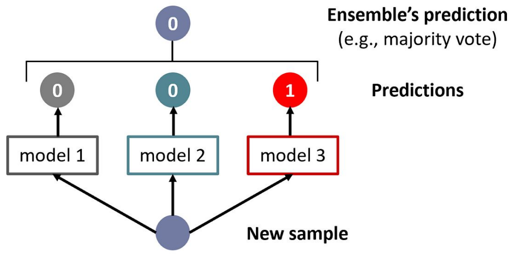
Random Forests & Boosted Forests

THE KNOWLEDGE HOUSE

Agenda - Schedule

1. Bootstrapping & Aggregating (Bagging)
2. Boosting
3. Break
4. Implementations



How do you make decisions?

Agenda - Goals

-

Bagging

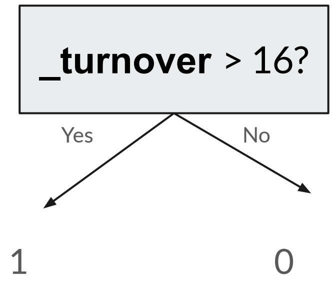
team1_3point	team2_turnover	team1_won
35	15	1
28.5	12	0
40	18	1
25.0	20.0	0

Let's continue with our exploration of college-basketball, except this time, let's also consider the stats of the other team in our dataset.

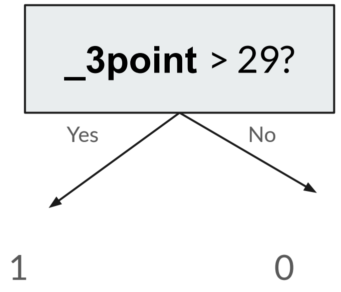
team1_3point is the average percent of games that team1 makes a 3 point
team2_turnover is the average percent of games that team2 achieve a turnover when the ball is in possession of another team

team1_3point	team2_turnover	team1_won
35	15	1
28.5	12	0
40	18	1
25.0	20.0	0

DT if we begin our impurity measure on **team2_turnover**



DT if we begin our impurity measure on **team1_3point**



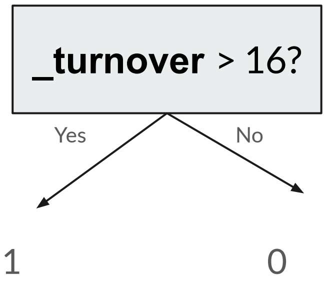
And this is only for our 4-sample dataset. What if we have 100, 1000 or 1,000,000 samples?

Going back to the formulation of our decision trees, our trees will **overfit to this noisy data and fail to generalize on new test samples.**

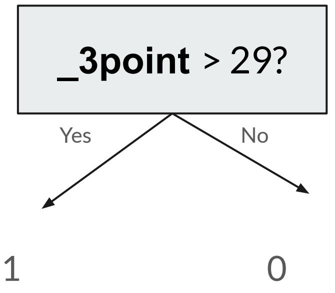
Is this considered high bias or high variance?

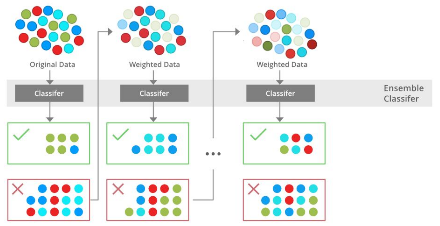
team1_3point	team2_turnover	team1_won
35	15	1
28.5	12	0
40	18	1
25.0	20.0	0

DT if we begin our impurity measure on **team2_turnover**



DT if we begin our impurity measure on **team1_3point**

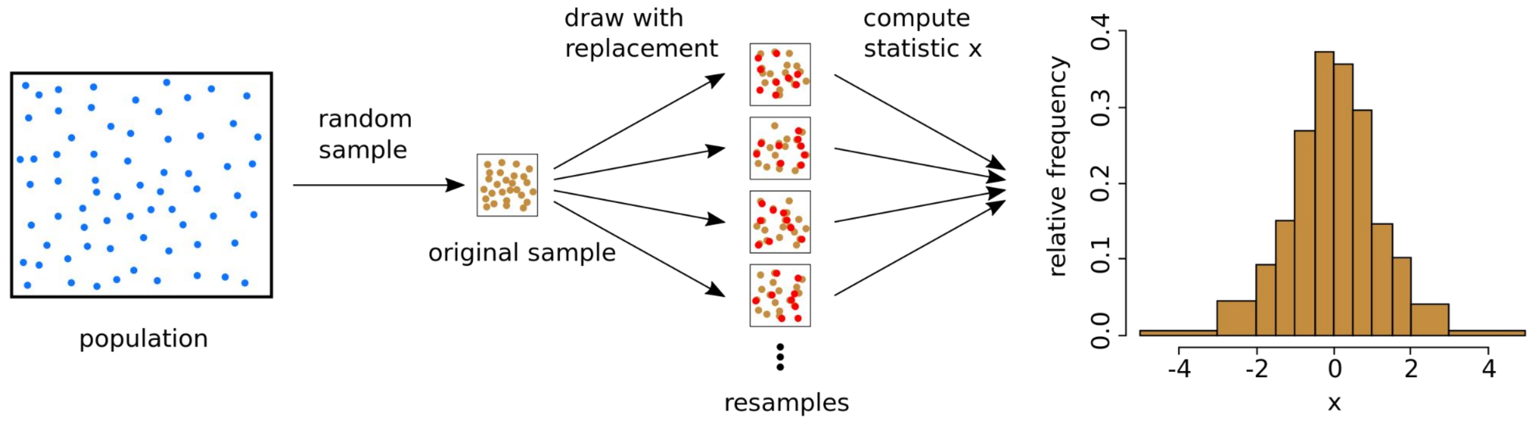




Therefore, we need a method to cut through all this noise and somehow **extract the true signal from this dataset**. The first technique we will learn is **bagging**.

Bagging - Bootstrapping

Bagging is a combination of **bootstrapping + aggregating.** Let's review **bootstrapping**.



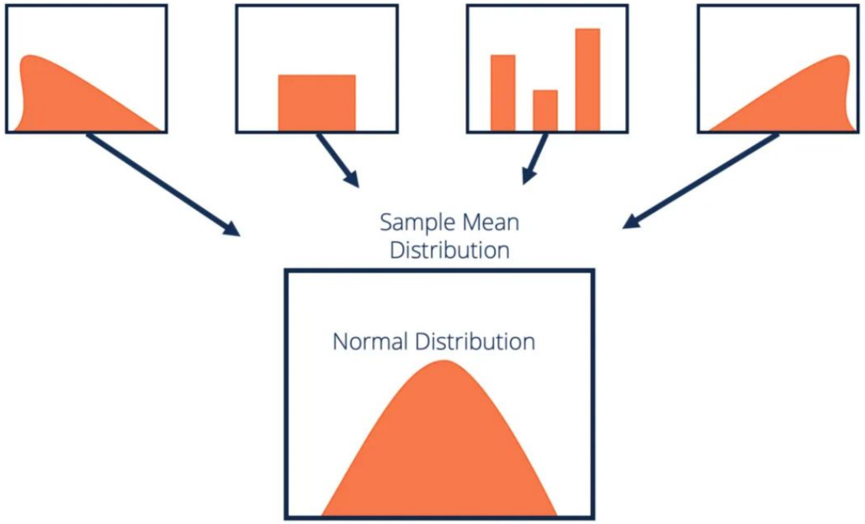
Bagging - Bootstrapping

The bootstrap solves a common problem in data collection: **we have a limited population**,

how do we estimate the true variance of this population given our constraints?

Let’s think back to the **central limit theorem**…

which “action” eventually reduces the variance of a non-normally (high variance) distributed population?

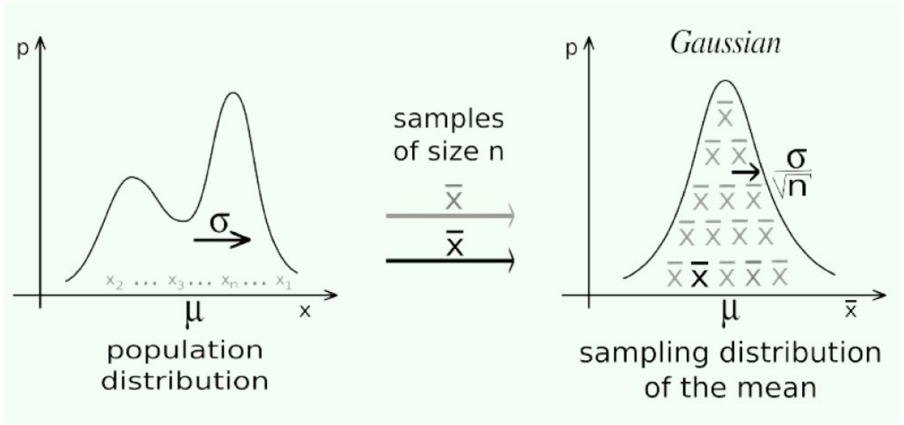


Bagging - Bootstrapping

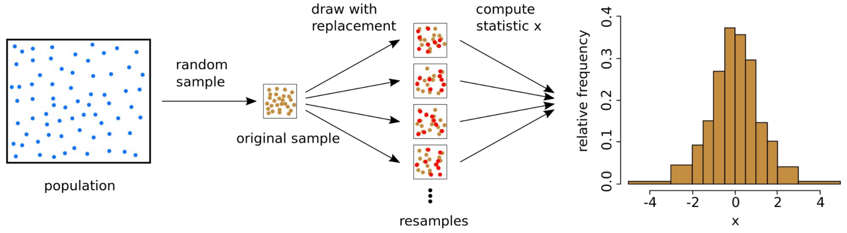
GRABBING SAMPLES AND ESTIMATING THEIR SUMMARY STATISTIC! (mean)

In other words, *averaging a set of observations reduces our variance.*

So, why don't we simply **resample our population (dataset)** with replacement and calculate the average of these samples in order to reduce variance and **therefore create better models.**


p
σ
$x_2 \dots x_3 \dots x_n \dots x_1$
μ
x
population distribution
samples of size n
$\bar{x}$
$\bar{x}$
p
Gaussian
$\frac{\sigma}{\sqrt{n}}$
μ
$\bar{x}$
sampling distribution of the mean

Bagging - Bootstrapping



This describes our “bootstrapping” algorithm:

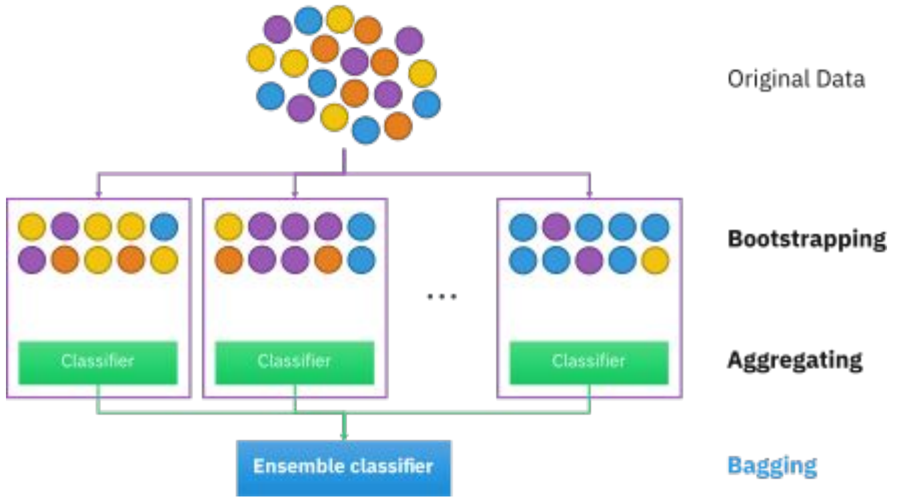
1. Iteratively select **a random observation from your dataset** with **replacement** until you get *n* observations
2. Calculate a summary statistic such as mean on this sample
   a. Add it to the distribution
3. Calculate the variance on this **distribution after a set amount of iterations**

Bagging - Bootstrapping + Aggregating

Let's apply this technique to our decision trees in order to implement what we call "**bagged trees.**"

The algorithm for **bagging (bootstrapping + aggregating)** is applied through the following steps:

1. Select *n* random subsets of datasets with replacement (**bootstrap**)
2. Grow arbitrarily large trees on each subset
3. "*Democratically*" select the prediction which has the most votes across all trees (**aggregate**).



team1_3point	team2_turnover	team1_won
35	15	1
28.5	12	0
40	18	1
25.0	20.0	0
37.5	14.0	1
30.0	16.0	0
42.0	19.5	1
33.5	21.0	0

Let’s bring back our basketball dataset to see how we can apply **bagging** and strengthen the true signal of our dataset. We include a few more data points to make this a little more interesting.

team1_3point	team2_turnover	team1_won
35	15	1
28.5	12	0
40	18	1
25.0	20.0	0
37.5	14.0	1
30.0	16.0	0
42.0	19.5	1
33.5	21.0	0

28.5	12	0
37.5	14.0	1
30.0	16.0	0

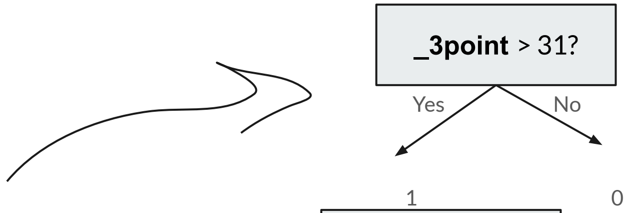
35	15	1
42.0	19.5	1
28.5	12	0

40	18	1
33.5	21.0	0
25.0	20.0	0

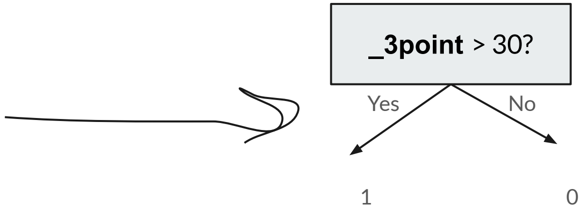
28.5	12	0
42.0	19.5	1
28.5	12	0

First, we will make 4 random subsets of data (with replacement). This is the **bootstrap** step.

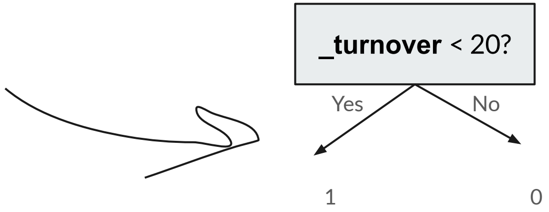
28.5	12	0
37.5	14.0	1
30.0	16.0	0



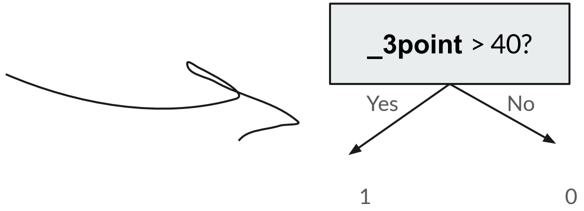
35	15	1
42.0	19.5	1
28.5	12	0



40	18	1
33.5	21.0	0
25.0	20.0	0

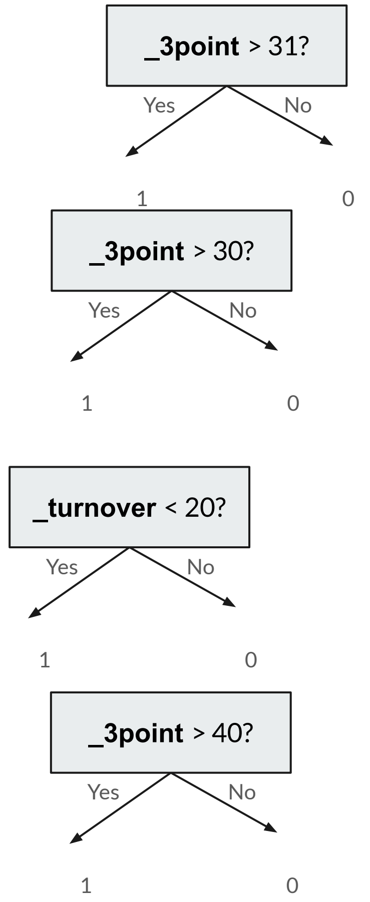


28.5	12	0
42.0	19.5	1
28.5	12	0



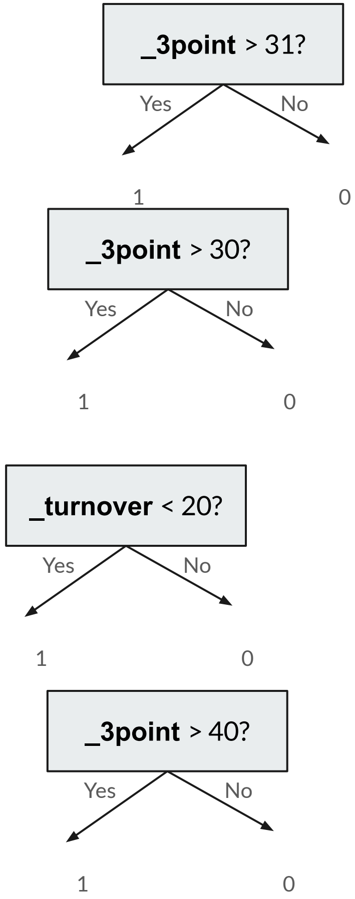
Note: keep in mind that this example is too simple to make any meaningful tree

Next, we will grow trees on each subset of data. Notice that individual trees might form different nodes due to the randomness of tree generation.



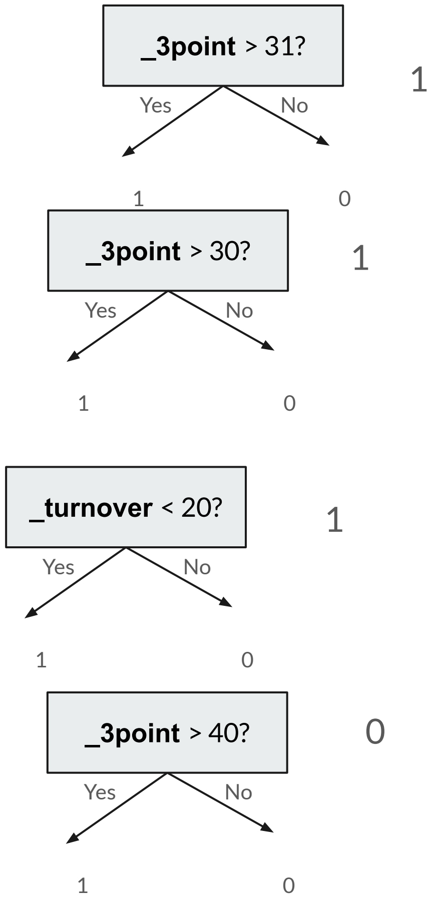
team1_3point	team2_turnover	team1_won_pred
35	15	???
28.5	12	???
40	18	???
25.0	20.0	???
37.5	14.0	???
30.0	16.0	???
42.0	19.5	???
33.5	21.0	???

Finally, we aggregate predictions for each sample using our generated bootstrapped trees. **Let's walk through a few samples.**



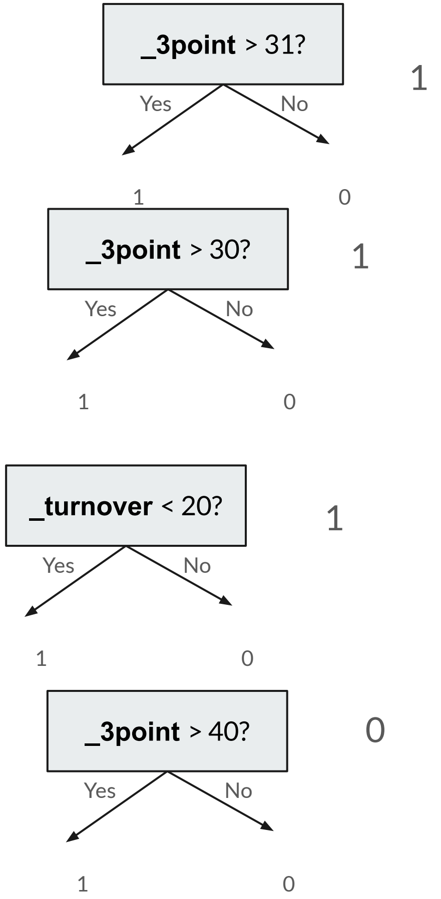
team1_3point	team2_turnover	team1_won_pred
35	15	???
28.5	12	???
40	18	???
25.0	20.0	???
37.5	14.0	???
30.0	16.0	???
42.0	19.5	???
33.5	21.0	???

For sample 1, what will the majority of trees classify this sample as? We observe a **team1_3point of 35**, and a **team2_turnover of 15**.



team1_3point	team2_turnover	team1_won_pred
35	15	???
28.5	12	???
40	18	???
25.0	20.0	???
37.5	14.0	???
30.0	16.0	???
42.0	19.5	???
33.5	21.0	???

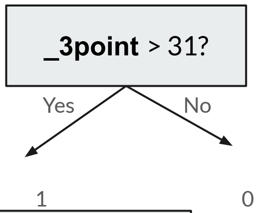
Much like a democratic election system, we will ask each tree of their prediction.
What do most trees "*believe*" about this sample?

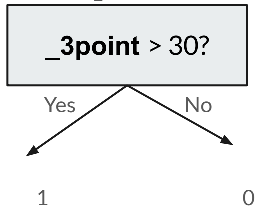


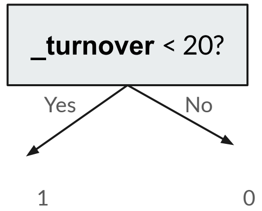
team1_3point	team2_turnover	team1_won_pred
35	15	**1**
28.5	12	???
40	18	???
25.0	20.0	???
37.5	14.0	???
30.0	16.0	???
42.0	19.5	???
33.5	21.0	???

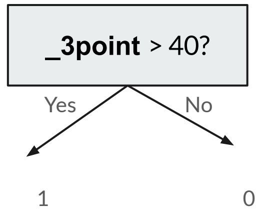
One question that you might be thinking is "wait so now we've introduced a parameter' to indicate how many trees we will train, is this another hyperparameter that we must find via GridSearch?"

Seems like most trees classify this as a win.









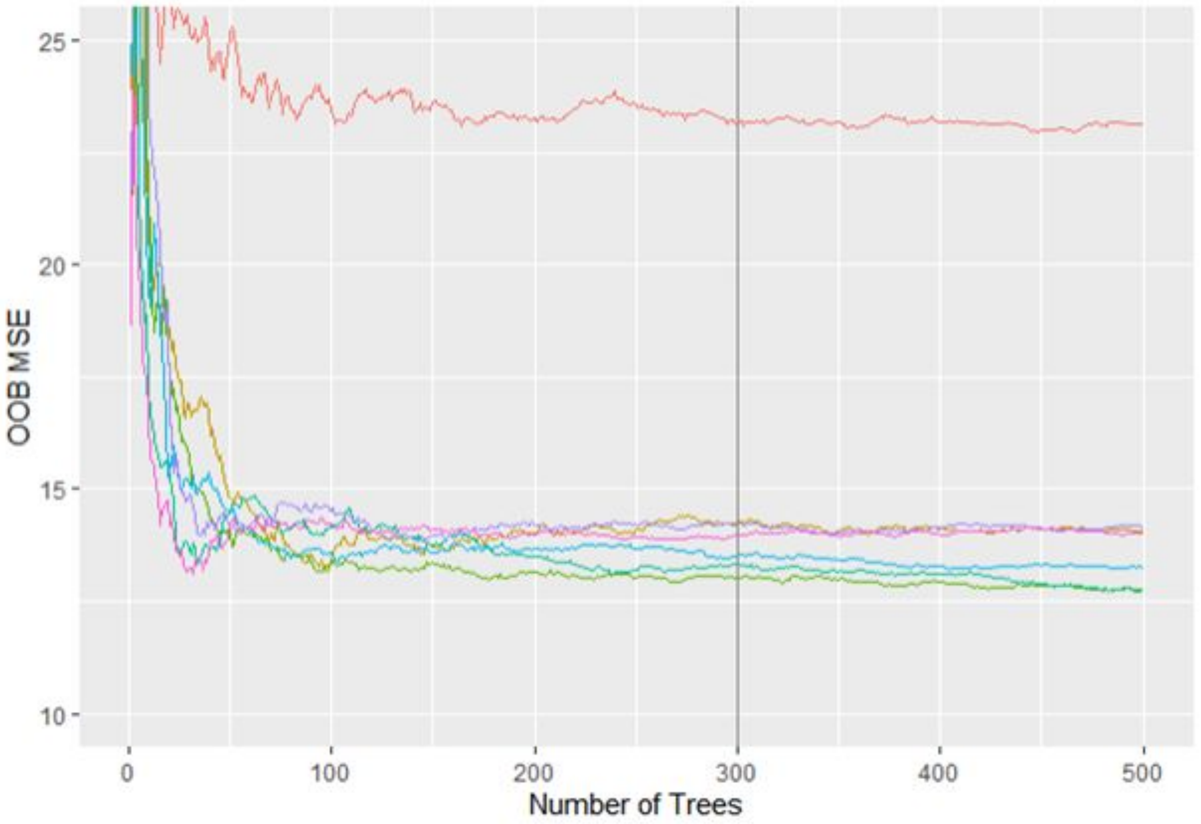
team1_3point	team2_turnover	team1_won_pred	team1_won
35	15	**1**	1
28.5	12	**0**	0
40	18	**1**	1
25.0	20.0	**0**	0
37.5	14.0	**1**	1
30.0	16.0	**0**	0
42.0	19.5	**1**	1
33.5	21.0	**0 or 1**	0

tie

Worst Case Accuracy = 85%

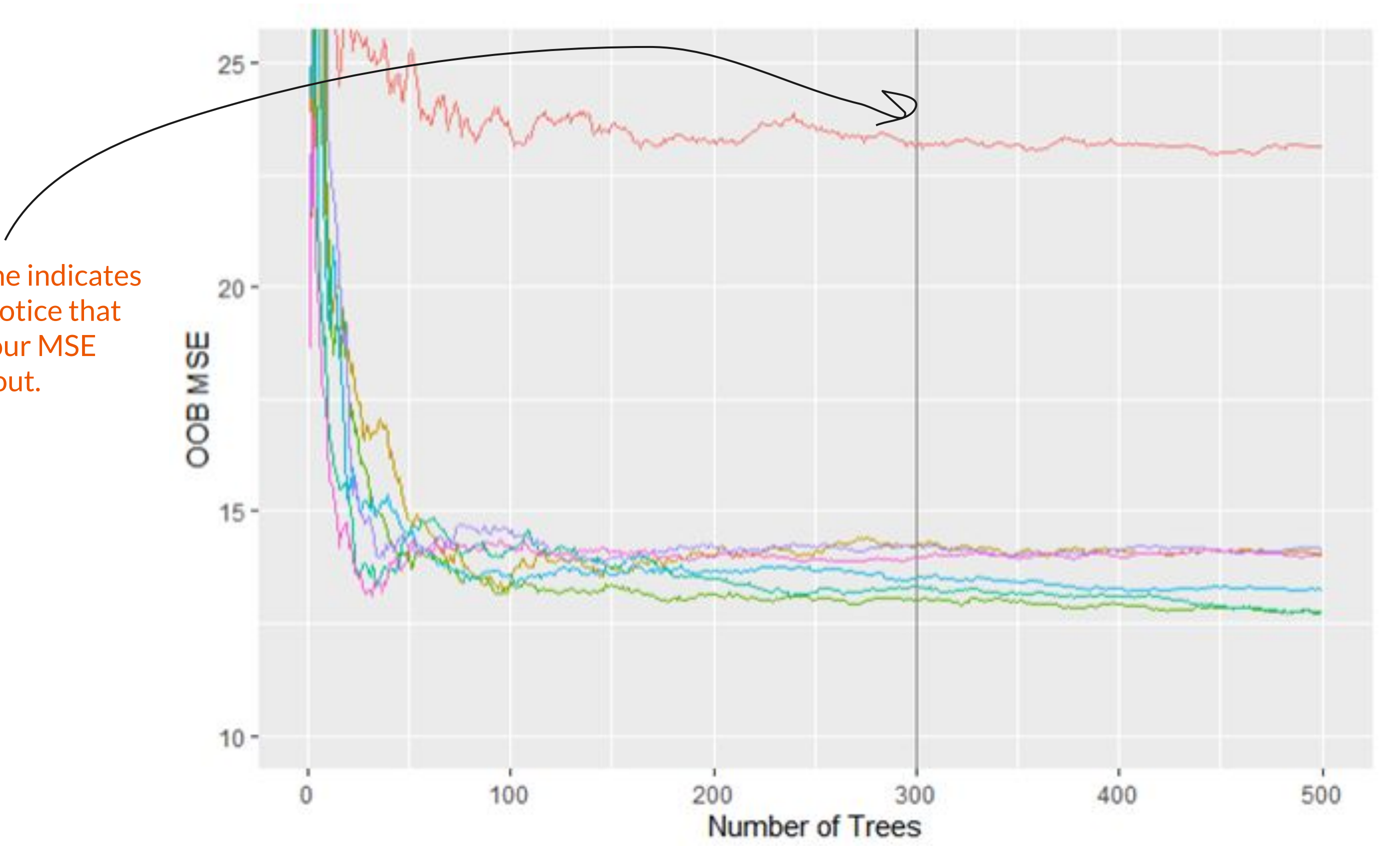
Best Case Accuracy = 100%

Just for fun, let's fill out the rest of these samples and observe the accuracy. Notice that we occasionally get **ties.** In this case, we decide randomly which class to choose which leads us to an accuracy that ranges from **85% to 100%**



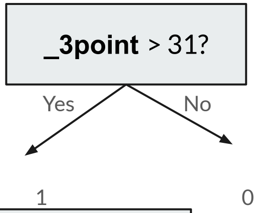
As it turns out, the number of trees that we train ***"B"*** does not suffer from overfitting (high variance) as we increase this amount! Instead, error simply *settles* after enough trees are generated. **WOW!**

The read line indicates test MSE, notice that after 300, our MSE "plateaus" out.

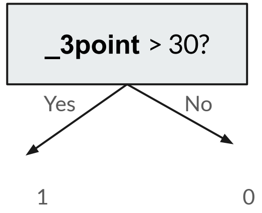


Therefore, we simply use a **sufficiently large B** until we do not see large gains in test accuracy.

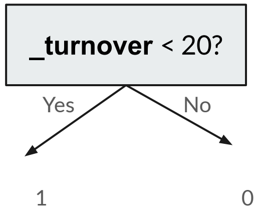
One technique we can use is to start large (100), and then see how many trees we can **remove for the same test accuracy rate.**



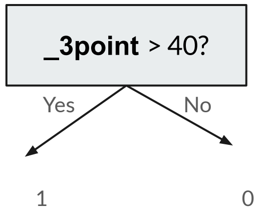
Impurity Removed = 0.875



Impurity Removed = 0.875

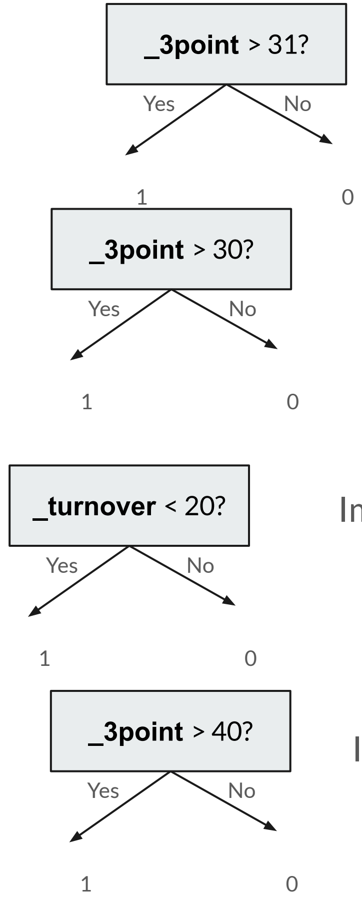


Impurity Removed = 0.33



Impurity Removed = 1

This is not the only utility of bagged trees. In addition, we can measure **the amount of impurity removed** for each predictor across all decision trees. This will show us a vital statistic called **variable importance.**

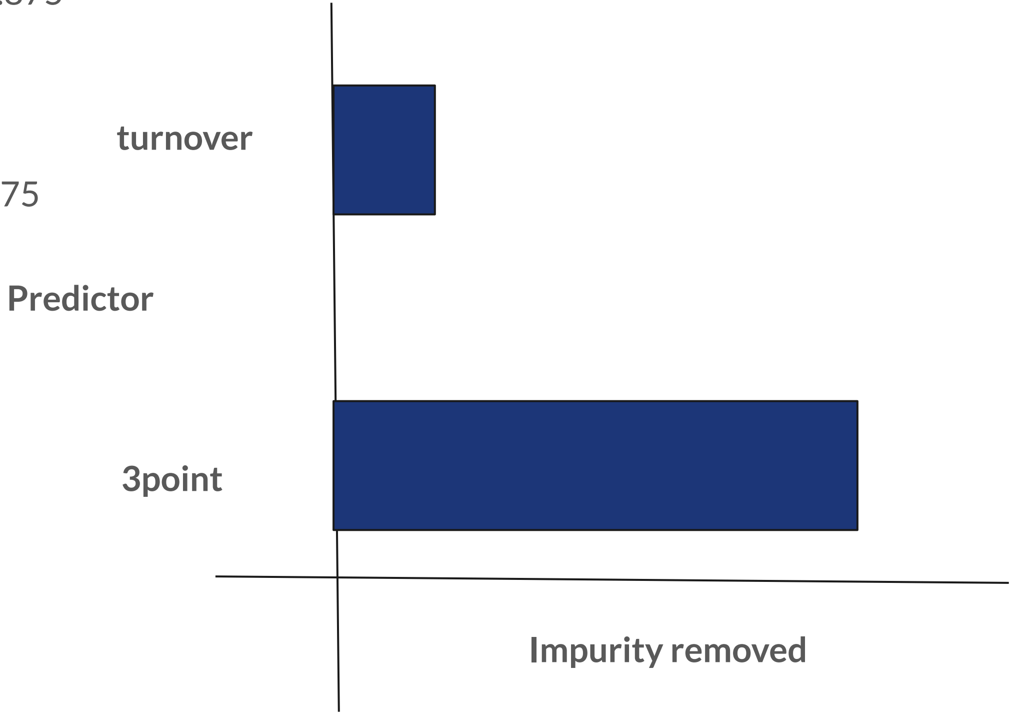


Impurity Removed = 0.875

Impurity Removed = 0.875

Impurity Removed = 0.33

Impurity Removed = 1



By adding all the impurity removed, we can observe which predictor is the **most important**. In this dataset, which feature seems to remove the most impurity?

*Keep in mind that bagging is applicable to many machine learning models, not just decision trees.

Bagged Trees

This might seem like a completely separate machine learning algorithm itself, however, there is one slight problem:

*As we consider all predictors in our random subsets, the strongest (most reliable) predictor will be preferred for all trees. This will result in **highly correlated*** predictions across decision trees, *i.e. prediction will still exhibit high variance.*

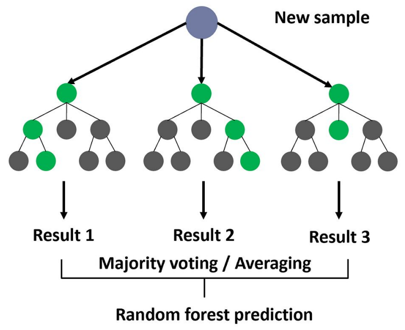
To fix this, we introduce **random forests**.

Random Forests

Random Forests

The random forest algorithm is an improvement over bagging, we simply specify a select number of predictors to use:

1. Select *n* random subsets of datasets with replacement (**bootstrap**)
2. Grow arbitrarily large trees on each subset using a subset of ***p predictors (commonly the square root of p)***
3. "*Democratically*" select the prediction which has the most votes across all trees (**aggregate**).



team1_3point	team2_turnover	team1_won
35	15	1
28.5	12	0
40	18	1
25.0	20.0	0
37.5	14.0	1
30.0	16.0	0
42.0	19.5	1
33.5	21.0	0

Let’s go back to our basketball example once more, and walk through the steps of bagging with this additional caveat

team1_3point	team2_turnover	team1_won
35	15	1
28.5	12	0
40	18	1
25.0	20.0	0
37.5	14.0	1
30.0	16.0	0
42.0	19.5	1
33.5	21.0	0

28.5	12	0
37.5	14.0	1
30.0	16.0	0

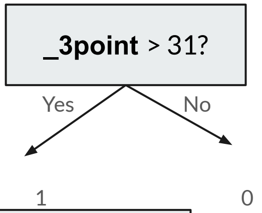
35	15	1
42.0	19.5	1
28.5	12	0

40	18	1
33.5	21.0	0
25.0	20.0	0

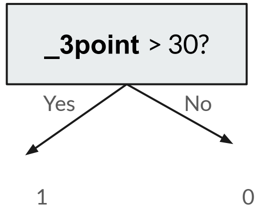
28.5	12	0
42.0	19.5	1
28.5	12	0

First, we will make 4 random subsets of data (with replacement). This is the **bootstrap** step.

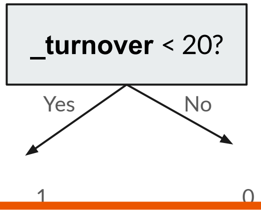
28.5	12	0
37.5	14.0	1
30.0	16.0	0



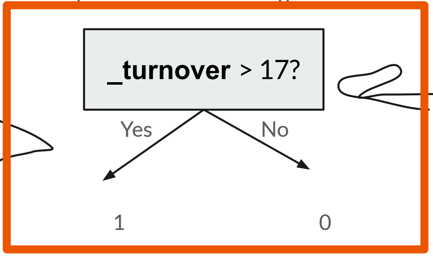
35	15	1
42.0	19.5	1
28.5	12	0



40	18	1
33.5	21.0	0
25.0	20.0	0



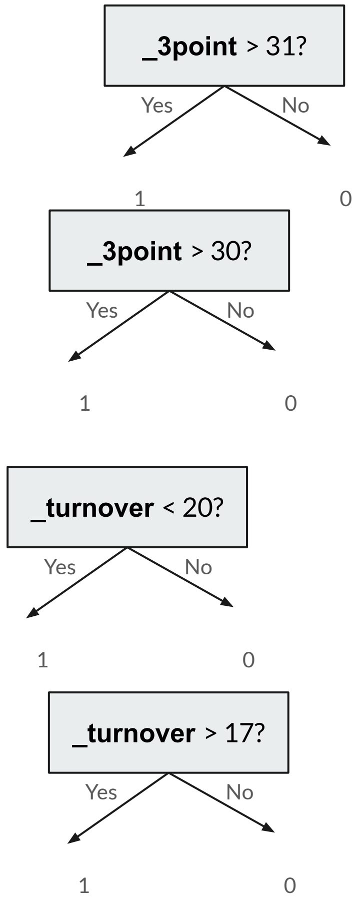
28.5	12	0
42.0	19.5	1
28.5	12	0



Once again, the scope of this dataset leads to uninteresting trees.

We will instead consider that we have an even split of trees considering both "turnover" and "3point", each candidate predictor gets a fair shot at proving its worth.

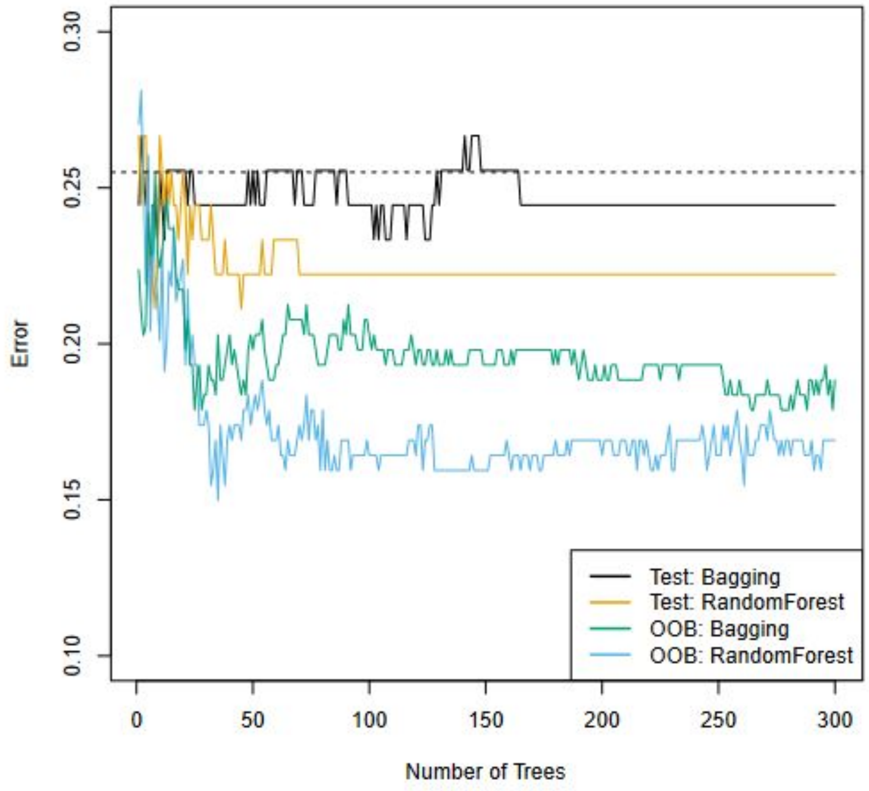
Next, we will grow trees on each subset of data, however this time we *decorrelate* our trees by limiting ourselves to only 1 select predictor over this training set.



team1_3point	team2_turnover	team1_won_pred	team1_won
35	15	**1**	1
28.5	12	**0**	0
40	18	**1**	1
25.0	20.0	**0**	0
37.5	14.0	**1**	1
30.0	16.0	**0**	0
42.0	19.5	**1**	1
33.5	21.0	**1**	0

Accuracy = 85%

Once again, we allow these trees to democratically elect the prediction based on their **bootstrapped samples and limited predictors.** **Notice that we get no ties! We walk away with one accuracy!**



This outcome is not unique to our dataset. Notice the **Test:RandomForest** error rate performs noticeably **better than the simple bagged model**.

Random Forests

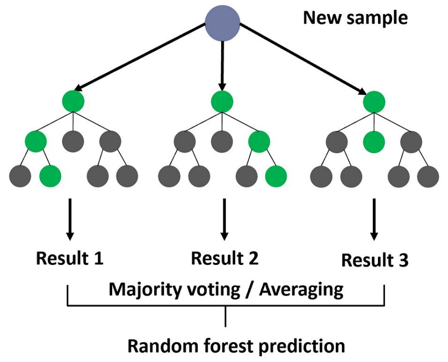
Random Forests is a **non-parametric multiclass supervised learning algorithm** **that learns optimal splits by creating multiple bagged DT's on a subset of predictors.**

Pros

- Used for both **classification & regression**
- Implicit **feature selection**
- Model **non-linear decision boundaries**
- **Highly robust** against outliers & dimensionality
- No need to **normalize data** (most of the time)

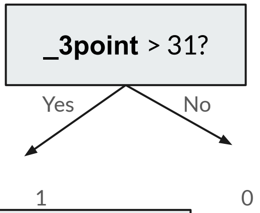
Cons

- For regression tasks, **predictions are limited to range of target**
- Not **easily** interpretable
- Computationally **expensive**

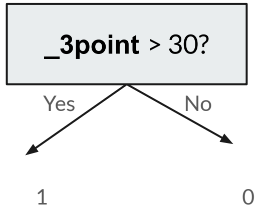


Boosting - AdaBoost

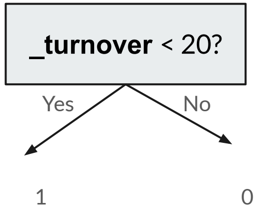
28.5	12	0
37.5	14.0	1
30.0	16.0	0



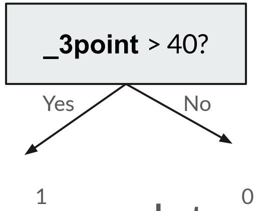
35	15	1
42.0	19.5	1
28.5	12	0



40	18	1
33.5	21.0	0
25.0	20.0	0



28.5	12	0
42.0	19.5	1
28.5	12	0



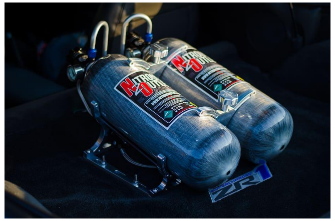
Trees with better information on which mistakes they should avoid should also be better classifiers!

Metaphors get tough in this context but consider the following:

You, a New Yorker/Connecticuter, are voting for public transportation policies in your city. Might you have better context than an extra-state voter?

In random forests/bagged trees each tree gets 1 vote. This presents an opportunity, what if we can determine which trees are **better predictors** and grant them greater leverage within the classification process.

Boosting



Boosting was (still is?) one of the best ways to win ML competitions.

This leads us to another **ensemble technique** called **boosting.**

Instead of creating **multiple bagged trees**, we grow trees **sequentially** using **errors** from the previously grown trees.

In other words, we *slowly* grow by **iteratively** considering gains in gini impurity and by **correcting errors** of previous trees.

In essence, we learn from the mistakes of previous trees to **grow better trees, while still keeping previous iterations**!

Boosting - AdaBoost

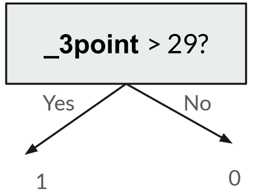
Let’s first consider **ada**ptive **boost**ing (**adaboost**).

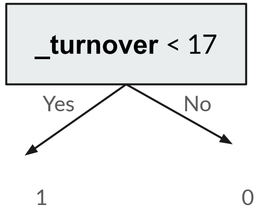
1. Grow weak decision trees (aka **stumps**) on your dataset.
2. Loop until **n trees** are grown:
   a. Calculate which stump is the **best classifier**, adjust your dataset based on the errors this stump made.
   b. Grow **another set of stumps** on your dataset using this adjusted dataset.
3. Utilize this set of stumps to make classifications by **voting considering a stumps importance**.

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	1/8
28.5	12	0	1/8
40	18	1	1/8
25.0	20.0	0	1/8
37.5	14.0	1	1/8
30.0	16.0	0	1/8
42.0	19.5	1	1/8
33.5	21.0	0	1/8

Again, let's consider our **march madness basketball dataset.** The first thing we do is assign **weights to each row**. This will simply be the frequency of this sample (1/size of dataset) at first, but we will update this value.

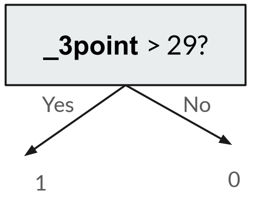
team1_3point	team2_turnover	team1_won	sample weight
35	15	1	1/8
28.5	12	0	1/8
40	18	1	1/8
25.0	20.0	0	1/8
37.5	14.0	1	1/8
30.0	16.0	0	1/8
42.0	19.5	1	1/8
33.5	21.0	0	1/8





Just like with random forests, we will grow **stumps** on a subset of predictors (one tree per predictor). However this time we use the **entire dataset.**

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	1/8
28.5	12	0	1/8
40	18	1	1/8
25.0	20.0	0	1/8
37.5	14.0	1	1/8
30.0	16.0	0	1/8
42.0	19.5	1	1/8
33.5	21.0	0	1/8



True pos = 4

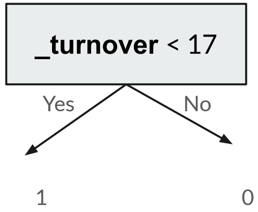
False pos = 2

False neg = 0

True neg = 2

Gini = 1 - ((4/6)^2+(2/6)^2)

Gini = 1 - ((0/2)^2+(2/2)^2)



True pos = 2

False pos = 2

False neg = 2

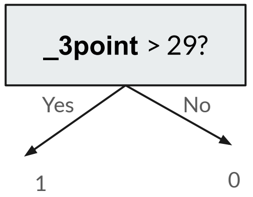
True neg = 2

Gini = 1 - ((2/4)^2+(2/4)^2)

Gini = 1 - ((2/4)^2+(2/4)^2)

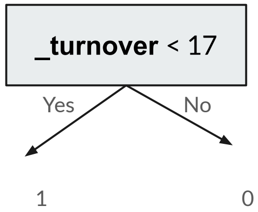
We then calculate the **gini index** for each stump and select the one with the lowest impurity.

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	1/8
28.5	12	0	1/8
40	18	1	1/8
25.0	20.0	0	1/8
37.5	14.0	1	1/8
30.0	16.0	0	1/8
42.0	19.5	1	1/8
33.5	21.0	0	1/8



True pos = 4 | False pos = 2 | False neg = 0 | True neg = 2

Gini = 0.44 | Gini = 0

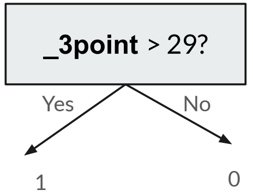


True pos = 2 | False pos = 2 | False neg = 2 | True neg = 2

Gini = 0.5 | Gini = 0.5

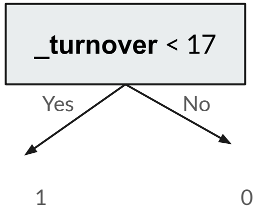
We then calculate the **gini index** for each stump and select the one with the lowest impurity.

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	1/8
28.5	12	0	1/8
40	18	1	1/8
25.0	20.0	0	1/8
37.5	14.0	1	1/8
30.0	16.0	0	1/8
42.0	19.5	1	1/8
33.5	21.0	0	1/8



True pos = 4 | False pos = 2 | False neg = 0 | True neg = 2

Total Gini = 0.44 *(6/8) + 0 * (2/8)

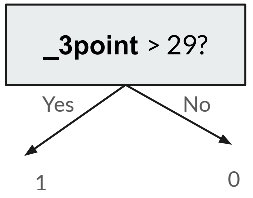


True pos = 2 | False pos = 2 | False neg = 2 | True neg = 2

Total Gini = 0.5 *(4/8) + 0.5 *(4/8)

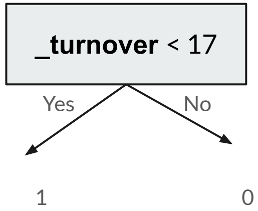
We then calculate the **gini index** for each stump and select the one with the lowest impurity.

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	1/8
28.5	12	0	1/8
40	18	1	1/8
25.0	20.0	0	1/8
37.5	14.0	1	1/8
30.0	16.0	0	1/8
42.0	19.5	1	1/8
33.5	21.0	0	1/8



True pos = 4 | False pos = 2 | False neg = 0 | True neg = 2

Total Gini = 0.33

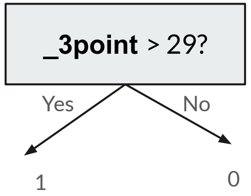


True pos = 2 | False pos = 2 | False neg = 2 | True neg = 2

Total Gini = 0.5

As this **_3point** stump has the lowest impurity, we select it as the winner tree of this round. We preserve it and use it to create a better tree in the next round.

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	1/8
28.5	12	0	1/8
40	18	1	1/8
25.0	20.0	0	1/8
37.5	14.0	1	1/8
30.0	16.0	0	1/8
42.0	19.5	1	1/8
33.5	21.0	0	1/8



True pos = 4

False pos = 2

False neg = 0

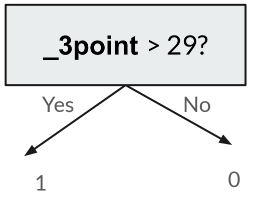
True neg = 2

Total Error = ⅛ + ⅛ = **0.25**

0 -> Perfect stump
1 -> Bad stump

We then calculate how **important** this stump is by summing the weights of all samples it incorrectly classified. This stump classified the following rows incorrectly as **false positives**.

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	1/8
28.5	12	0	1/8
40	18	1	1/8
25.0	20.0	0	1/8
37.5	14.0	1	1/8
30.0	16.0	0	1/8
42.0	19.5	1	1/8
33.5	21.0	0	1/8



True pos	False pos	False neg	True neg
= 4	= 2	= 0	= 2

Total Error = ⅛ + ⅛ = **0.25**

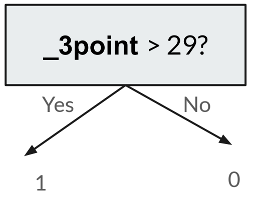
Importance = (½) * log((1-error)/error)

= (½) * log((1-0.25)/0.25)
= (½) * log(0.75/0.25)
~= **0.24**

We then plug this error into an importance metric. The importance of this stump is 0.24.

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	1/8
28.5	12	0	1/8
40	18	1	1/8
25.0	20.0	0	1/8
37.5	14.0	1	1/8
30.0	16.0	0	1/8
42.0	19.5	1	1/8
33.5	21.0	0	1/8

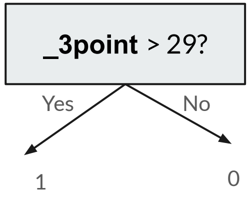
0.24



Next we will use this **stump** to update our sample weights which we will then use to grow another stump! We will use these update weights to direct our forest where to grow (which errors to minimize)! Double Bam (StatQuest Trademarked)

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	1/8
28.5	12	0	1/8
40	18	1	1/8
25.0	20.0	0	1/8
37.5	14.0	1	1/8
30.0	16.0	0	1/8
42.0	19.5	1	1/8
33.5	21.0	0	1/8

0.24



New Sample Weight of Erroneous Samples

(old weight) * e^(importance)

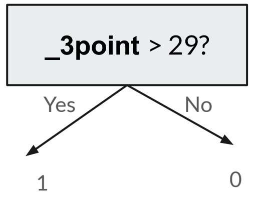
New Sample Weight of Correct Samples

(old weight) * e^(-importance)

We update both types of samples using this predefined formula. **What was the importance measure of our previous stump ?**

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	0.10
28.5	12	0	0.10
40	18	1	0.10
25.0	20.0	0	0.10
37.5	14.0	1	0.10
30.0	16.0	0	0.16
42.0	19.5	1	0.10
33.5	21.0	0	0.16

0.24



New Sample Weight of Erroneous Samples

(1/8) * e^(0.24) = **0.16**

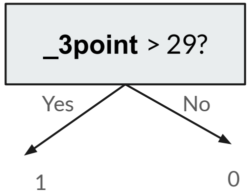
New Sample Weight of Correct Samples

(1/8) * e^(-0.24) = **0.10**

This equation is applied to each row, therefore the old weight is simply the former weight of each row (i.e. ⅛). We update each row sample weight respectively.

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	0.1086
28.5	12	0	0.1086
40	18	1	0.1086
25.0	20.0	0	0.1086
37.5	14.0	1	0.1086
30.0	16.0	0	0.1739
42.0	19.5	1	0.1086
33.5	21.0	0	0.1739

0.24

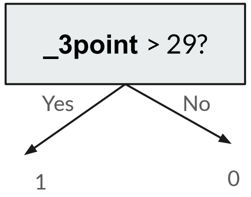


Each row is divided by **0.92** since this is the current sum of the sample weight column.

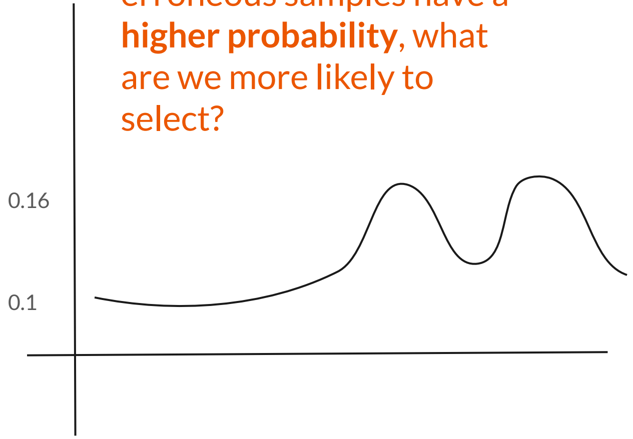
We must guarantee that these new sample weights add up to 1 since we will be treating them as a probability distribution. Therefore we **normalize** these weights.

team1_3point	team2_turnover	team1_won	sample weight
35	15	1	0.1086
28.5	12	0	0.1086
40	18	1	0.1086
25.0	20.0	0	0.1086
37.5	14.0	1	0.1086
30.0	16.0	0	0.1739
42.0	19.5	1	0.1086
33.5	21.0	0	0.1739

0.24



Considering the fact that erroneous samples have a **higher probability**, what are we more likely to select?

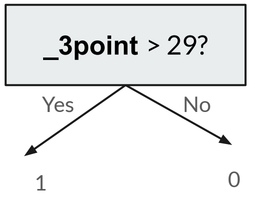


From here, we treat these **weights** as a **probability distribution of values**. We use this distribution to randomly select samples using a random number generator to build another dataset.

team1_3point	team2_turnover
35	15
28.5	12
40	18
25.0	20.0
37.5	14.0
30.0	16.0
42.0	19.5
33.5	21.0

team1_3point	team2_turnover
33.5	21.0
28.5	12
30.0	16.0
33.5	21.0
28.5	12
40	18
30.0	16.0
25.0	20.0

0.24

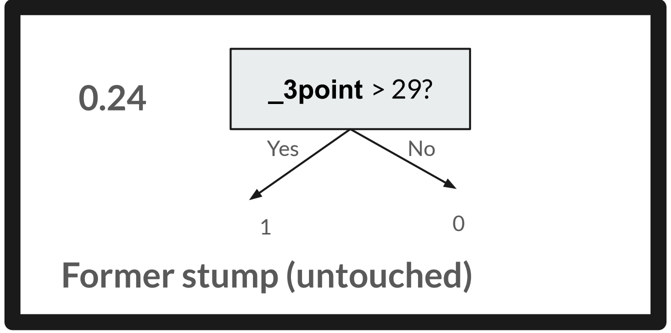


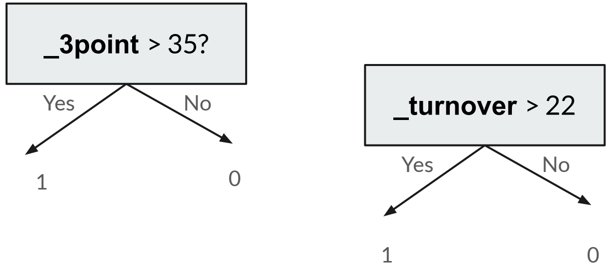
Consider what just happened! Our algorithm just figured out which **samples our trees need to look out for.**

This will allow us to grow better trees!

Since our erroneous samples have higher likelihood of selection, we're more likely to have duplicates of previous errors in our new dataset!

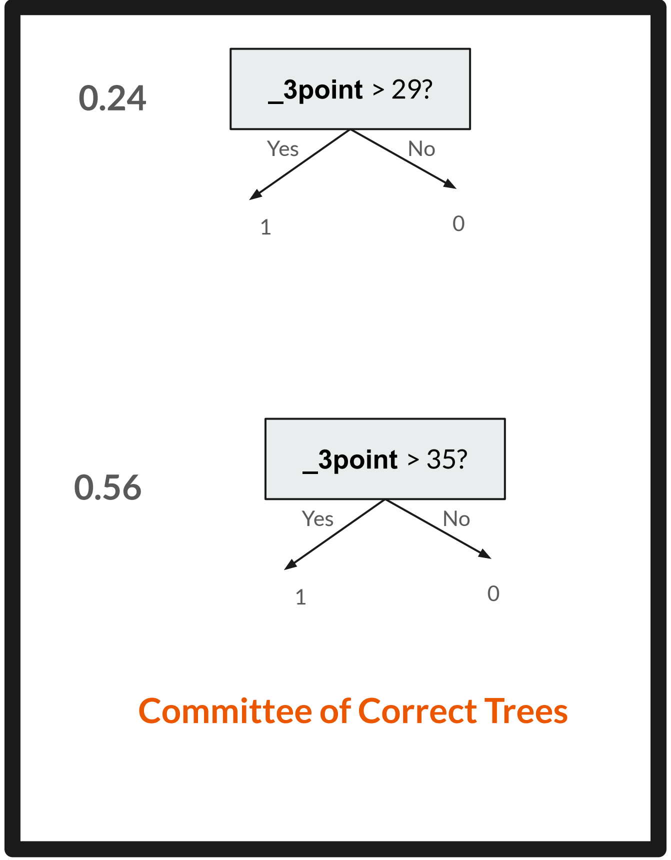
team1_3point	team2_turnover	team1_won	sample weight
33.5	21.0	0	1/8
28.5	12	0	1/8
30.0	16.0	0	1/8
33.5	21.0	0	1/8
28.5	12	0	1/8
40	18	1	1/8
30.0	16.0	0	1/8
25.0	20.0	0	1/8





From here we reset our sample weights and begin the stump growing process anew. However our dataset **now knows which samples to be wary of.**

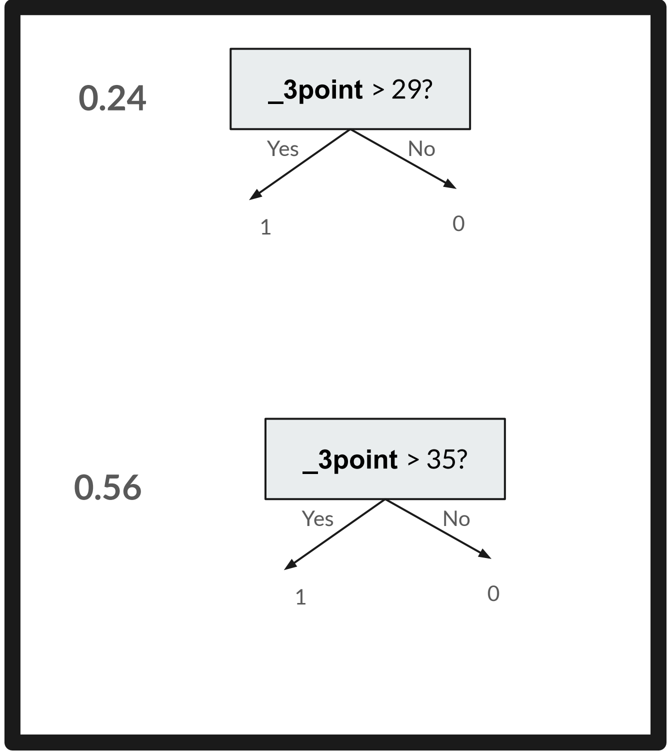
team1_3point	team2_turnover	team1_won	sample weight
33.5	21.0	0	1/8
28.5	12	0	1/8
30.0	16.0	0	1/8
33.5	21.0	0	1/8
28.5	12	0	1/8
40	18	1	1/8
30.0	16.0	0	1/8
25.0	20.0	0	1/8



We then begin this process anew until we have a host of trees, each with their own **importance** metric.

team1_3point	team2_turnover
21	18
36	17
39	15

This is a limited example. You should instead expect to have 10,20, or 30 trees on a dataset **of thousands if not millions of rows.**



We use these importance metrics when deciding which trees to listen to use on unseen data. In this example, which tree has the higher importance metric?

AdaBoost

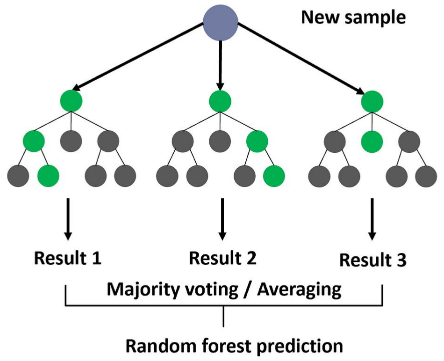
AdaBoost is a **non-parametric multiclass supervised learning algorithm** **that learns optimal splits by iteratively growing "better" trees using the errors of previous trees.**

Pros

- Used for both **classification & regression**
- Implicit **feature selection**
- Model **non-linear decision boundaries**
- **Highly robust** against noisy & dimensional data
- No need to **normalize data** (most of the time)

Cons

- For regression tasks, **predictions are limited to range of target**
- **High number of hyperparameters make training computationally expensive**



Boosting - Gradient Boosting

Gradient Boosting

While we will not dive further into boosted models, one more model we should be aware of is the **gradient boosted** tree.

Much like AdaBoost, we start out with an initial weak classifier.

We then instead calculate the **residuals** (errors) of our tree in order to update our predictions and generate our next tree.

Instead of stumps, we allow our tree to grow deeper.

While we will not explore this algorithm, we invite you to use all that you learned to walk through this tutorial:
https://www.datacamp.com/tutorial/guide-to-the-gradient-boosting-algorithm

End of Class Announcements

Lab (Due 7/23)

Zurich, Switzerland

You are a data scientist working for a Zurich-based international bank called Caishen. The company announced in an all-hands meeting that they are aiming to develop a classification algorithm **that can identify 99% of all fraudulent activity within customer-facing bank accounts.**

For this project, you will use a **dataset of 1 million bank transactions to create a classifier** that will detect if fraudulent activity has occurred for a transaction.

Your second checkpoint will be to complete your EDA (of template files) by **7/16.**

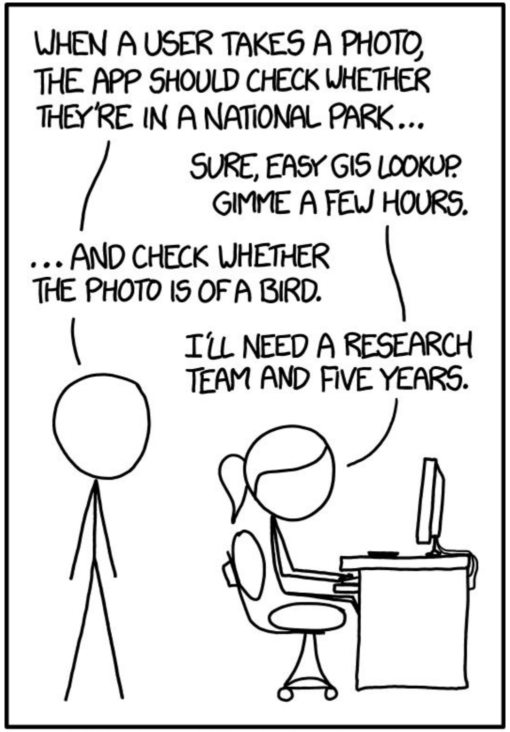
Next Week

Pre-class content for **Week 7** will be due **7/21**.

We will review:

- Linear algebra **again**
- Principal Component Analysis
- Unsupervised machine learning

If you understand what you're doing, you're not learning anything. - Anonymous


WHEN A USER TAKES A PHOTO, THE APP SHOULD CHECK WHETHER THEY'RE IN A NATIONAL PARK...
SURE, EASY GIS LOOKUP. GIMME A FEW HOURS.
...AND CHECK WHETHER THE PHOTO IS OF A BIRD.
I'LL NEED A RESEARCH TEAM AND FIVE YEARS.

IN CS, IT CAN BE HARD TO EXPLAIN THE DIFFERENCE BETWEEN THE EASY AND THE VIRTUALLY IMPOSSIBLE.