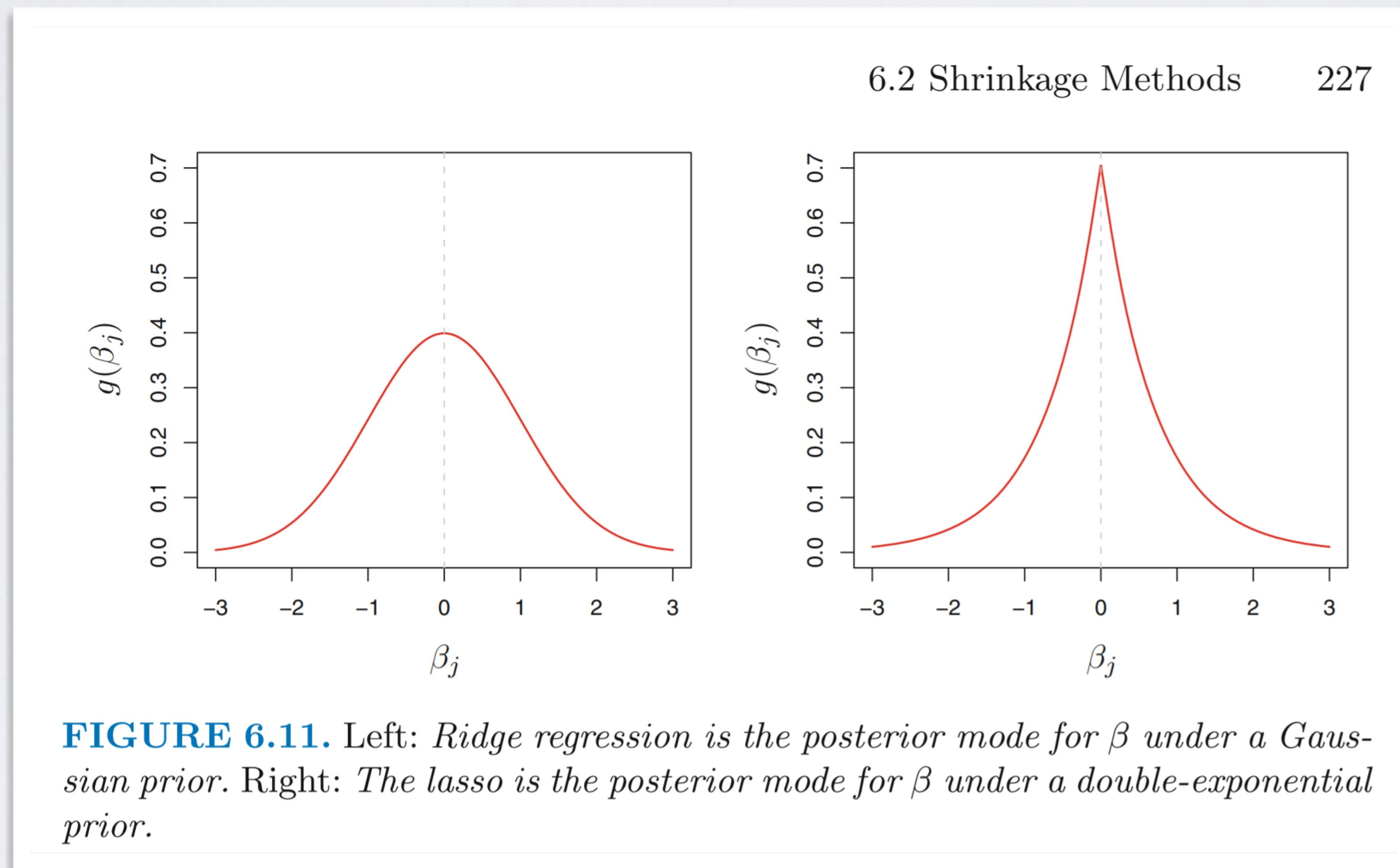

6.2 Shrinkage Methods 227

FIGURE 6.11. Left: *Ridge regression is the posterior mode for β under a Gaussian prior.* Right: *The lasso is the posterior mode for β under a double-exponential prior.*

https://ekamperi.github.io/mathematics/2020/08/02/bayesian-connection-to-lasso-and-ridge-regression.html

ELASTIC NET

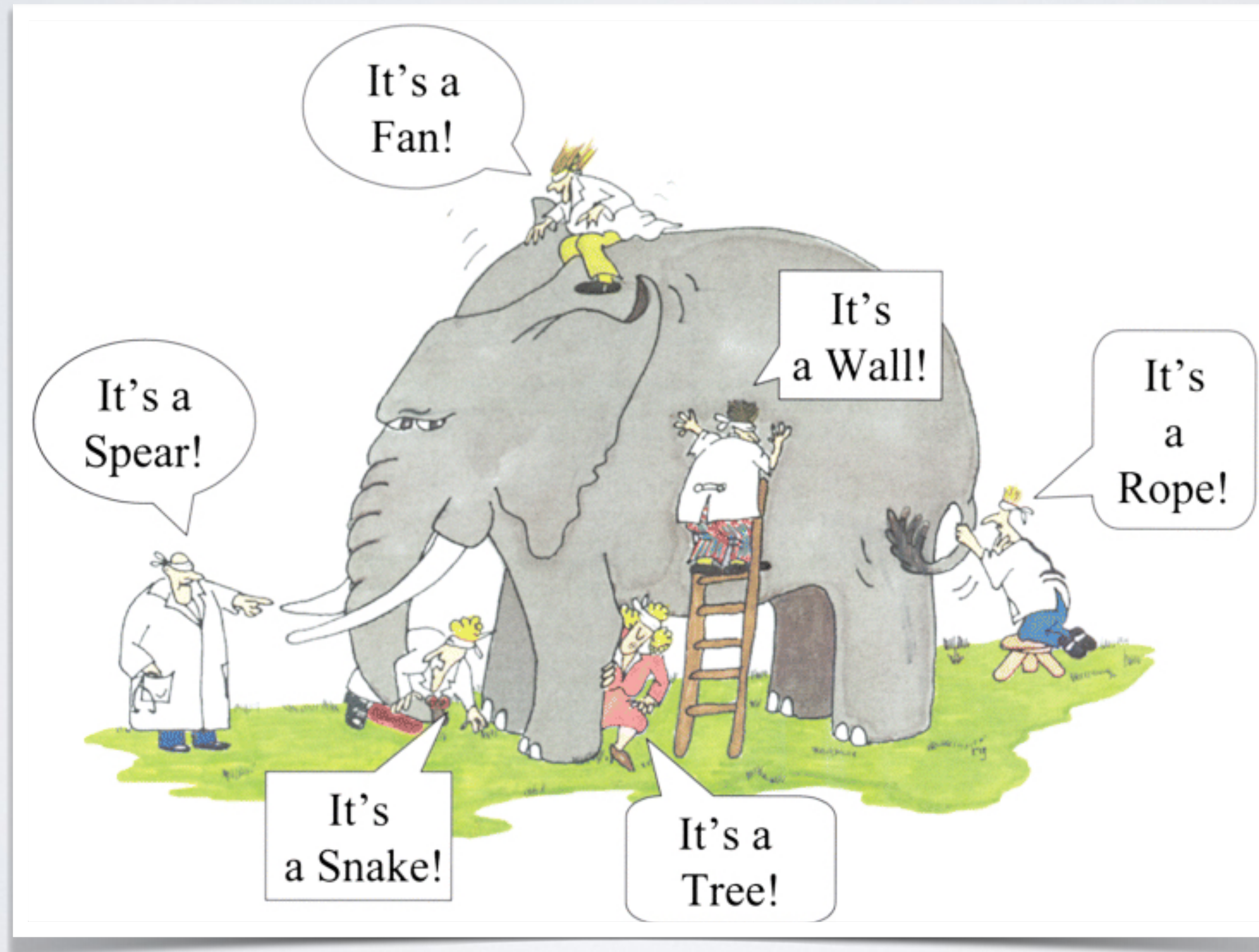
- Best of both worlds :)
- $\ell(b, w) = \frac{1}{N} \sum_{i}^{N} (y_i - \hat{y}_i) + \lambda_1 \sum_{j}^{p} |w_j| + \lambda_2 \sum_{j}^{p} w_j^2$

ENSEMBLE LEARNING

ENSEMBLE LEARNING

- Ensemble learning is a general principle:
  - All models have strengths and weaknesses
    - e.g., linear models struggle with non-linearities but are good at extrapolation
    - Decision trees are good at capturing non-linearities, but struggle with extrapolation
  - Could we combine the strengths of various models?
    - Direct application: **Stacking**
    - Using multiple times the same model: **Bagging**
    - Training models specifically to solve other weaknesses: **Boosting**

ENSEMBLE LEARNING



STACKING

- In the simplest approach, various models (different approaches, same approach with different parameters) are trained on the same dataset

- Their predictions are then combined:
  - Regression: averaging. Average values of the classifiers (possibly weighted)
  - Classification:
    - Voting: class with the most vote
    - Soft / Averaging: average of probabilities yielded by the classifier

- Weaknesses:
  - What if several models make the same mistake? (Correlation of errors…)
  - What if we merge good models and poor models?

STACKING

- A possible solution to stacking is to use a meta-model:
  - The prediction made by each individual model is considered as a feature for the meta-model
  - The meta-model is trained as any ML model with the original target, but using sub-models outputs as features.
- Any model can be used as meta-model
- Famous for winning the $1M prize of the 2009 Netflix prize.
  - 100+ individual predictors

BAGGING

- Bagging is an ensemble methods, but differ from stacking in two main ways:
  - The various individual predictors are made of the same algorithm
  - Each algorithm is trained on a subset of the original data
    - Different subsets on all variables
    - And/Or trained only on some variables
    - => Various strategies exist.

- Advantages over stacking:
  - All models are comparable, less chances to average "good" and "bad" models
  - Can be understood as "lower the Variance", i.e., prevent overfit.
    - Remember the Bias/Variance tradeoff ? Expressive models overfits => high variance.
    - The definition of variance is high variation over the "average" of multiple models…

$$\mathrm{Bias}_D\big[\hat{f}(x;D)\big] = \mathrm{E}_D\big[\hat{f}(x;D)\big] - f(x)$$

$$\mathrm{Var}_D\big[\hat{f}(x;D)\big] = \mathrm{E}_D[\big(\mathrm{E}_D[\hat{f}(x;D)] - \hat{f}(x;D)\big)^2].$$

D: subsets.

X: all elements in all subsets

BAGGING: RANDOM FOREST

- Random forest is the most famous bagging algorithm
  - It is based on decision trees (thus the name *forest*…)
  - A direct application of bagging
    - innovations to bagging came from random forests
- Trees are good candidates for bagging because overfit is their main problem
  - What is similar between trees will stay, and when they disagree, taking the average of all the errors should get close to right answer.
    - Similar to "Wisdom of the crowds"

RANDOM FOREST

- Set
  - Parameters of individual trees (not too simple, not too large…)
  - Averaging function
  - #trees

- What is specific is the subsamble strategy
  - What is key is to avoid correlation between trees, i.e., train on different data
  - Subsample observations: With replacement. Sample n at random among n items
    - Variants: m among n. Or without replacement: random samples, or "folds" (each observation used in a single tree, but requires lot of data)…
  - Specific to trees: subsample of variables at each node: to chose the best split, restrain to a random fraction of variables.
    - Impose diversity in the trees

BOOSTING

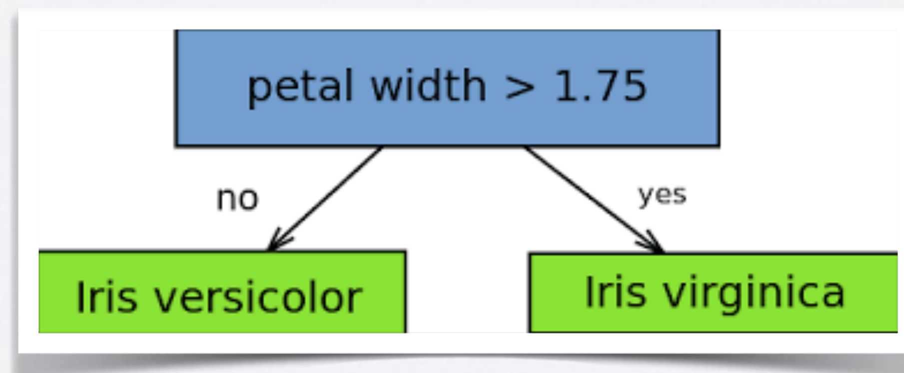
BOOSTING

- Again, a general principle
- We train various models in sequence
  - First, train a normal, first model
    - Usually, this model will be tuned to be relatively simple, and thus underfit=>Weak learners
  - Then, extract the errors of the model (incorrect classes/residuals).
  - Train a second model, focusing on predicting the errors missed by the first model
  - Update the main model and recompute the errors
  - Repeat until we cannot improve anymore
- Final prediction is the sum of all weak learners (not average: each method *corrects*, *complement* previous ones)

$$F_T(x) = \sum_{t=1}^{T} f_t(x)$$

ADABOOST

- First boosting method to reach wide recognition
- Method for classification
- Weak learners are *decision stumps*
  - Choose only one variable. Split it only once



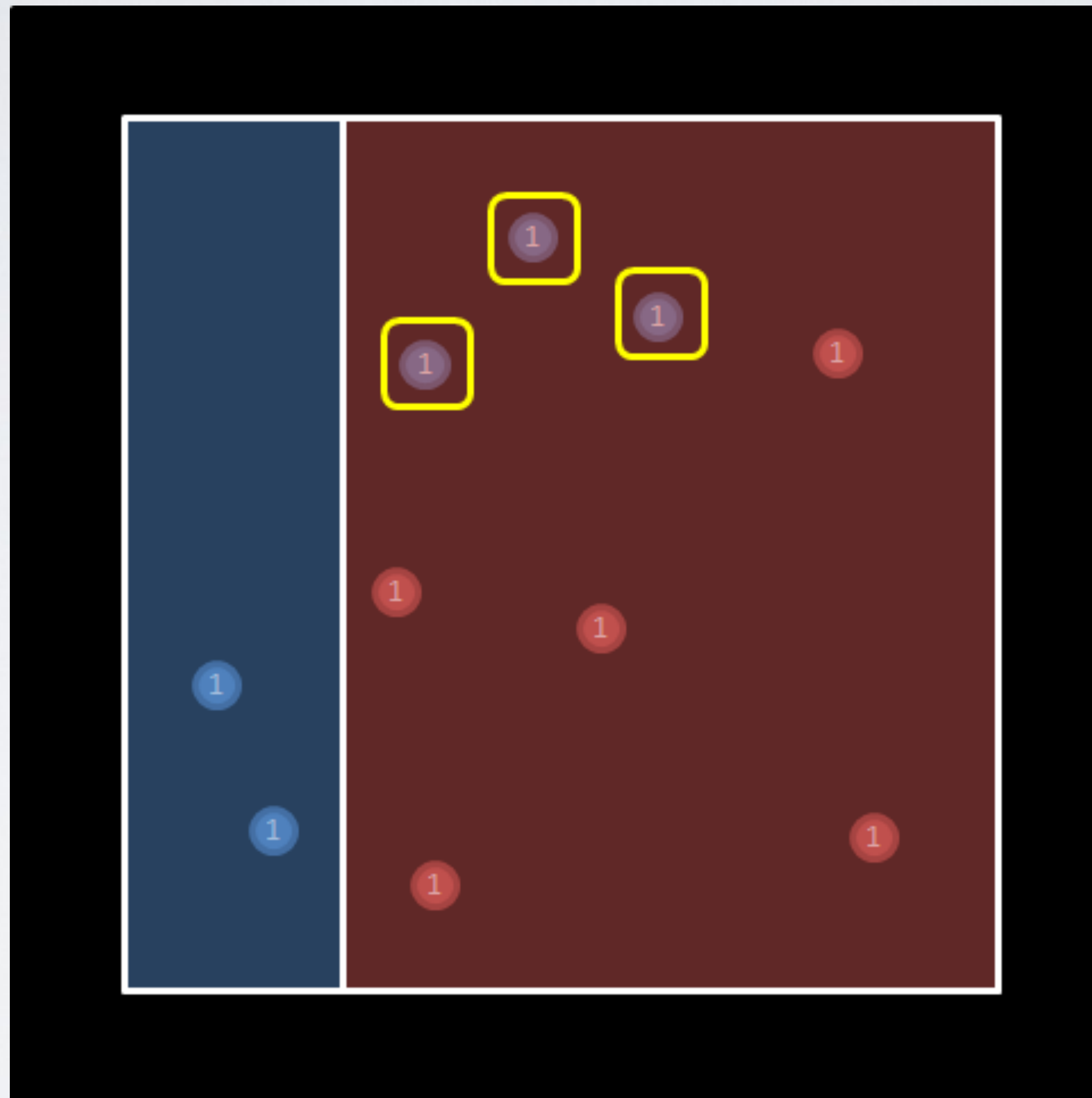
ADABOOST

- Error to minimize at each step m: $\epsilon_m = \dfrac{\sum_i^N w_i^m I(f_m(x_i) \neq y_i)}{\sum_i^N w_i^m}$
  - I(true)=1, I(false)=0
  - w_i^m: weight of element i at step m
  - Interpretation: fraction of weights w_i^m for misclassified elements
  - Weights are initialized at 1: first, minimize fraction of errors

- $w_i^{m+1} = w_i^m e^{\alpha_m I(fm(x_i) \neq y_i)}$
  - Updates weights of <u>misclassified</u> items($e^0 = 1$) by a coefficient proportional to the error
  - With $\alpha_m = \ln\left(\dfrac{1-\epsilon_m}{\epsilon_m}\right)$
  - e^{α_m}: correct / incorrect =>
    - Sum of Weights of correct pts: correct *(w=1)=correct
    - Sum of Weigths of incorrect pts: incorrect *(w=correct/incorrect) =correct
    - Incorrectly classified now weights equal to correctly classified.

ADABOOST

0.7/0.3=2.33

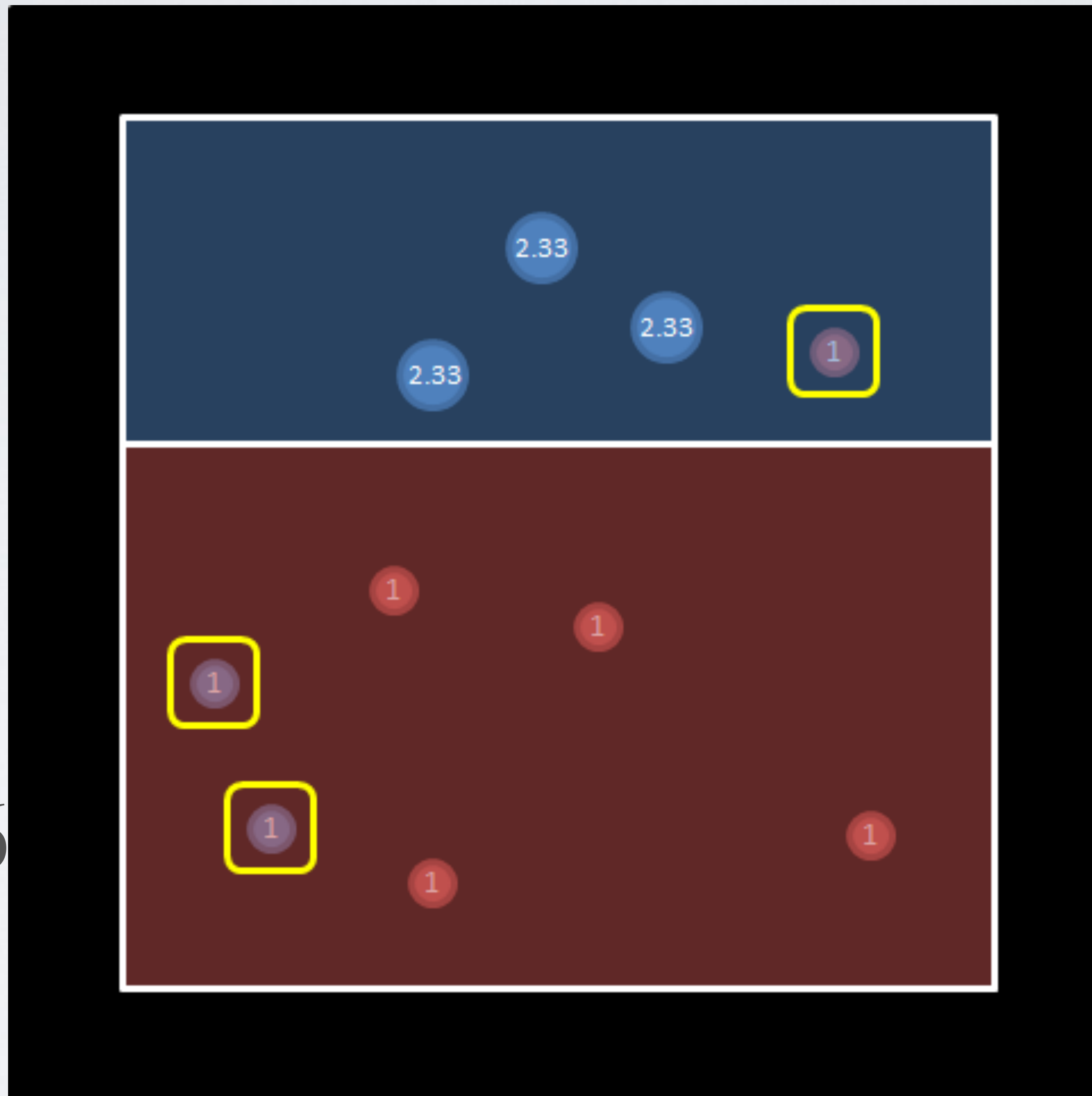


Weight incorrect=3*2.33=7
Weight correct=7*1=7

https://medium.datadriveninvestor.com/how-do-boosting-algorithms-work-5d2c734aca4c

ADABOOST

$$0.7/0.3=2.33$$



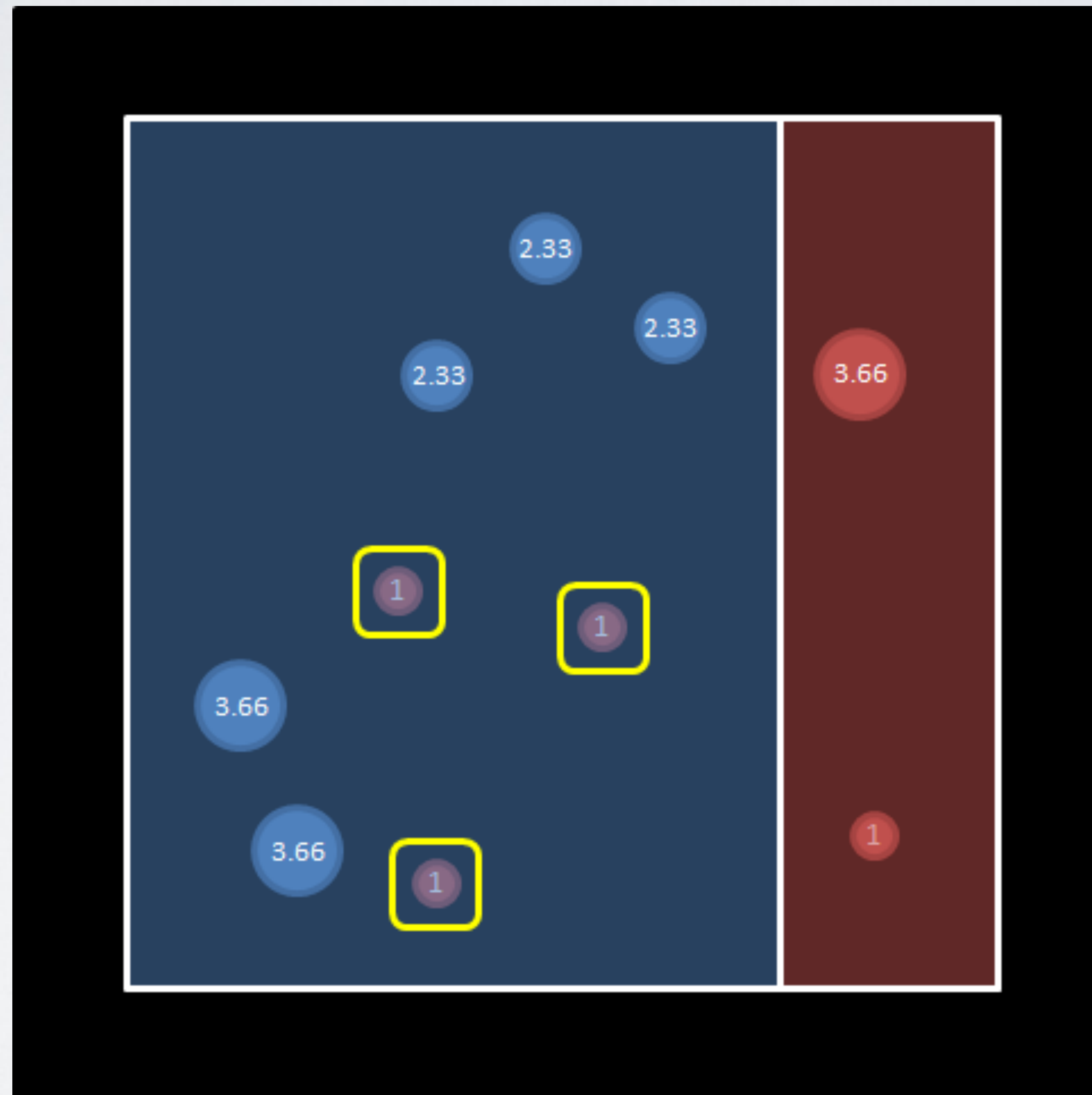
Weight incorrect=3*3.66=11
Weight correct=4*1+3*2.33=11

$$\frac{11}{14}/\frac{3}{14}=3.66$$

https://medium.datadriveninvestor.com/how-do-boosting-algorithms-work-5d2c734aca4c

ADABOOST

$$\frac{11}{14} / \frac{3}{14} = 3.66$$



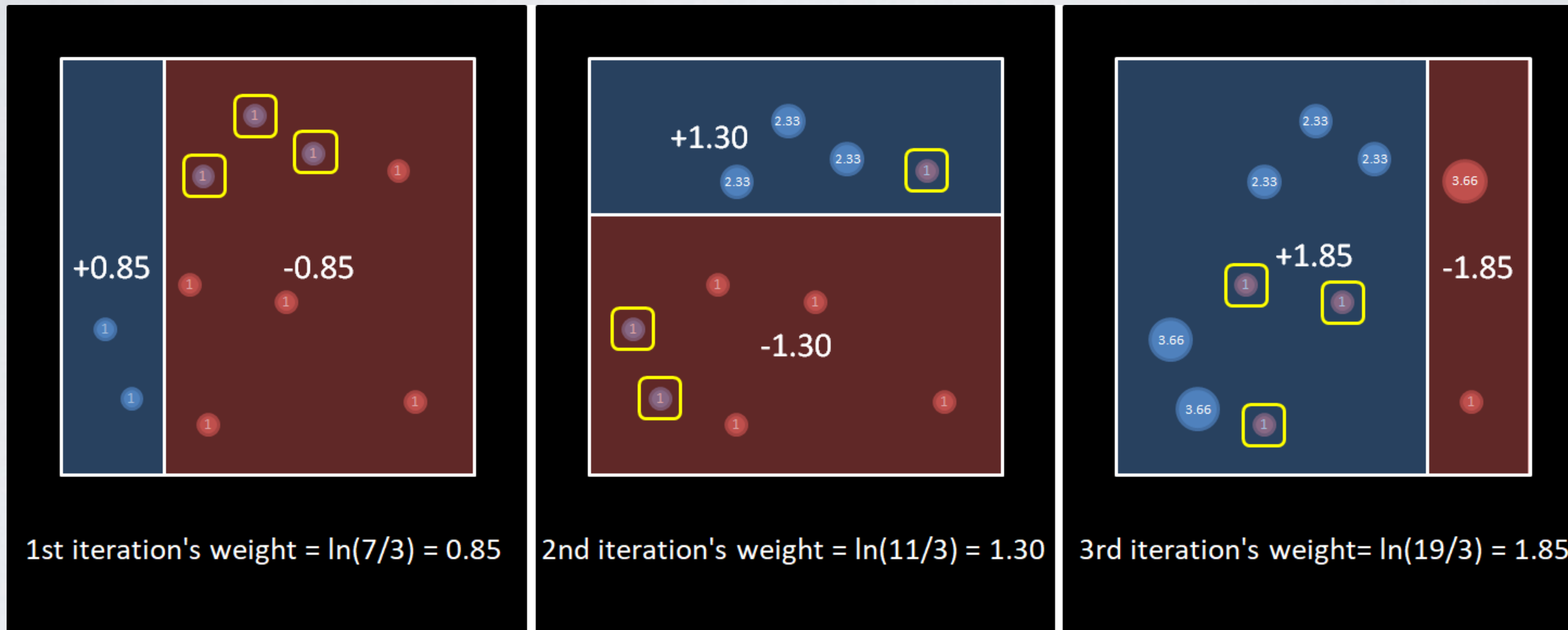
$$\frac{19}{22} / \frac{3}{22} = 6.33$$

https://medium.datadriveninvestor.com/how-do-boosting-algorithms-work-5d2c734aca4c

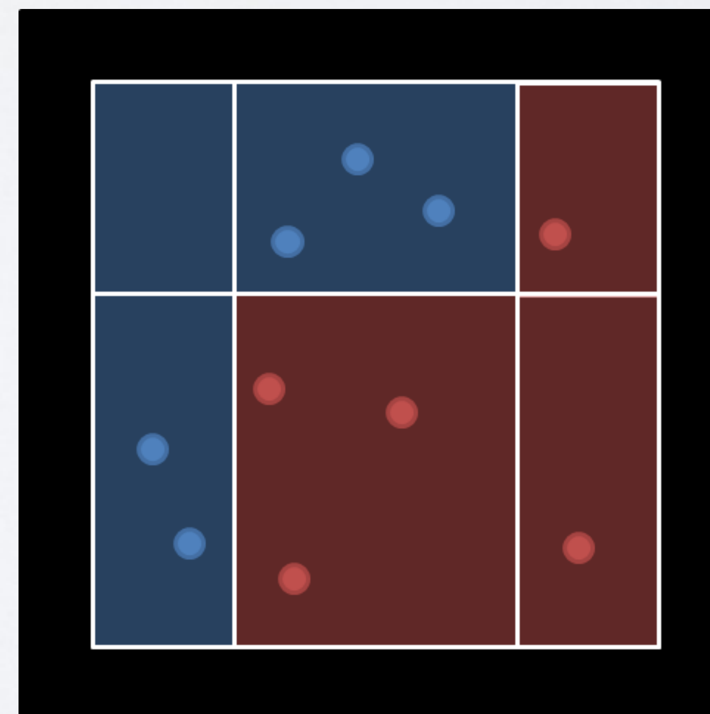
ADABOOST

- Finally, we need to combine our various weak learners into a single prediction
  - $F_m(x) = F_{m-1}(x) + \alpha_m h_m(x) = \sum_m \alpha_m h_m(x)$
    - The new set of rules at step m is the previous set of rules to which we add the new rule weighted by coefficient $\alpha_m = \ln\left(\frac{1-\epsilon_m}{\epsilon_m}\right)$
    - ln(Correct /errors): 0 if as many correct as error (ignore if rangom…), the more we have correct results, the higher the value. (Infinite with perfect solution…)

ADABOOST



+0.85 +1.30 +1.85 **4.00**	-0.85 +1.30 +1.85 **2.30**	-0.85 +1.30 -1.85 **-1.40**
+0.85 -1.30 +1.85 **1.40**	-0.85 -1.30 +1.85 **-0.30**	-0.85 -1.30 -1.85 **-4.00**



https://medium.datadriveninvestor.com/how-do-boosting-algorithms-work-5d2c734aca4c

ADABOOST

- Why does it work?
  - Intuitively:
    - We force the latest weak learner to focus on what was missed by others.
    - The weights of models are stronger when we solve errors found in many other models
    - "ADA: adaptative (weights adapt based on previous step)"
  - Theoretically:
    - It can be shown that Adaboost minimizes the Exponential loss, which is a way to estimate the probability of having a given class given the data
    - $\underset{f(X)}{\operatorname{argmin}} \mathbb{E}_{Y|X} e^{-Yf(X)} = \frac{1}{2} \log \frac{\mathbb{P}(Y = 1 | X)}{\mathbb{P}(Y = -1 | X)},$
    - (A posteriori improvement and explanations, everything not fully clear IMO…)

XGBOOST

XGBOOST

- As of today, certainly the most popular method among those not using neural networks
- Used in winning solution in countless ML challenges
  - And at Google, Amazon, Uber…
- Both :
  - A method described in a scientific paper
  - A library developed and improved by a community
    - Changes in the ML scientific culture…

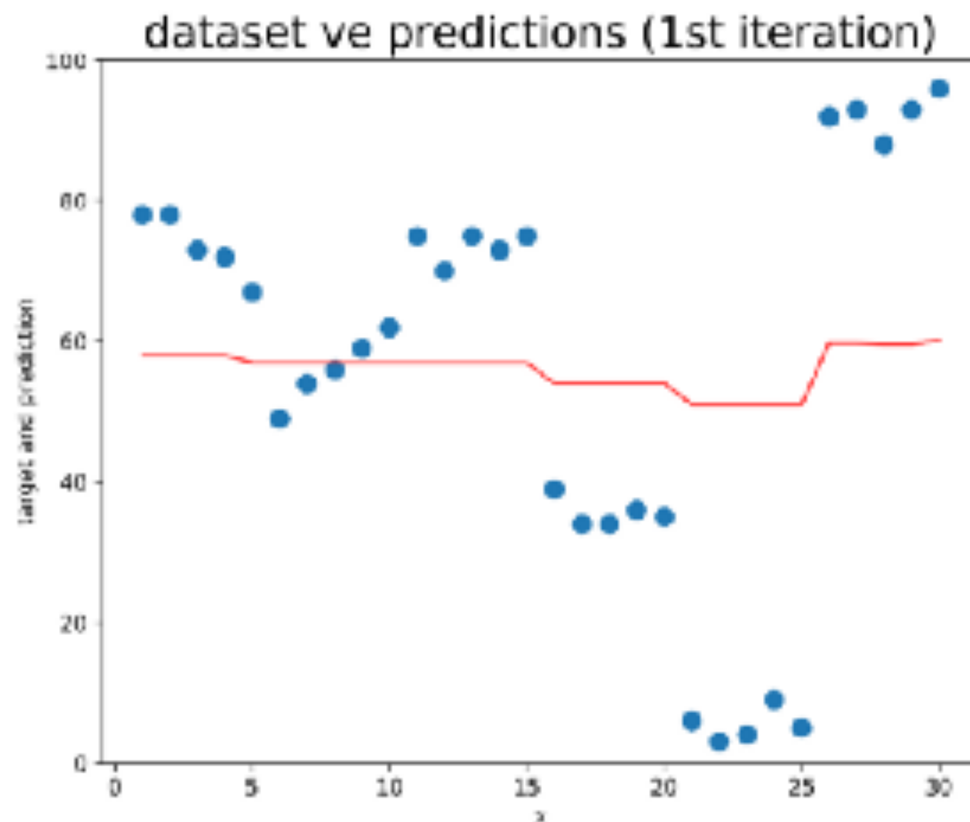
XGBOOST

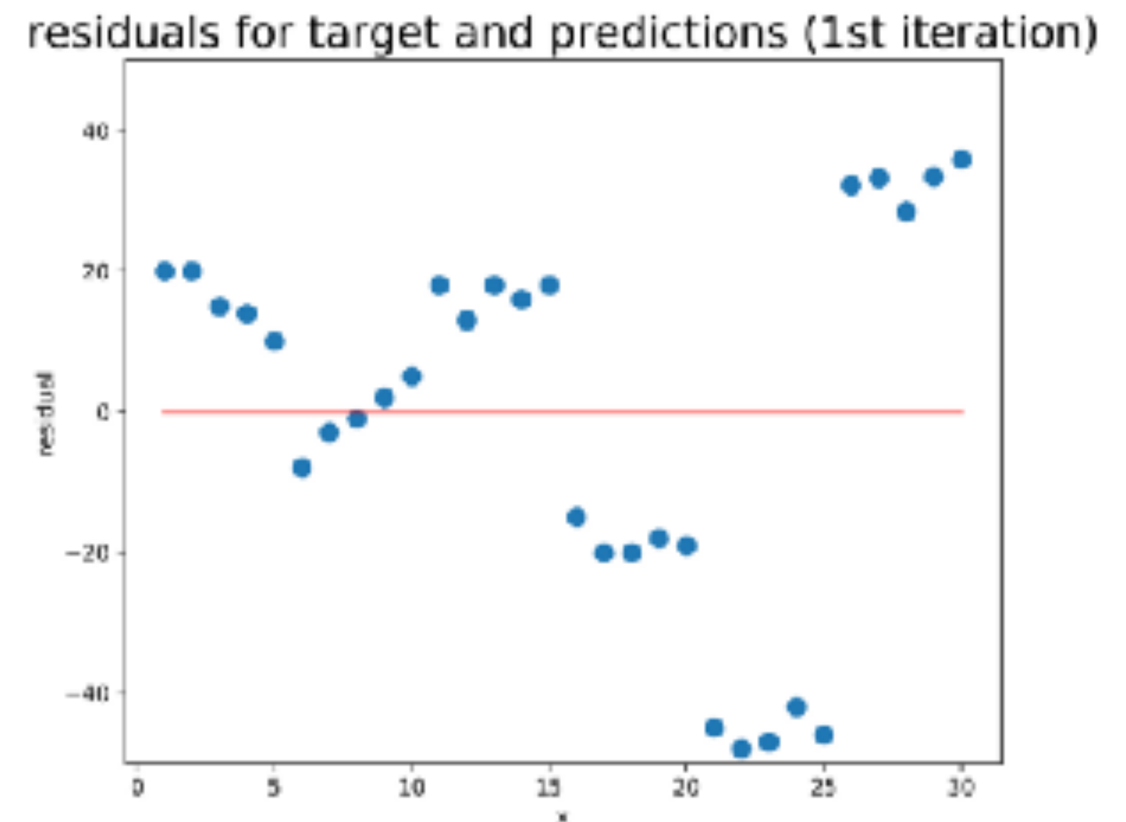
- In a few words:
  - A tree boosting methods
    - Can be used for classification and regression
  - Weak learners not as weak as in AdaBoost
    - Default to 3 or 6 levels max
  - Introduces Regularization
    - Each new leaf add some regularization cost
  - Gradient Boosting method:
    - Explicitly do a gradient-descent like approach

GRADIENT BOOSTING

- Gradient boosting is the application of boosting to explicit gradient descent

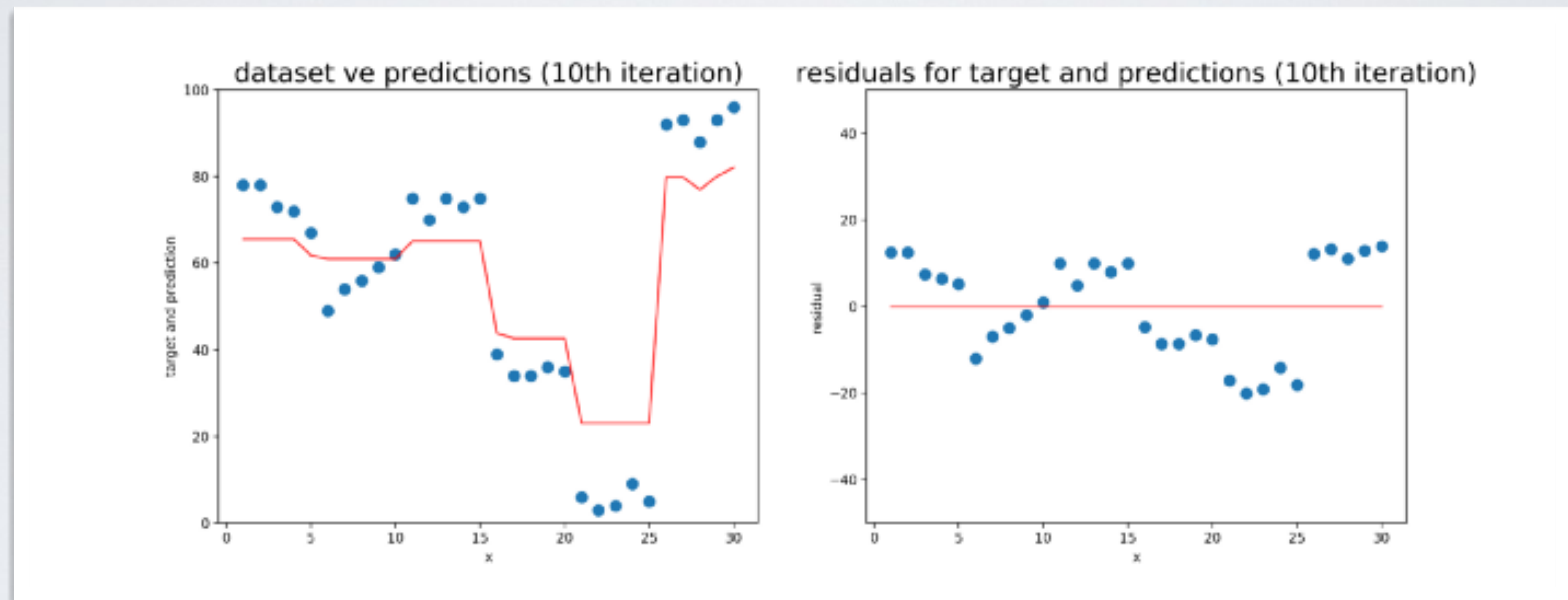
GRADIENT BOOSTING





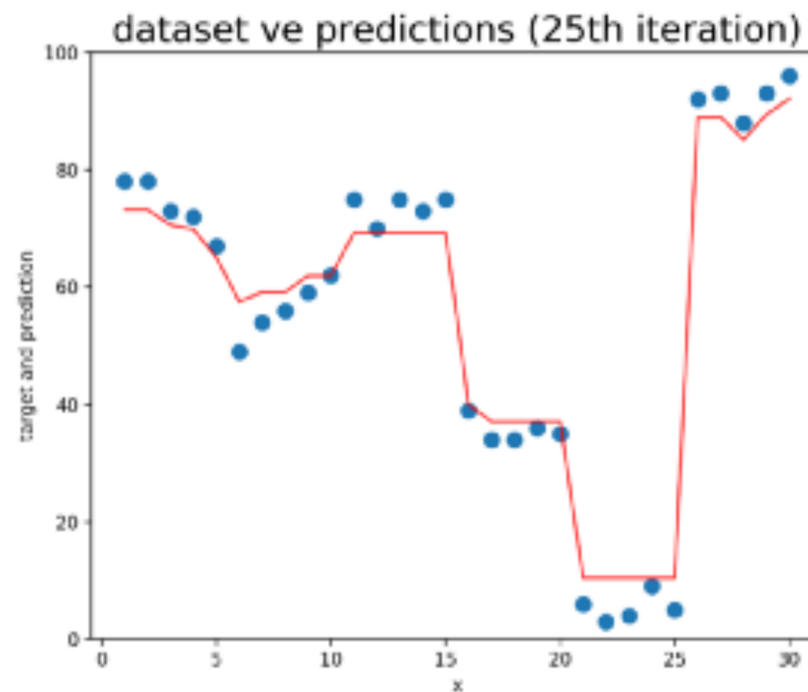
https://medium.datadriveninvestor.com/how-do-boosting-algorithms-work-5d2c734aca4c

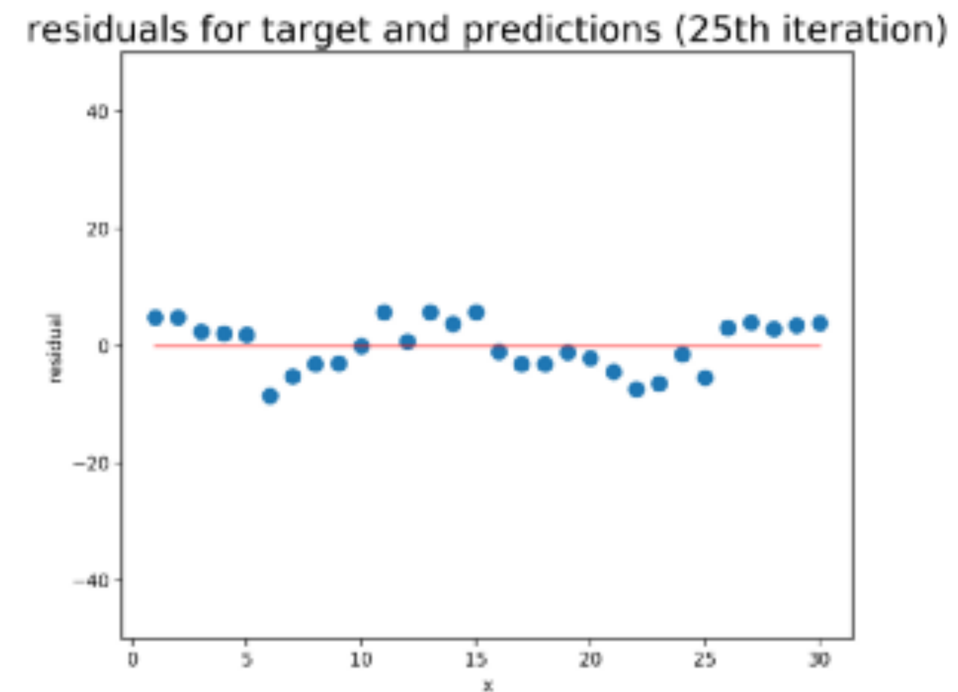
GRADIENT BOOSTING



https://medium.datadriveninvestor.com/how-do-boosting-algorithms-work-5d2c734aca4c

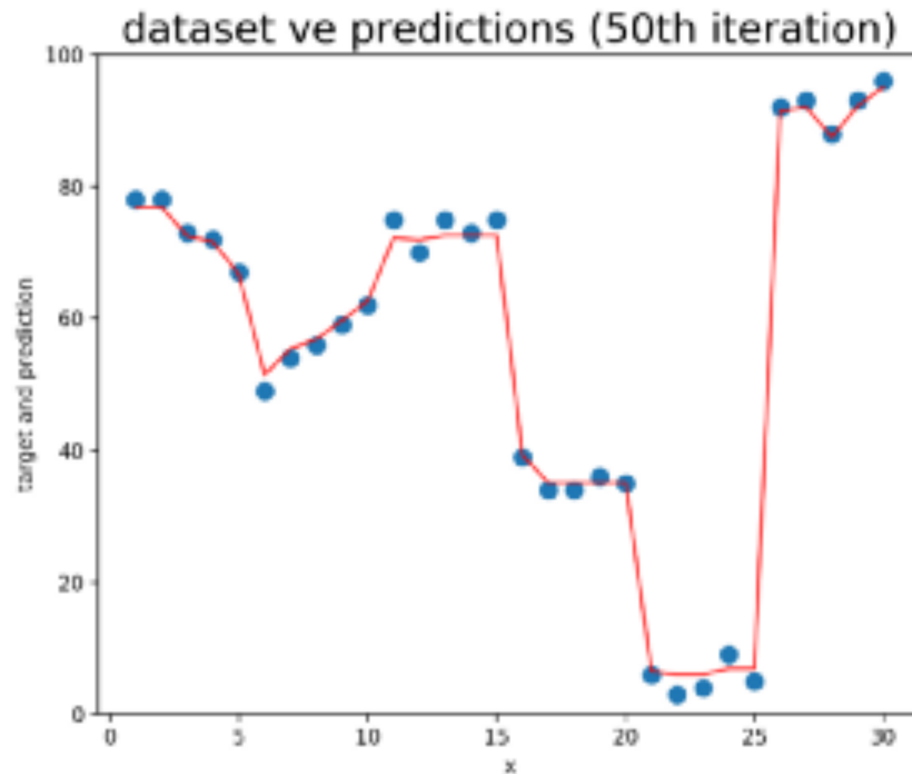
GRADIENT BOOSTING

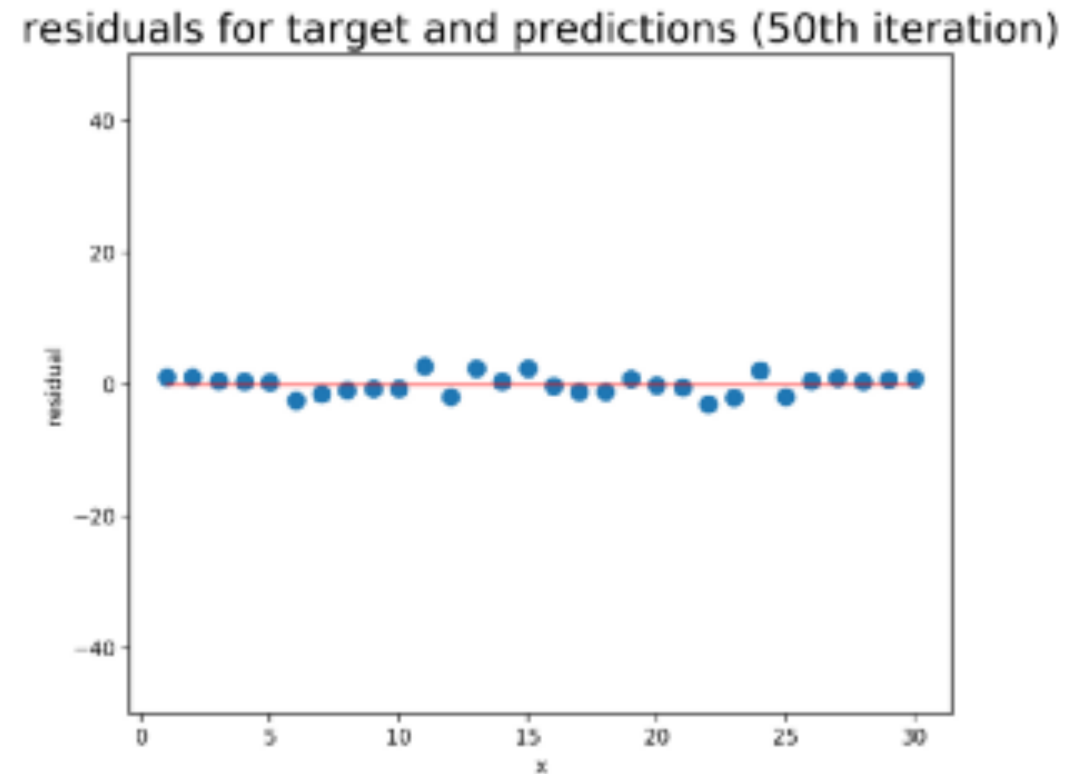




https://medium.datadriveninvestor.com/how-do-boosting-algorithms-work-5d2c734aca4c

GRADIENT BOOSTING





https://medium.datadriveninvestor.com/how-do-boosting-algorithms-work-5d2c734aca4c

XGBOOST IN A NUTSHELL

$$\text{obj}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \omega(f_i)$$

$$= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \sum_{i=1}^{t} \omega(f_i)$$

- In our loss for the tree, we decompose the prediction $\hat{y}$ as
  - Prediction given by previous tree + prediction of new tree.
  - ω regularization, explained later

XGBOOST IN A NUTSHELL

$$w_j = -\frac{G_j}{H_j + \lambda}$$

- w_j : score of a leaf (like AdaBoost)
- Using RMSE as objective:
  - G_j: Sum of errors (to residuals)
  - H_j: Number of items in the leaf
  - λ: Regularization parameter

GAIN ON A SPLIT

- $Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma$

- L, R=> Left and Right children

- Sum of regularized averaged error of the children squared, minus that of parent, minus regularization γ

https://xgboost.readthedocs.io/en/stable/tutorials/model.html

XGBOOST IN A NUTSHELL

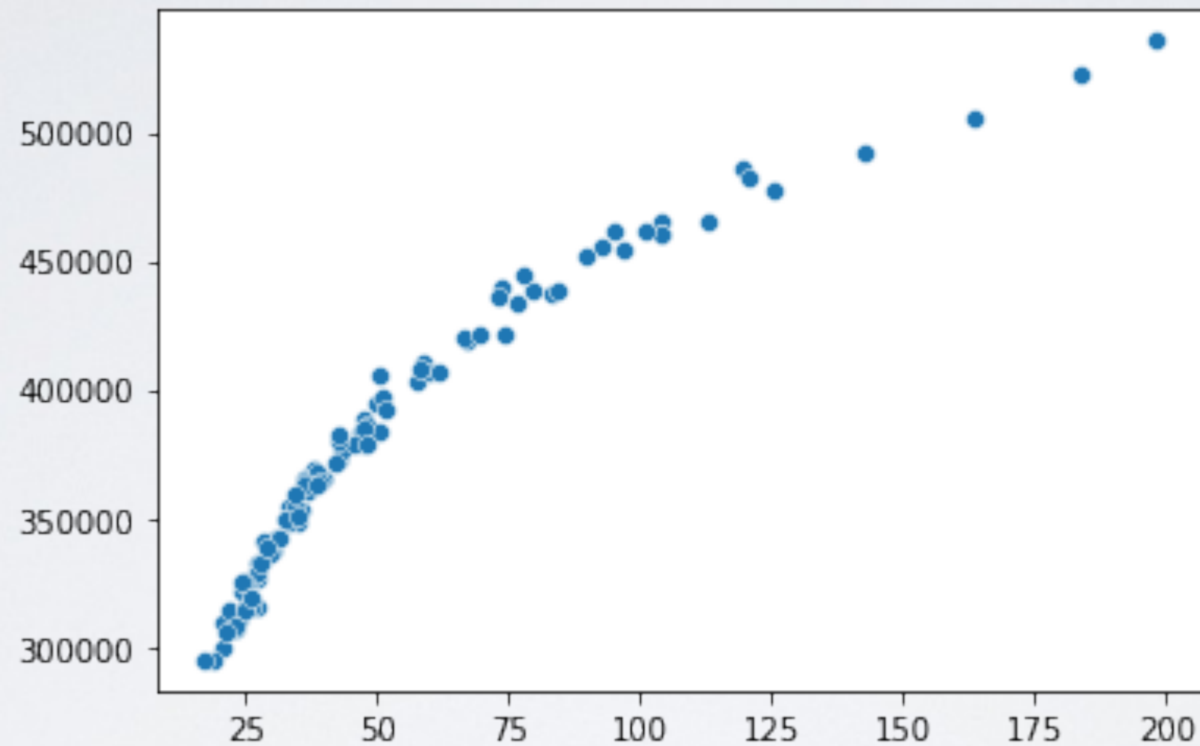
- For First tree:
  - For each leaf
    - We compute the gain to find the best possible split,
    - If regularization makes the gain negative, do nothing
    - If we reach the maximal tree depth, do nothing
    - Compute the final score of the leaf : signed error. To add to the final prediction
- Next tree: same process, but compute error relatively to previous tree (residuals)
- When finished, for each prediction, sum the (signed) prediction of each tree (weighted by learning rate η)

LEARNING RATE

- As in most gradient descent methods, there is a learning rate η (eta) parameter, allowing to tune how fast we converge
  - To avoid the "ping-pong" effect around global minimum
  - In practice, the prediction of the previous tree is shrinked by η

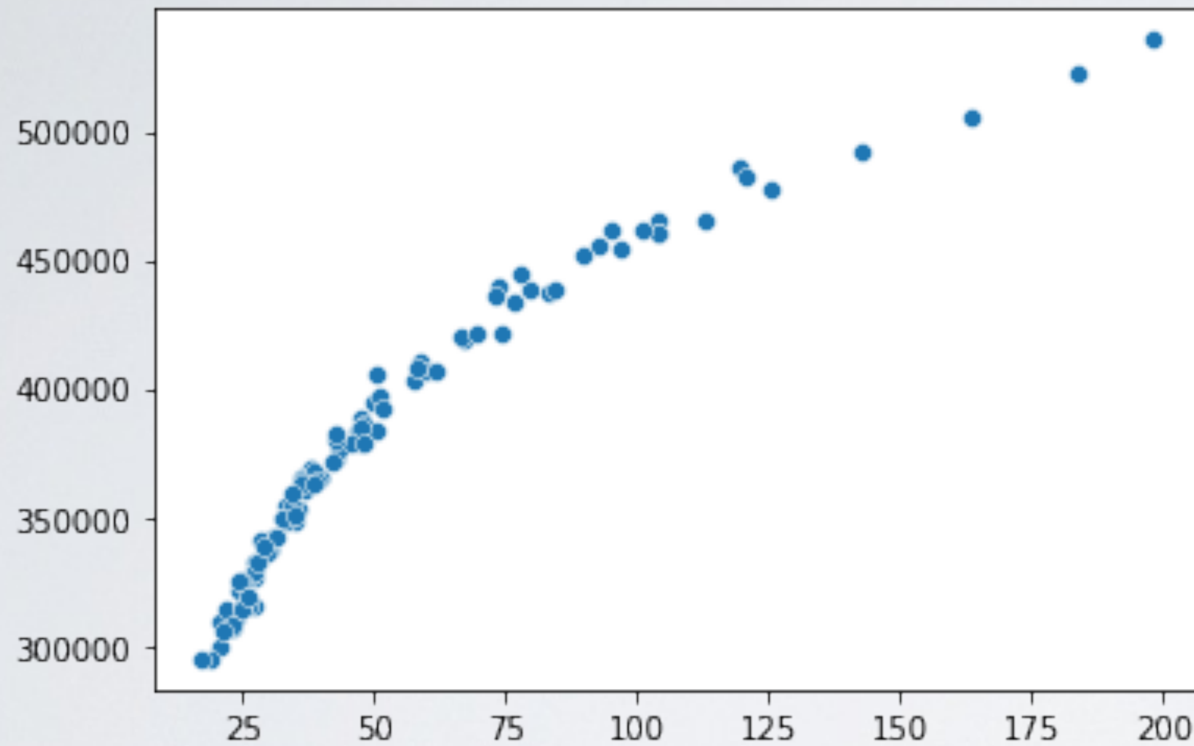
- $\hat{y} = \eta\hat{y}_i^{(t-1)} + f_t(x_i)$

XGBOOST: EXAMPLE

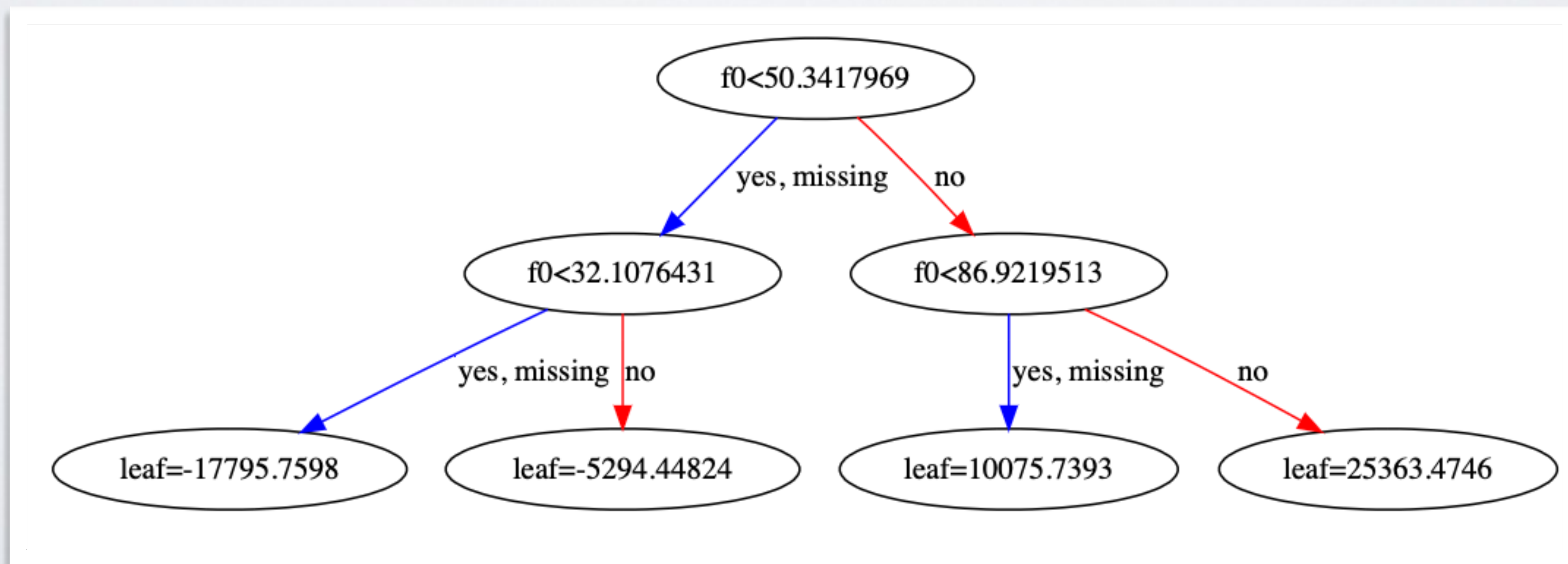


objective="reg:squarederror",
learning_rate=0.3,
base_score=np.mean(Ytrain),
max_depth=2

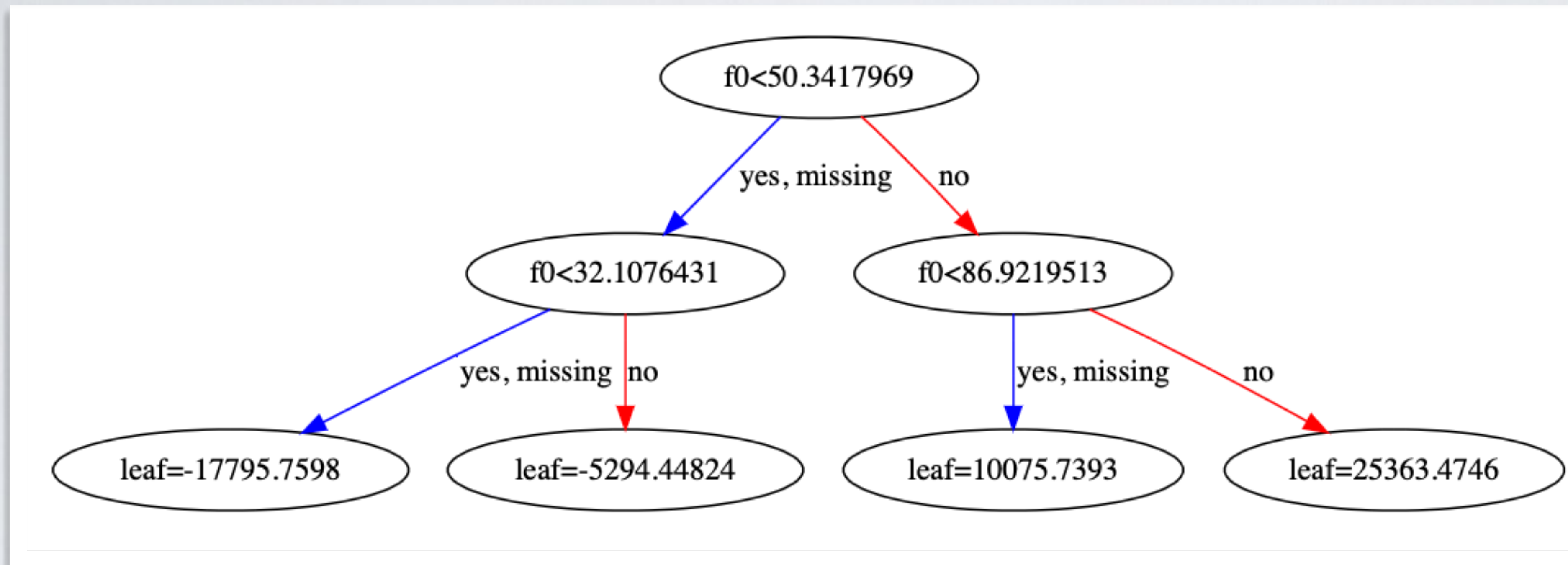
XGBOOST: EXAMPLE

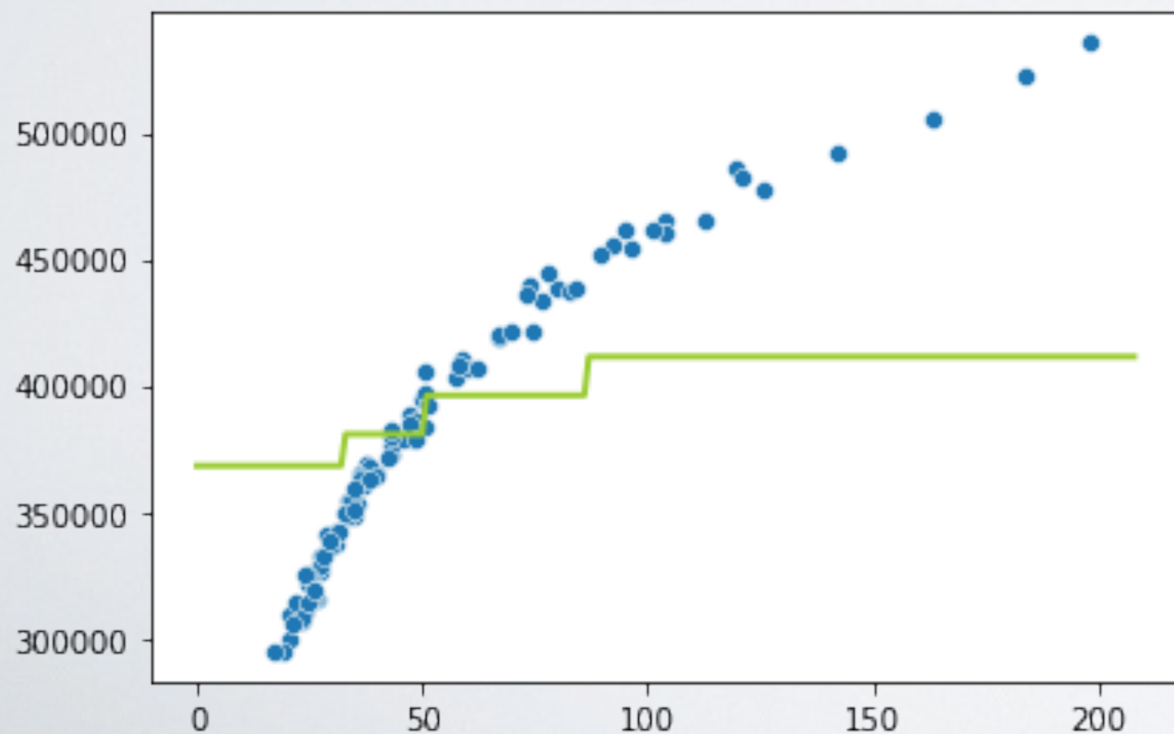


First tree



XGBOOST: EXAMPLE

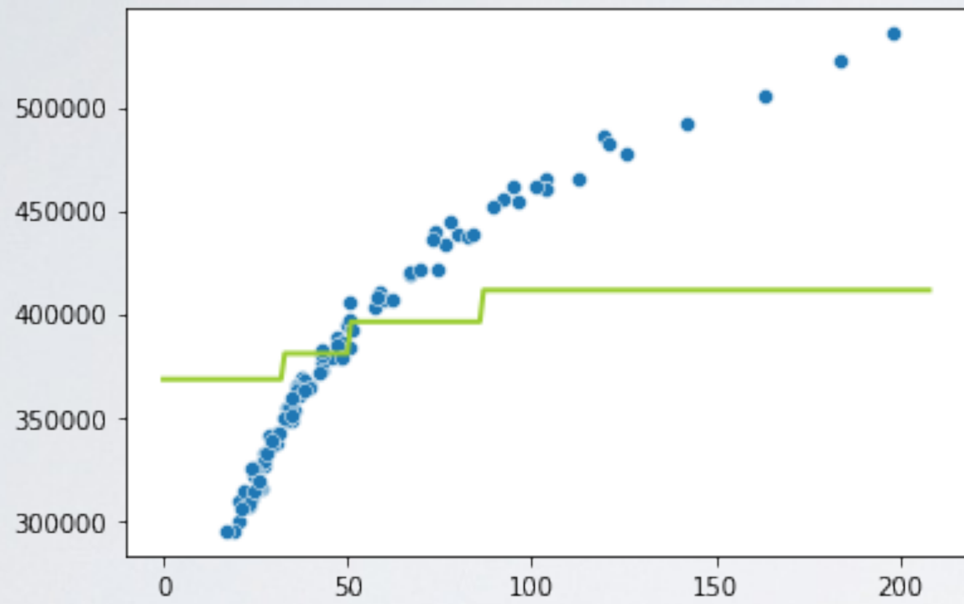




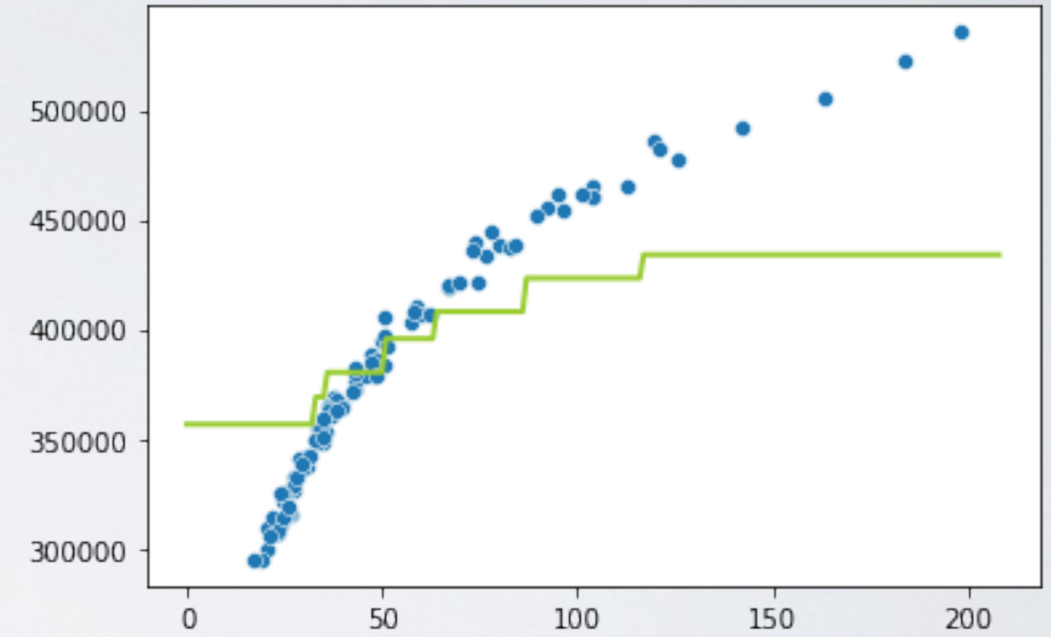
1 single tree for prediction:
Learning rate effect…

XGBOOST: EXAMPLE

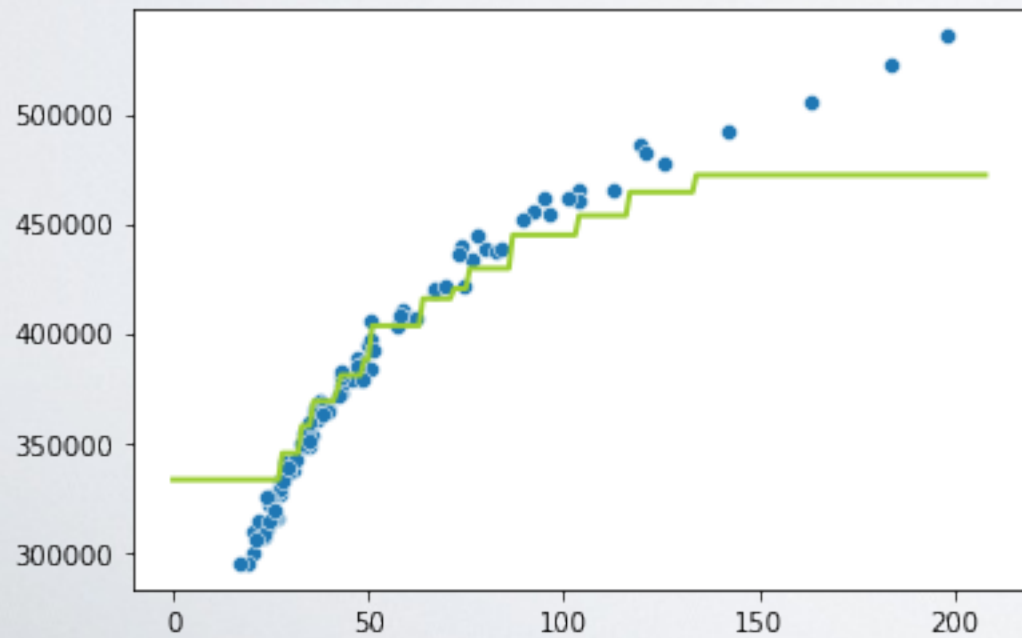
1



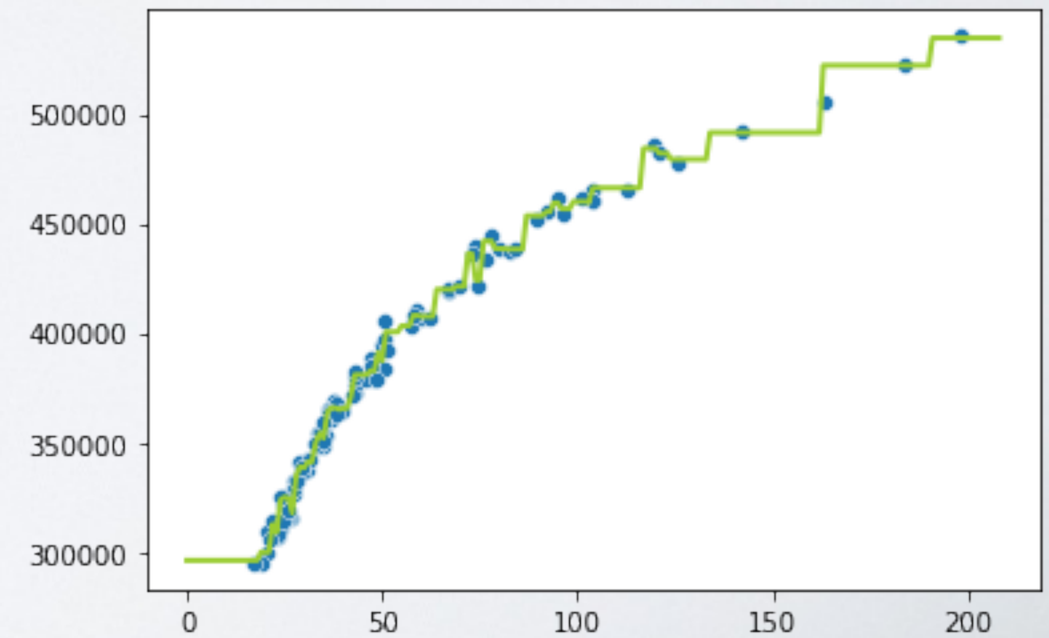
2



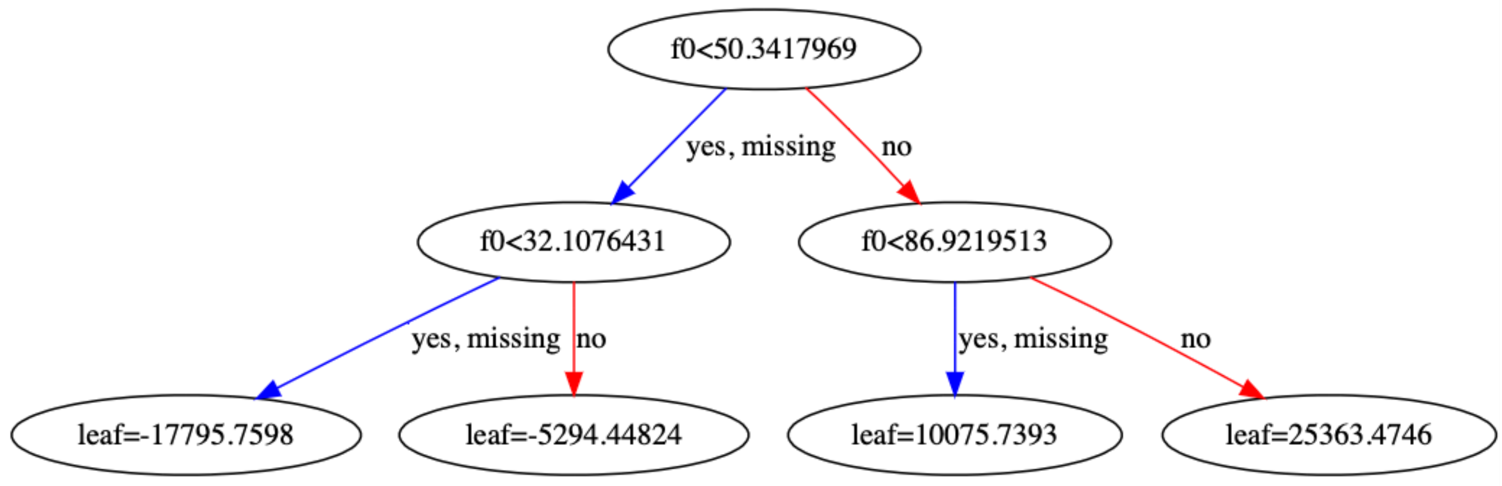
5



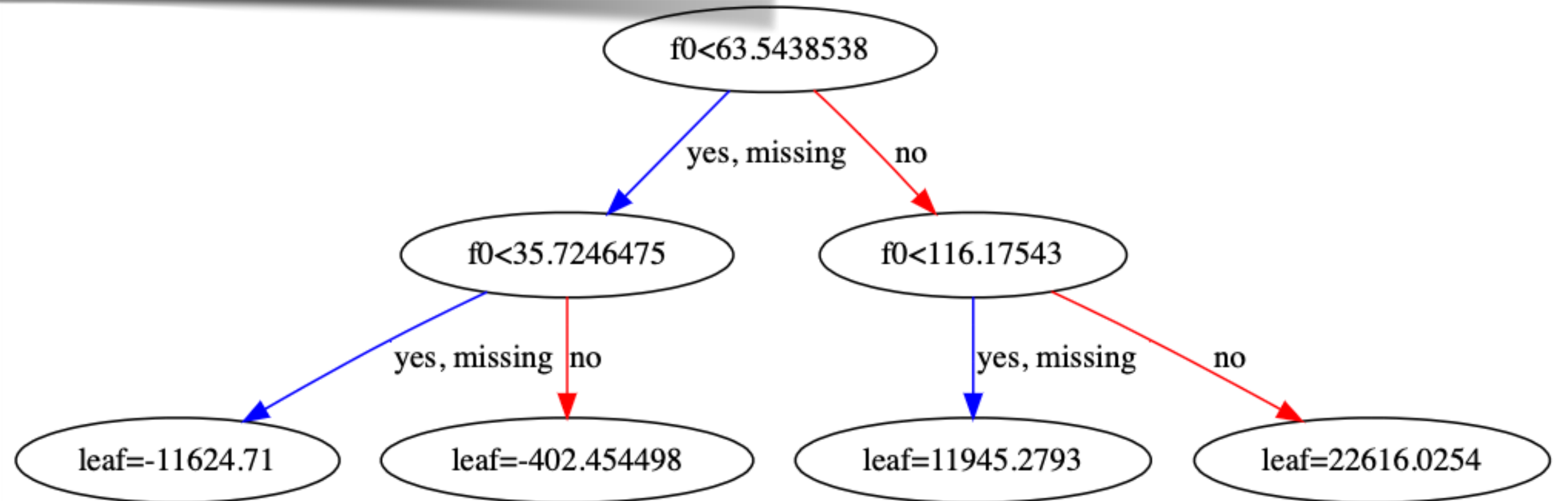
50



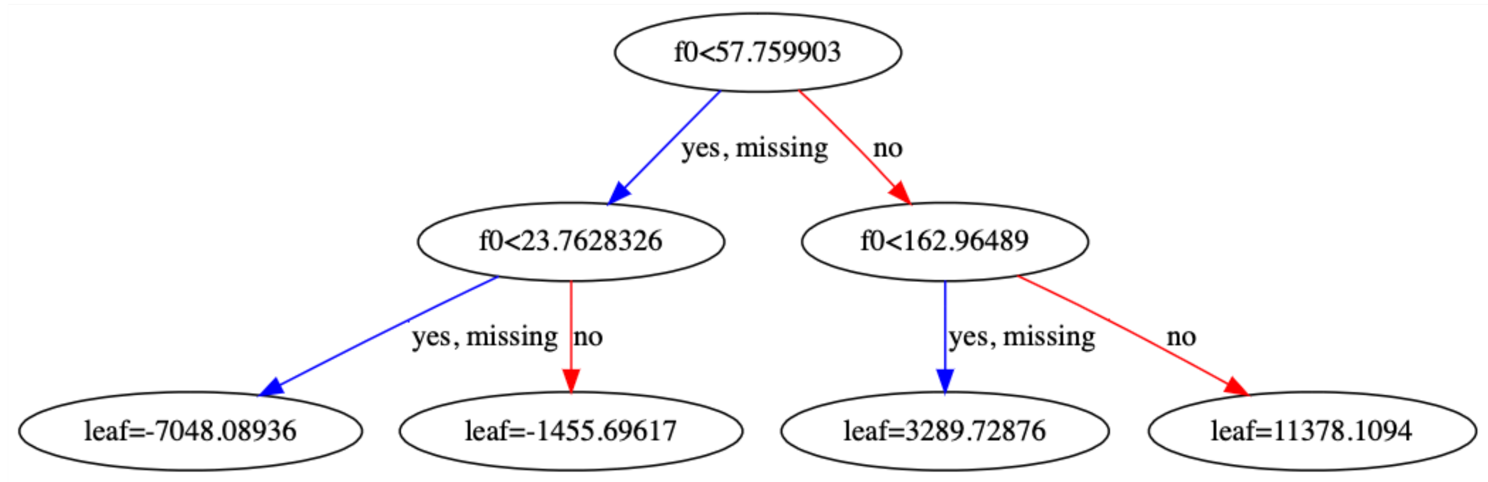
1



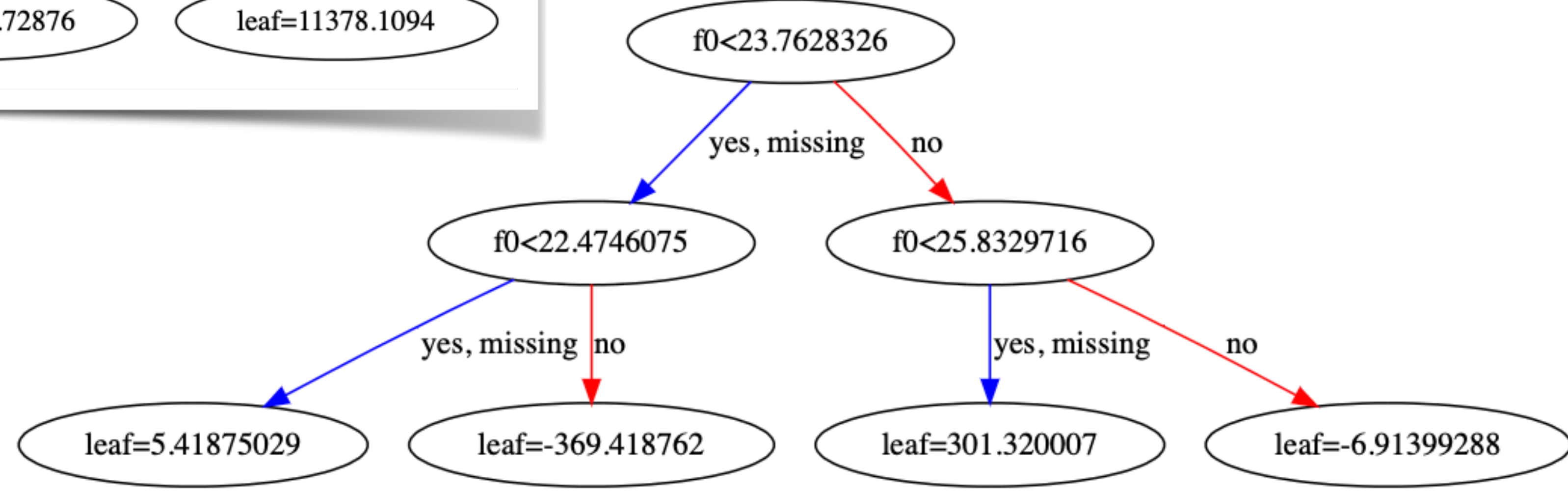
2



5



50



DETAILS ON WHY

https://xgboost.readthedocs.io/en/stable/tutorials/model.html

REGULARIZATION TERM

XGB: REGULARIZATION

$$\text{obj} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \omega(f_i)$$

- $$\omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$
  - T: number of leaves
  - λ: parameter for the strength of the regularization
  - γ: gain threshold below which we choose not to split a leaf
  - w_j "score" of leaf j, next slide

=>Chosen to simplify computations

DEFINITION USING ANY LOSS FUNCTION

GRADIENT BOOSTING

$$\text{obj}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \omega(f_i)$$

$$= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \omega(f_t)$$

- In our loss for the tree, we decompose the prediction $\hat{y}$ as
  - Prediction given by previous tree + prediction of new tree.
  - ω regularization, explained later

GRADIENT BOOSTING

$$\sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \omega(f_t)$$

Instead of classic gradient descent, uses Taylor series to compute an approximation, allowing any error function

$$\text{obj}^{(t)} = \sum_{i=1}^{n} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \omega(f_t) + \text{constant}$$

With g_i, h_i first and second derivatives

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

$$h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

LEAF AND TREE SCORES

After development:

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

The score of a leaf
(What we will sum to make the prediction)

$$\text{obj}^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T$$

The score of a tree
T: leaves

With $G_j = \sum_{i \in I_j} g_i \quad H_j = \sum_{i \in I_j} h_i$

Score of leaf I_j, sum for items inside it

FROM GENERIC TO MSE

SCORES WITH MSE

$$w_j = -\frac{G_j}{H_j + \lambda}$$

Looks complicated…
In practice, meaning for squared loss?

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

$$h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

$$\partial_{\hat{y}_i^{t-1}} (y_i - \hat{y}_i^{t-1})^2 = 2(y_i - \hat{y}_i^{t-1})(-1) = 2(\hat{y}_i^{t-1} - y_i)$$

$$\partial^2_{\hat{y}_i^{t-1}} (y_i - \hat{y}_i^{t-1})^2 = \partial_{\hat{y}_i^{t-1}} 2(\hat{y}_i^{t-1} - y_i) = 2$$

$$w_j = -\frac{G_j}{H_j + \lambda}$$

$\approx$ avg (signed) error

CLASSIC ML VS DNN

- Until now, I have presented "classic" methods.
- In the news, we hear often about Neural networks methods when talking about IA. Are classic obsolete?
  - DNN are mostly "chained" classic methods. Nothing different in the theory
  - DNN are good for problems with
    - Huge quantity of data
    - Huge quantity of attributes
    - Attributes being semantically related to each other (adjacent pixels, following words…)
    - Attributes are of the same nature
    - => Currently, extremely specialized for tasks on images, text, audio, etc.
  - If limited data, set of unrelated, loosely known features: XGboost & Co. are the most used and usually most efficient methods