XGBoost for Regression

Gradient Boost

Regularization

A Unique Regression Tree

Approximate Greedy Algorithm

Weighted Quantile Sketch

Sparsity-Aware Split Finding

Parallel Learning

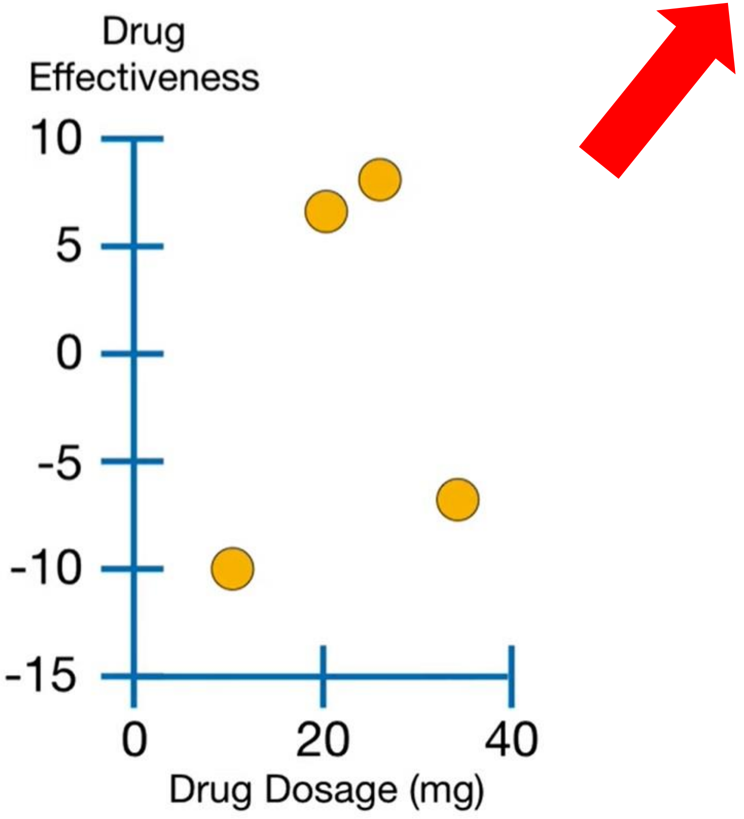
Cache-Aware Access

Blocks for Out-of-Core Computation

XGBoost is a big **Machine Learning** algorithm with lots of parts.

A Unique Regression Tree

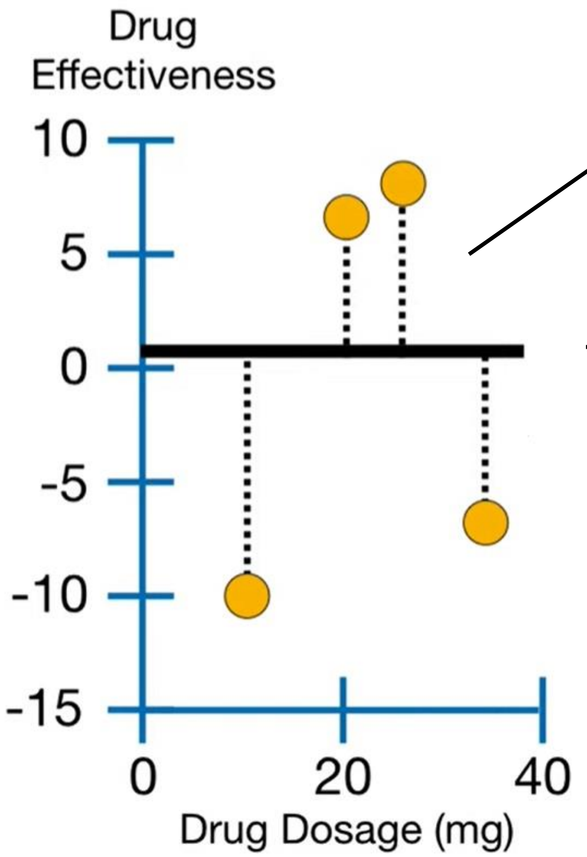
Training Data



A Unique Regression Tree

Initial Predicted Drug Effectiveness

0.5



Residual

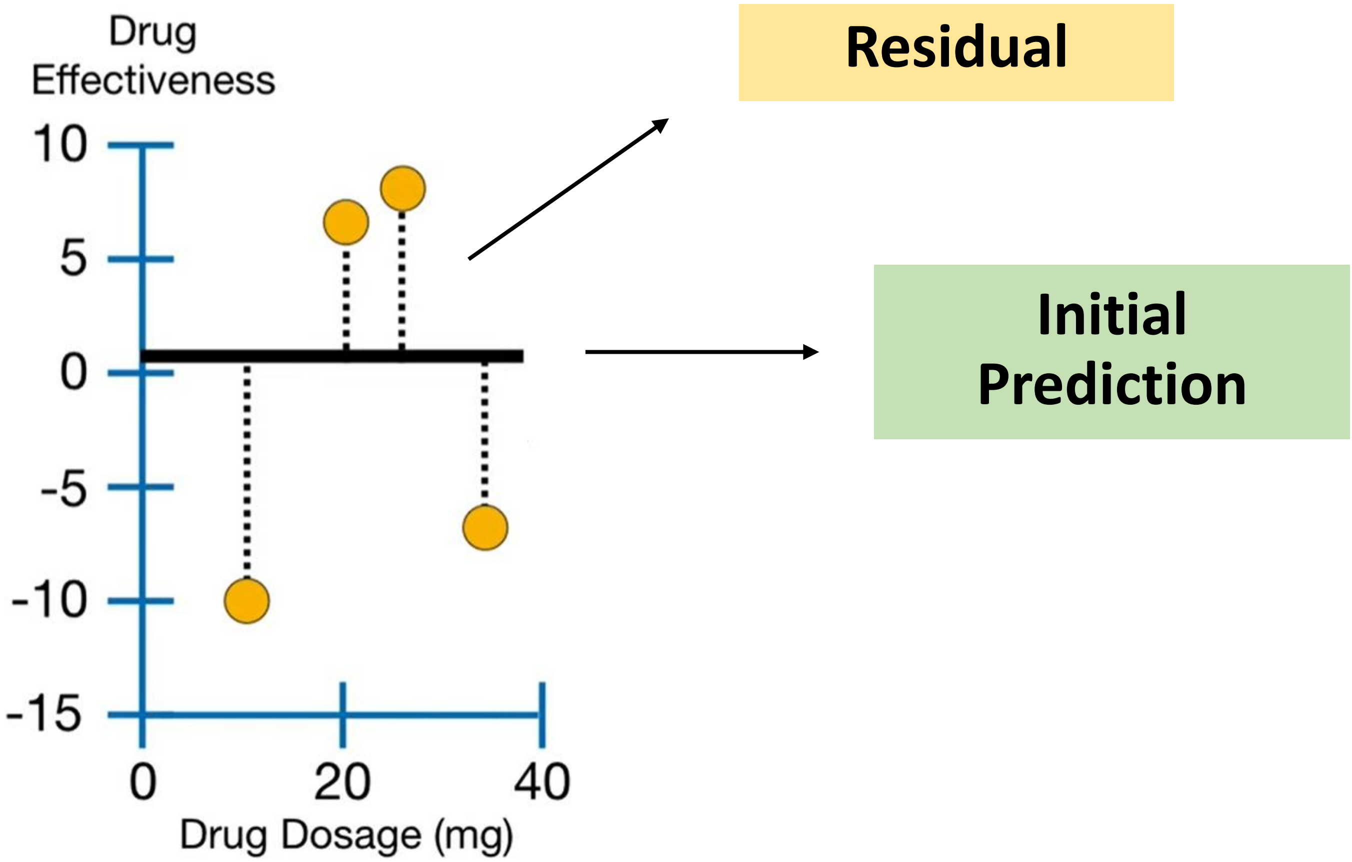
Initial Prediction

A Unique Regression Tree

Initial Predicted Drug Effectiveness

0.5

XGBoost fits a **Regression Tree** to the **residuals**.



Residual

Initial Prediction

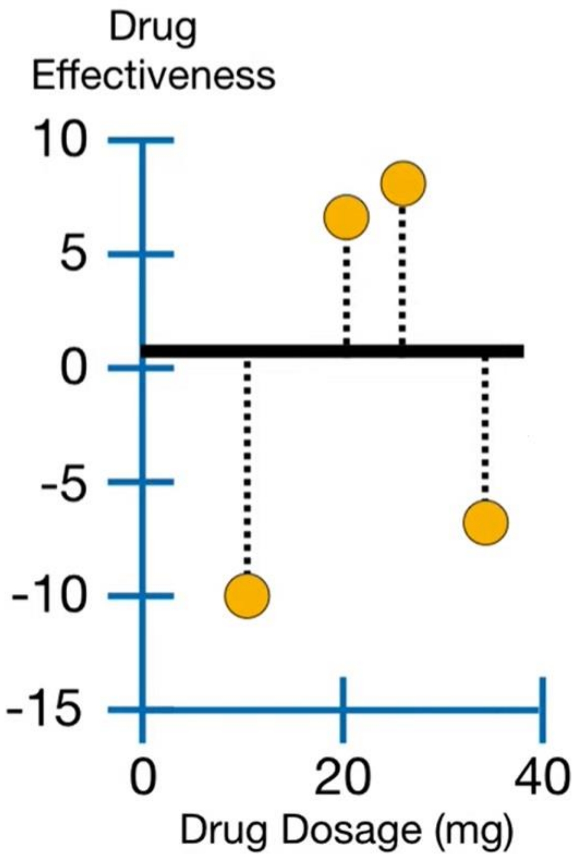
NOTE: There are many ways to build **XGBoost Trees**. This presentation focuses on the most common way to build them for **Regression**.

A Unique Regression Tree

Initial Predicted Drug Effectiveness

0.5

-10.5, 6.5, 7.5, -7.5



$$Similarity\ Score = \frac{(Sum\ of\ Residuals)^2}{Number\ of\ Residuals + \lambda}$$

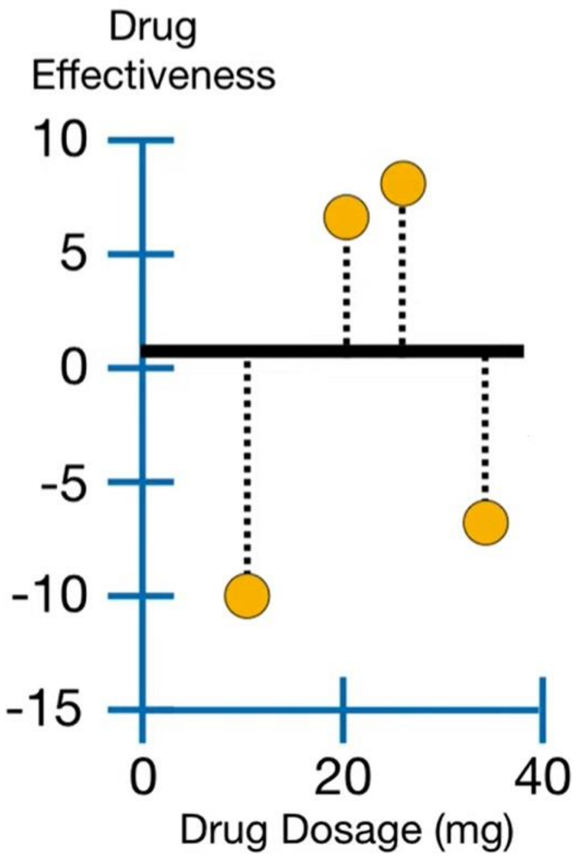
λ = Regularization Parameter

A Unique Regression Tree

Initial Predicted Drug Effectiveness

0.5

-10.5, 6.5, 7.5, -7.5



$$Similarity\ Score = \frac{(Sum\ of\ Residuals)^2}{Number\ of\ Residuals +\ \lambda}$$

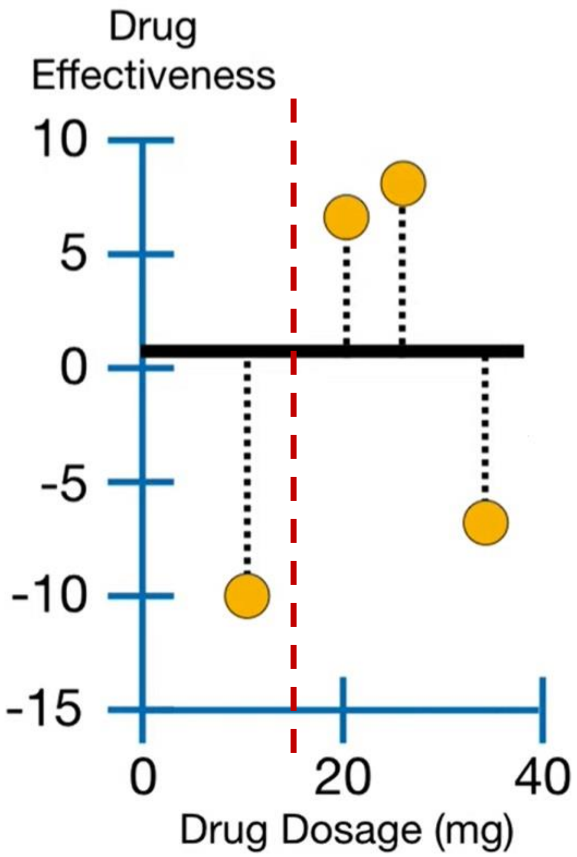
Let $\lambda = 0$

$$Similarity\ Score = \frac{(-10.5 + 6.5 + 7.5 + (-7.5))^2}{4 + 0} = 4$$

A Unique Regression Tree

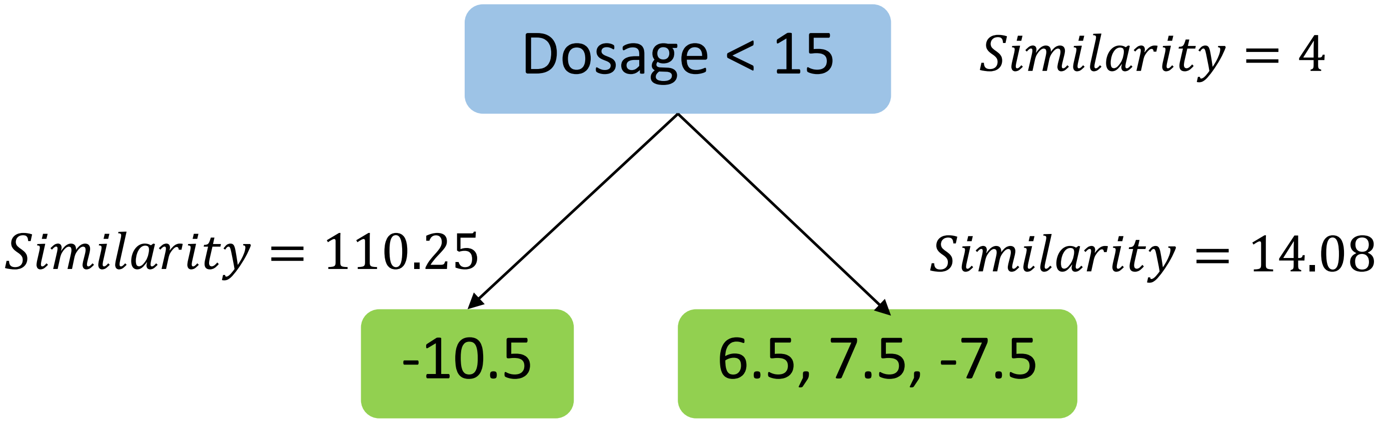
Initial Predicted Drug Effectiveness

0.5



-10.5, 6.5, 7.5, -7.5

$Similarity = 4$



Dosage < 15

$Similarity = 4$

$Similarity = 110.25$

$Similarity = 14.08$

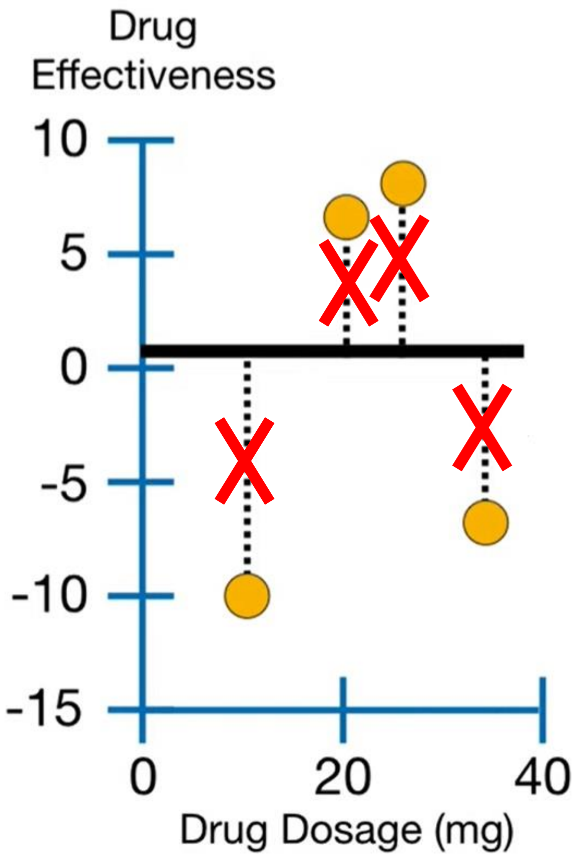
-10.5

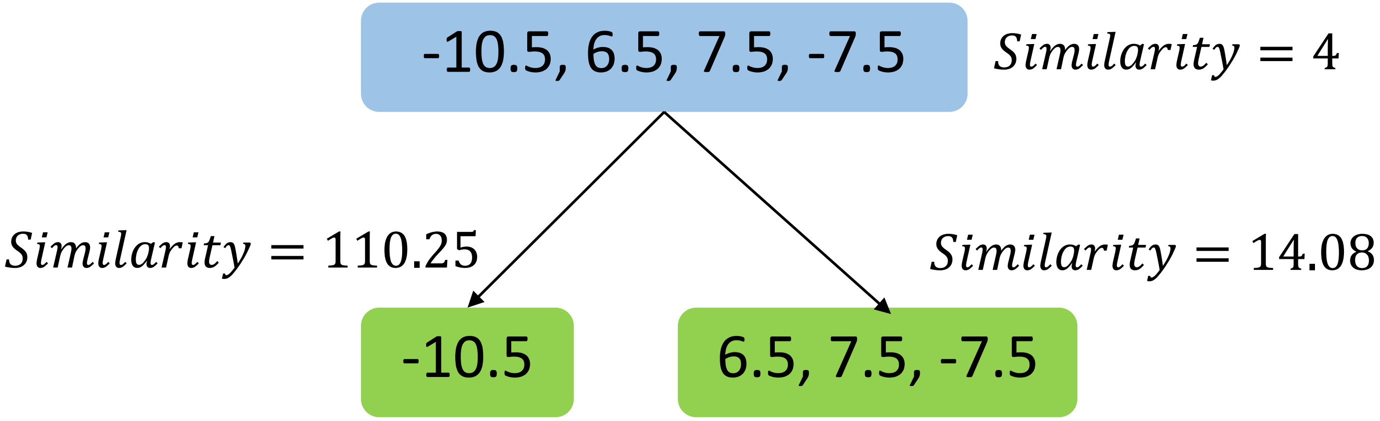
6.5, 7.5, -7.5

A Unique Regression Tree

Initial Predicted Drug Effectiveness

0.5





-10.5, 6.5, 7.5, -7.5

$Similarity = 4$

$Similarity = 110.25$

$Similarity = 14.08$

-10.5

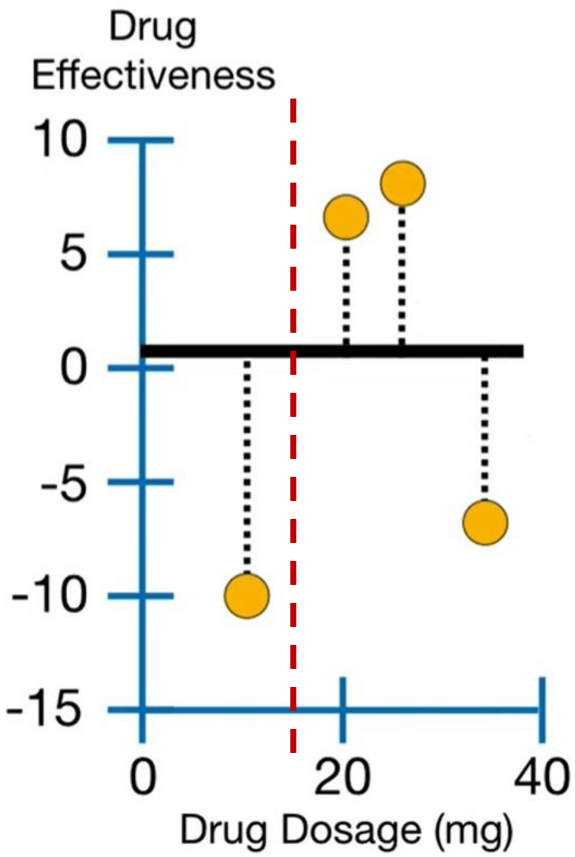
6.5, 7.5, -7.5

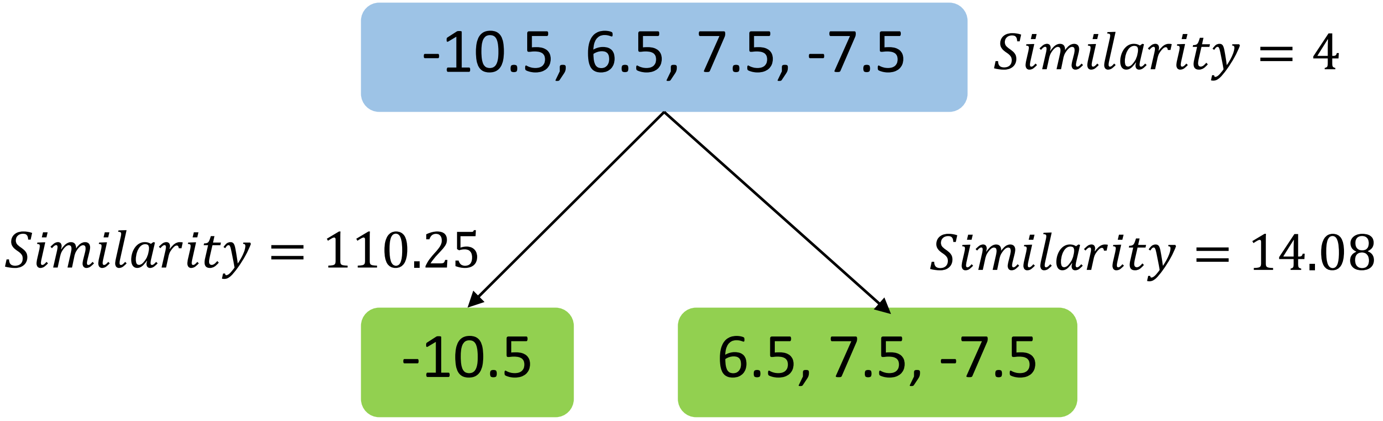
When the **Residuals** in a node are very different, they cancel each other out and the **Similarity Score** is relatively small.

A Unique Regression Tree

Initial Predicted Drug Effectiveness

0.5





Need to quantify how much better the leaves cluster similar **Residuals** than the root.

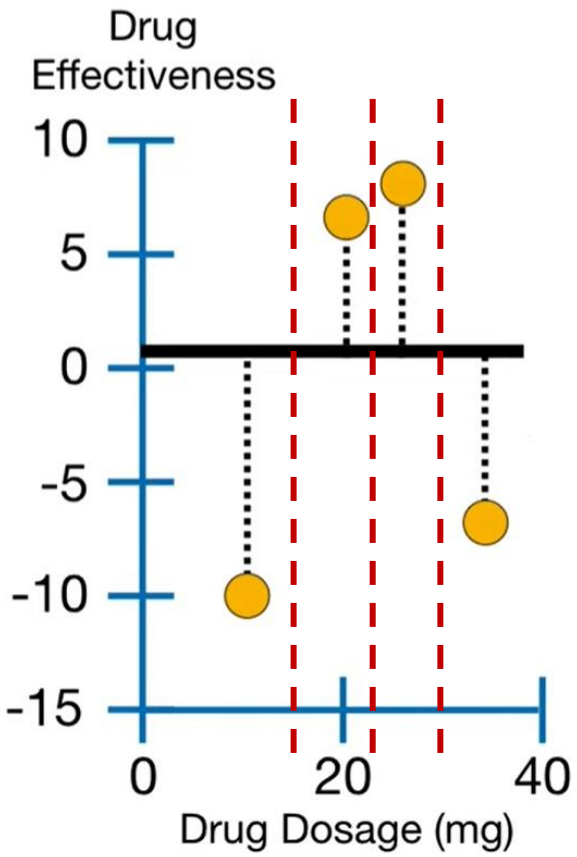
$$Gain = Left_{Similarity} + Right_{Similarity} - Root_{Similarity}$$

$$Gain = 110.25 + 14.08 - 4 = 120.33$$

A Unique Regression Tree

Initial Predicted Drug Effectiveness

0.5

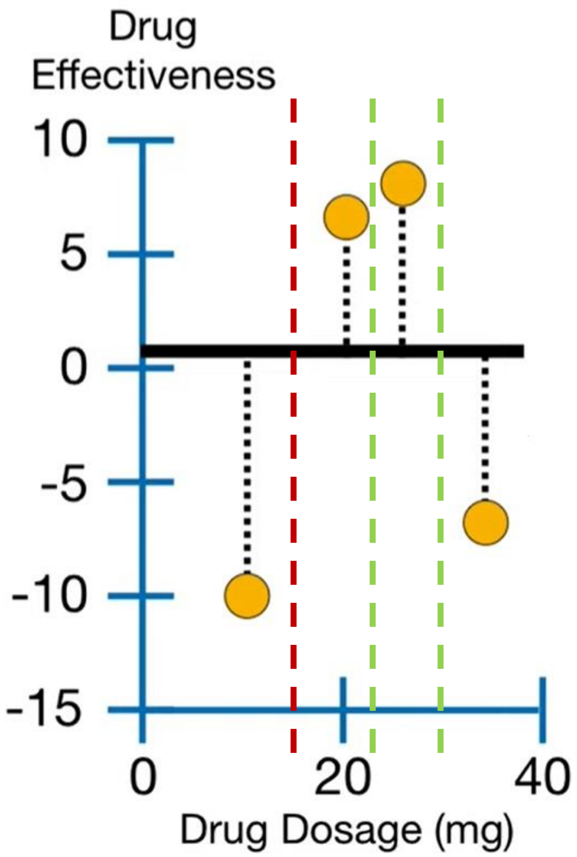


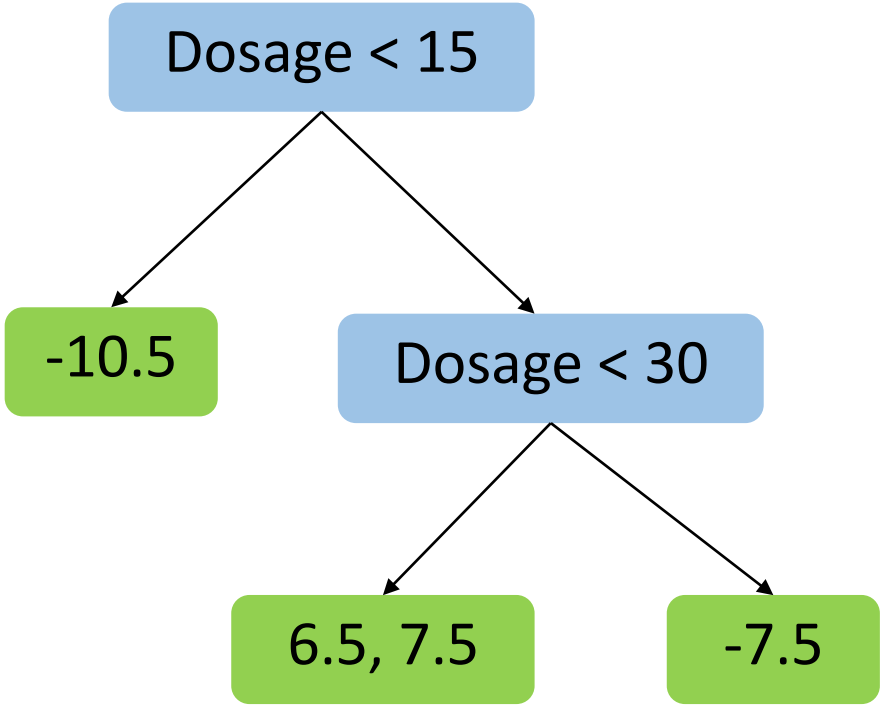
Threshold	Gain
15	**120.33**
22.5	4
30	56.33

A Unique Regression Tree

Initial Predicted Drug Effectiveness

0.5



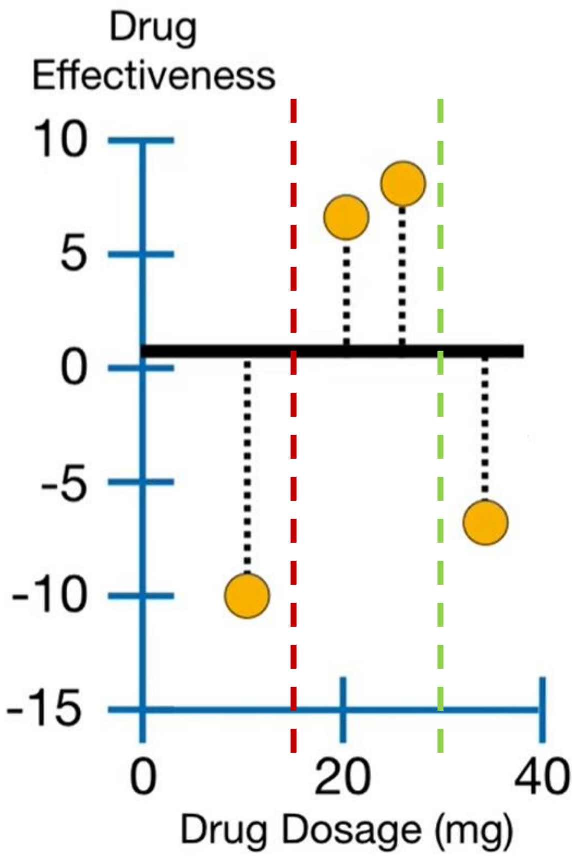


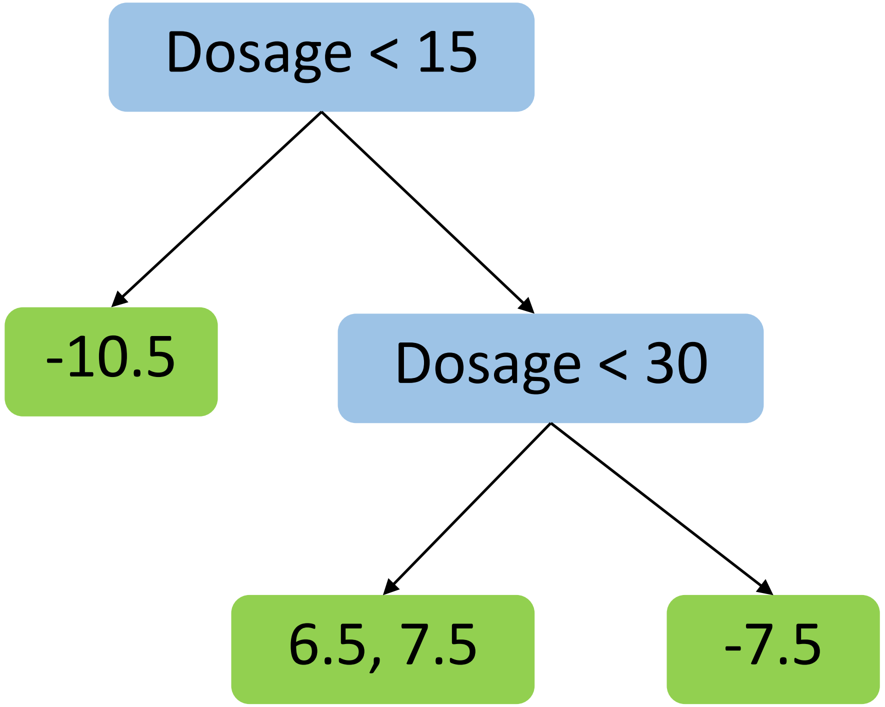
Threshold	Gain
22.5	28.17
30	140.17

A Unique Regression Tree

Initial Predicted Drug Effectiveness

0.5





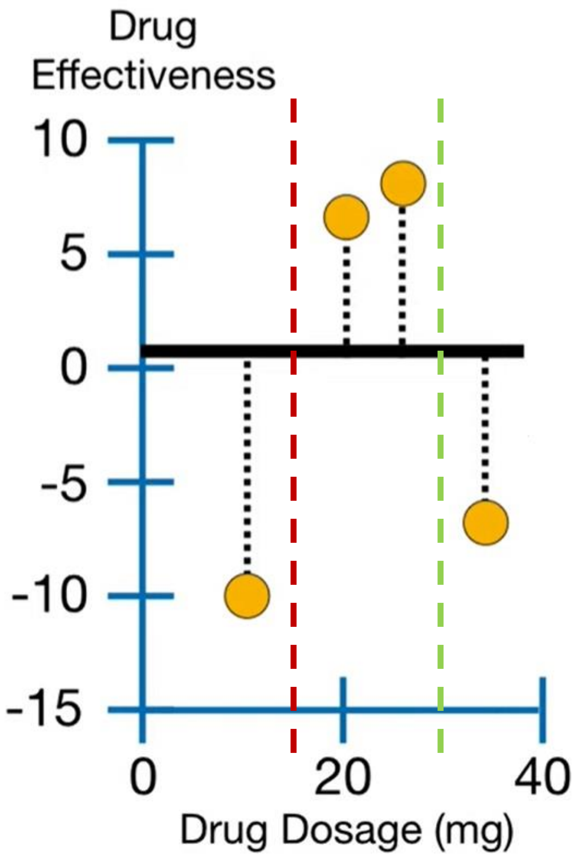
NOTE: To keep this example from getting out of hand, I've limited the **tree depth** to **two levels**…

However, the default is to allow up to **6 levels**.

A Unique Regression Tree

Initial Predicted Drug Effectiveness

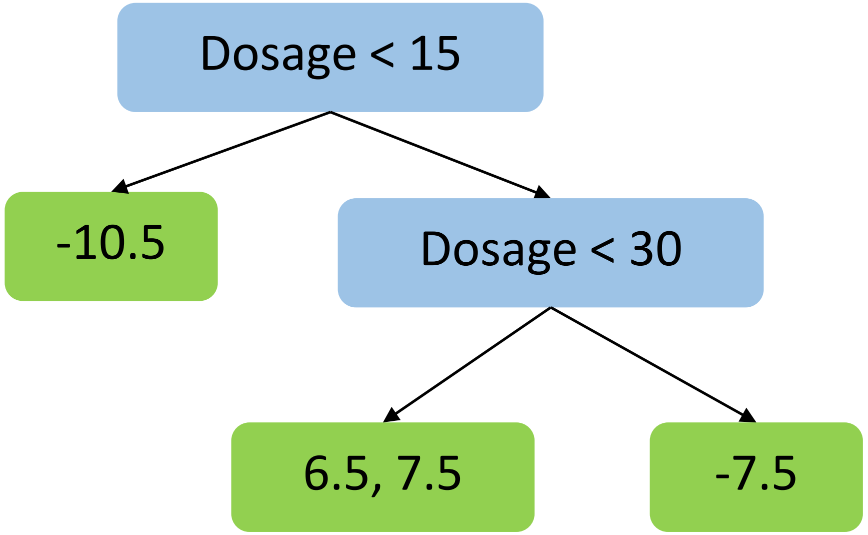
0.5



Pruning an XGBoost Tree

γ = Tree Complexity Parameter

γ = 130



Dosage < 15

-10.5

Dosage < 30

6.5, 7.5

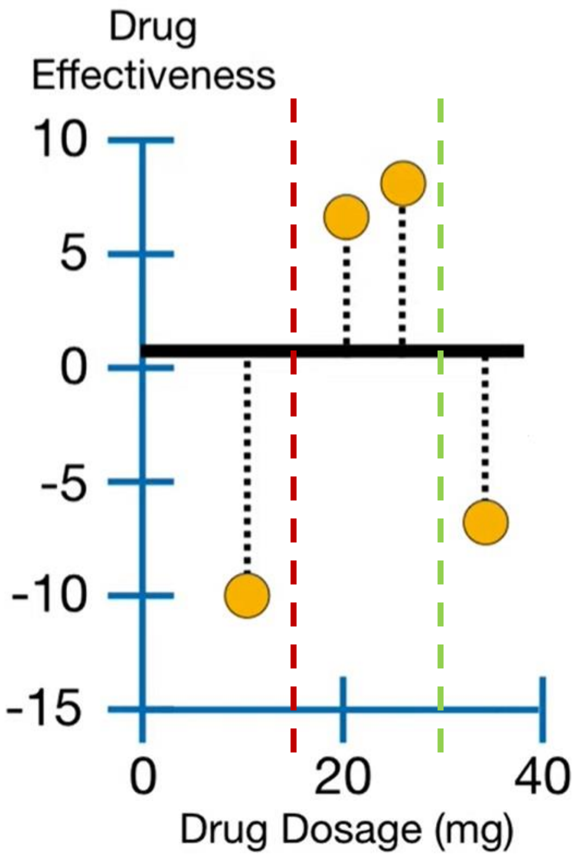
-7.5

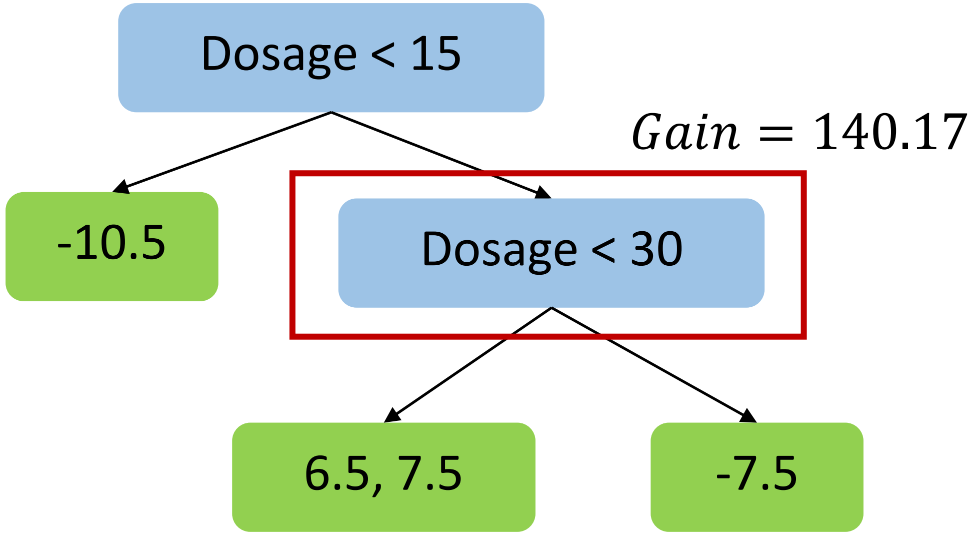
A Unique Regression Tree

Pruning an XGBoost Tree

Initial Predicted Drug Effectiveness

0.5





$Gain = 140.17$

Calculate $Gain - \gamma$

If $Gain - \gamma$ is *negative*, we will *remove* the branch…

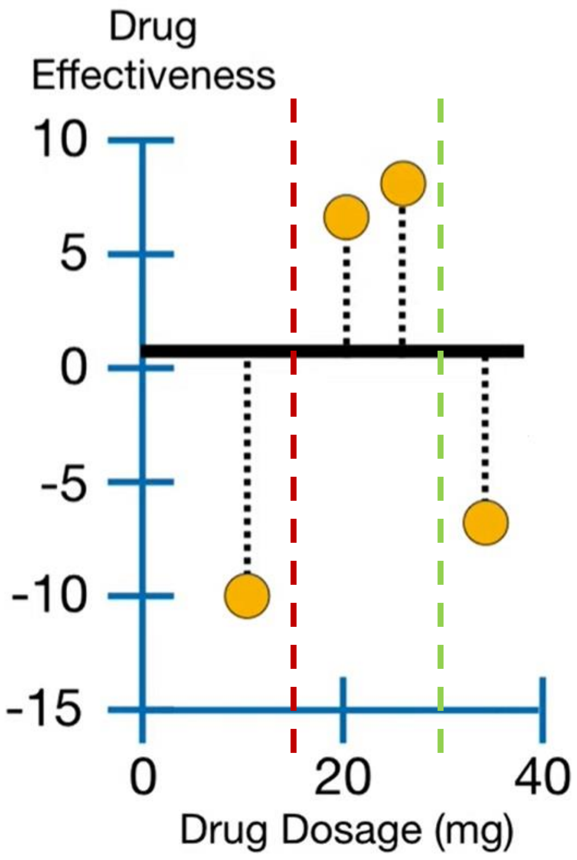
If $Gain - \gamma$ is *positive*, we will *not remove* the branch…

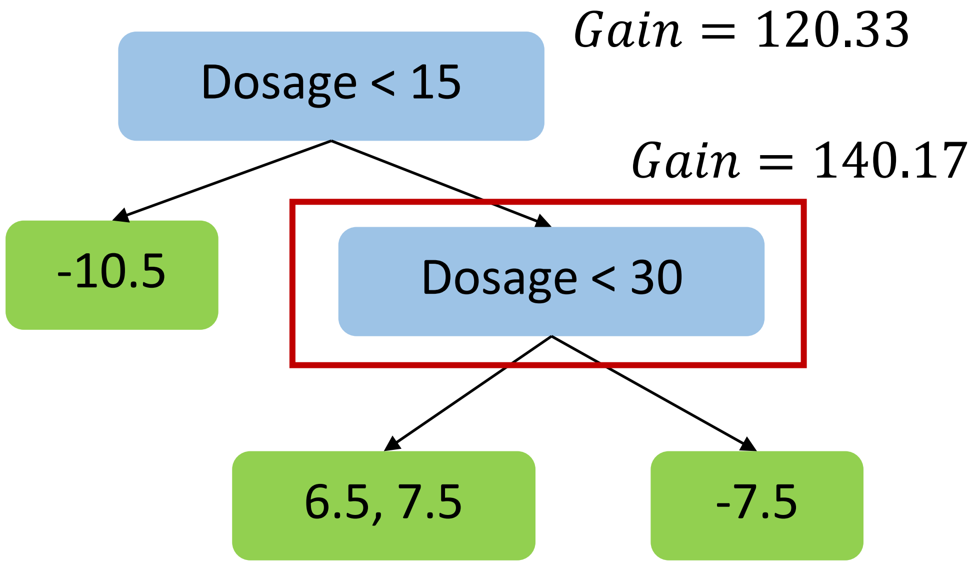
A Unique Regression Tree

Pruning an XGBoost Tree

Initial Predicted Drug Effectiveness

0.5





$$Gain - \gamma = 140.17 - 130 = 10.17$$

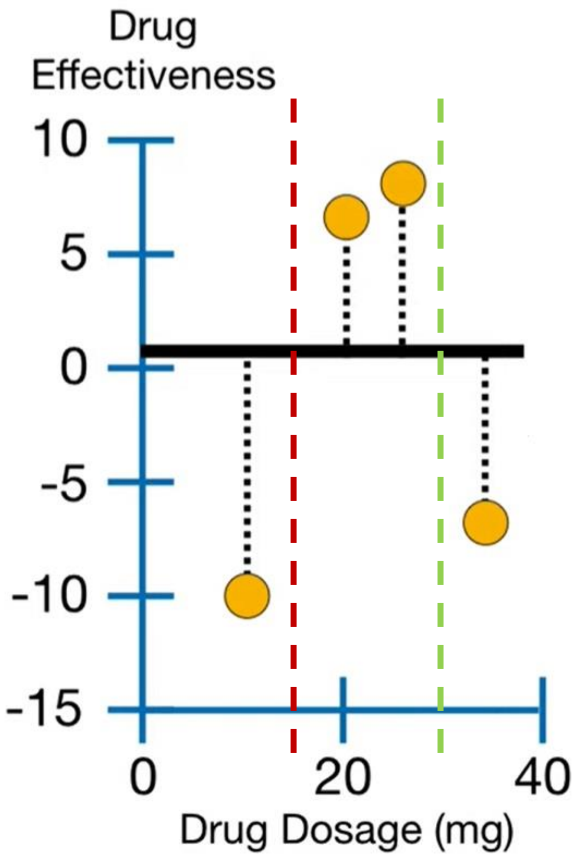
NOTE: The Gain for the root, 120.33, so the difference will be *negative*. However, because we did not remove the first branch, we will not remove the root.

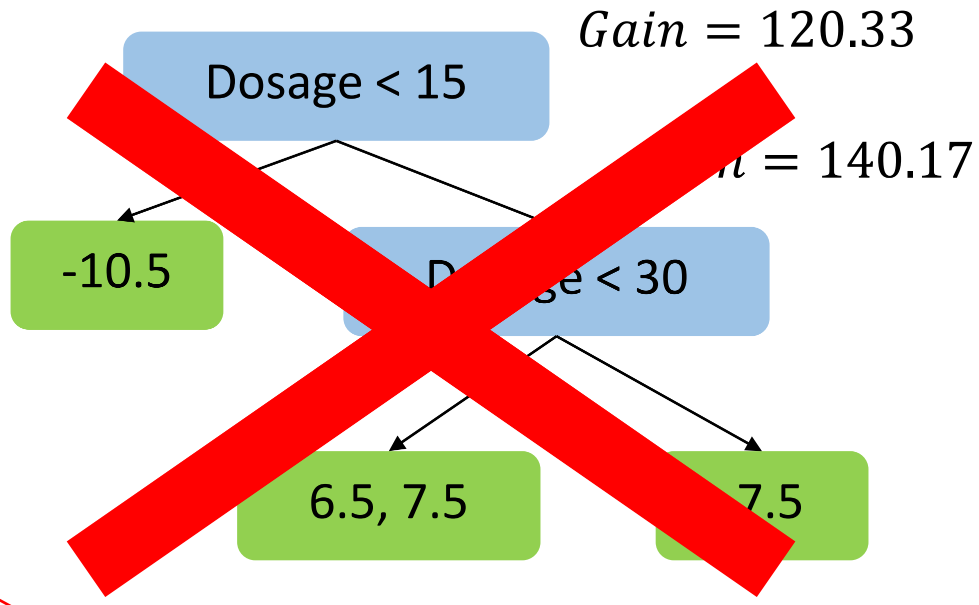
A Unique Regression Tree

Pruning an XGBoost Tree

Initial Predicted Drug Effectiveness

0.5





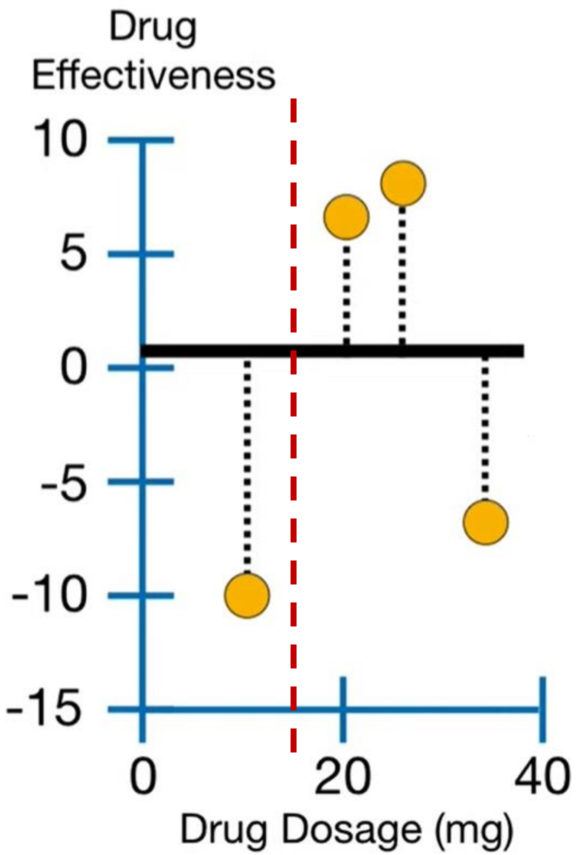
In contrast, if we set γ = 150, then we would remove the branch and the root. And all we would be left with is the original prediction, which is pretty extreme pruning.

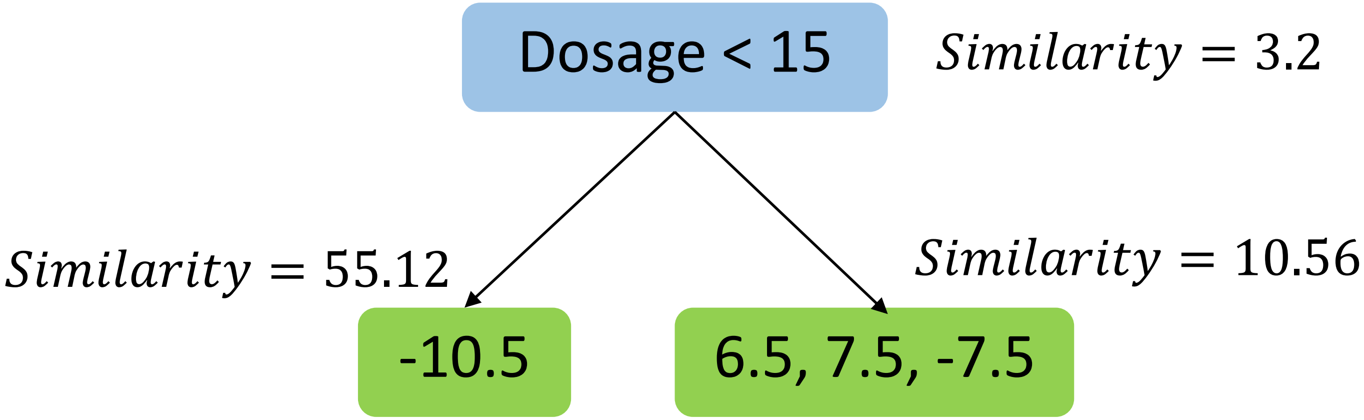
A Unique Regression Tree

Building an XGBoost Tree with Regularization

Initial Predicted Drug Effectiveness

0.5





Set $\lambda = 1$

$$Similarity\ Score = \frac{(Sum\ of\ Residuals)^2}{Number\ of\ Residuals + 1}$$

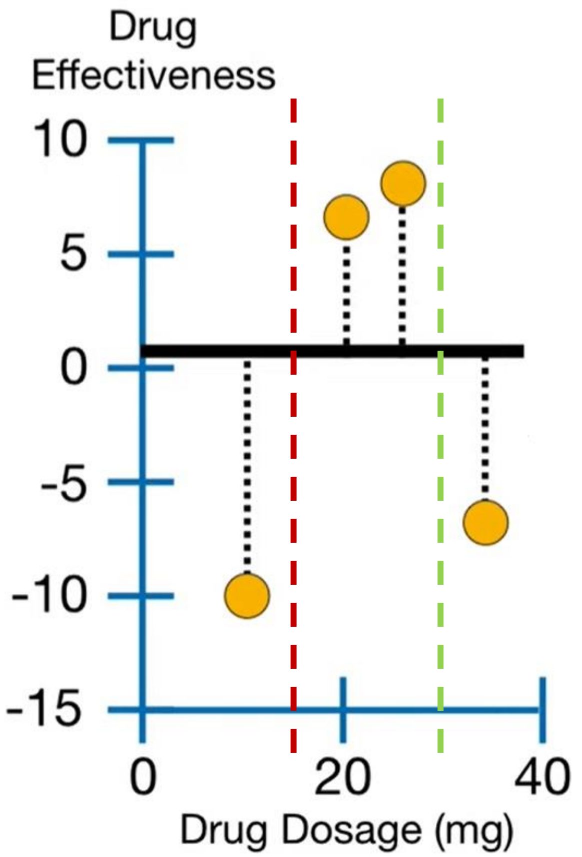
When $\lambda > 0$, the Similarity Scores are smaller.

A Unique Regression Tree

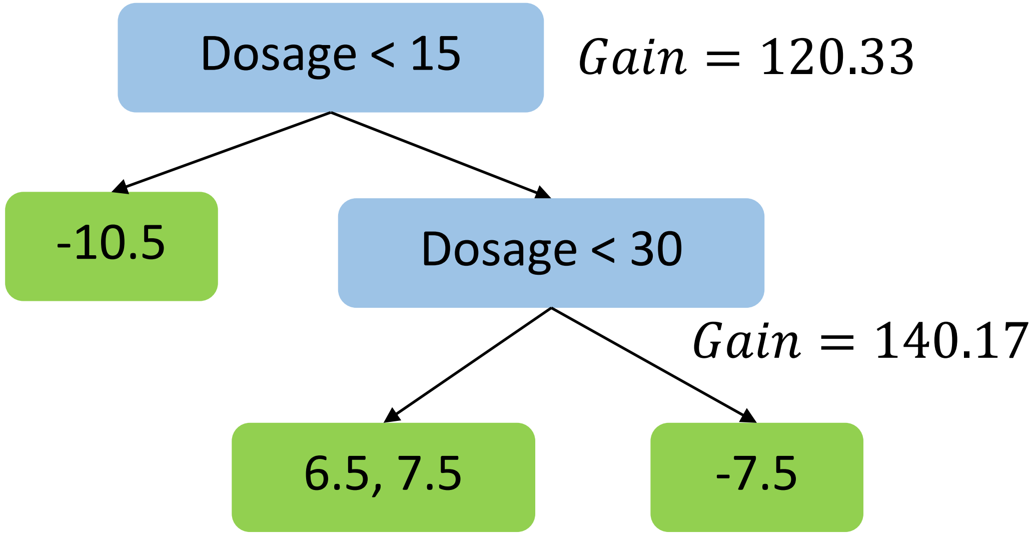
Building an XGBoost Tree with Regularization

Initial Predicted Drug Effectiveness

0.5



Set $\lambda = 0$



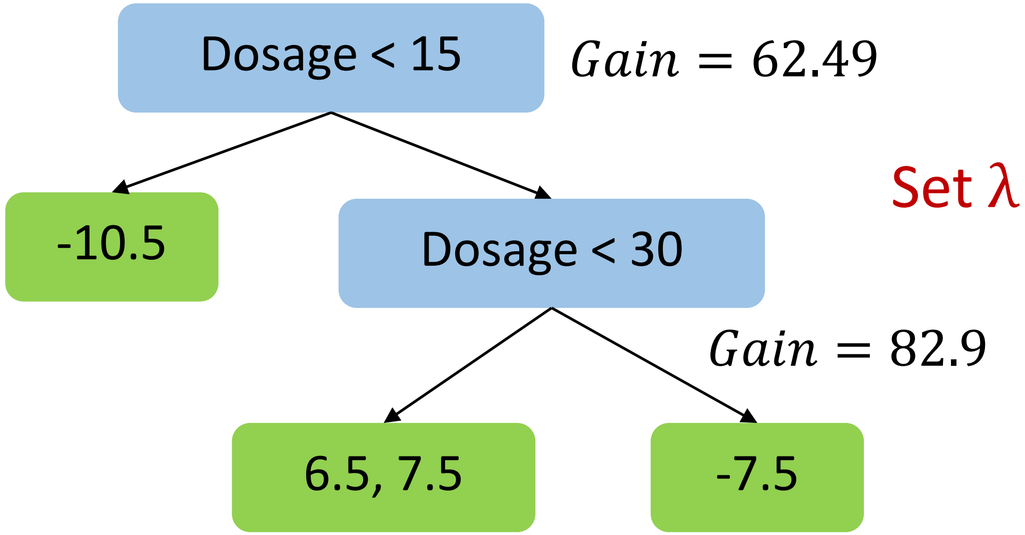
Dosage < 15 $Gain = 120.33$

-10.5

Dosage < 30 $Gain = 140.17$

6.5, 7.5

-7.5



Dosage < 15 $Gain = 62.49$

-10.5

Dosage < 30 $Gain = 82.9$

6.5, 7.5

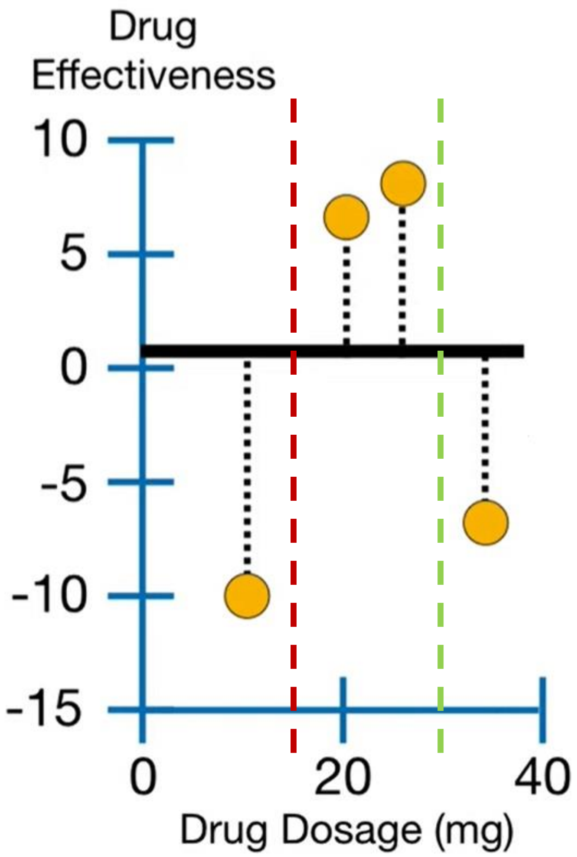
-7.5

Set $\lambda = 1$

A Unique Regression Tree

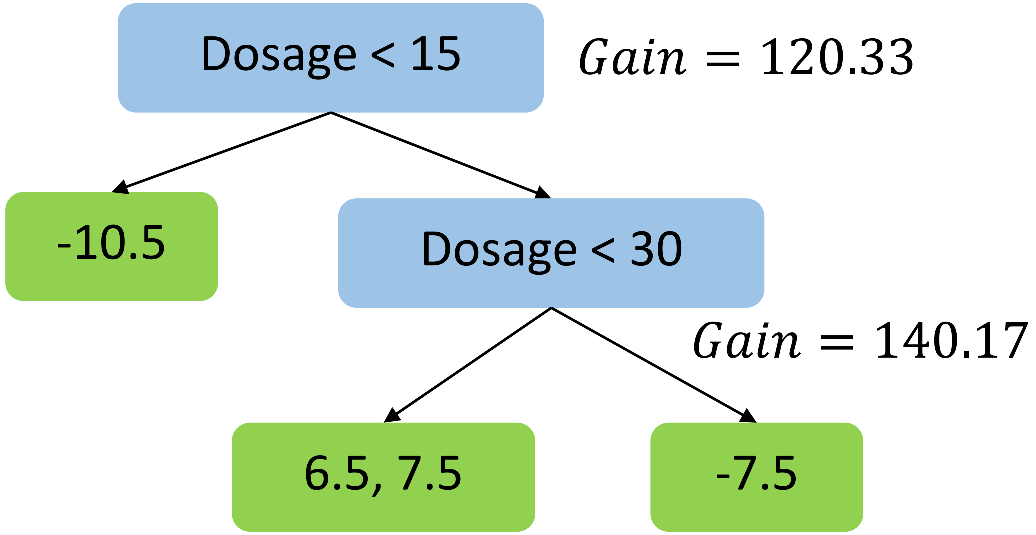
Initial Predicted Drug Effectiveness

0.5

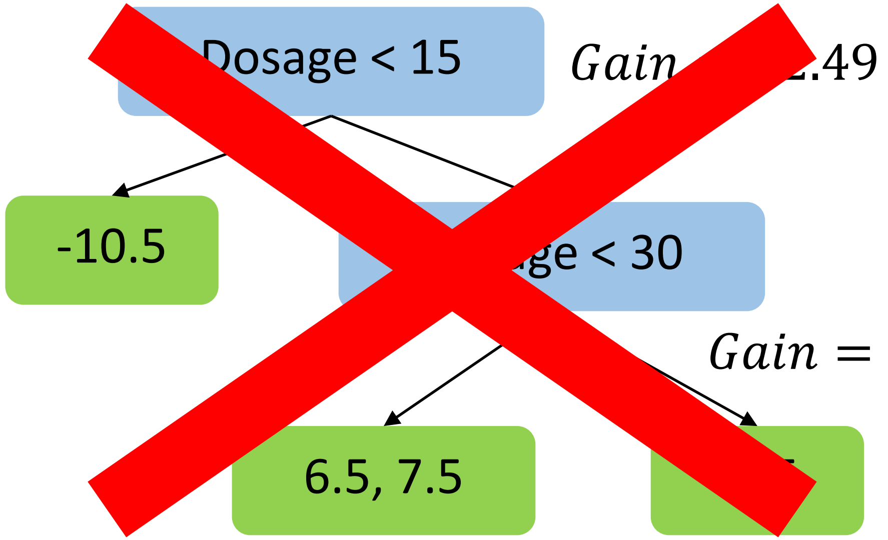


Building an XGBoost Tree with Regularization

Set $\lambda = 0$



$\gamma = 130$

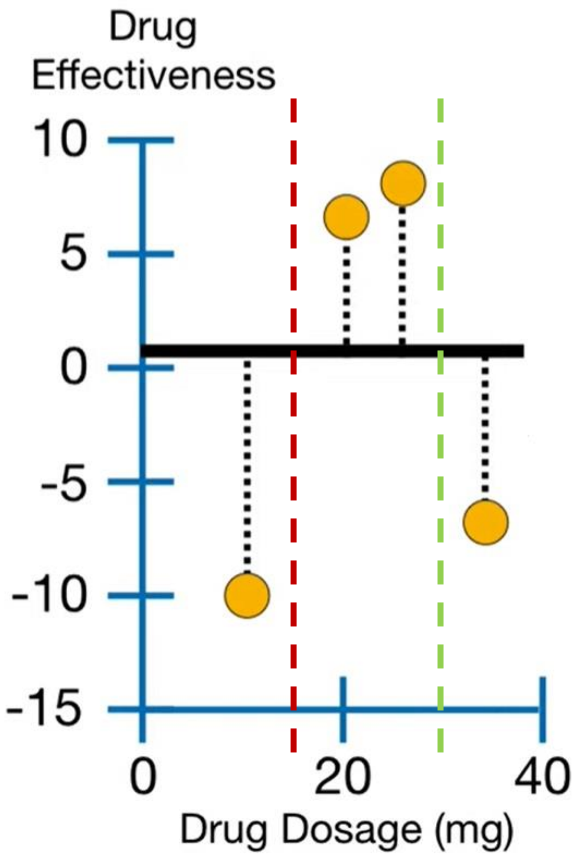


Set $\lambda = 1$

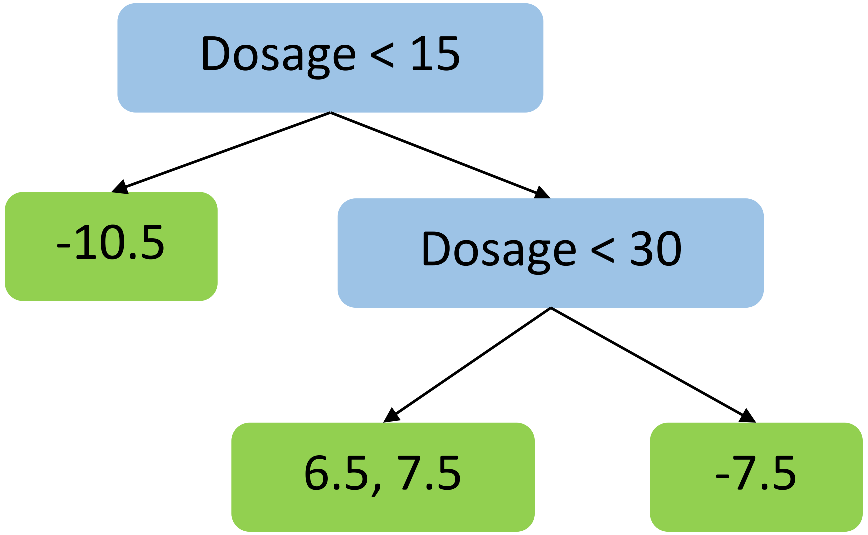
A Unique Regression Tree

Initial Predicted Drug Effectiveness

0.5



Calculating output values for an XGBoost Tree


Dosage < 15

-10.5

Dosage < 30

6.5, 7.5

-7.5

$$Output\ Value = \frac{Sum\ of\ Residuals}{Number\ of\ Residuals + \lambda}$$

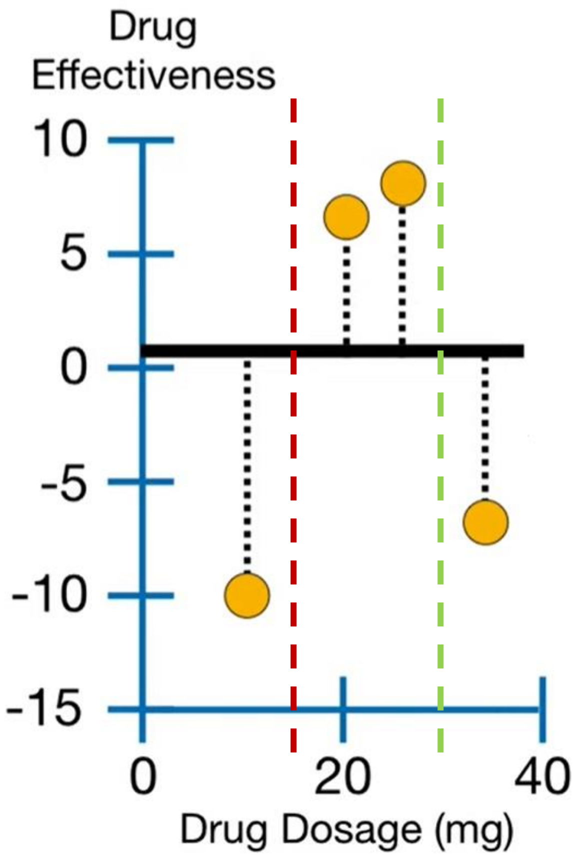
$$Output\ Value = \frac{-10.5}{1+1} = -5.25$$

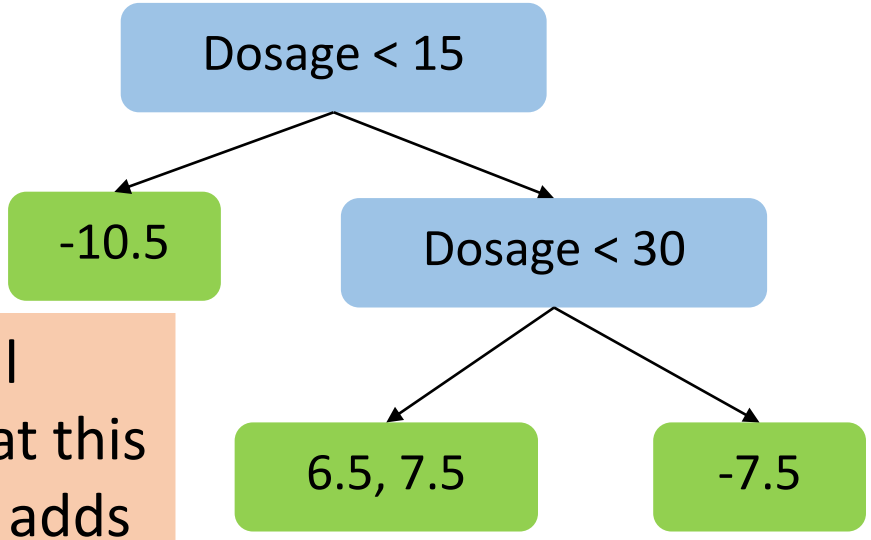
A Unique Regression Tree

Calculating output values for an XGBoost Tree

Initial Predicted Drug Effectiveness

0.5





When λ > 0, then it will reduce the amount that this individual observation adds to the overall prediction.

$$Output\ Value = \frac{Sum\ of\ Residuals}{Number\ of\ Residuals + \lambda}$$

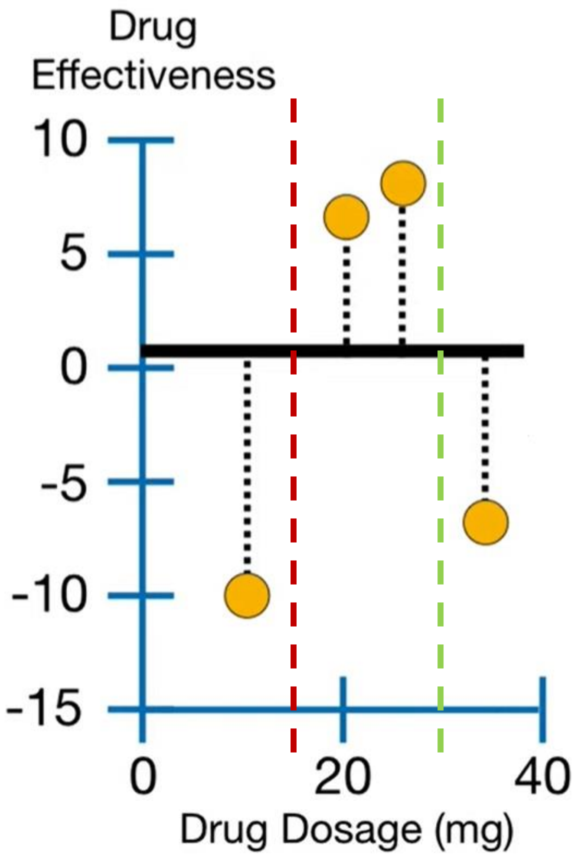
$$Output\ Value = \frac{-10.5}{1+1} = -5.25$$

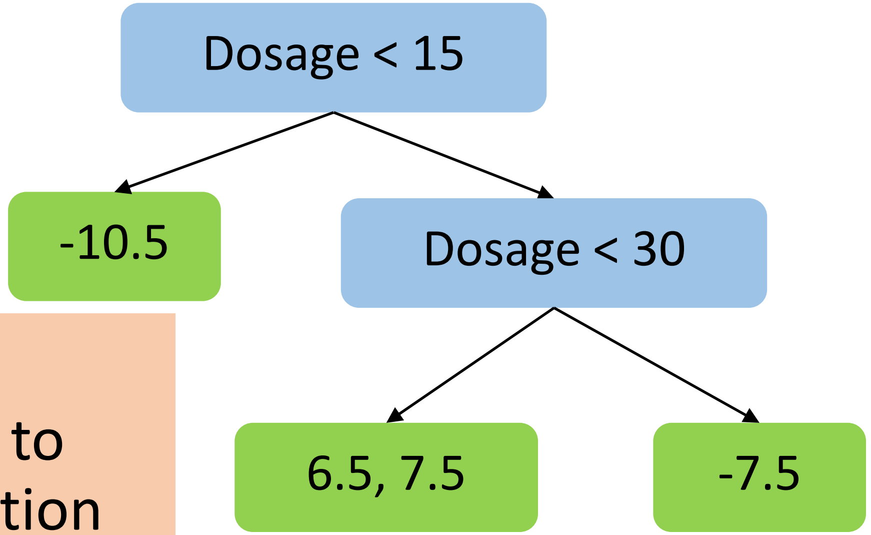
A Unique Regression Tree

Calculating output values for an XGBoost Tree

Initial Predicted Drug Effectiveness

0.5





Dosage < 15

-10.5

Dosage < 30

6.5, 7.5

-7.5

Thus, λ will reduce the prediction's sensitivity to this individual observation

$$Output\ Value = \frac{Sum\ of\ Residuals}{Number\ of\ Residuals + \lambda}$$

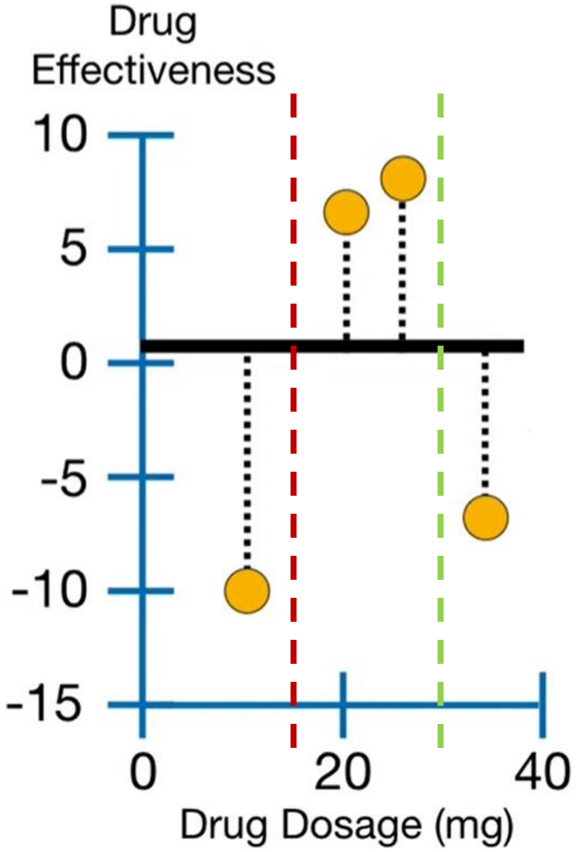
$$Output\ Value = \frac{-10.5}{1+1} = -5.25$$

A Unique Regression Tree

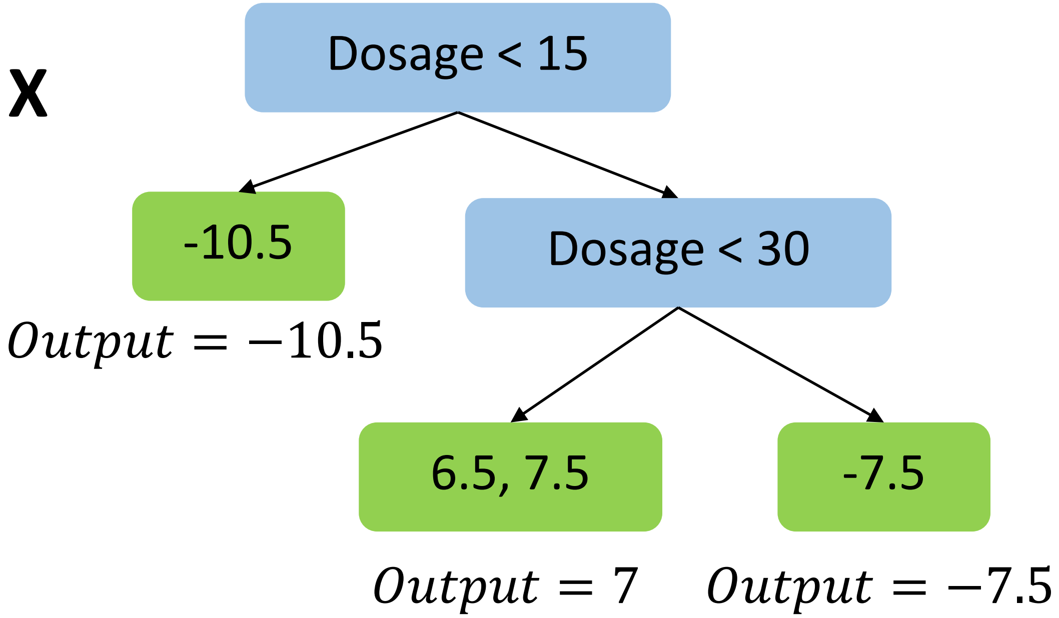
Initial Predicted Drug Effectiveness

0.5

+ **Learning Rate X**



Making predictions with XGBoost Tree



Dosage < 15

-10.5

$Output = -10.5$

Dosage < 30

6.5, 7.5

$Output = 7$

-7.5

$Output = -7.5$

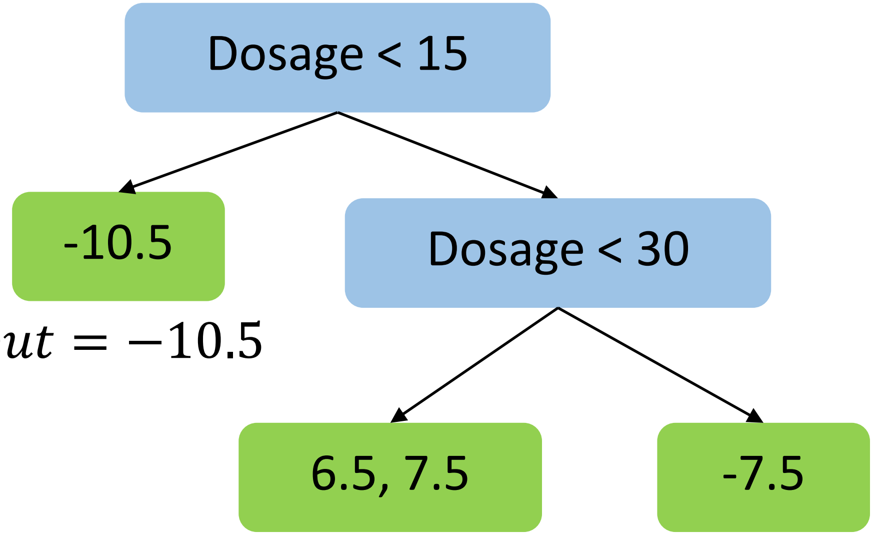
XGBoost calls the **Learning Rate, *ε*(eta),** and the default value is 0.3.

A Unique Regression Tree

Making predictions with XGBoost Tree

Initial Predicted Drug Effectiveness

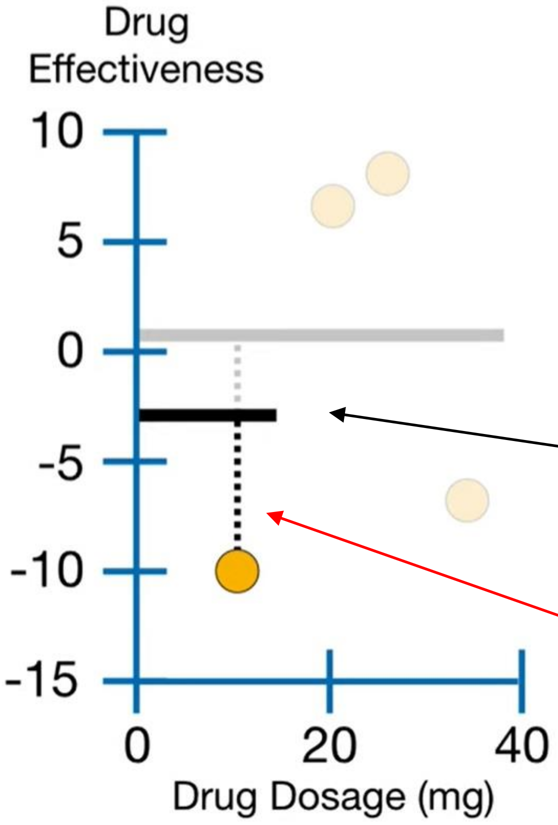
0.5 + **0.3** **X**



$Output = -10.5$

$Output = 7$

$Output = -7.5$



$$0.5 + \left(0.3 \times (-10.5)\right) = -2.65$$

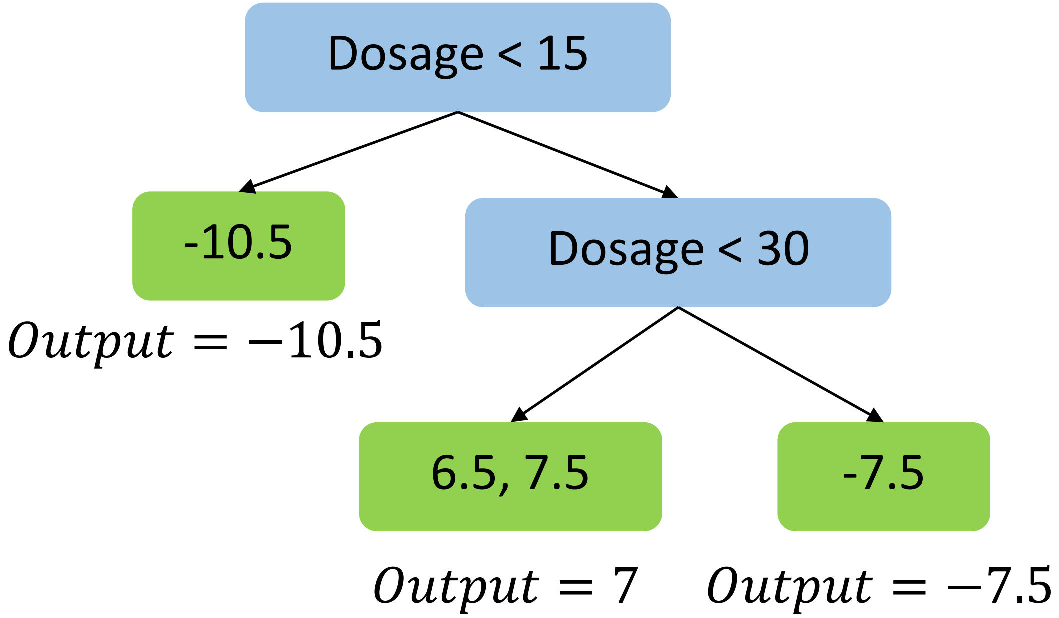
The new Residual is smaller than before, so we've taken a small step in the right direction.

A Unique Regression Tree

Making predictions with XGBoost Tree

Initial Predicted Drug Effectiveness

0.5 + **0.3** **X**



Dosage < 15

-10.5

$Output = -10.5$

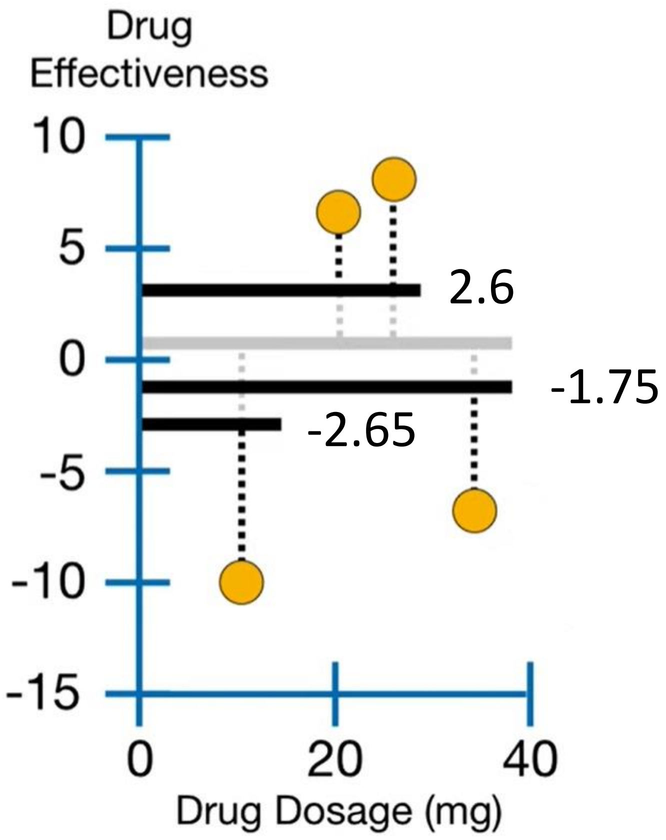
Dosage < 30

6.5, 7.5

$Output = 7$

-7.5

$Output = -7.5$

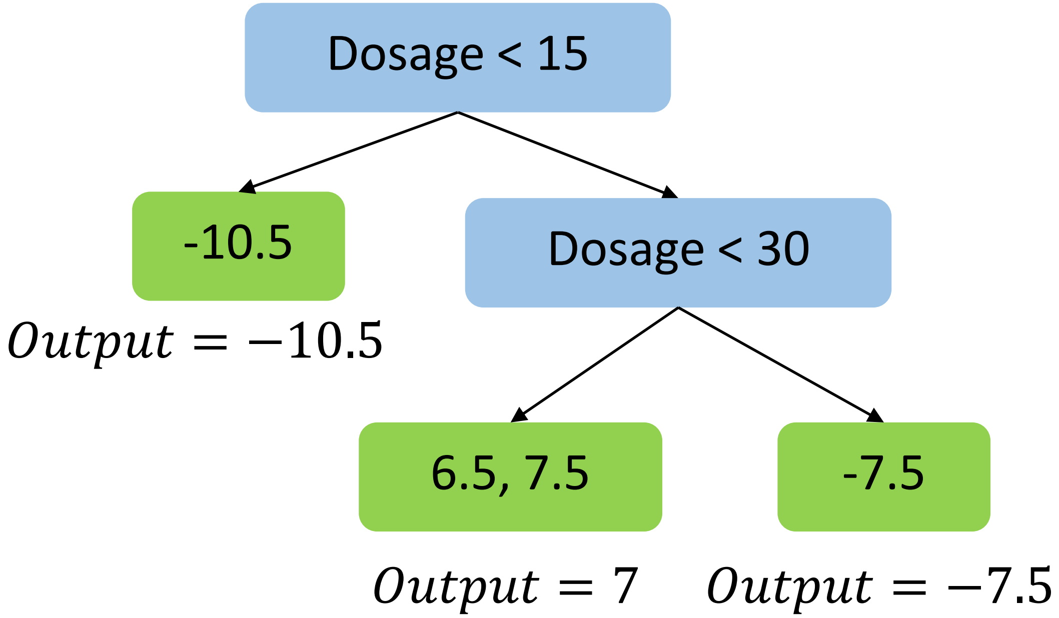


A Unique Regression Tree

Making predictions with XGBoost Tree

Initial Predicted Drug Effectiveness

0.5 + 0.3 X



Dosage < 15

-10.5

$Output = -10.5$

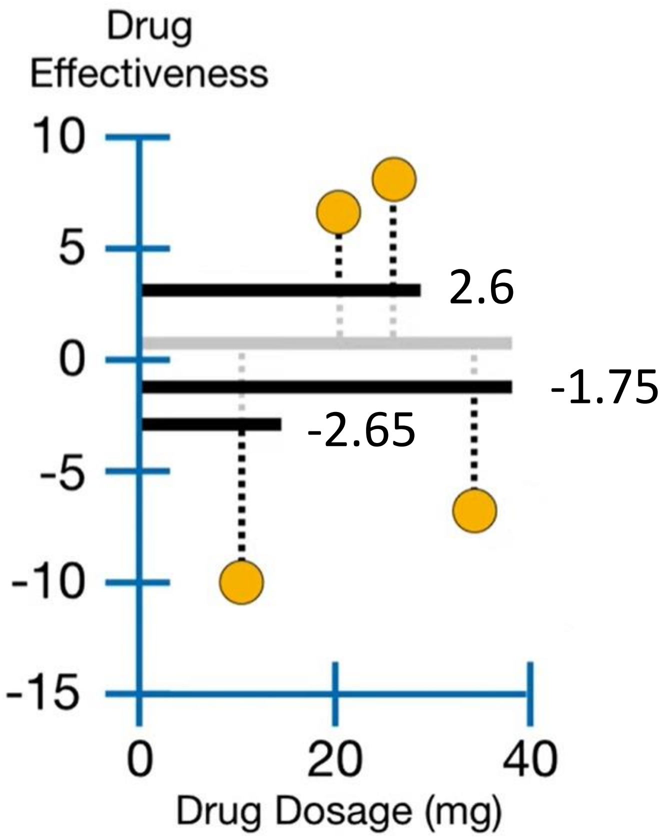
Dosage < 30

6.5, 7.5

$Output = 7$

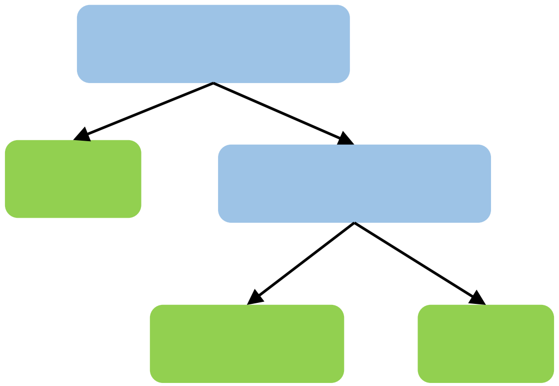
-7.5

$Output = -7.5$



Now we build another tree based on the new Residuals

+ 0.3 X

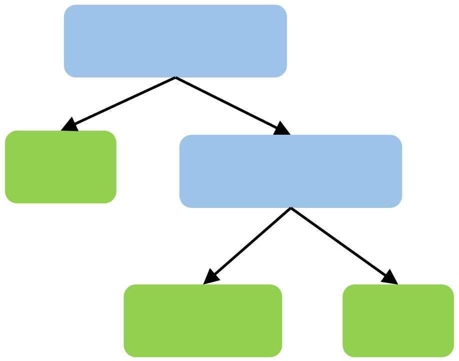


A Unique Regression Tree

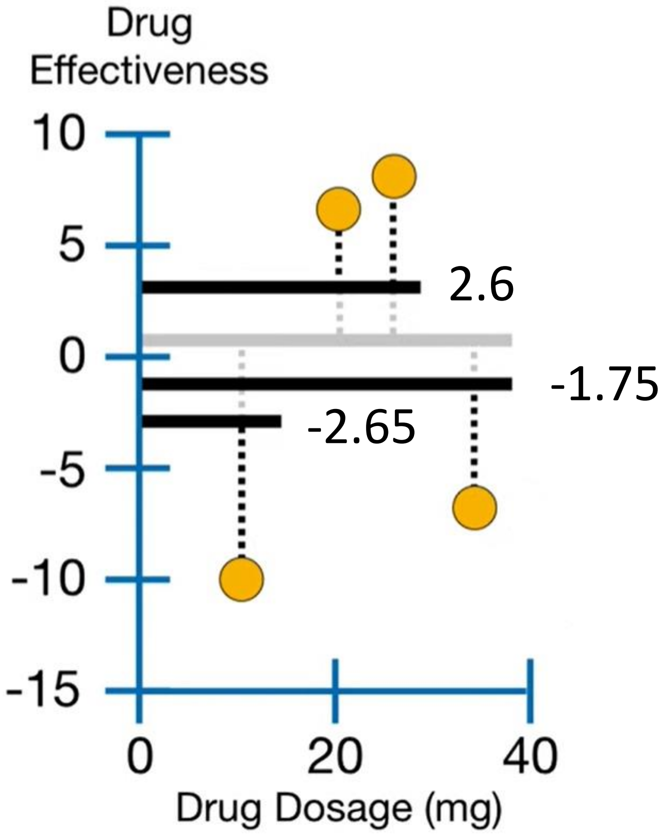
Making predictions with XGBoost Tree

Initial Predicted Drug Effectiveness

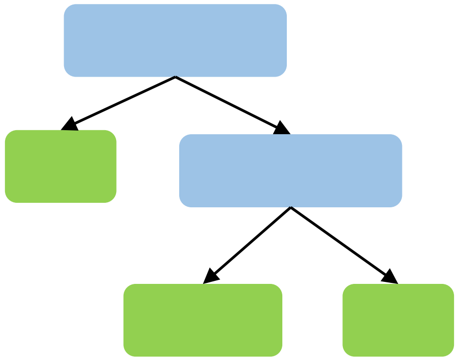
0.5 + 0.3 X



Now we build another tree based on the new Residuals



+ 0.3 X



...

...and we keep building trees until the Residuals are super small, or we have reached the maximum number.

Gradient Boost

Regularization

A Unique Regression Tree

Approximate Greedy Algorithm

Weighted Quantile Sketch

Sparsity-Aware Split Finding

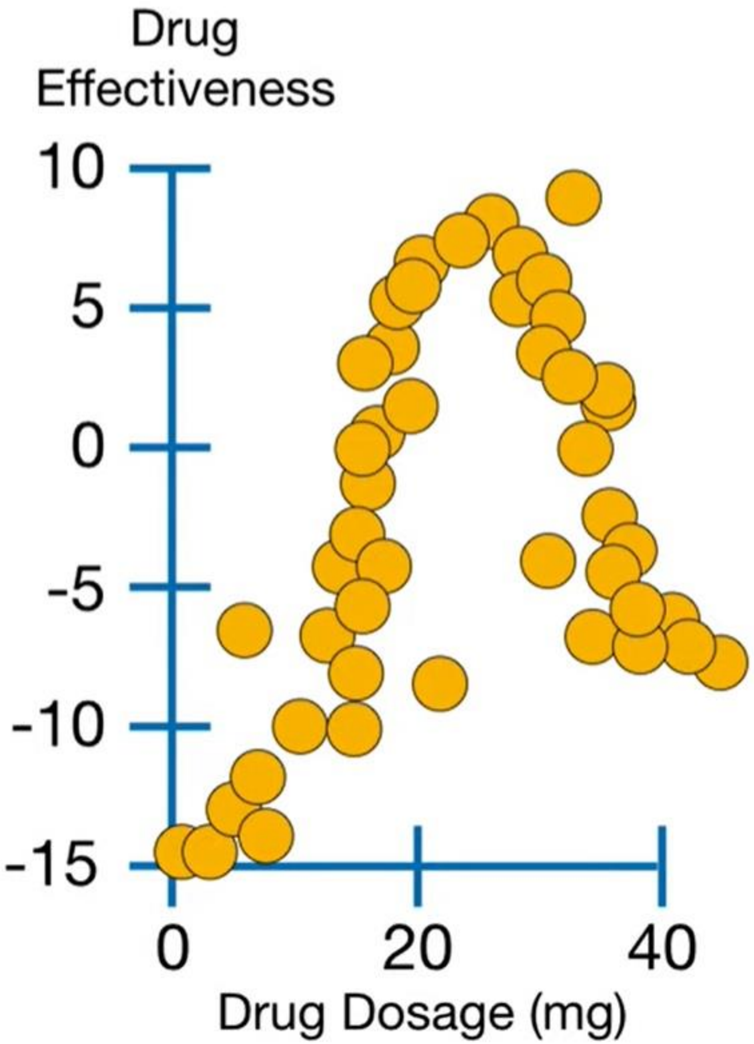
Parallel Learning

Cache-Aware Access

Blocks for Out-of-Core Computation

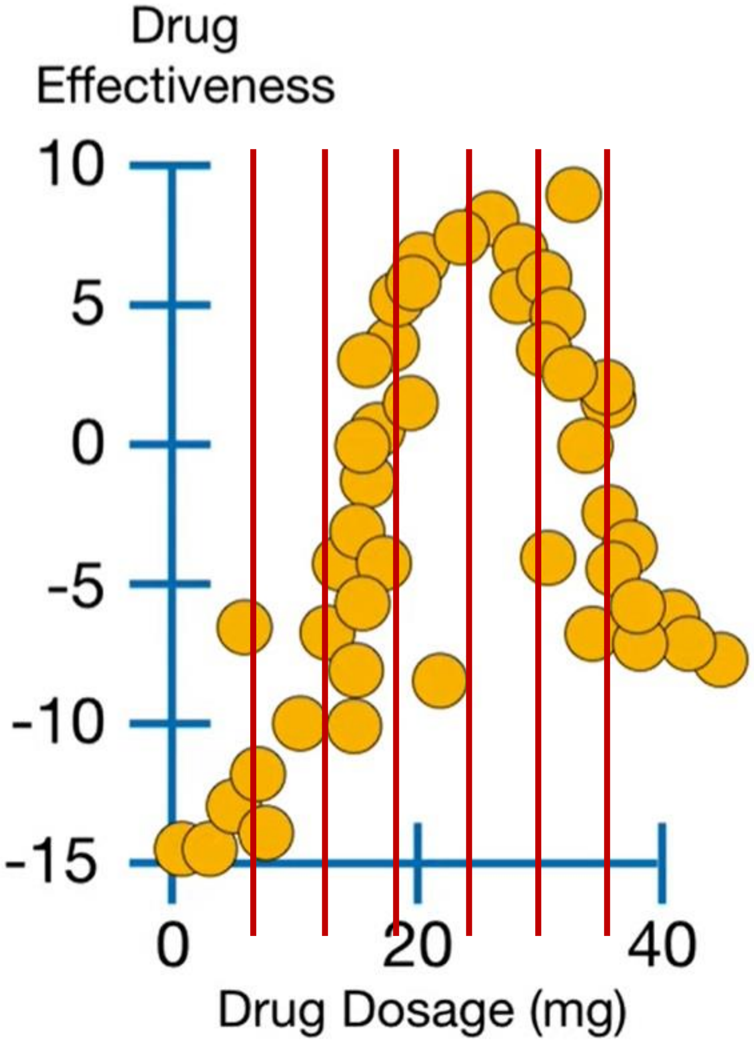
These parts are what make **XGBoost** relatively **efficient** with relatively **large training datasets**.

Approximate Greedy Algorithm



Instead of testing every single threshold, we could divide the data into **Quantiles**…

Approximate Greedy Algorithm

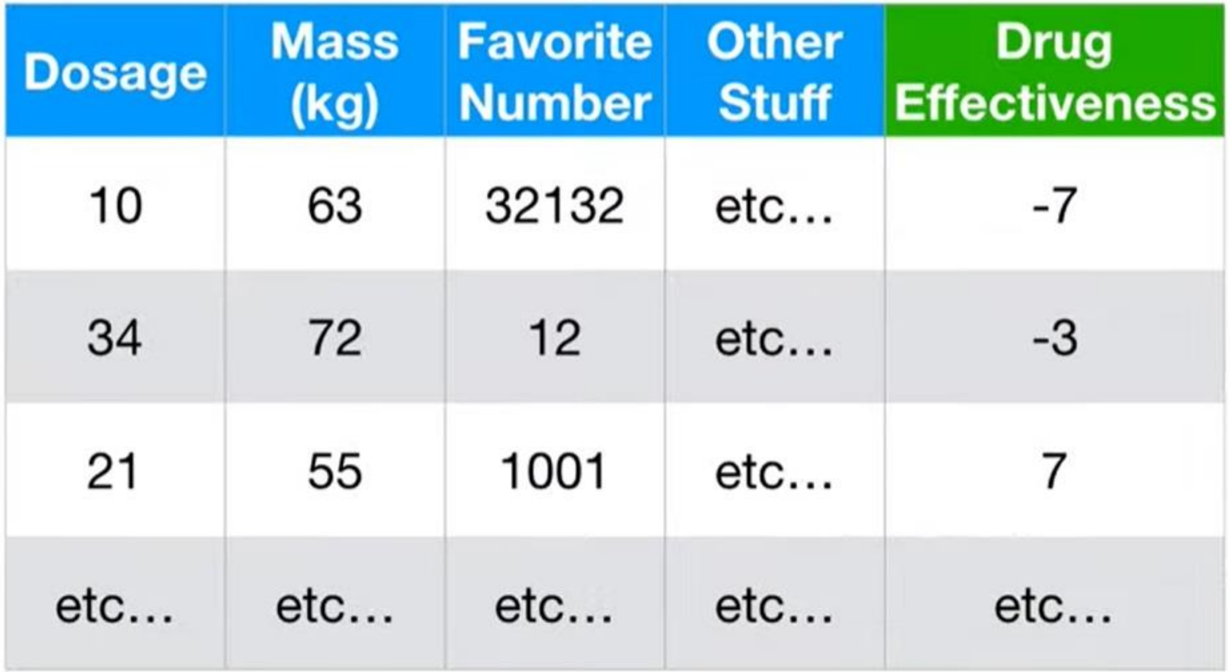


Instead of testing every single threshold, we could divide the data into **Quantiles**…

…and only use the quantiles as candidate thresholds to split the observations.

By default, the **Approximate Greedy Algorithm** uses about **33** quantiles.

Weighted Quantile Sketch



Dosage	Mass (kg)	Favorite Number	Other Stuff	Drug Effectiveness
10	63	32132	etc…	-7
34	72	12	etc…	-3
21	55	1001	etc…	7
etc…	etc…	etc…	etc…	etc…

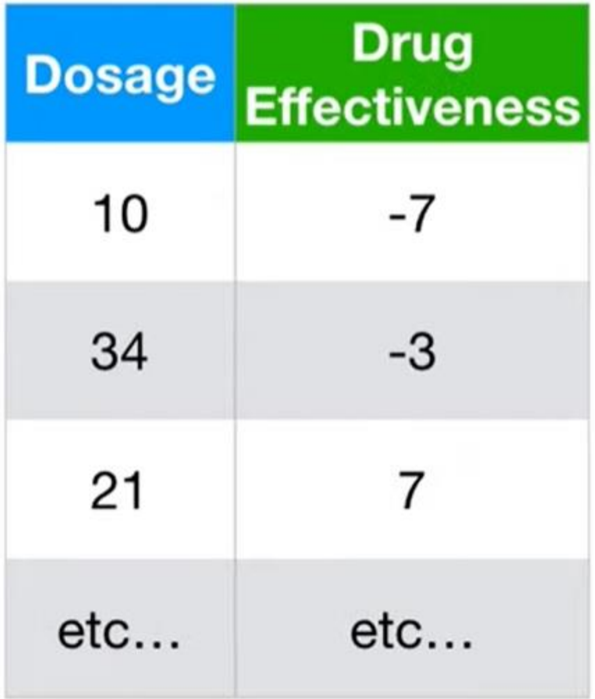
When you have tons and tons of data…

…so much data that you can't fit it all into a computer memory at one time…

…then things that seem simple, like sorting a list of numbers and finding quantiles, become really slow.

To get around this problem, a class of algorithms, called **Sketches**, can quickly create ***approximate*** solutions.

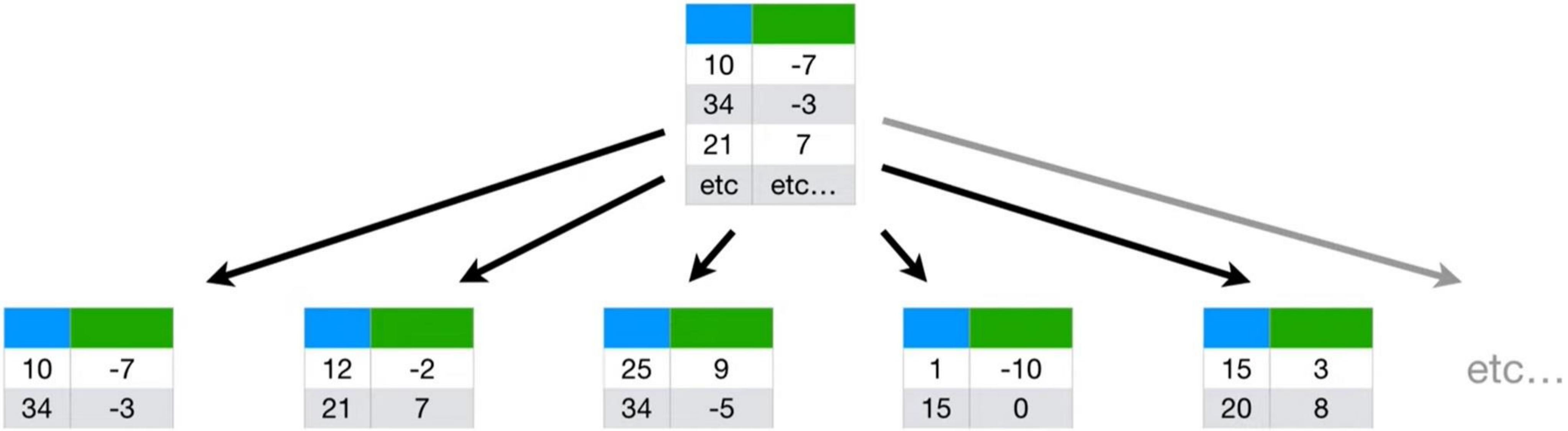
Weighted Quantile Sketch



Dosage	Drug Effectiveness
10	-7
34	-3
21	7
etc…	etc…

For this example, imagine we are just using a ton of **Dosages** to predict **Drug Effectiveness**.

Weighted Quantile Sketch + Parallel Learning



🟦	🟩
10	-7
34	-3
21	7
etc	etc…

🟦	🟩
10	-7
34	-3

🟦	🟩
12	-2
21	7

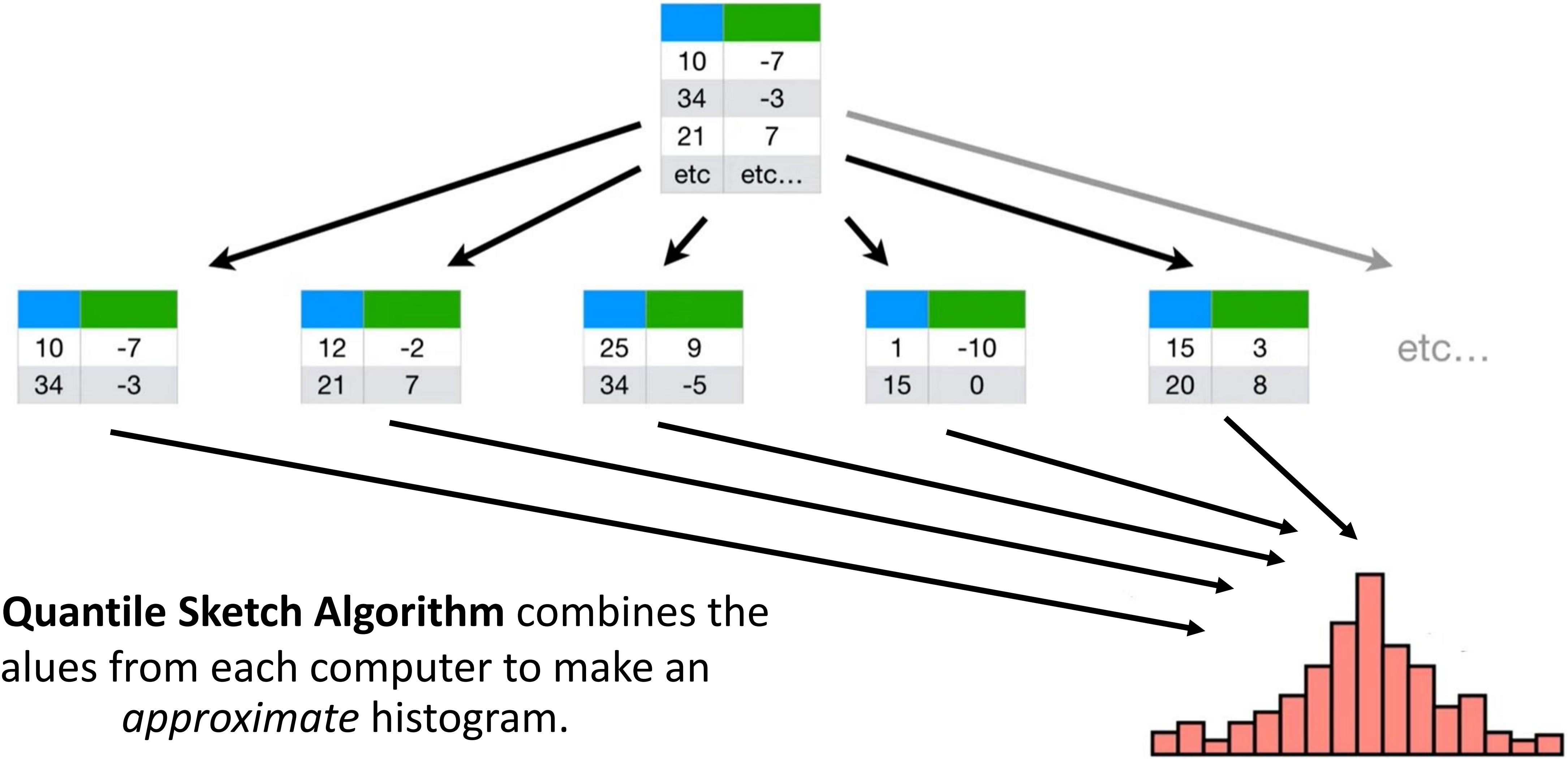
🟦	🟩
25	9
34	-5

🟦	🟩
1	-10
15	0

🟦	🟩
15	3
20	8

Imagine splitting it into small pieces and putting the pieces on different computers on a network.

Weighted Quantile Sketch + Parallel Learning



🟦	🟩
10	-7
34	-3
21	7
etc	etc…

🟦	🟩
10	-7
34	-3

🟦	🟩
12	-2
21	7

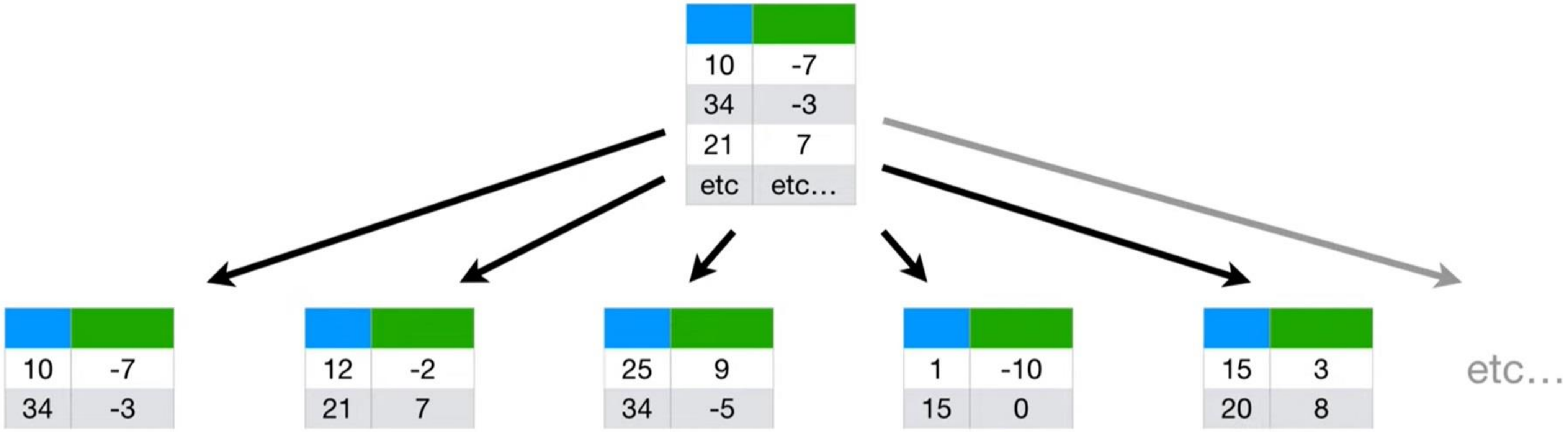
🟦	🟩
25	9
34	-5

🟦	🟩
1	-10
15	0

🟦	🟩
15	3
20	8

The **Quantile Sketch Algorithm** combines the values from each computer to make an *approximate* histogram.

Weighted Quantile Sketch + Parallel Learning



🟦	🟩
10	-7
34	-3
21	7
etc	etc…

🟦	🟩
10	-7
34	-3

🟦	🟩
12	-2
21	7

🟦	🟩
25	9
34	-5

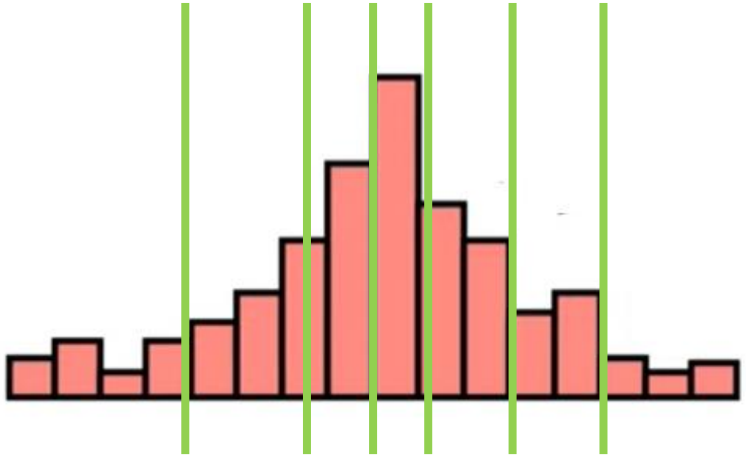
🟦	🟩
1	-10
15	0

🟦	🟩
15	3
20	8

etc…

Then, the approximate histogram is used to calculate *approximate quantiles*.

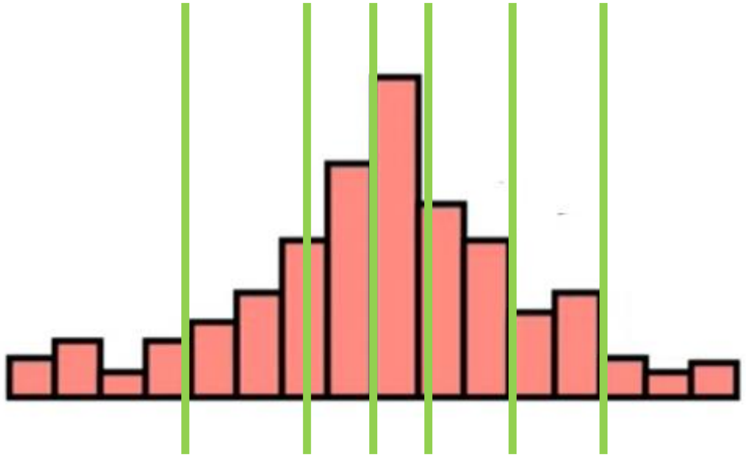
And the Approximate Greedy Algorithm uses approximate quantiles.



Weighted Quantile Sketch

But **XGBoost** uses a **Weighted Quantile Sketch**.

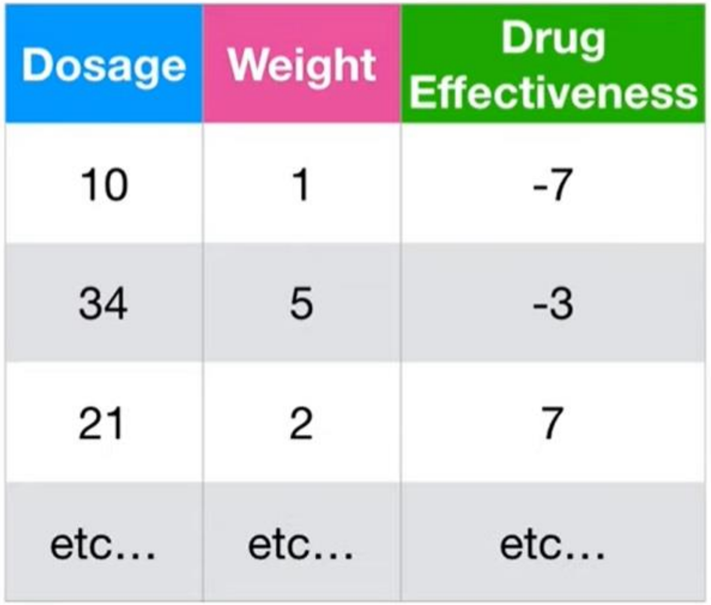
Usually quantiles are set up so that the same number of observations are in each one.



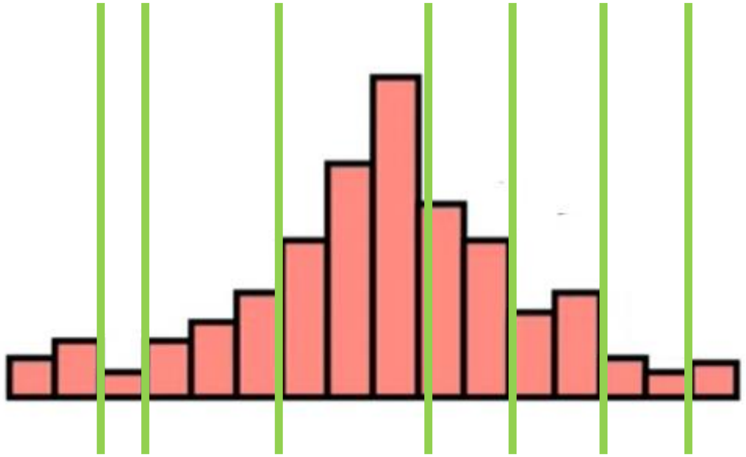
Weighted Quantile Sketch

In contrast, with weighted quantiles, each observation has a corresponding **Weight** and the sum of the **Weights** are the same in each quantiles.

Specifically, the weight for each observation is the 2nd derivative of the **Loss Function**, what we are referring as the **Hessian**.

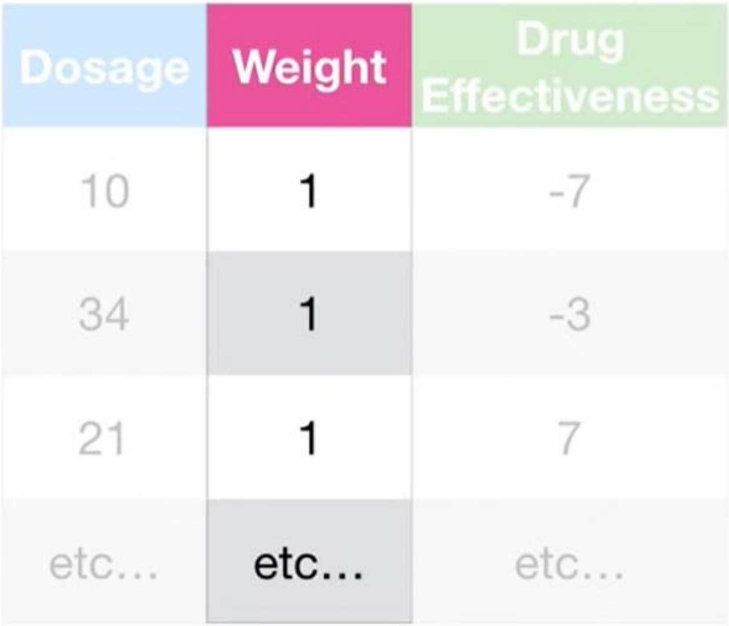


Dosage	Weight	Drug Effectiveness
10	1	-7
34	5	-3
21	2	7
etc…	etc…	etc…



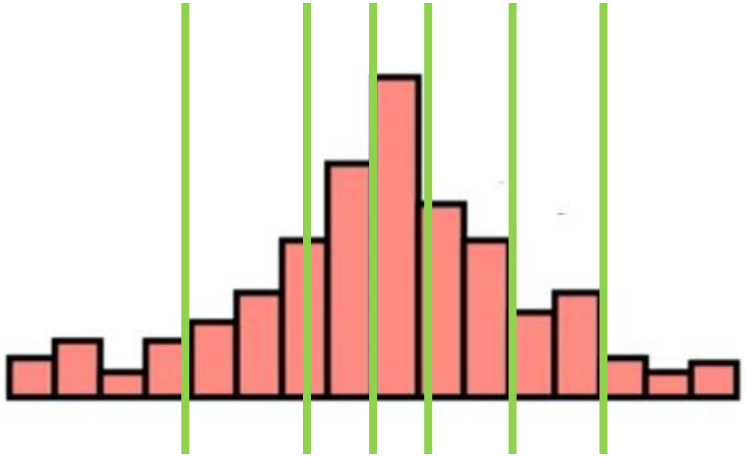
Weighted Quantile Sketch

For **Regression**, the **Weights** are all equal to **1**…



Dosage	Weight	Drug Effectiveness
10	1	-7
34	1	-3
21	1	7
etc…	etc…	etc…

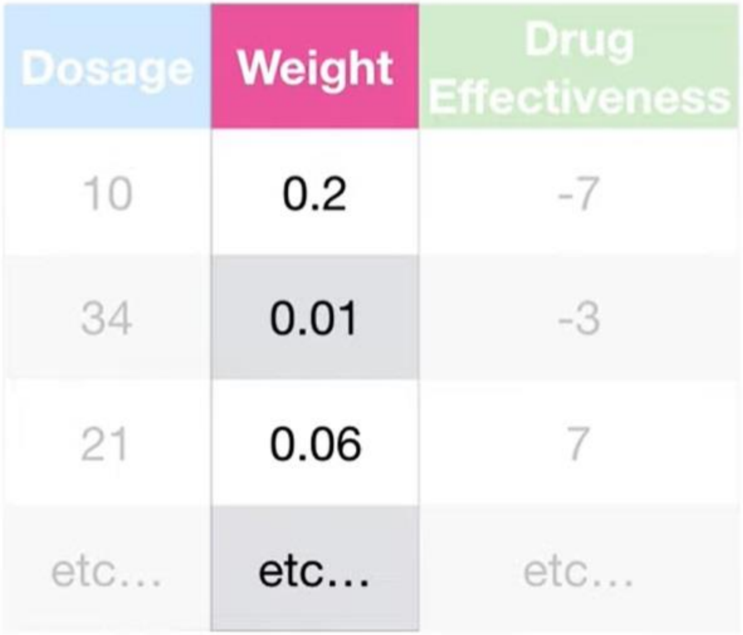
And that means the weighted quantiles are just like normal quantiles and contain an equal number of observations.



Weighted Quantile Sketch

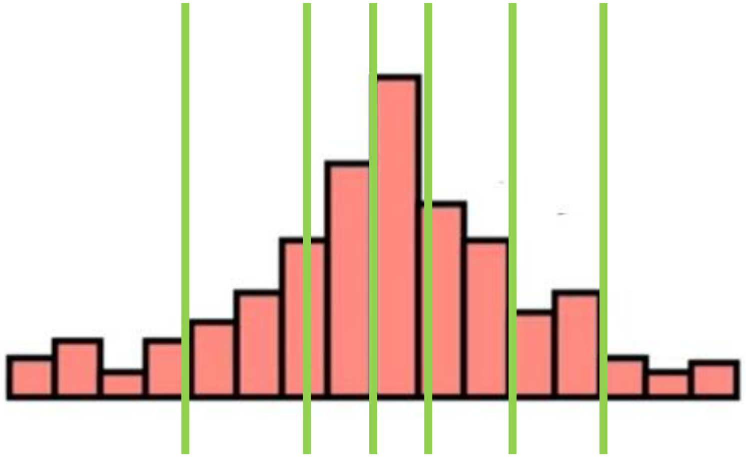
In contrast, for **Classification**, the **Weights** are…

$$Weight = Previous\ Probability_i \times (1 - Previous\ Probability_i)$$



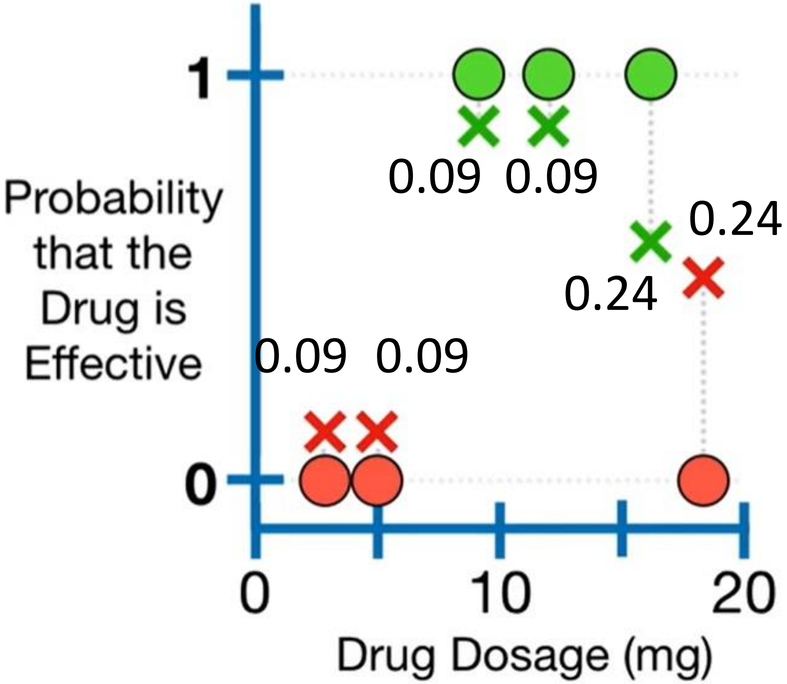
Dosage	Weight	Drug Effectiveness
10	0.2	-7
34	0.01	-3
21	0.06	7
etc…	etc…	etc…

And that means the weighted quantiles are just like normal quantiles and contain an equal number of observations.



Weighted Quantile Sketch

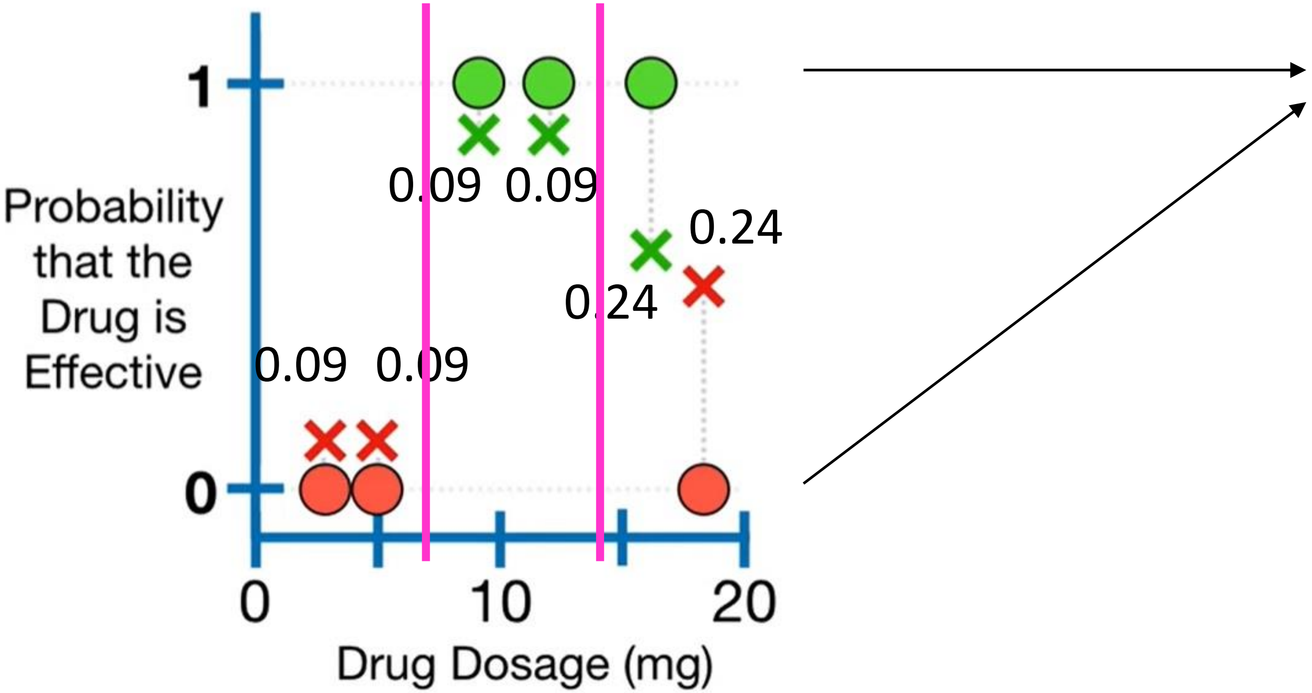
$$Weight = Previous\ Probability_i \times (1 - Previous\ Probability_i)$$



Weighted Quantile Sketch

$$Weight = Previous\ Probability_i \times (1 - Previous\ Probability_i)$$

If we split this data into equal quantiles…

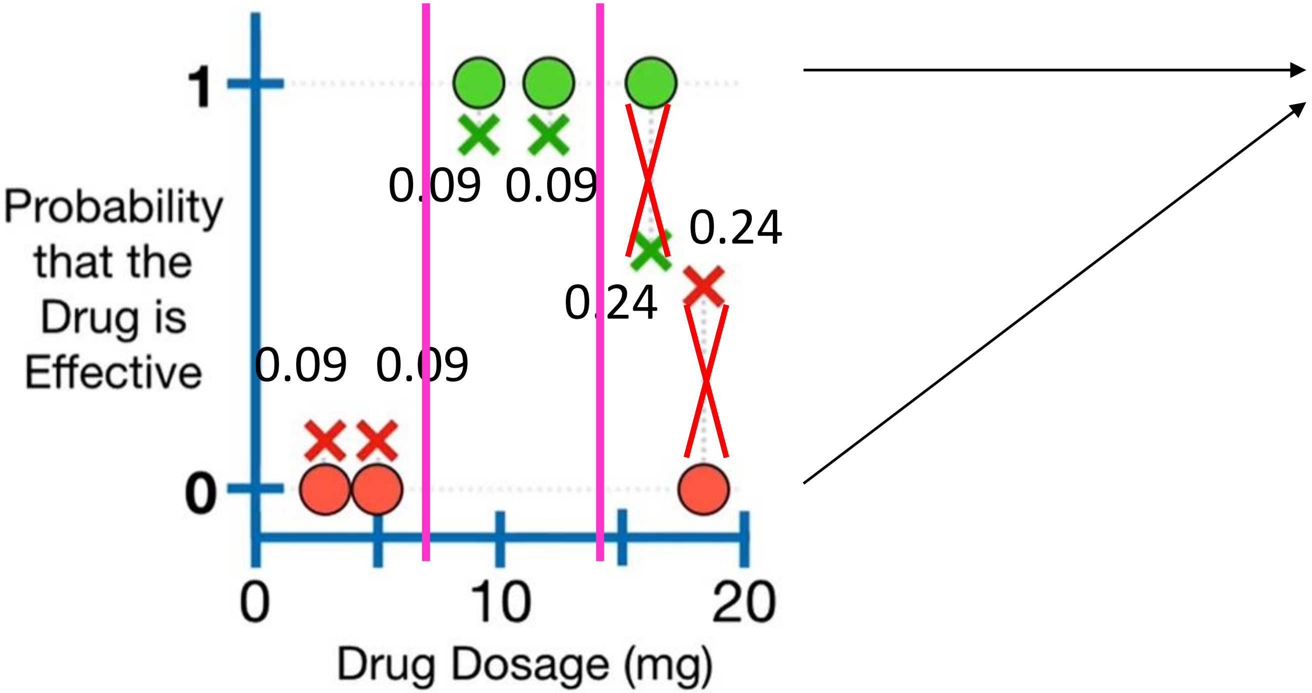


Last two observations will end up in the same leaf.

Weighted Quantile Sketch

$$Weight = Previous\ Probability_i \times (1 - Previous\ Probability_i)$$

If we split this data into equal quantiles…



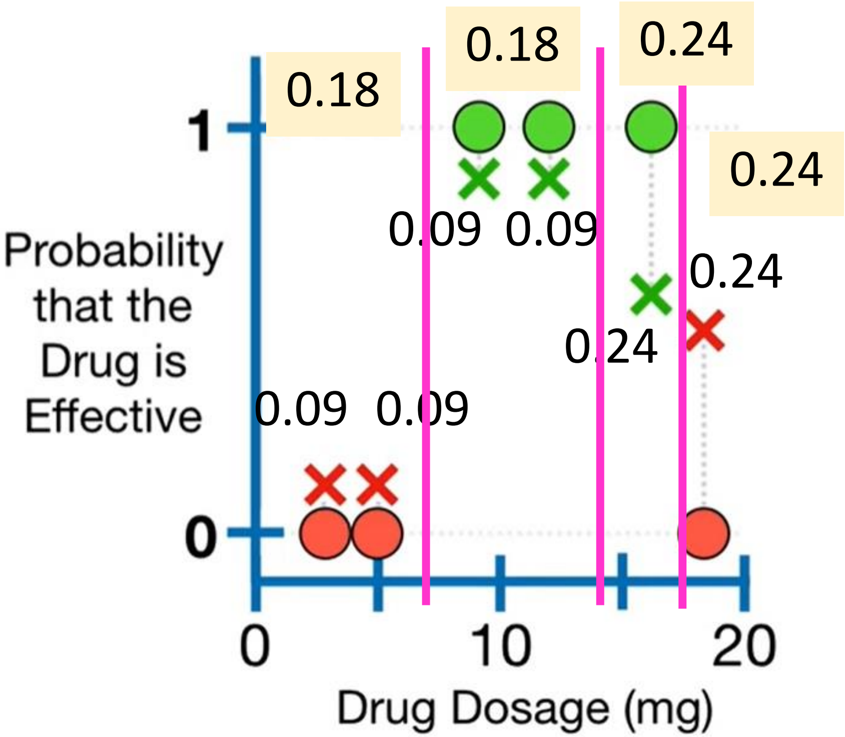
Last two observations will end up in the same leaf.

Moreover, the positive residual will cancel out the negative residual, it will be very difficult to improve the predicted probabilities.

Weighted Quantile Sketch

$$Weight = Previous\ Probability_i \times (1 - Previous\ Probability_i)$$

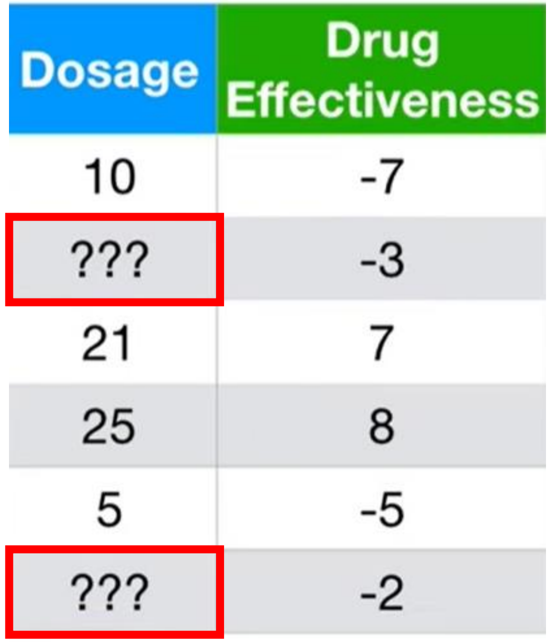
If we split this data into equal quantiles…



So, instead of using equal quantiles, XGBoost tries to make quantiles that have a similar ***sum of weights***.

The advantage of using the **Weighted Quantile Sketch** is that we get smaller quantiles when we need them.

Sparsity-Aware Split Finding



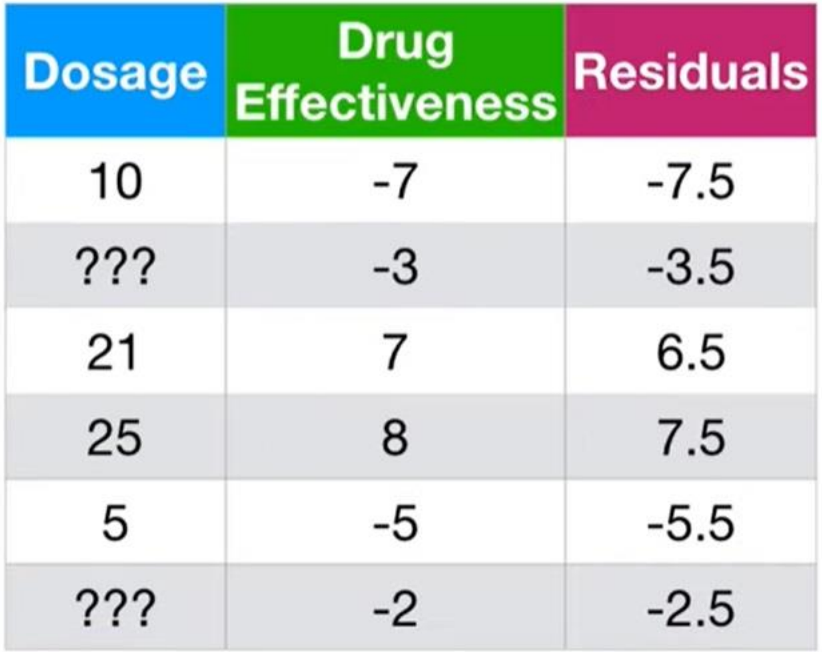
Dosage	Drug Effectiveness
10	-7
???	-3
21	7
25	8
5	-5
???	-2

We have a few missing values

Sparsity-Aware Split Finding

Initial Predicted Drug Effectiveness

0.5



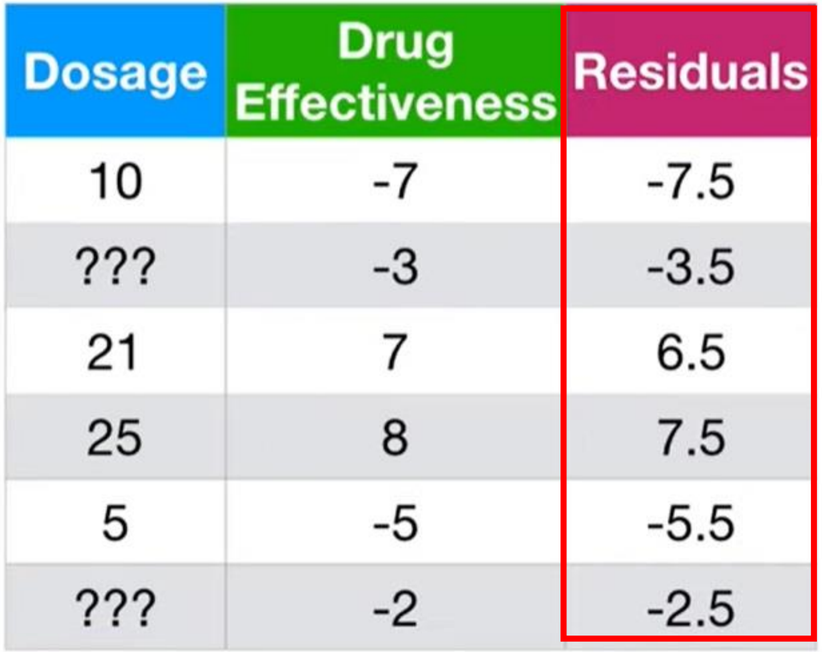
Dosage	Drug Effectiveness	Residuals
10	-7	-7.5
???	-3	-3.5
21	7	6.5
25	8	7.5
5	-5	-5.5
???	-2	-2.5

Even though we have missing values, we can calculate the Residuals.

Sparsity-Aware Split Finding

Initial Predicted Drug Effectiveness

0.5



Dosage	Drug Effectiveness	Residuals
10	-7	-7.5
???	-3	-3.5
21	7	6.5
25	8	7.5
5	-5	-5.5
???	-2	-2.5

-7.5, -3.5, 6.5, 7.5, -5.5, 2.5

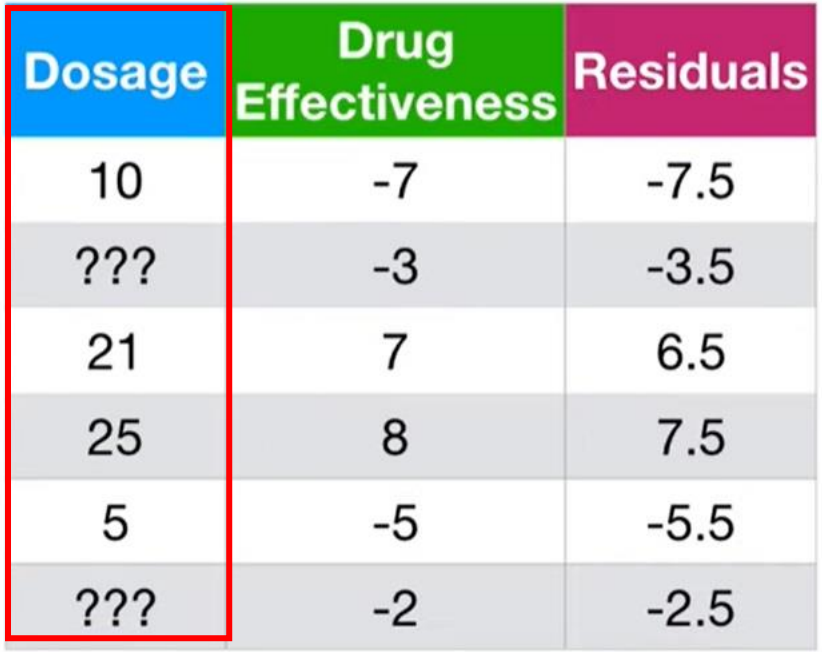
And just like we normally do when we build **XGBoost Trees**, we can put all of the **Residuals** into a single leaf.

Sparsity-Aware Split Finding

-7.5, -3.5, 6.5, 7.5, -5.5, 2.5

Initial Predicted Drug Effectiveness

0.5



Dosage	Drug Effectiveness	Residuals
10	-7	-7.5
???	-3	-3.5
21	7	6.5
25	8	7.5
5	-5	-5.5
???	-2	-2.5

So, just like we always do for continuous data, we need to sort the **Dosages** from low to high.

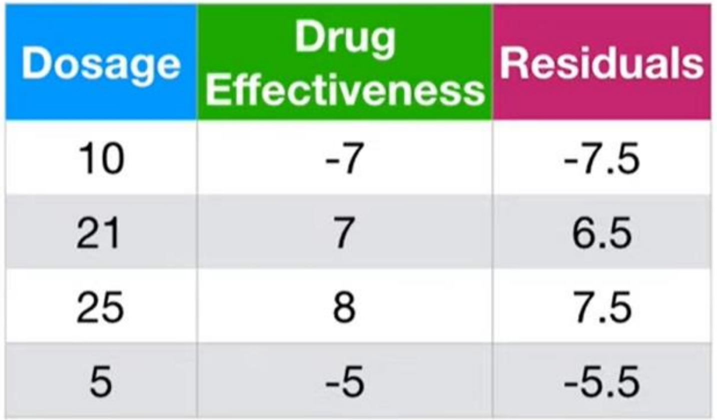
Unfortunately, it’s unclear how to sort the **Dosages** with missing values.

Sparsity-Aware Split Finding

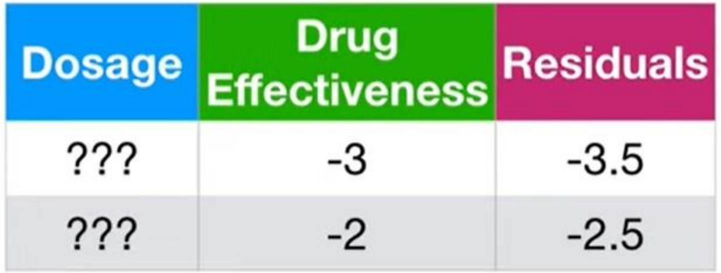
-7.5, -3.5, 6.5, 7.5, -5.5, 2.5

Initial Predicted Drug
Effectiveness

0.5



Dosage	Drug Effectiveness	Residuals
10	-7	-7.5
21	7	6.5
25	8	7.5
5	-5	-5.5



Dosage	Drug Effectiveness	Residuals
???	-3	-3.5
???	-2	-2.5

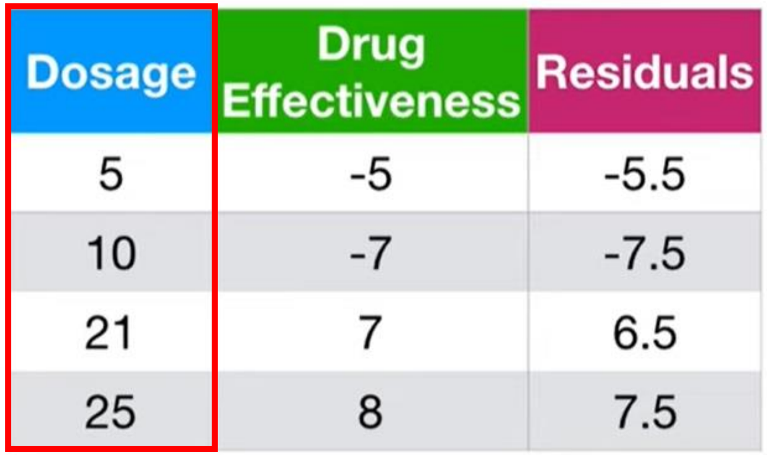
So, we'll split the data into two tables.

Sparsity-Aware Split Finding

-7.5, -3.5, 6.5, 7.5, -5.5, 2.5

Initial Predicted Drug Effectiveness

0.5



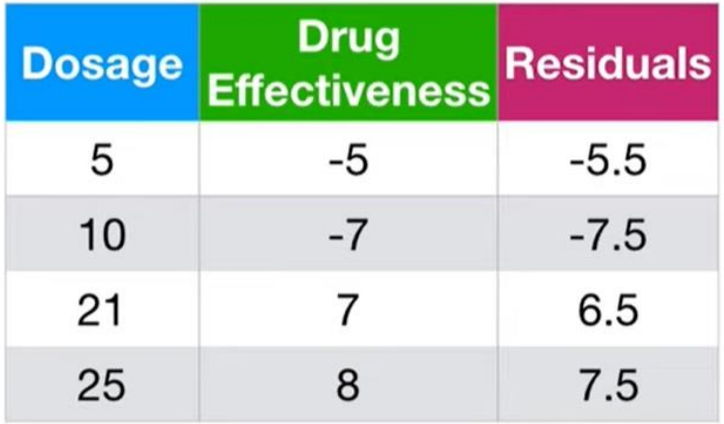
Dosage	Drug Effectiveness	Residuals
5	-5	-5.5
10	-7	-7.5
21	7	6.5
25	8	7.5

Focusing on the table that has **Dosage** values for every observation, we sort rows by **Dosage**, from low to high.

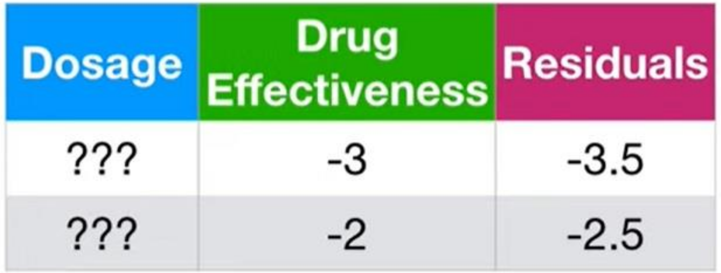
Sparsity-Aware Split Finding

Initial Predicted Drug Effectiveness

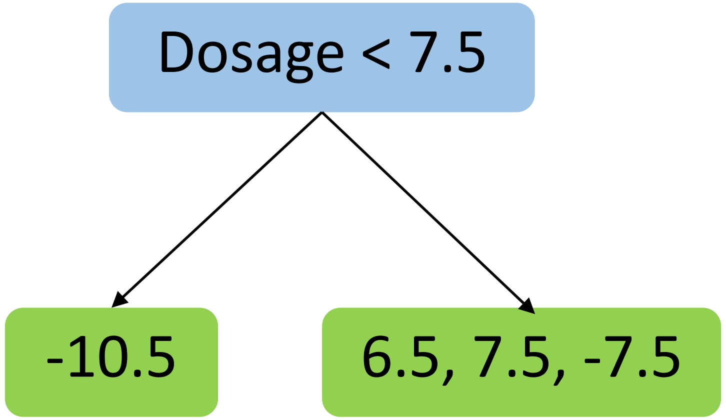
0.5



Dosage	Drug Effectiveness	Residuals
5	-5	-5.5
10	-7	-7.5
21	7	6.5
25	8	7.5



Dosage	Drug Effectiveness	Residuals
???	-3	-3.5
???	-2	-2.5



Dosage < 7.5

-10.5

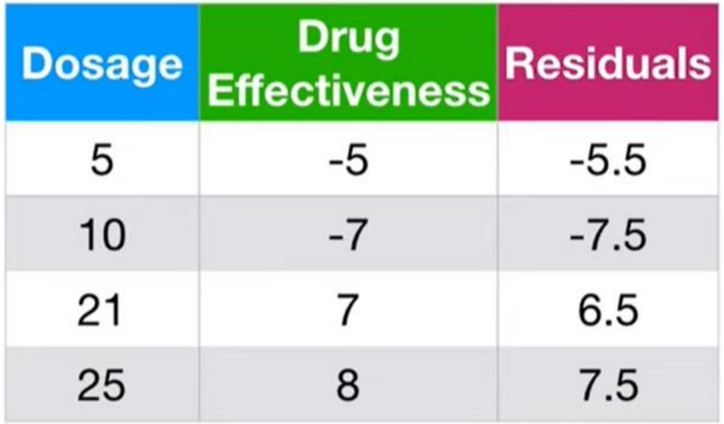
6.5, 7.5, -7.5

Now that we have all of the **Residuals** with known **Dosages** in the tree, we calculate two separate **Gain** values.

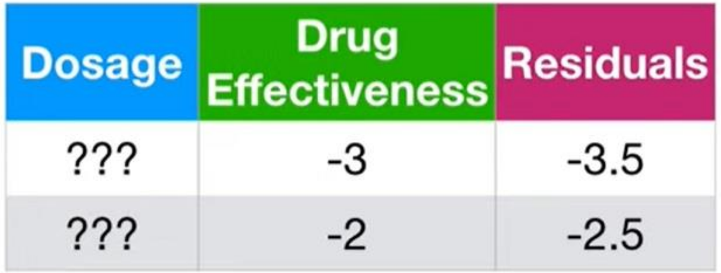
Sparsity-Aware Split Finding

Initial Predicted Drug
Effectiveness

0.5

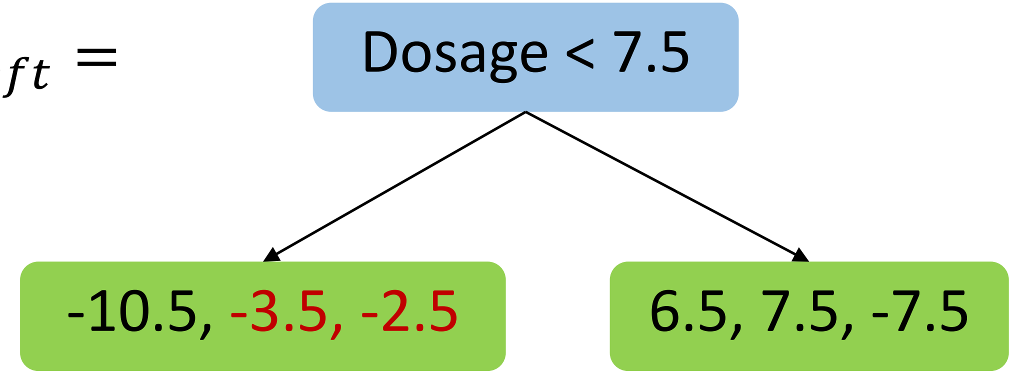


Dosage	Drug Effectiveness	Residuals
5	-5	-5.5
10	-7	-7.5
21	7	6.5
25	8	7.5



Dosage	Drug Effectiveness	Residuals
???	-3	-3.5
???	-2	-2.5

$Gain_{Left} =$

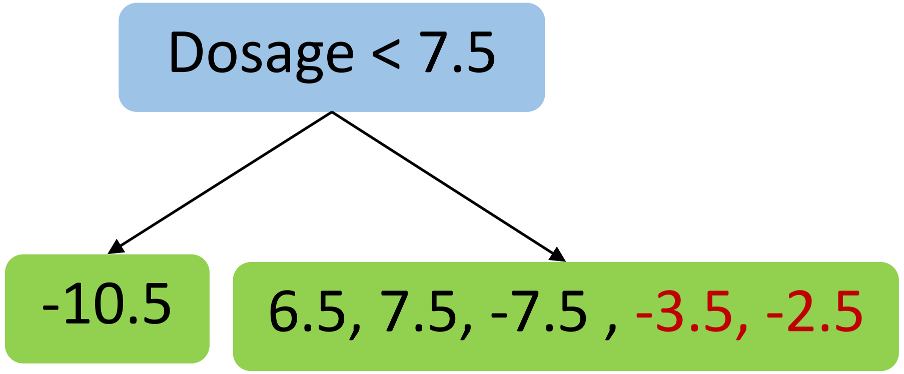


Dosage < 7.5

-10.5, -3.5, -2.5

6.5, 7.5, -7.5

$Gain_{Right} =$



Dosage < 7.5

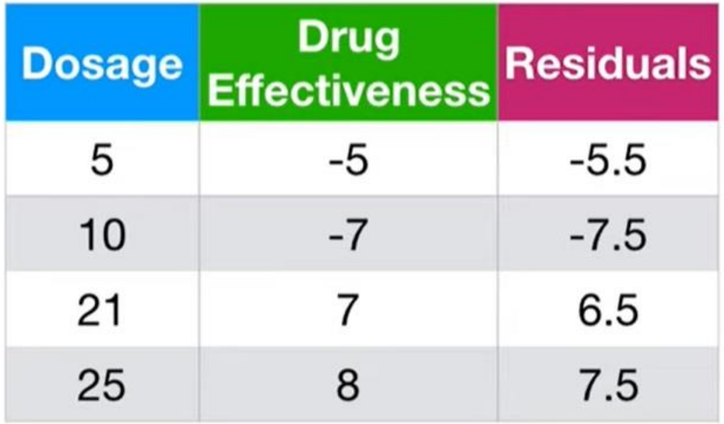
-10.5

6.5, 7.5, -7.5 , -3.5, -2.5

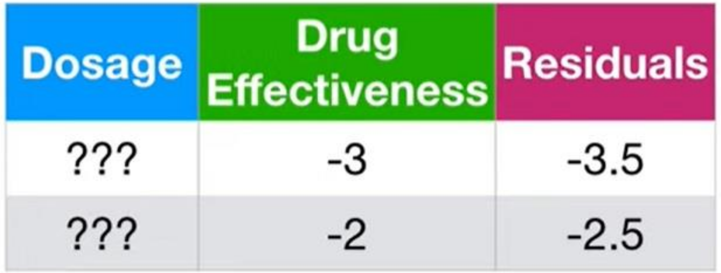
Sparsity-Aware Split Finding

Initial Predicted Drug Effectiveness

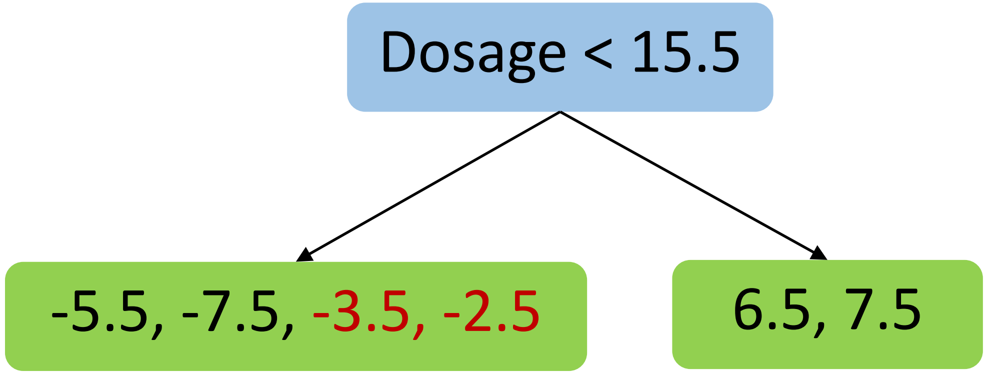
0.5



Dosage	Drug Effectiveness	Residuals
5	-5	-5.5
10	-7	-7.5
21	7	6.5
25	8	7.5



Dosage	Drug Effectiveness	Residuals
???	-3	-3.5
???	-2	-2.5



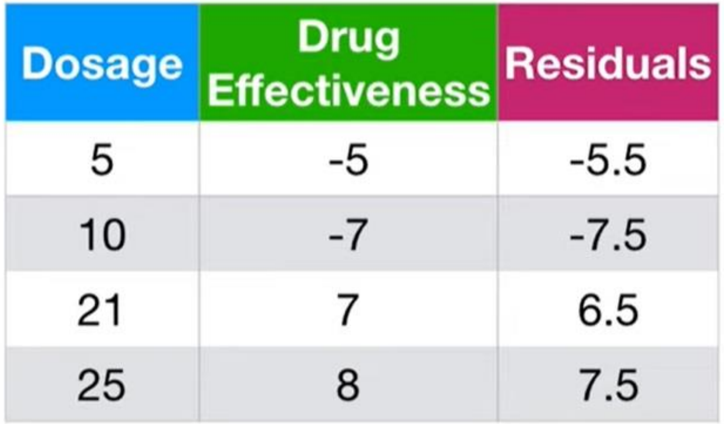
In the end, we choose the threshold that gave us the largest value for **Gain**, overall.

In this case, that meant picking $\boldsymbol{Gain_{Left}}$ when the threshold was **Dosage < 15.5**.

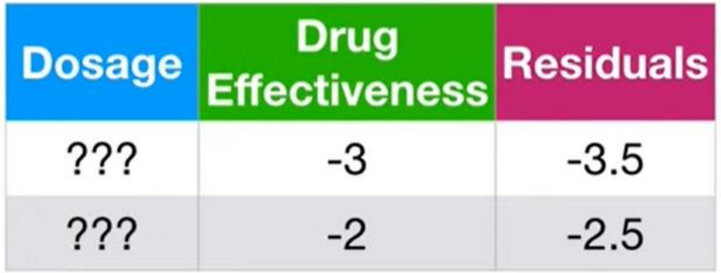
Sparsity-Aware Split Finding

Initial Predicted Drug Effectiveness

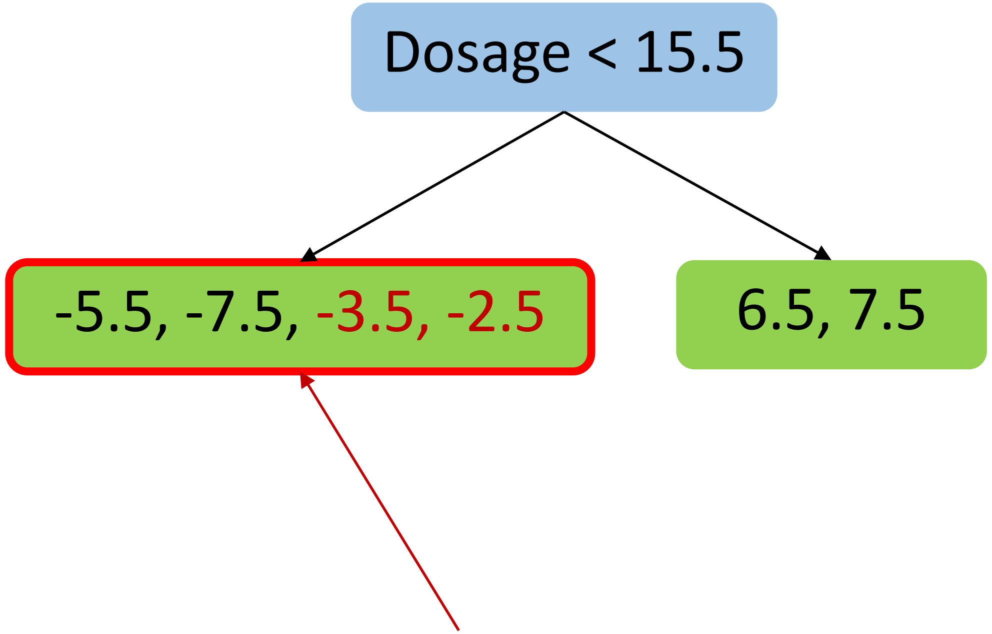
0.5



Dosage	Drug Effectiveness	Residuals
5	-5	-5.5
10	-7	-7.5
21	7	6.5
25	8	7.5

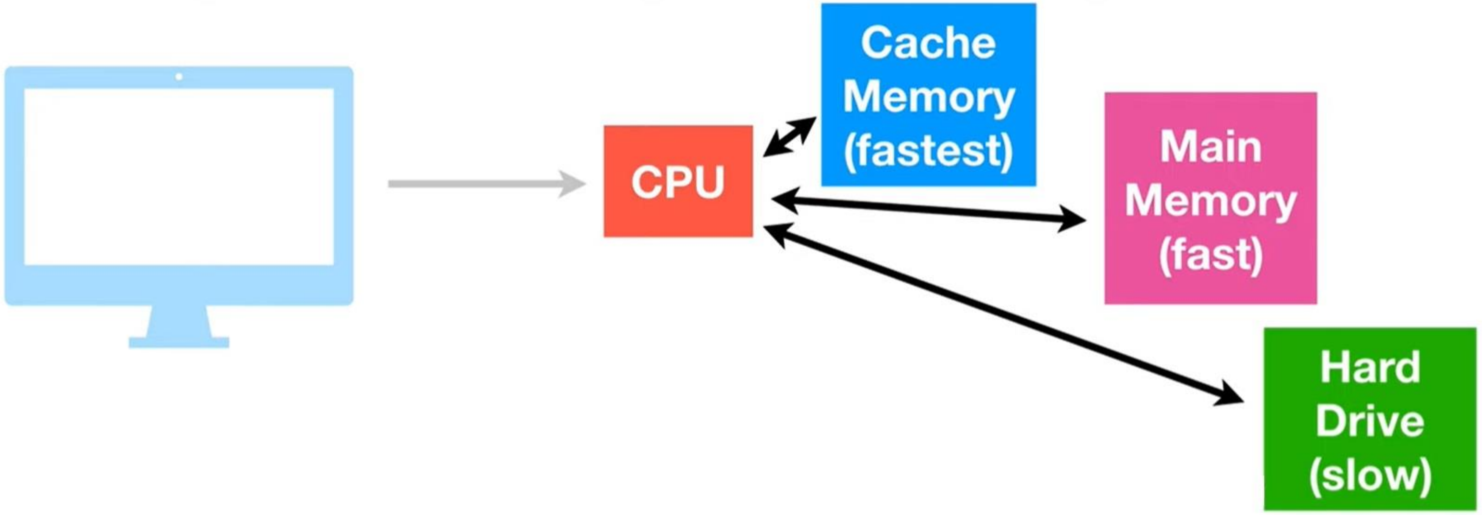


Dosage	Drug Effectiveness	Residuals
???	-3	-3.5
???	-2	-2.5



NOTE: This path, going to the **left leaf** when **Dosage < 15.5**, will be the default path for all future observations that are **missing Dosage values**.

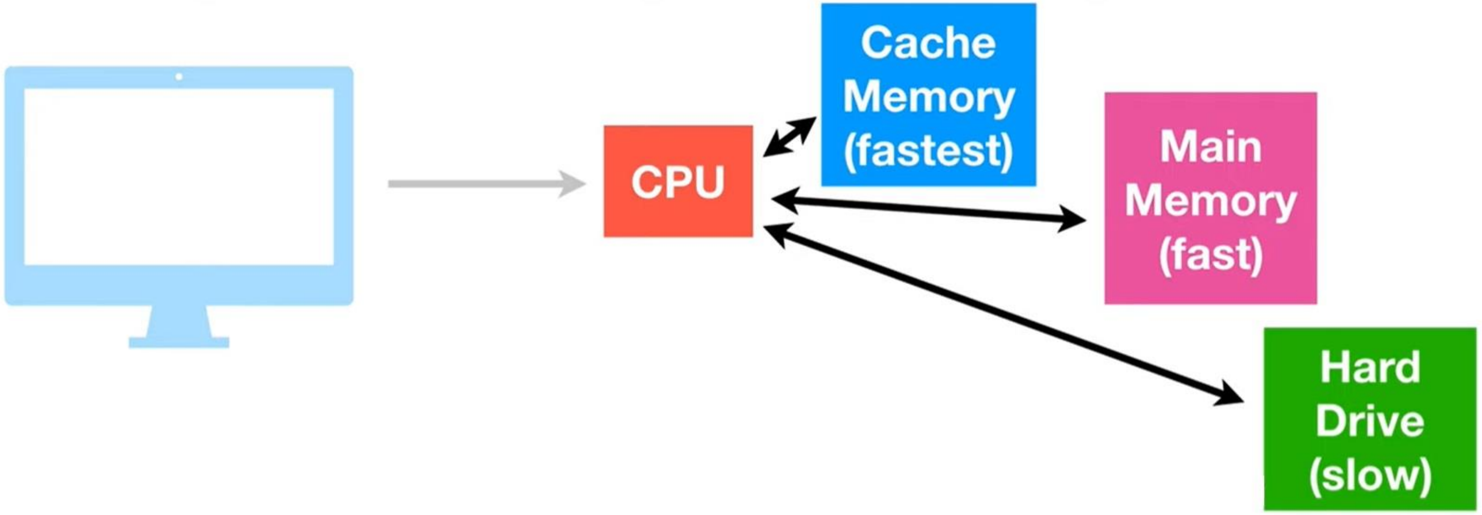
Cache-Aware Access



Inside each computer, we have a **CPU** (Central Processing Unit) and that **CPU** has a small amount of **Cache Memory**.

The **CPU** can use this memory **faster** than any other memory in the computer.

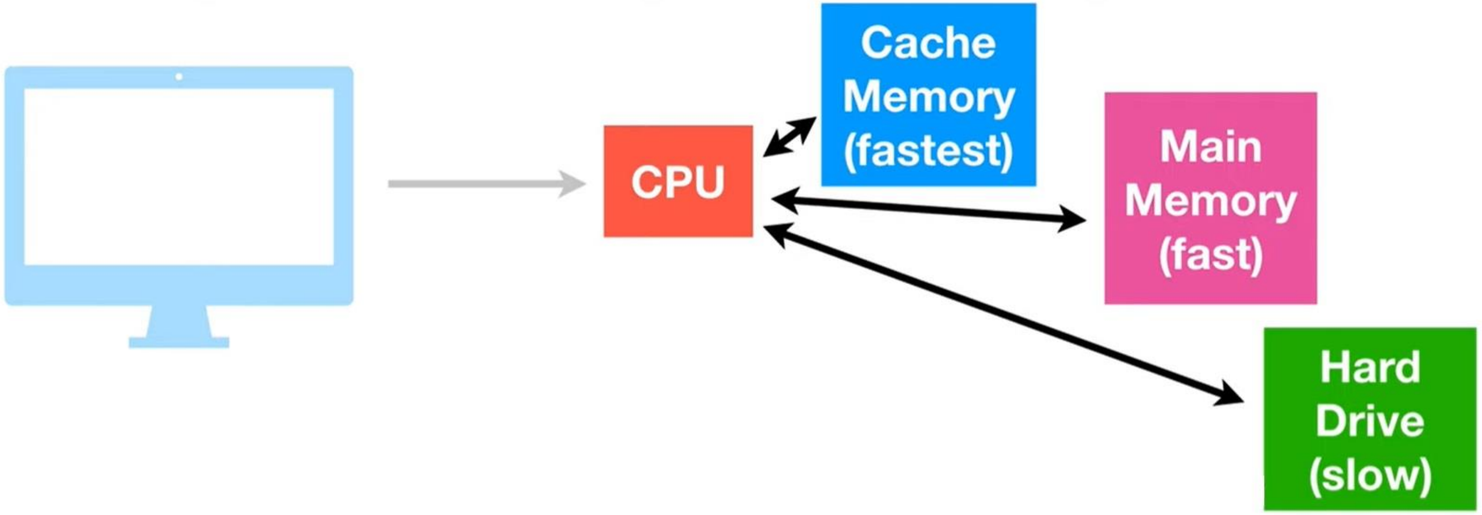
Cache-Aware Access



If you want your program to run really fast, the goal is to maximize what you can do with the **Cache Memory**.

So, **XGBoost** puts the **Gradients** and **Hessians** in the **Cache** so that it can rapidly calculate **Similarity Scores** and **Output Values**.

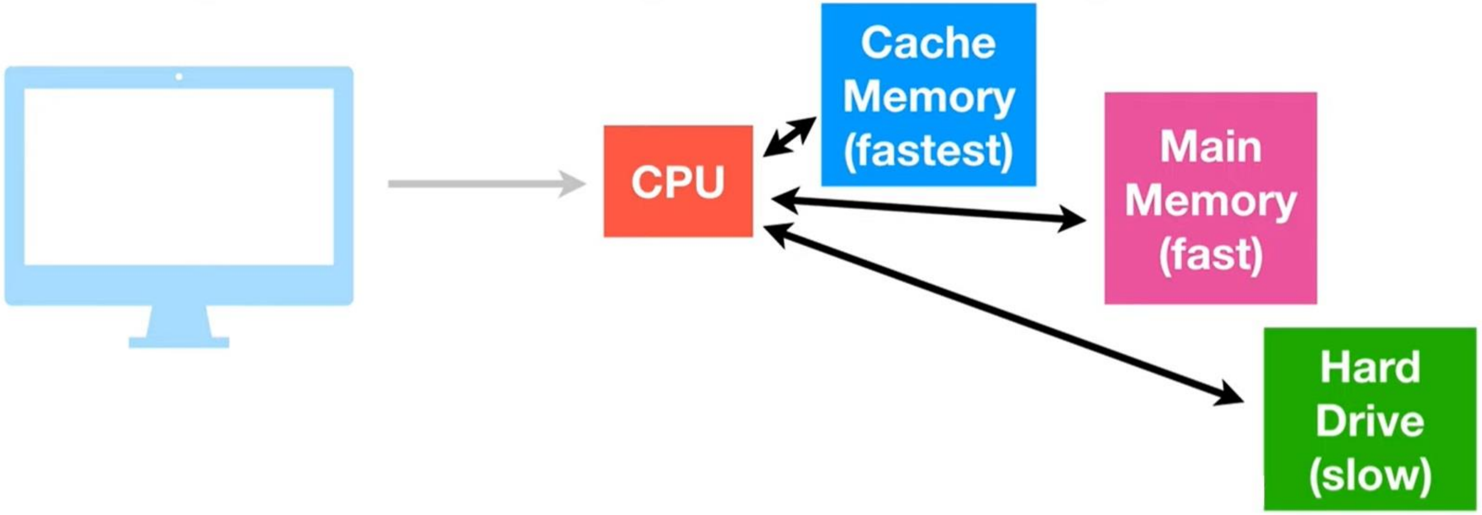
Blocks for Out-of-Core Computation



When the dataset is too large for **Cache** and **Main Memory**, then at least some of it, must be stored on the **Hard Drive**.

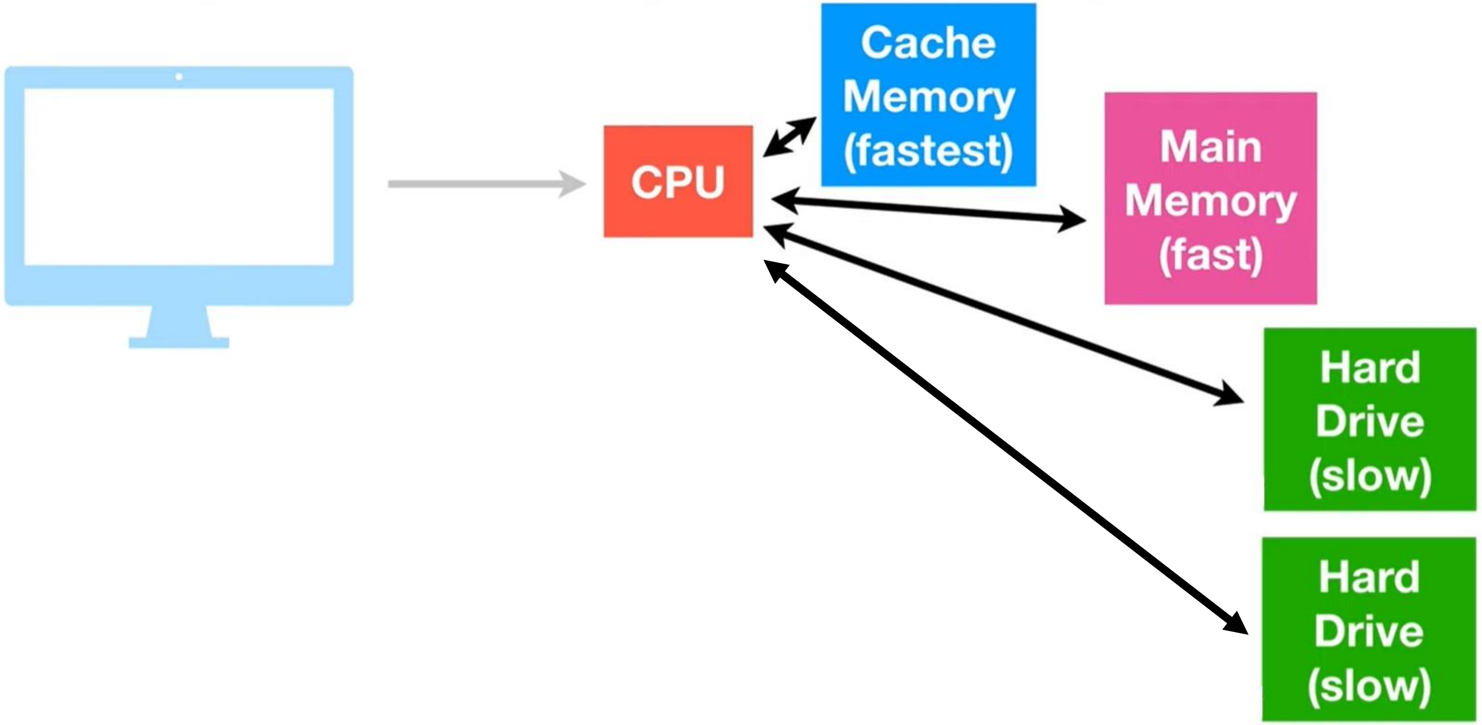
Because reading and writing data to the **Hard Drive** is super slow, **XGBoost** tries to minimize these actions by compressing the data.

Blocks for Out-of-Core Computation



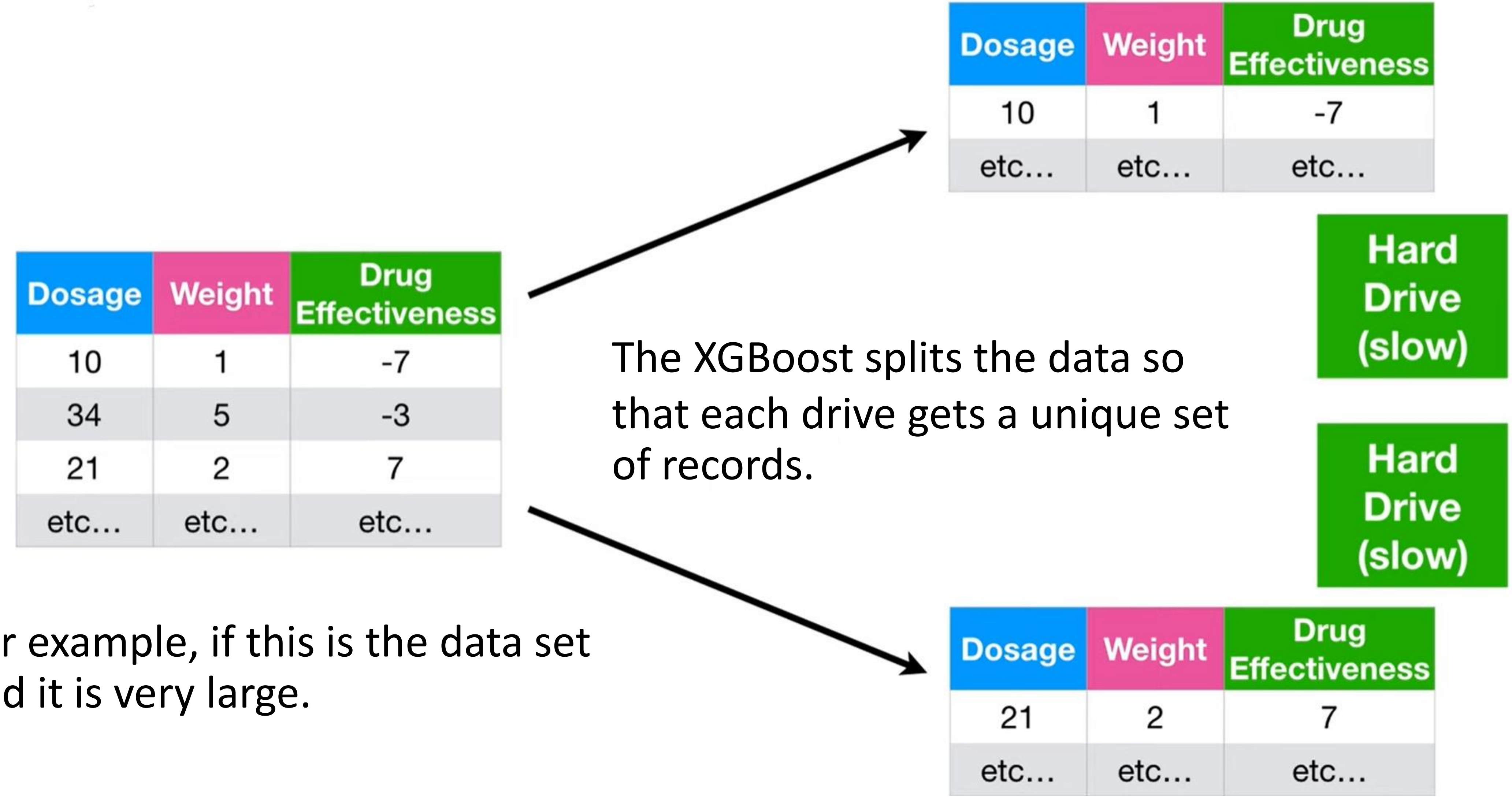
Even though the **CPU** must spend some time decompressing the data that comes from the **Hard Drive**, it can do this faster than the **Hard Drive** can read the data.

Blocks for Out-of-Core Computation



Also, when there is more than one **Hard Drive** available for storage, **XGBoost** uses a database technique called **Sharding** to speed up disk access.

Blocks for Out-of-Core Computation



Dosage	Weight	Drug Effectiveness
10	1	-7
34	5	-3
21	2	7
etc…	etc…	etc…

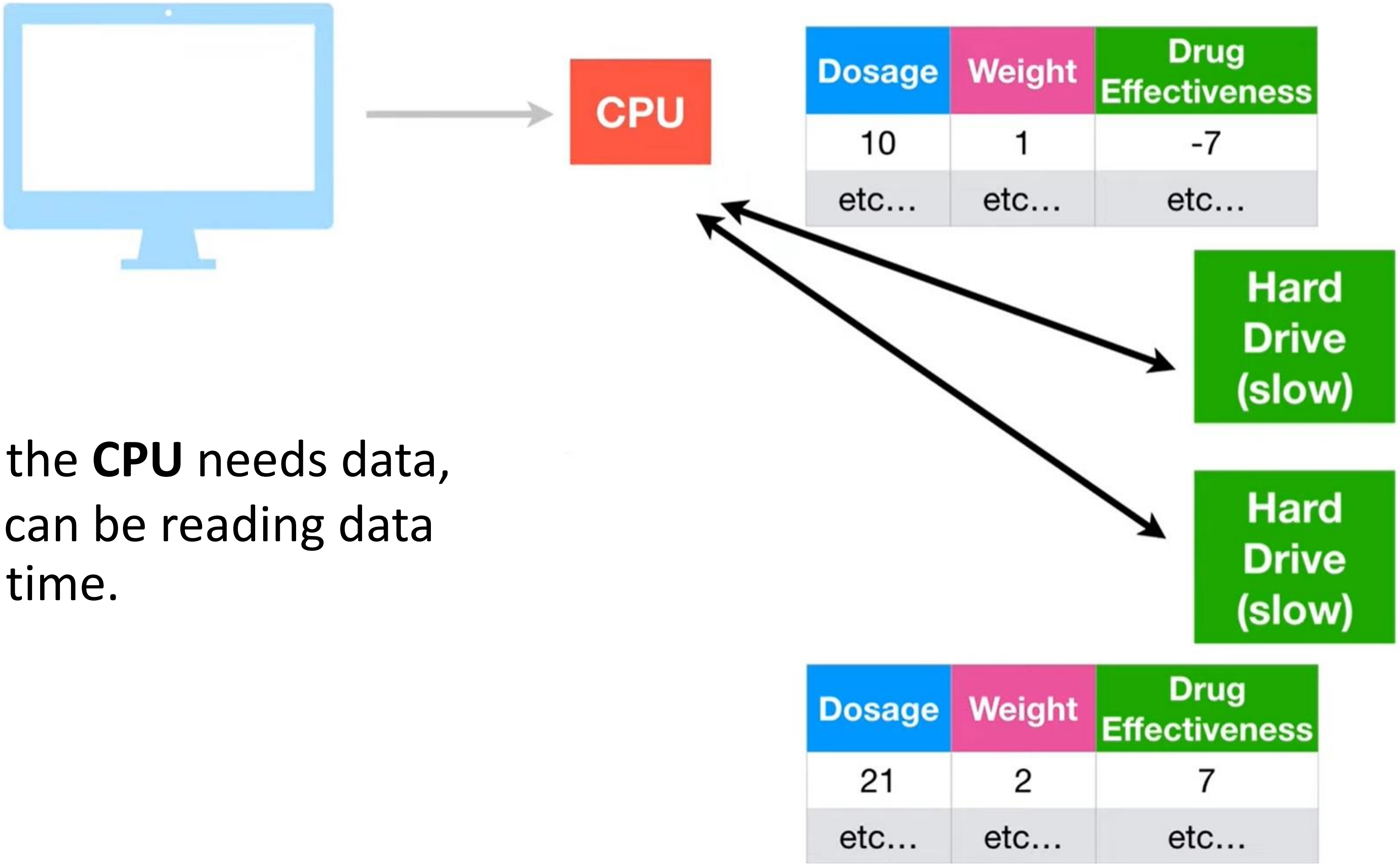
For example, if this is the data set and it is very large.

Dosage	Weight	Drug Effectiveness
10	1	-7
etc…	etc…	etc…

Dosage	Weight	Drug Effectiveness
21	2	7
etc…	etc…	etc…

The XGBoost splits the data so that each drive gets a unique set of records.

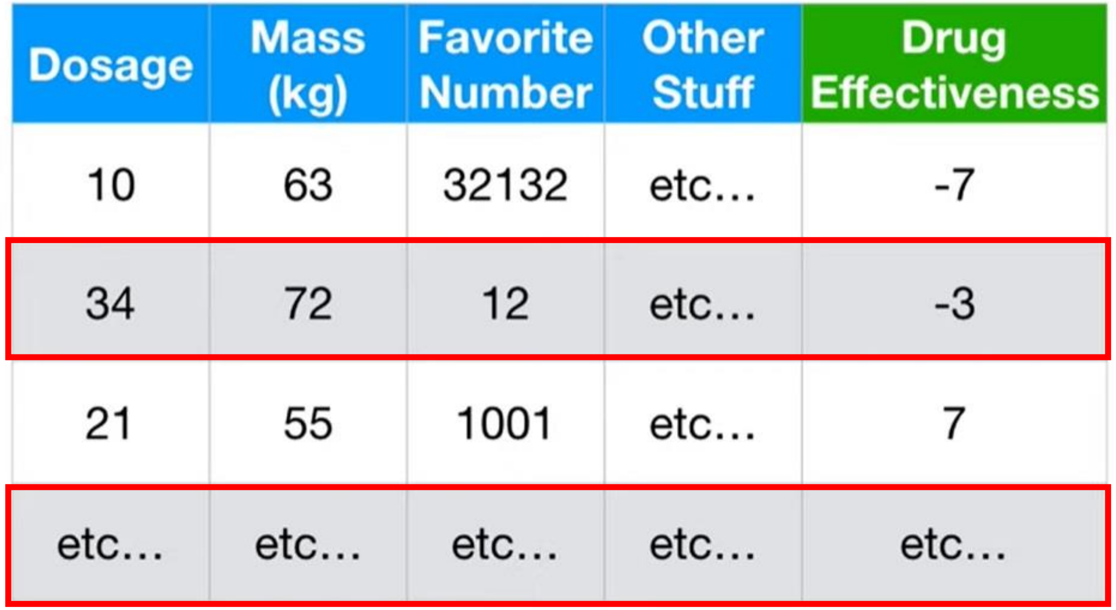
Blocks for Out-of-Core Computation



Dosage	Weight	Drug Effectiveness
10	1	-7
etc...	etc...	etc...

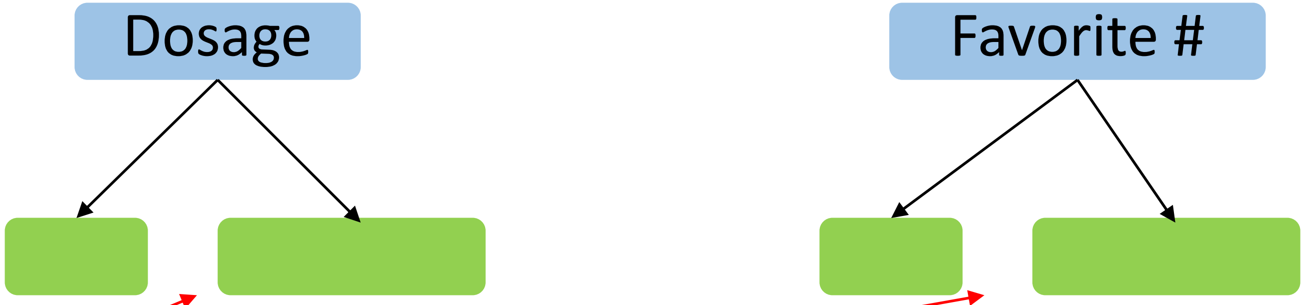
Dosage	Weight	Drug Effectiveness
21	2	7
etc...	etc...	etc...

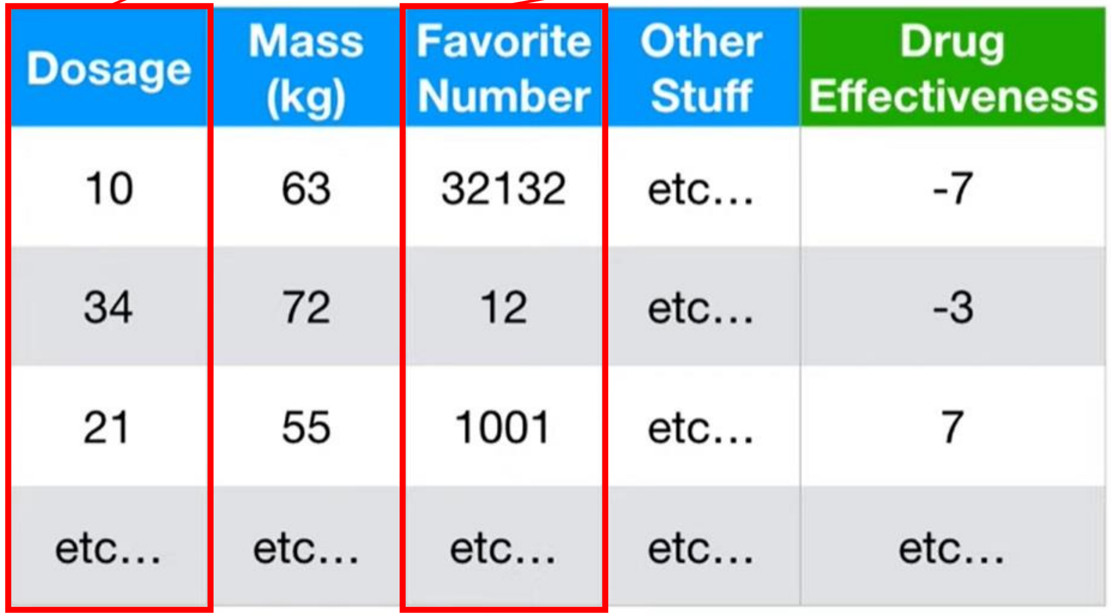
Then, when the **CPU** needs data, both **Drives** can be reading data at the same time.



Dosage	Mass (kg)	Favorite Number	Other Stuff	Drug Effectiveness
10	63	32132	etc…	-7
34	72	12	etc…	-3
21	55	1001	etc…	7
etc…	etc…	etc…	etc…	etc…

XGBoost can also speed things up by allowing you to build each tree with only a random subset of data.





Dosage	Mass (kg)	Favorite Number	Other Stuff	Drug Effectiveness
10	63	32132	etc…	-7
34	72	12	etc…	-3
21	55	1001	etc…	7
etc…	etc…	etc…	etc…	etc…

And **XGBoost** can speed up building trees by only looking at a random subset of features when deciding how to split the data.

Thank you for your attention