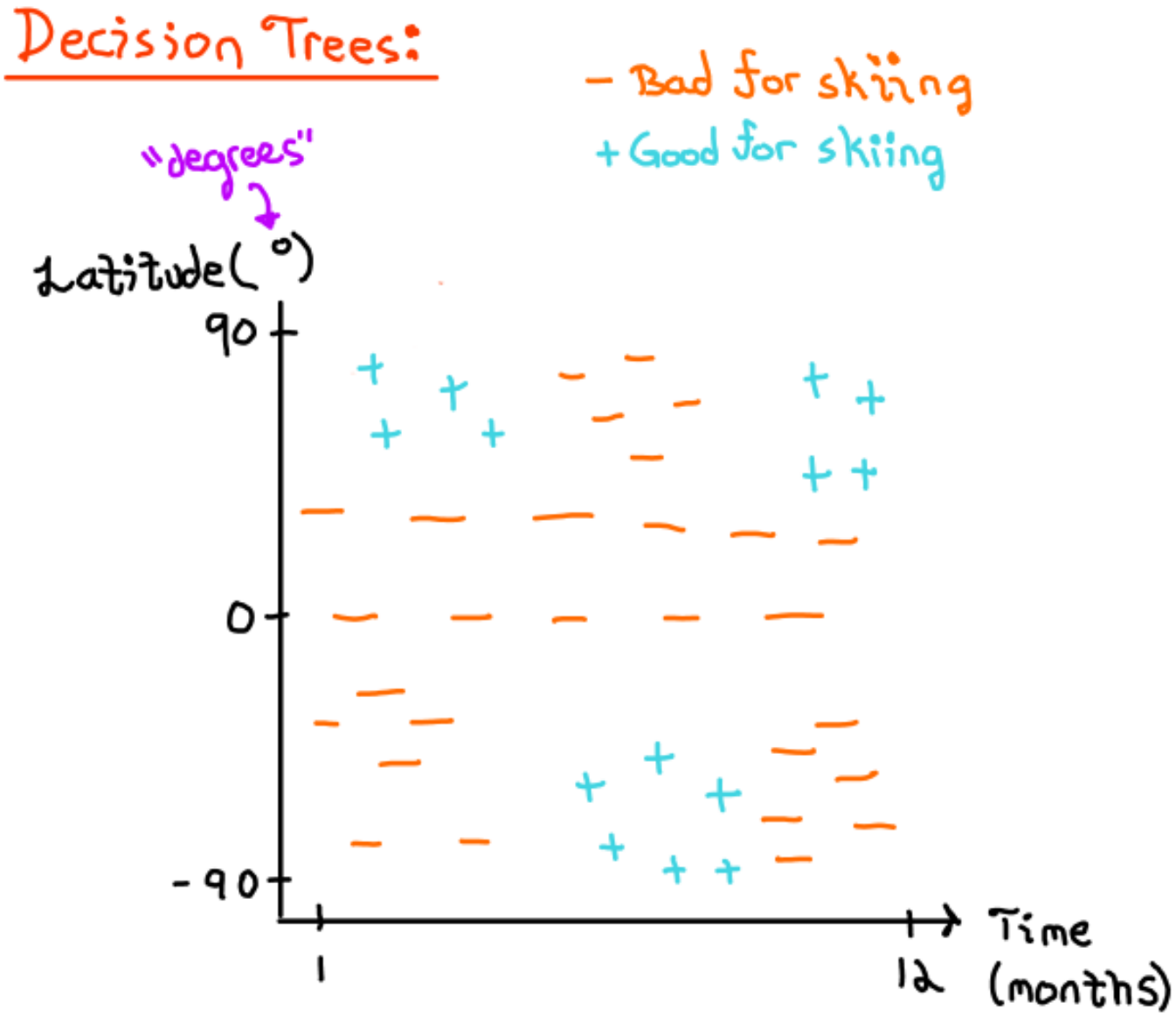
Lecture 10 [Decision Tress and Ensemble Methods]

Decision Trees:

Decision trees are sort of a classic example model class to use with various ***ensembling methods*** and is one of our first examples of a ***non-linear model***

Pretend you have a classifier, that given a time and a location, tells you whether or not you can ski (it's a ***binary classifier*** saying yes or no) and a graph like this:



When you look at a dataset like this, you have these separate regions that you're looking at and you want to isolate out those regions of positive examples. If you had a linear classifier, you'd be hard-pressed to come up with any sort of decision boundary that would separate this reasonably or an SVM or something where you'd come up with a kernel that could project it into a higher feature space that would make it linearly separable

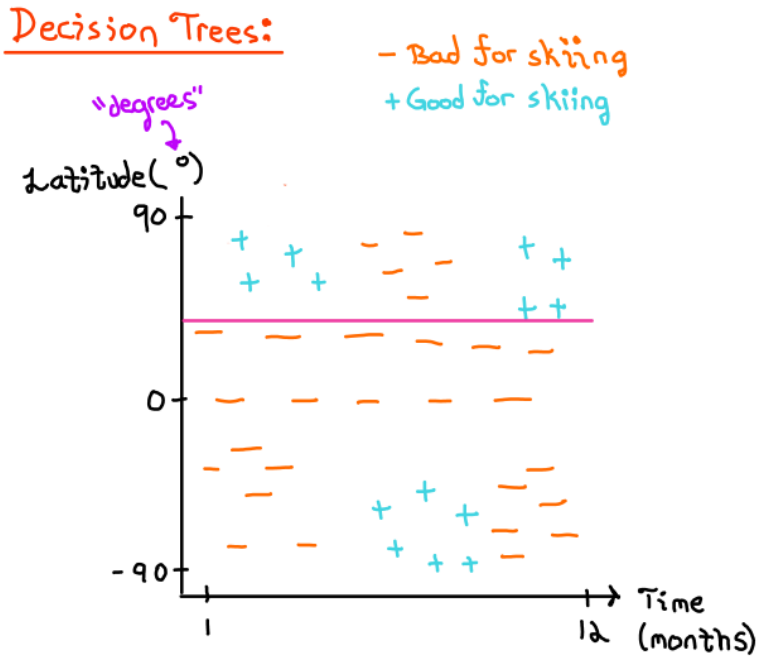
With ***Decision Trees***, you have a very natural way to do this. What we want to do with decision trees is we want to partition the space into individual regions to isolate out the positive examples, for example

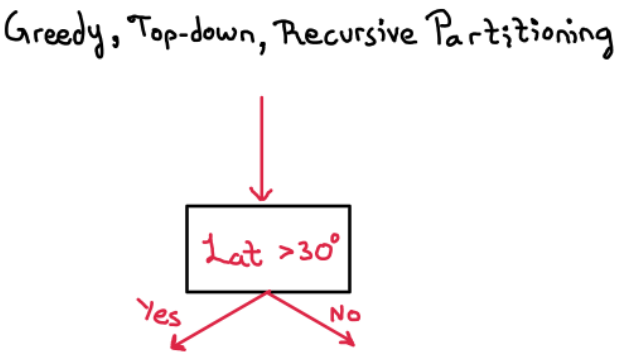
In general, this problem is fairly intractable, just coming up with the optimal regions, but how we do it with ***decision trees*** is we do it in this, basically greedy, top-down, recursive partitioning manner. It's ***top-down*** because we're starting with the overall region and we want to slowly partition it up. It's ***greedy*** because at each step, we want to pick the best partition possible

What a decision tree would do:

What we do is we start with the overall space and the tree is basically going to play ***'20 Questions'*** with this space

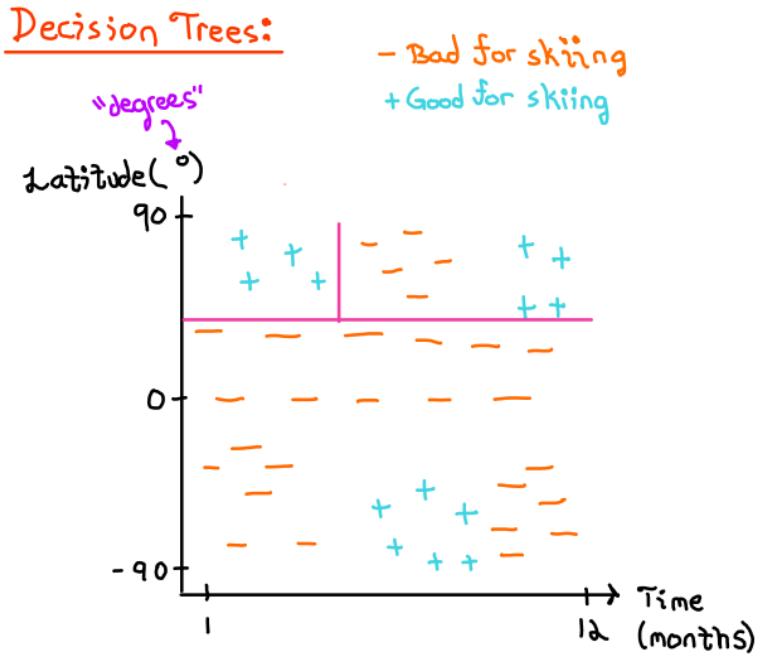
For example, one question it might ask is "**Is the Latitude > 30°?"** . This would involve cutting the space like this, for example:

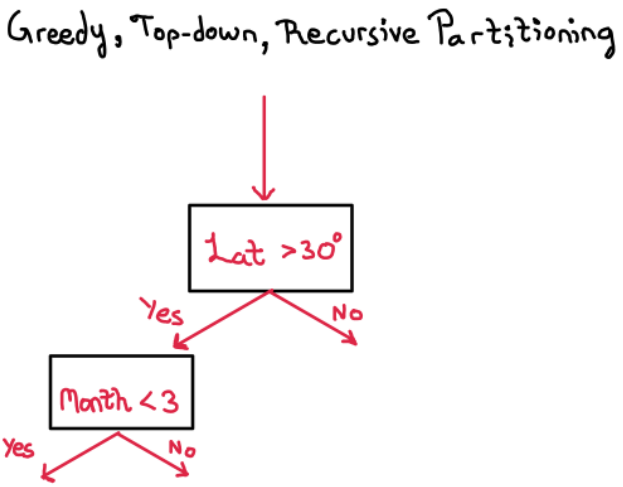




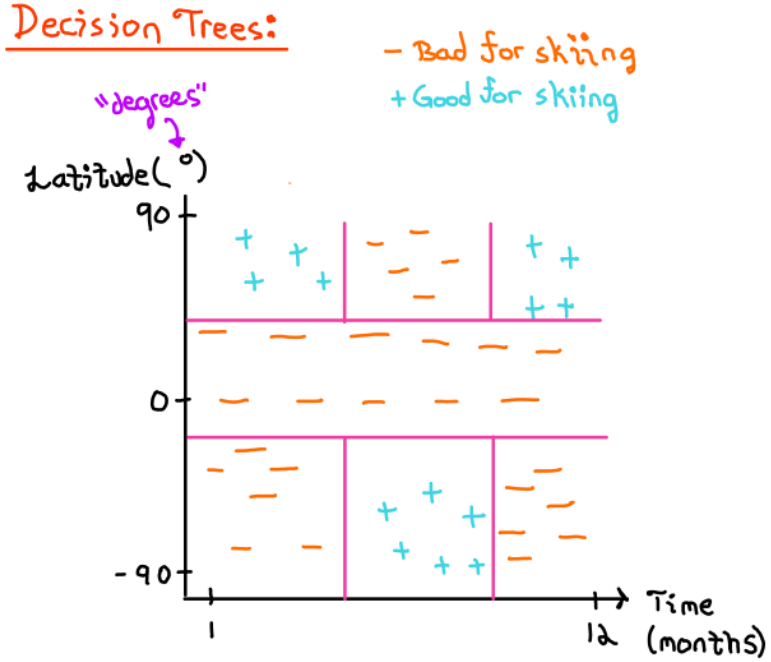
So, starting from the most general space, now we have partitioned the overall space into two separate spaces using this question, and this is where the recursive part comes in now. Now that you've split the space into two , you can then treat each individual space as a new problem to ask a new question about

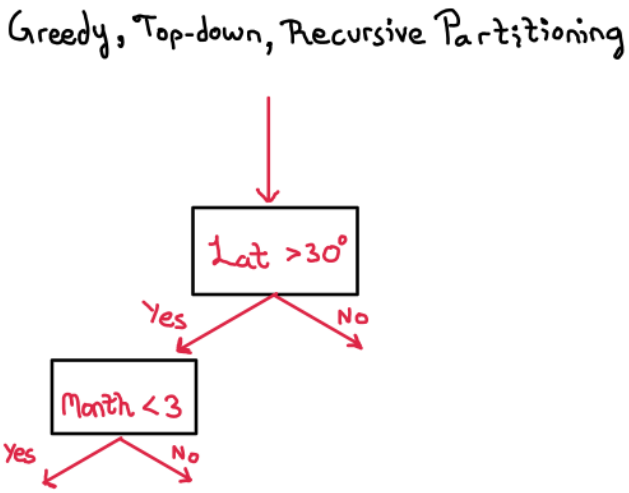
You could then ask something like:



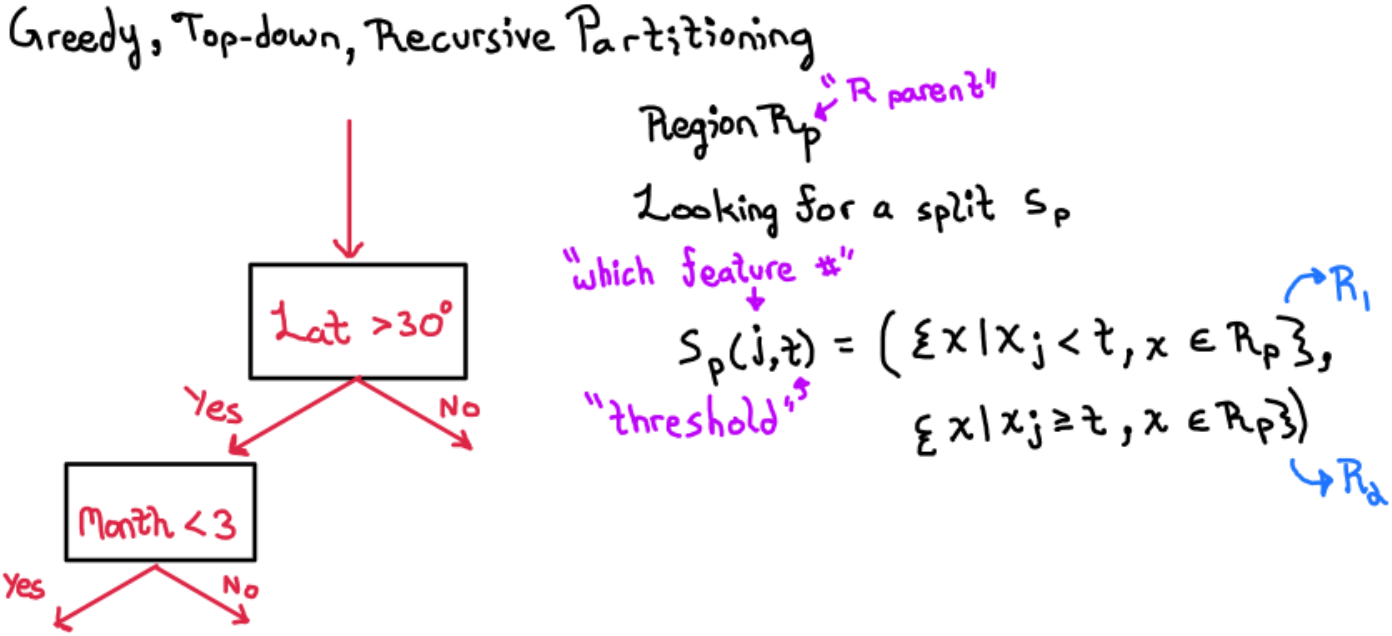


And then you could start splitting up the entire space into your individual regions like this:





Formally, we are looking for this ***split function***. You can define a region:



We can refer to each one of these sets as shown

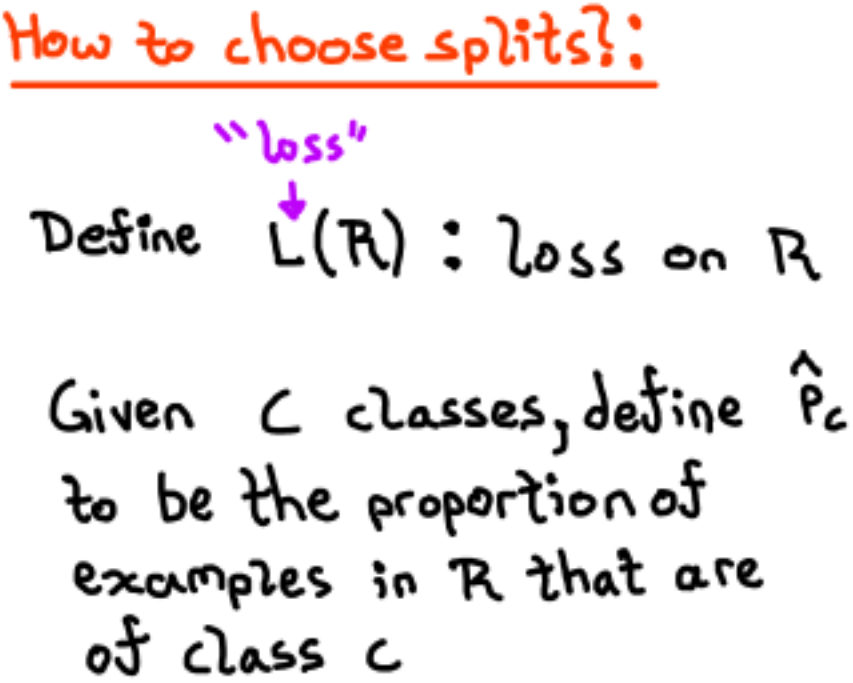
We would now define how we would do this:

We were trying to greedily pick these peaks that are partitioning our input space and the splits are defined by which feature you're looking at and the threshold that you're applying to that feature

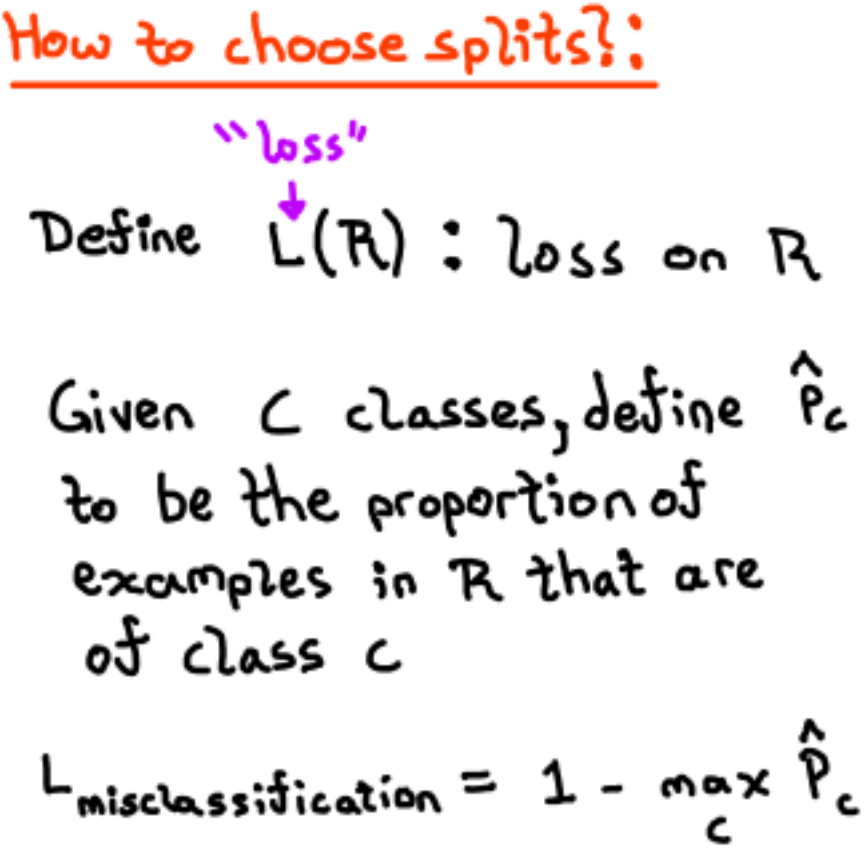
A natural question to ask now is is how do you choose these splits?

How do you choose these splits?:

We gave the intuitive explanation that really what you're trying to do is you're trying to isolate out the space of positives and negatives, in this case. It is useful to define a ***loss on a region***



Now that we've got this definition, we can try to define the loss of any region as:



The reasoning behind this is basically you can say that, for any region that you've subdivided, generally what you'll want to do is predict the most common class there which is just the maximum of $\hat{p}_c$, and then all the remaining probability just gets thrown onto ***misclassification errors***

Now that we have a ***loss*** defined, we want to pick a ***split*** that decreases the loss as much as possible

How to choose splits?:

Define "loss" $L(R)$: loss on R

Given C classes, define $\hat{P}_c$ to be the proportion of examples in R that are of class c

$$L_{misclassification} = 1 - \max_c \hat{P}_c$$

$$\max_{j,t} \underbrace{L(R_p)}_{\text{Parent loss}} - \underbrace{(L(R_1)+L(R_2))}_{\text{children loss}}$$

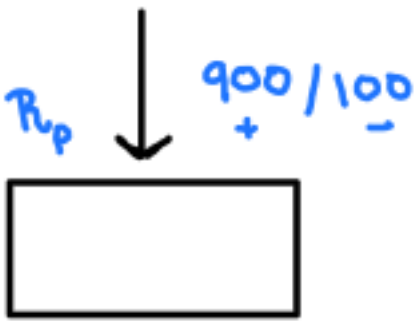
Since basically what you're minimizing over, in some case, is this **(j,t)** that we defined since this split is really what is going to define our two children regions

The loss of the parent doesn't really matter, in this case, because that's already defined. So really, all you're trying to do is minimize this negative sum of losses of your children

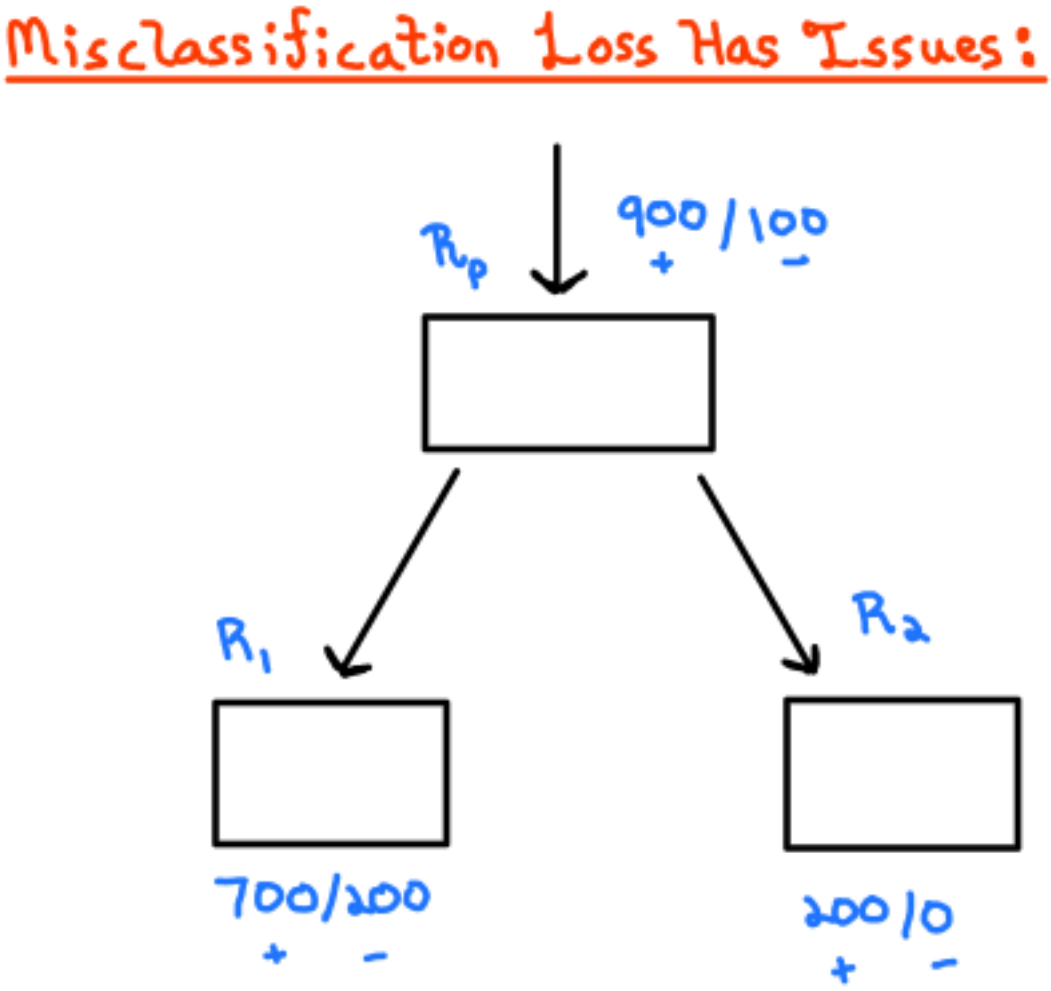
Misclassification Loss:

For a simple example, let's pretend that instead we have another setup where we're coming into a decision node and at this point we have 900 positives and 100 negatives

Misclassification Loss Has Issues:

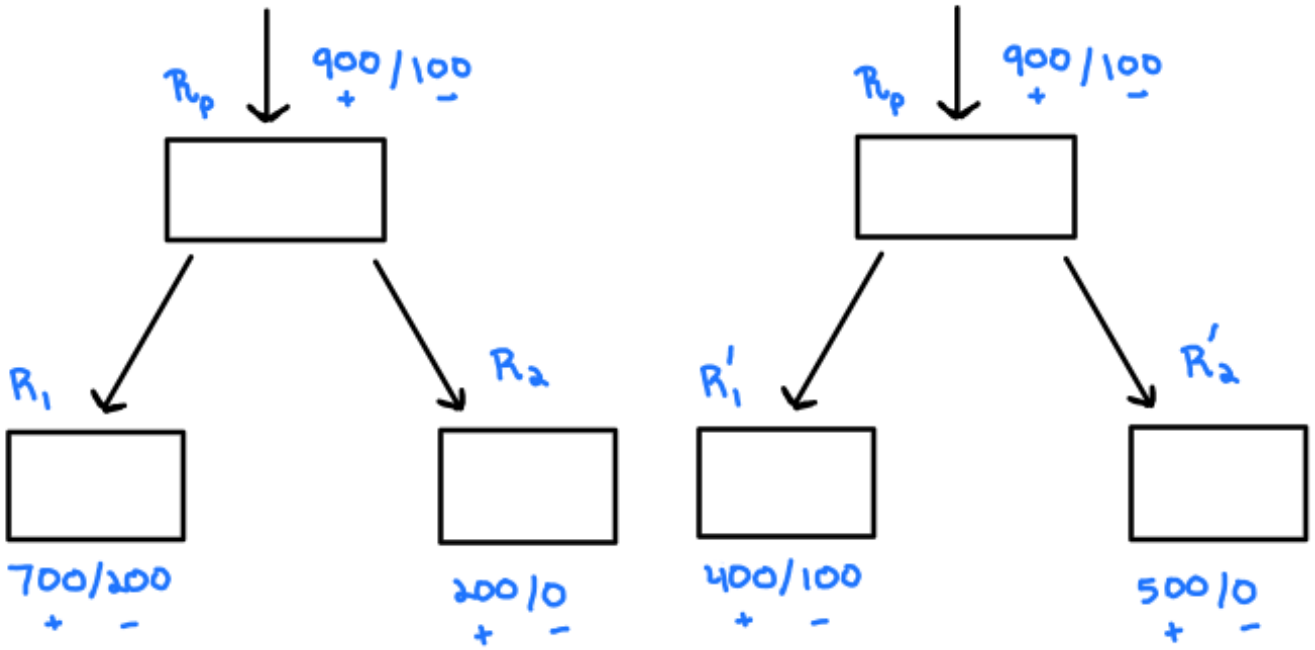


This is a misclassification loss of **100** in this case because you'd predict the most common class and end up with **100** misclassified examples. This would currently be your region $\mathbf{R_p}$, and so, you can split it into these two other regions



This seems like a pretty good split since you're getting out some more examples, but what you can see is that, if you just drew the same thing again:

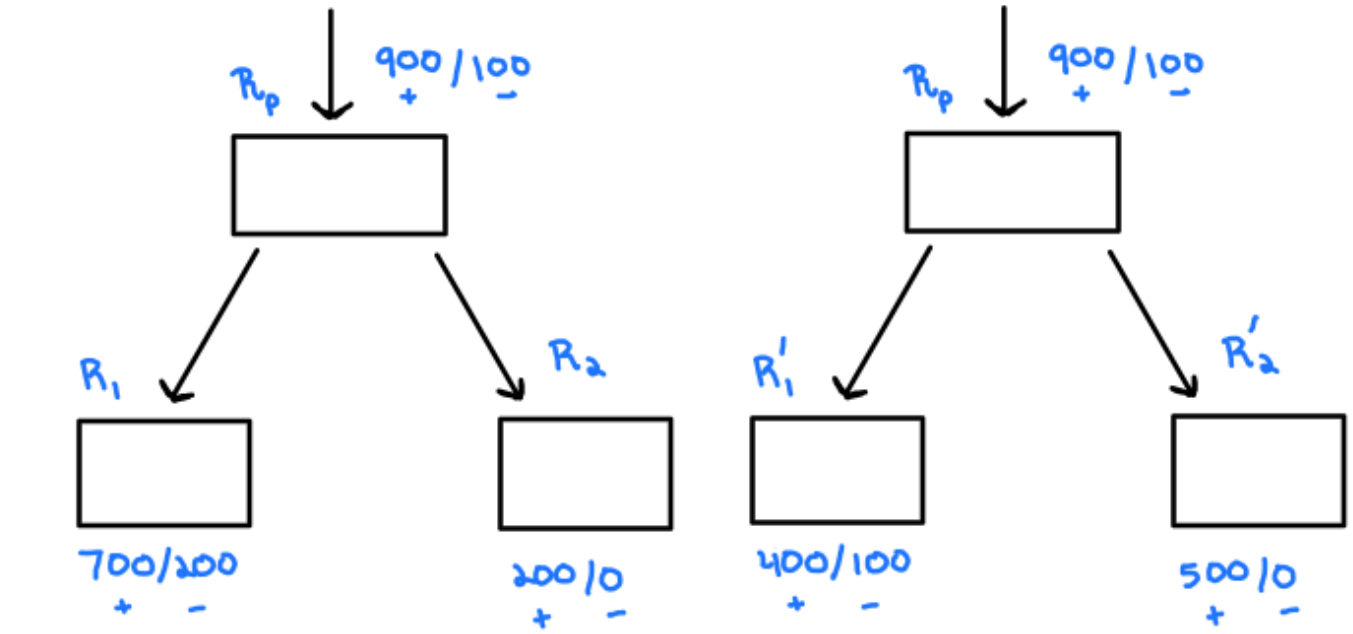
Misclassification Loss Has Issues:



Most people would argue that this decision boundary on the right is better than the one on the left because you're basically isolating out even more positives, in this case

However, if you're just looking at your ***misclassification loss***, it turns out that:

Misclassification Loss Has Issues:



$$L(R_1) + L(R_2) = 100 + 0 = 100$$

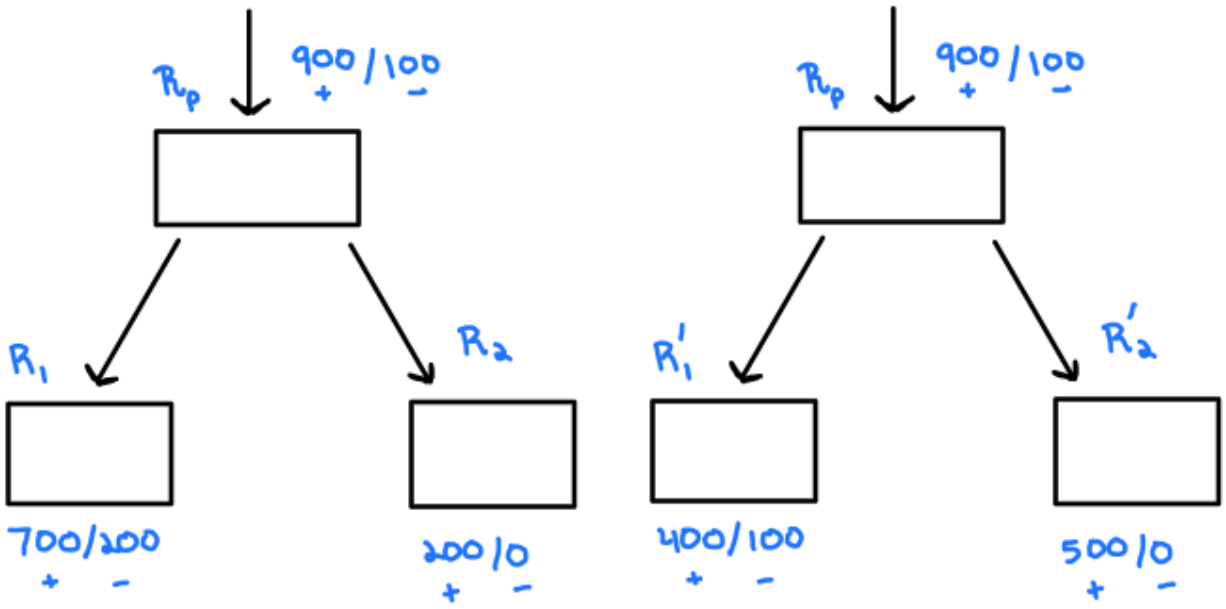
$$L(R_1') + L(R_2') = 100 + 0 = 100$$

$$L(R_p) = 100$$

So the left and right decision boundaries are actually the same, and if you'd look at the ***original loss*** of your parent, it's also just **100**. So you haven't really, according to this loss metric, changed anything at all

And so that brings up one problem with the ***misclassification loss***; it's nit really sensitive enough. Instead what we can do is we can define this ***cross-entropy loss***

Misclassification Loss Has Issues:



$$L(R_1) + L(R_2) = 100 + 0 = 100$$

$$L(R_1') + L(R_2') = 100 + 0 = 100$$

$$L(R_p) = 100$$

Instead, define cross-entropy loss

$$L_{cross} = -\sum_c \hat{p}_c \log_2 \hat{p}_c$$

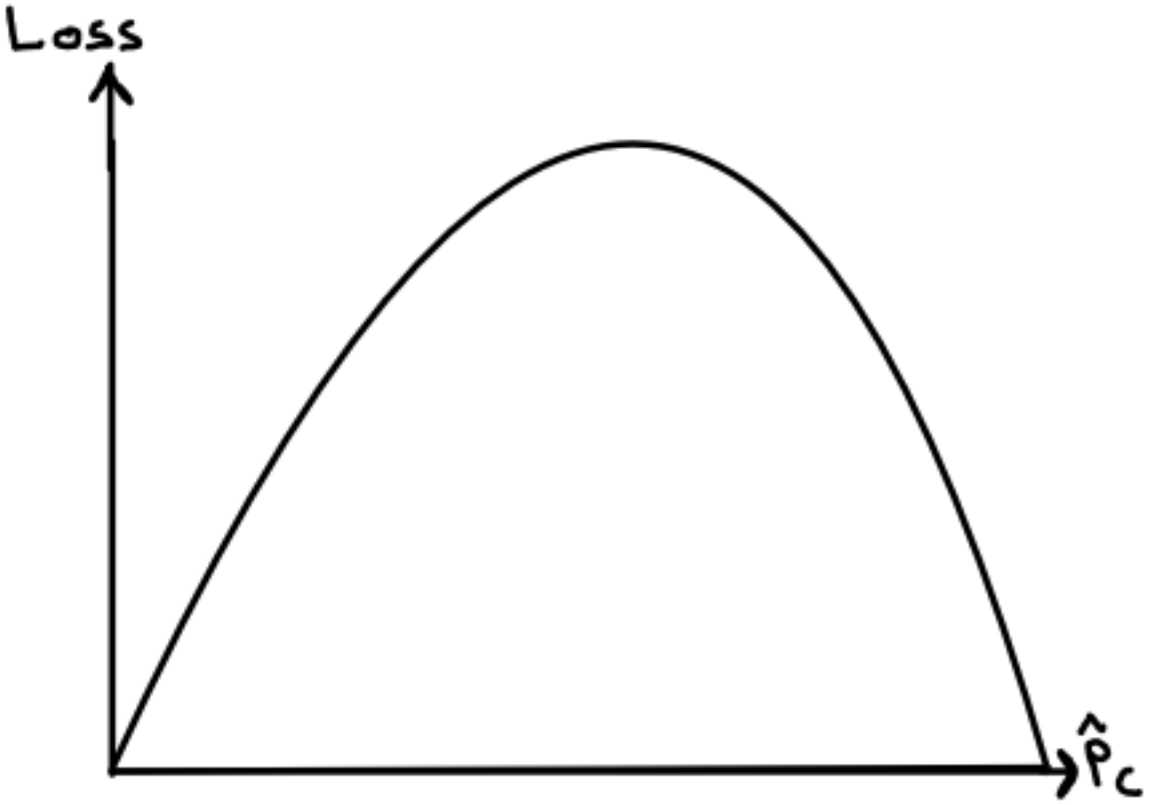
So really what you're doing is you're just summing over the classes and it's the proportion of elements in that class times the log of the proportion in that class, and how you can think of this is, it's this concept that we borrow from ***information theory*** “the number of bits you need to communicate to tell someone who already knows what the probabilities are, what class you are looking at”. You can think of this intuitively as “if someone already knows the probabilities, say it's a **100%** chance that it is of one class, then you don't need to communicate anything to tell them exactly which class it is because it's obvious that it is that one class, versus if you have a fairly even split, then you'd need to communicate a lot more information to tell someone exactly what class you were in”

We can get a fairly good intuition for why ***misclassification loss*** vs ***cross-entropy loss*** might be better or worse by looking at it from a ***geometric perspective***

Pretend now that you have a binary classification problem, and you can represent **p^** as the proportioin of positives in your set

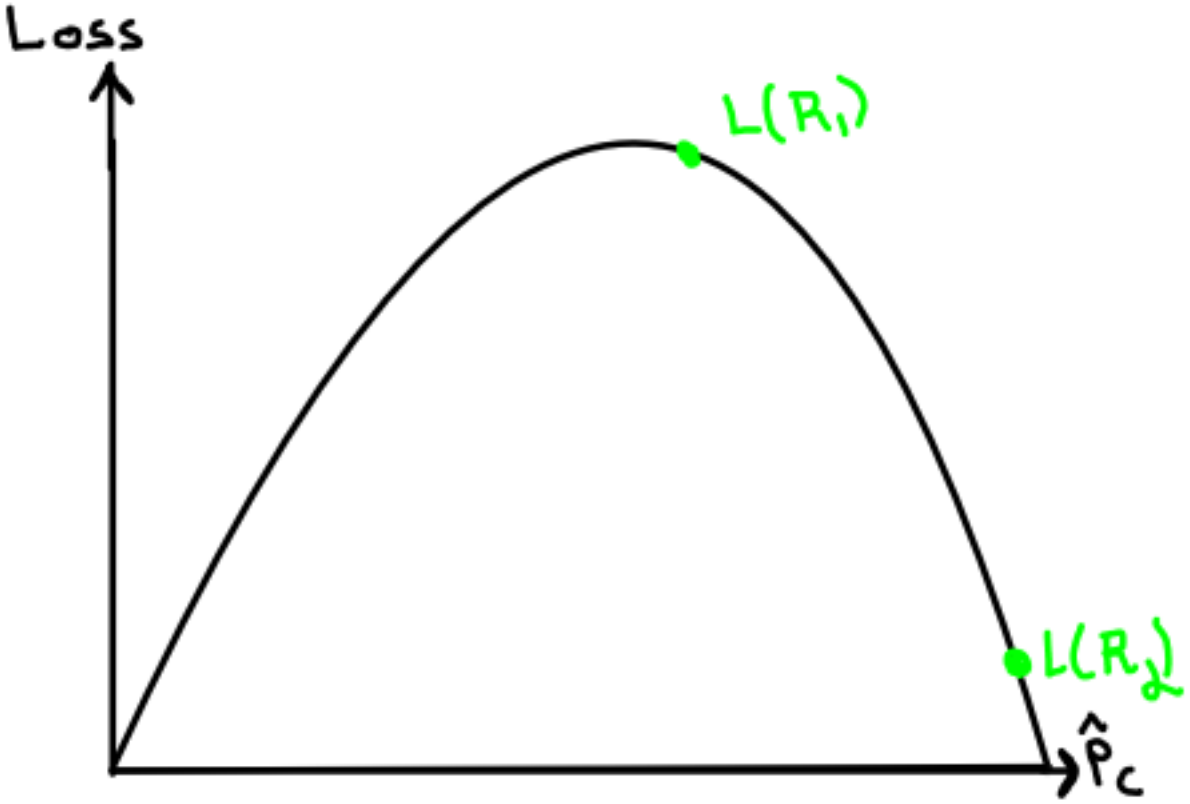
Cross-Entropy Loss:

For ***cross-entropy loss***, where your curve is gonna end up looking like this strictly concave curve

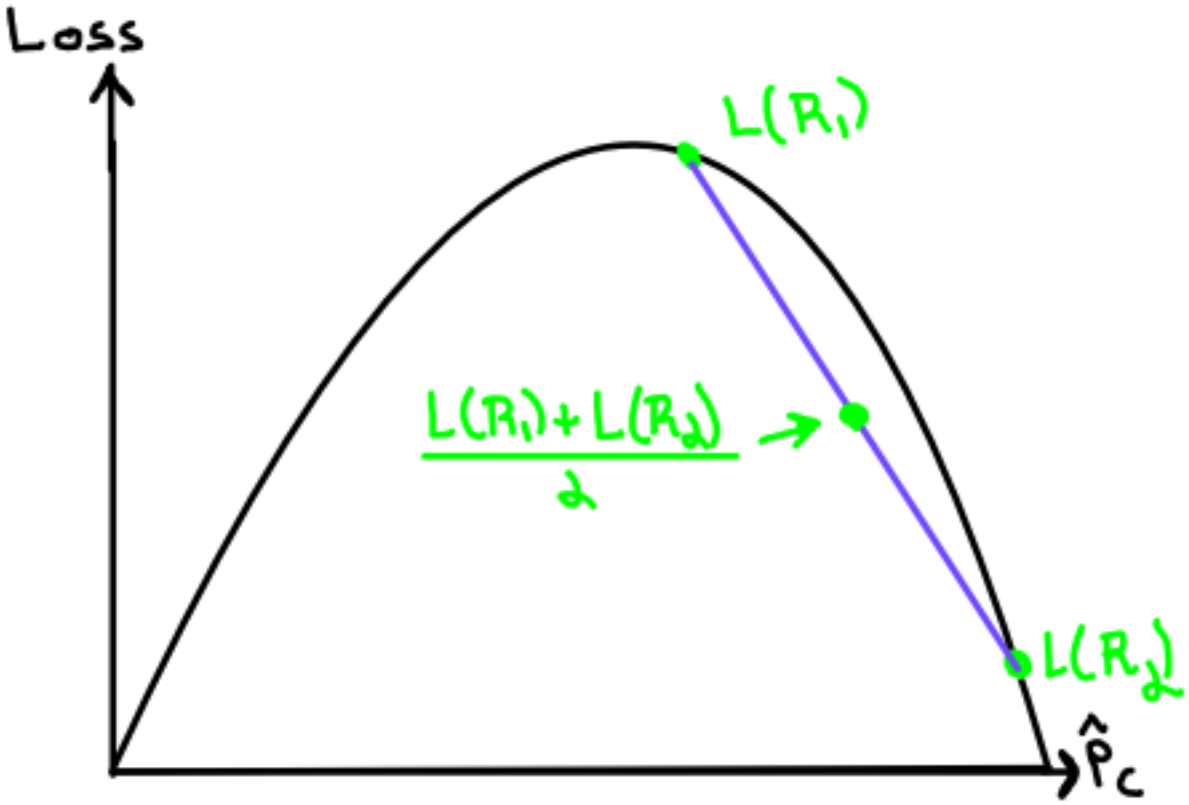


You can look at where ***your children vs your parent*** would fall on this curve

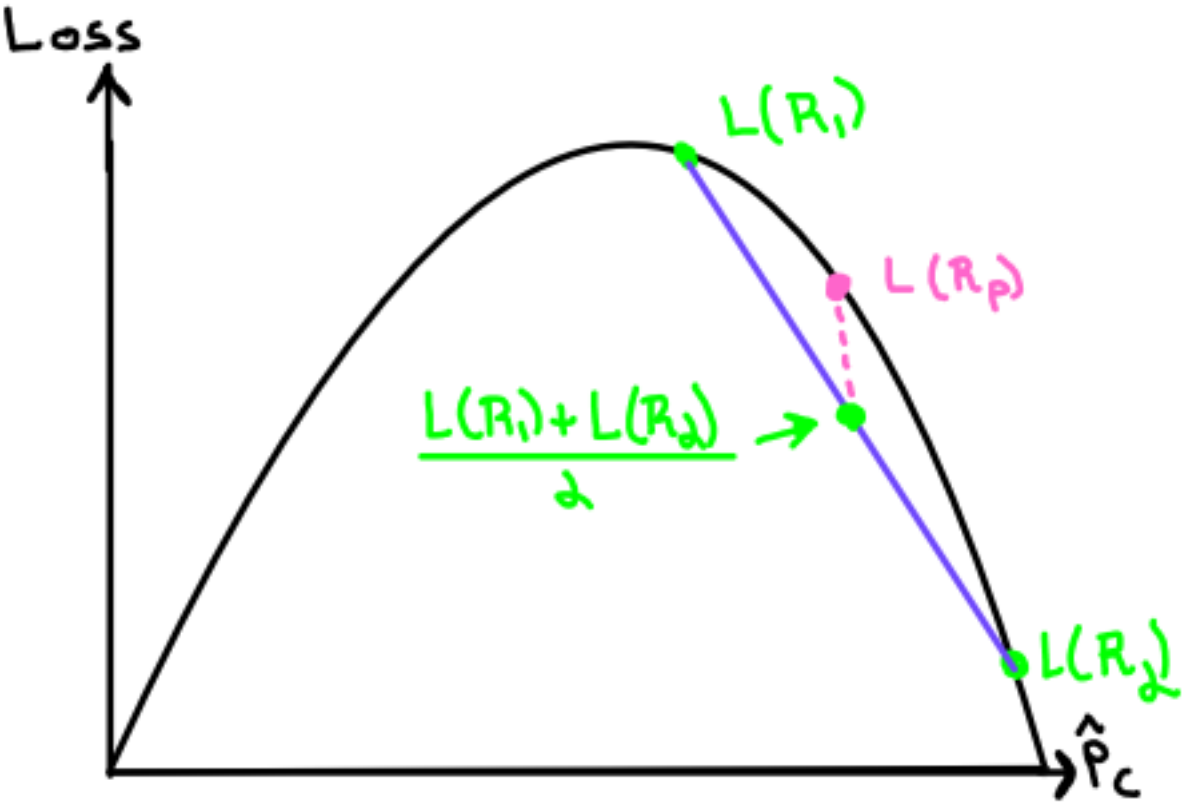
Say you have two children:



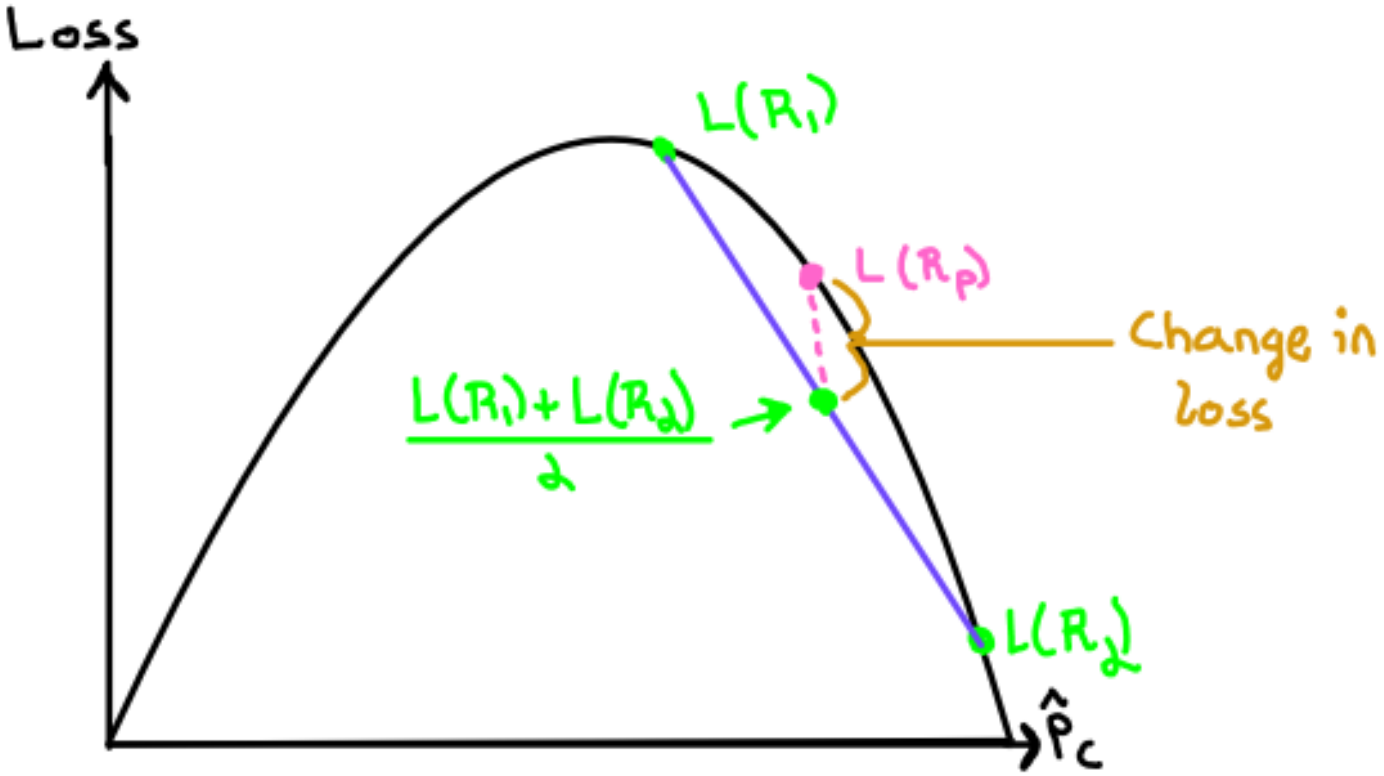
And say that you have an equal number of examples in both $\mathbf{R_1}$ and $\mathbf{R_2}$, so they're equally weighted. When you're looking at the overall loss between the two, that's really just the average of the two, so you can draw a line between these two and the midpoint turns out to be the average of your two losses



What you can notice is that, in fact, the loss of the parent node is actually just this point projected upwards here:



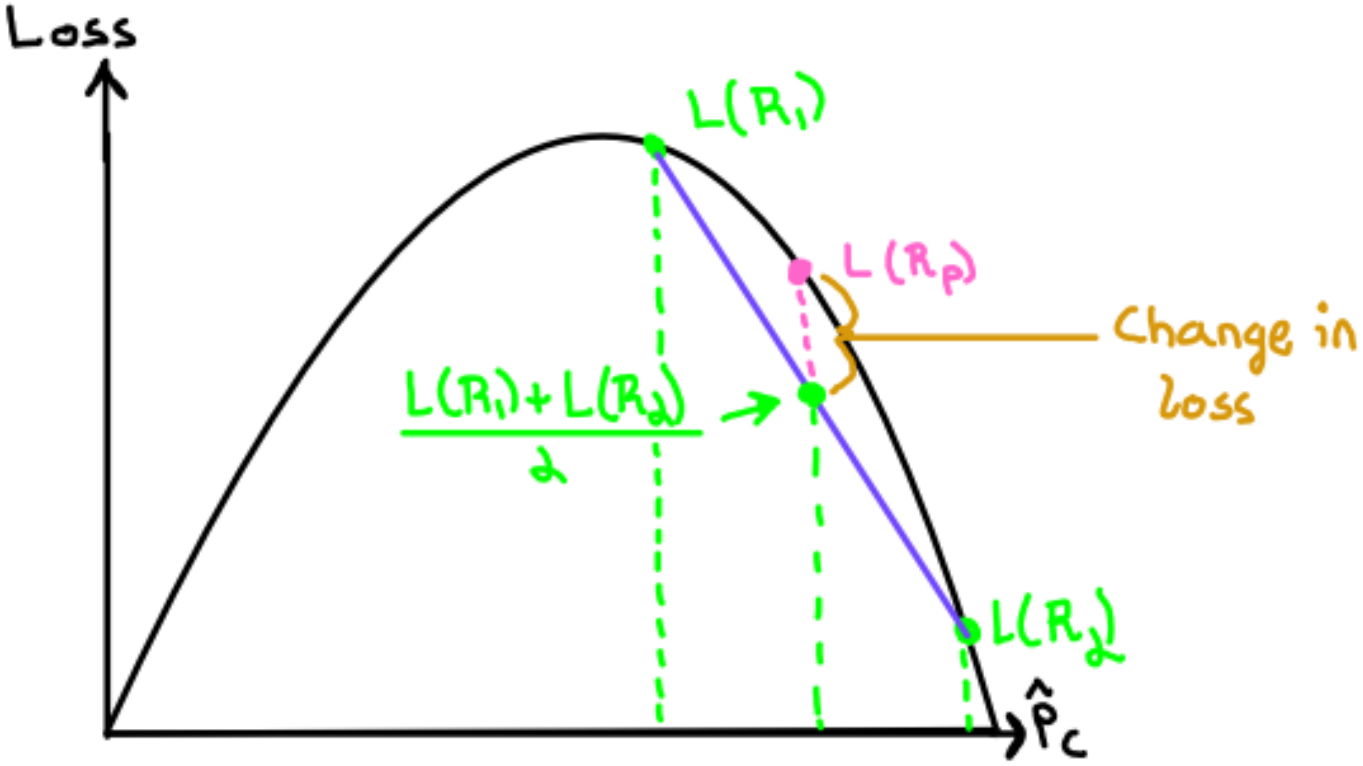
And this difference is your change in loss:



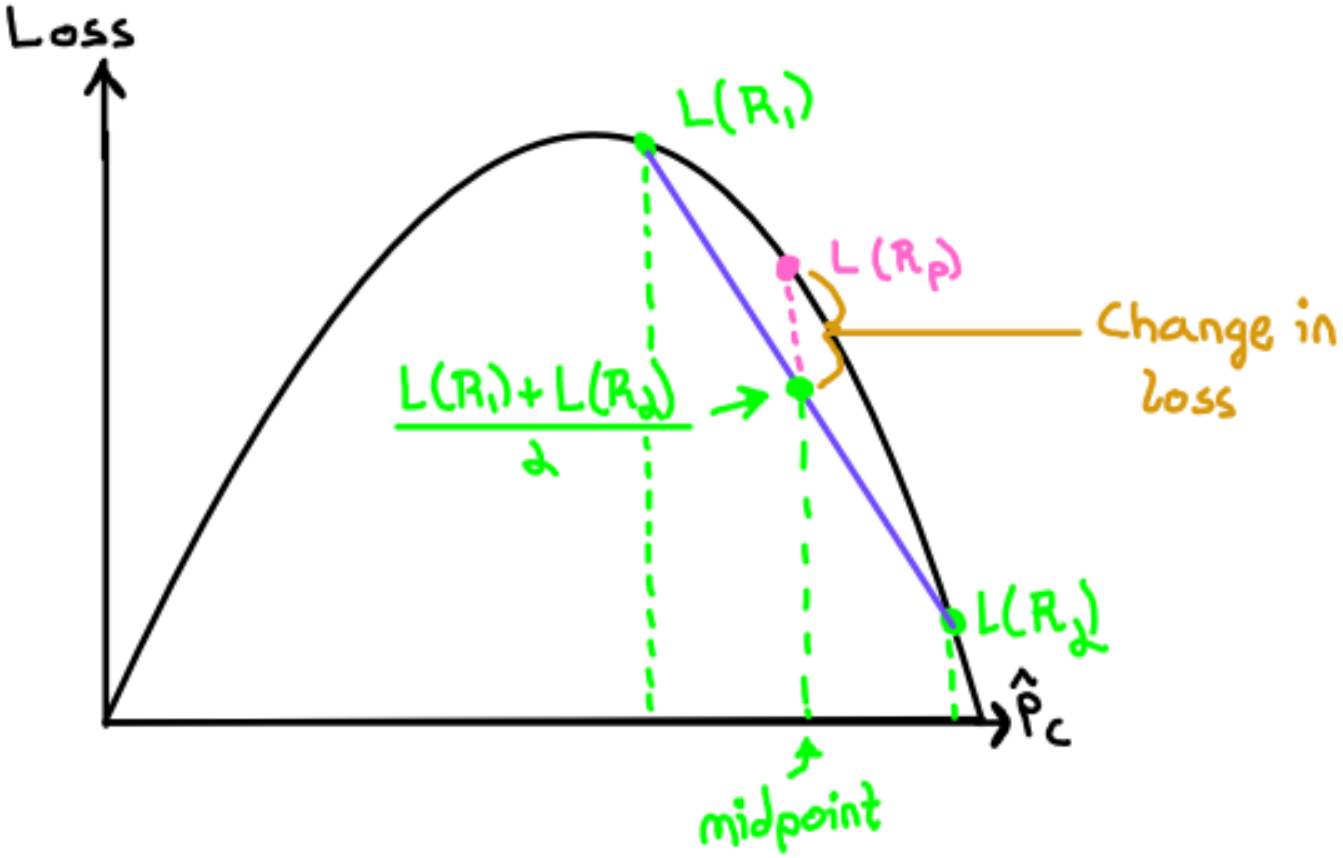
Recap:

Say we have two children regions and they have different probabilities of positive examples occuring. They would fall on the above points on the curve, then the average of the two losses falls on the midpoint between these two original losses. If you look at the parent, it's really just halfway between on the ***x-axis*** and you can project upwards for that as well and you end up with the loss of $\mathbf{R_p}$

If we projcted these points down:

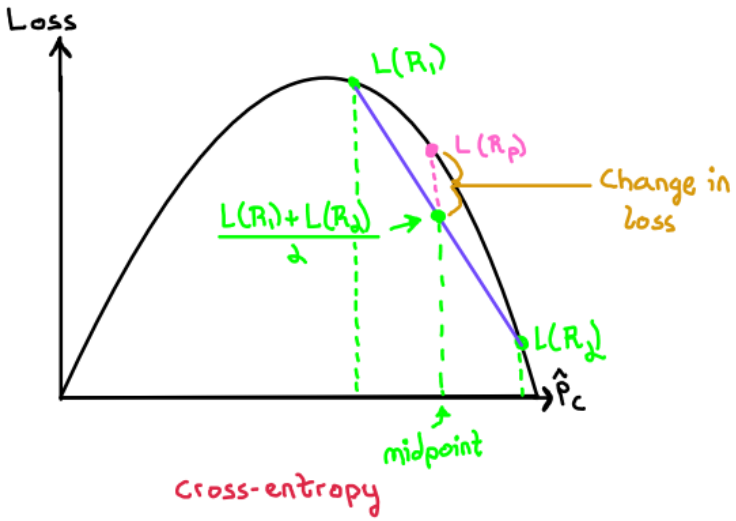


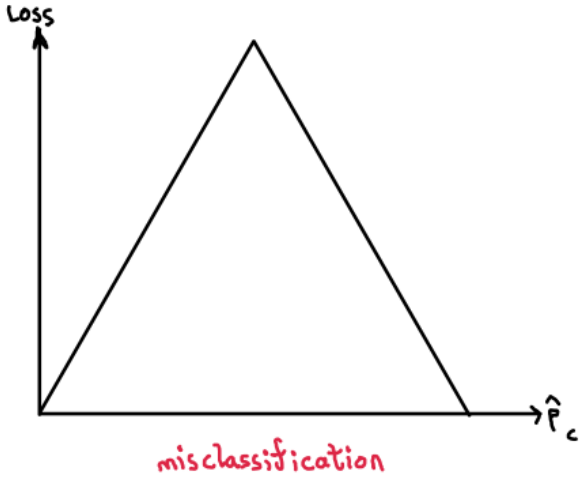
We'd see that this is the midpoint:



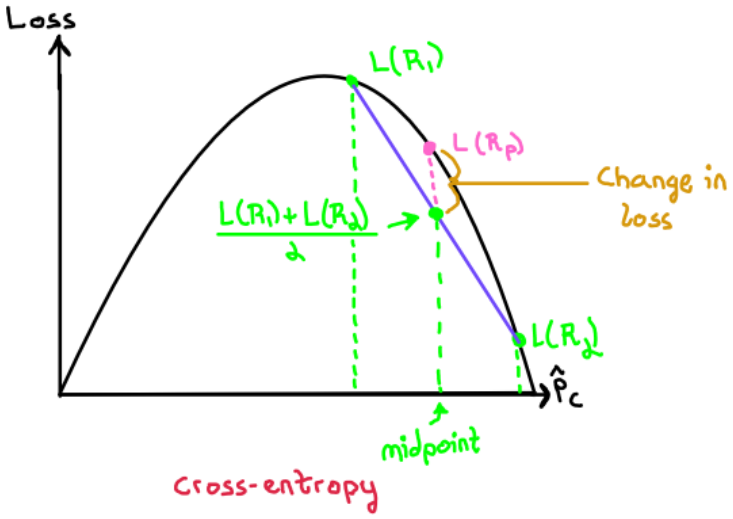
But then when you're averaging the two losses after you've done the split, then you're basically just taking the average loss

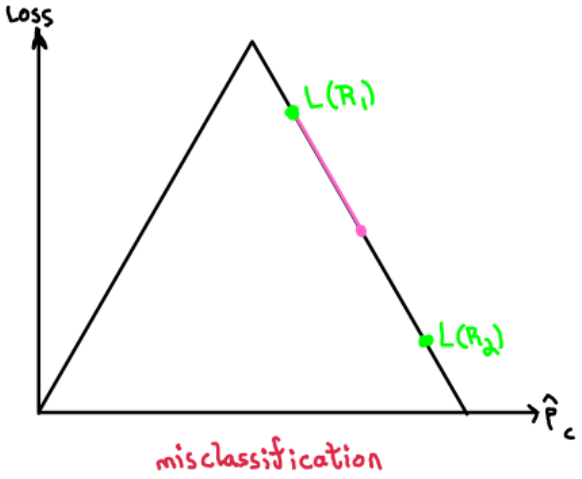
If you have an uneven split, the average would just be any point along this (**lavender**) line. And as you can see, the whole thing is strictly concave, so any point along that line is going to lie below the ***original loss curve*** for the parent. So as long as you're not picking the exact same points on the probability curve and not making any gain at all in your split, you're going to gain some amount of information through this split

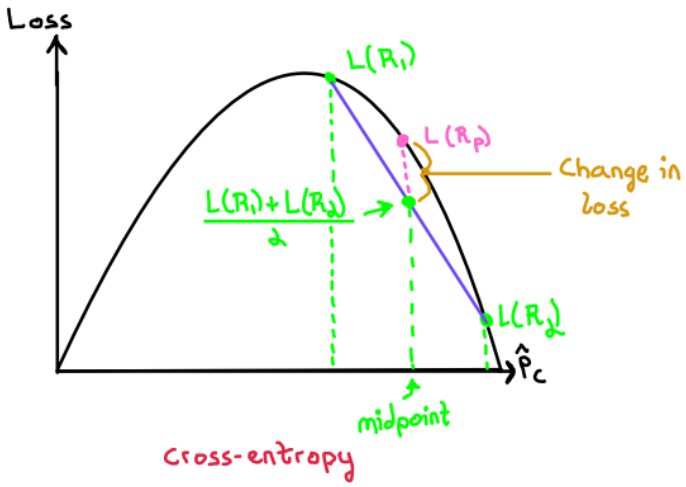


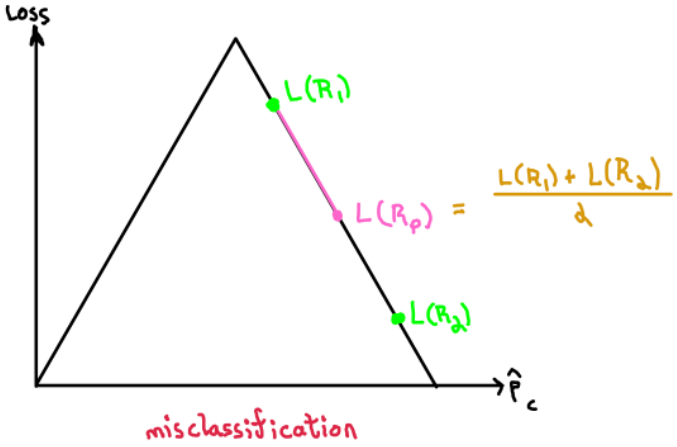


If instead we look at the misclassification loss, we can see that it's this pyramid kind of shape where it's just linear and then flips over once you start classifying the other side



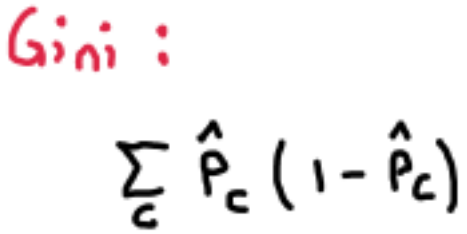






So in this case, even though according to the ***cross-entropy formulation***, you do have a gain in information and intuitively we do see a gain in information in the decision boundary of the decision tree. For the misclassification loss, since it's not very sensitive, if you end up with points on the same side of the curve then you actually don't see any sort of information gain, based on this kind of representation

There's also the ***Gini Loss***:



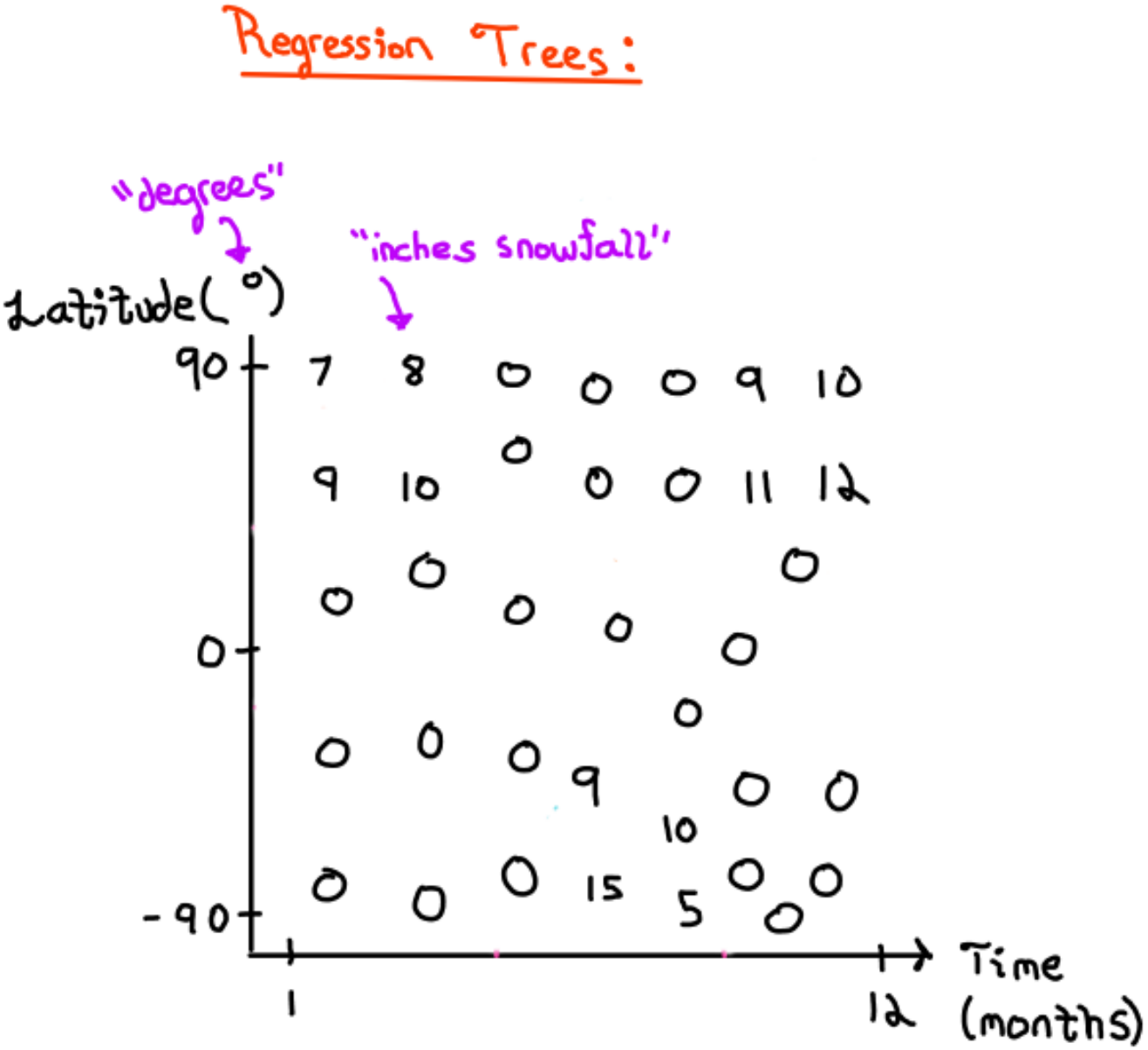
It turns out that this curve also looks very similar to the original ***cross-entropy curve***

What you'll see is that most curves, that are successfully used for ***decision splits***, look basically like the strictly concave function

So far we've been talking about decision trees for ***classification***. You could also imagine having decision trees for ***regression***, also known as ***Regression Trees***

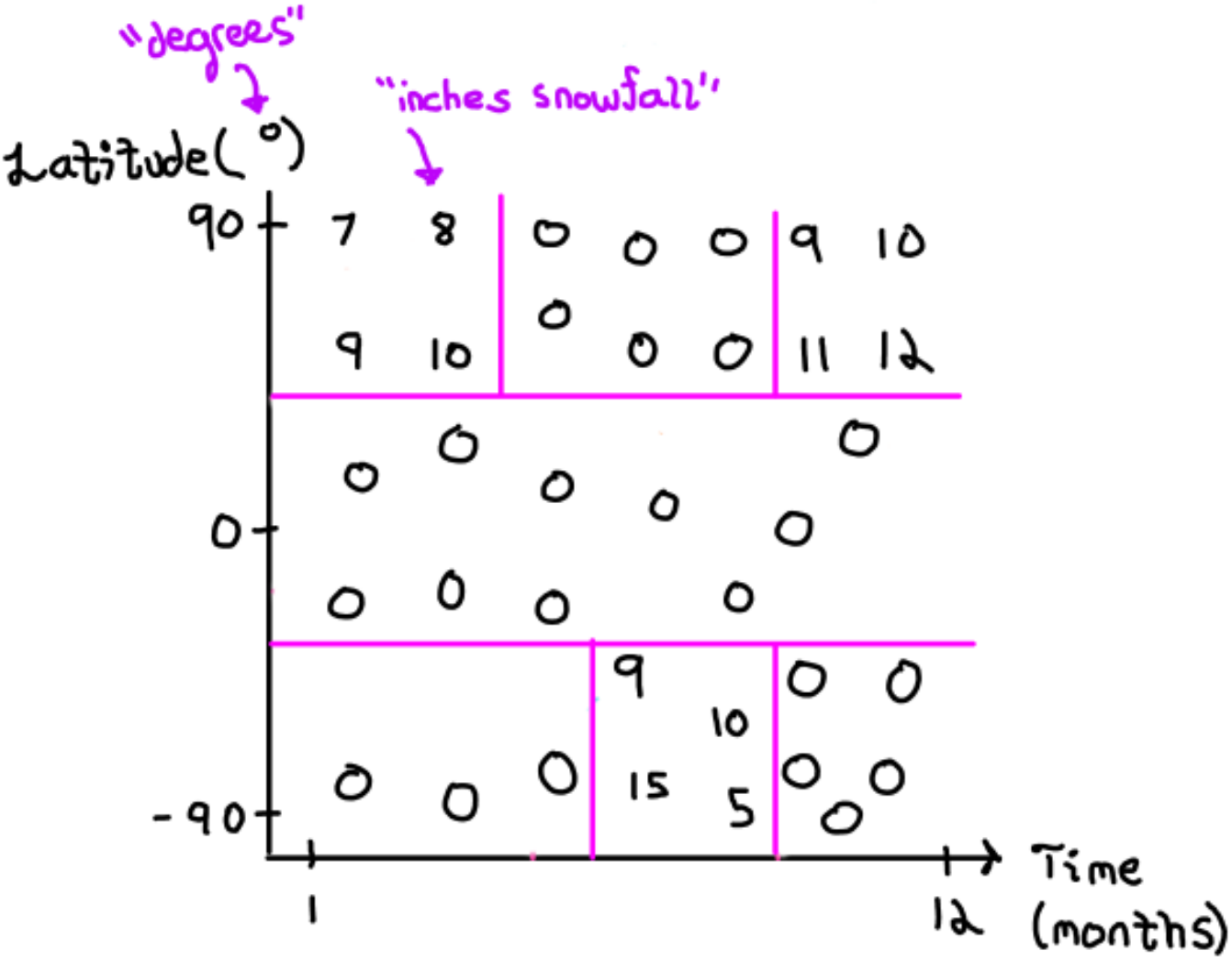
Regression Trees:

Taking the ski example again, let's pretend that instead of now predicting whether or not you can ski, you're predicting the amount of snowfall you would expect in that area around that time



You can sort of see how you would do just the exact same thing. You still want to isolate regions and sort of increase the purity of those regions. So you could still create your trees like this and split out like this, for example:

Regression Trees:

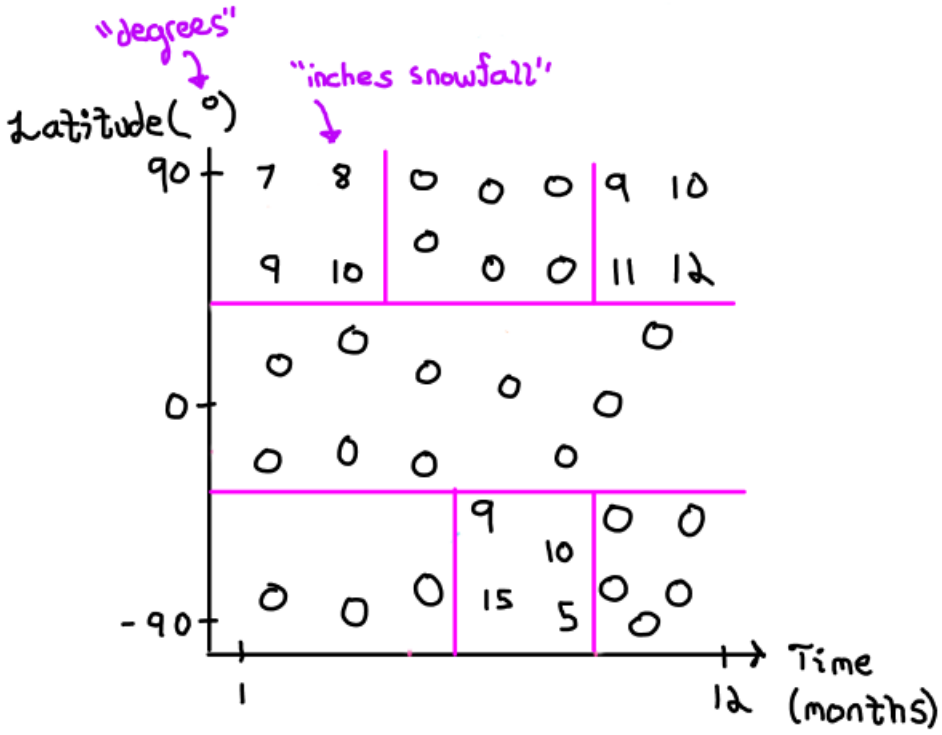


What you do when you get to one of your leaves is instead of just predicting a majority class, what you can do is predict the ***mean*** of the values left:

Regression Trees:

"Region" R_m

Predict $\hat{y}_m = \frac{\sum_{i \in R_m} y_i}{|R_m|}$



You're summing all the values within your region and then just taking the average of that

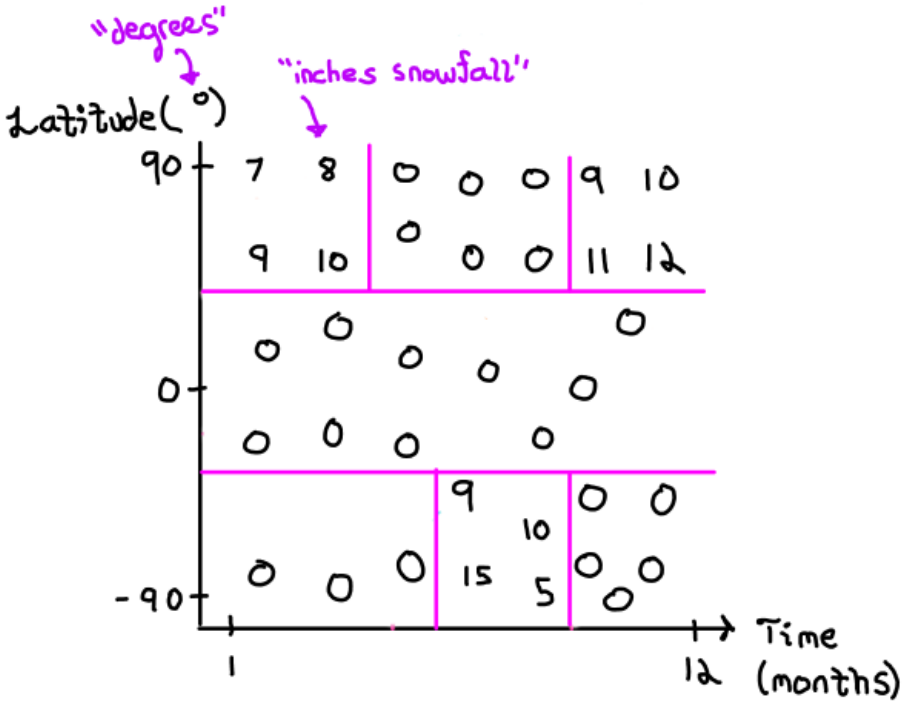
The loss that you would use is your ***squared loss***

Regression Trees:

"Region" R_m

Predict $\hat{y}_m = \frac{\sum_{i \in R_m} y_i}{|R_m|}$

$L_{squared} = \sum_{i \in R_m} \frac{(y_i - \hat{y}_m)^2}{|R_m|}$

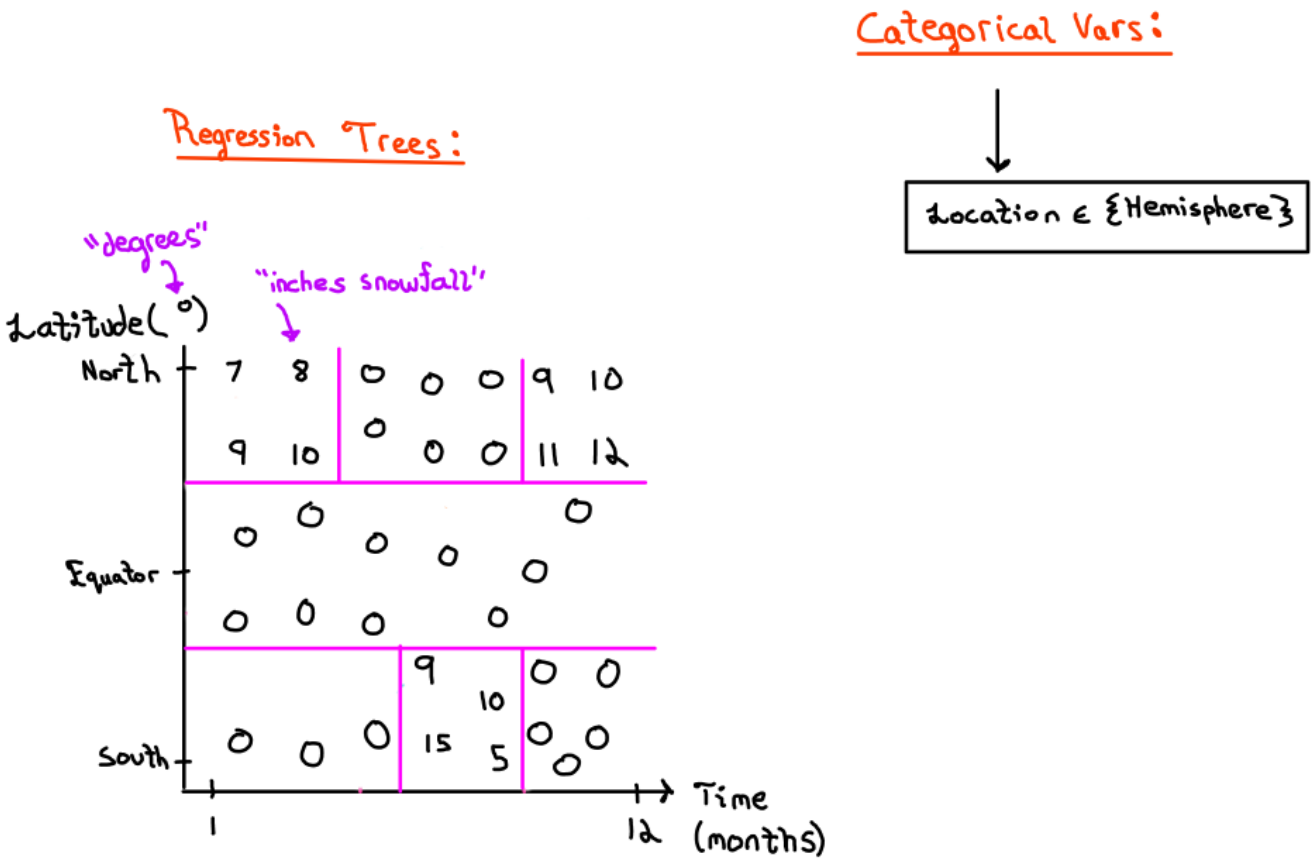


You have your ***mean prediction*** and then your ***loss*** in this case is how far off your ***mean prediction*** is from the overall predictions

#How do you actually search for these splits? How do you actually solve the optimization problem of finding these splits?

You can actually basically ***brute-force*** it very efficiently

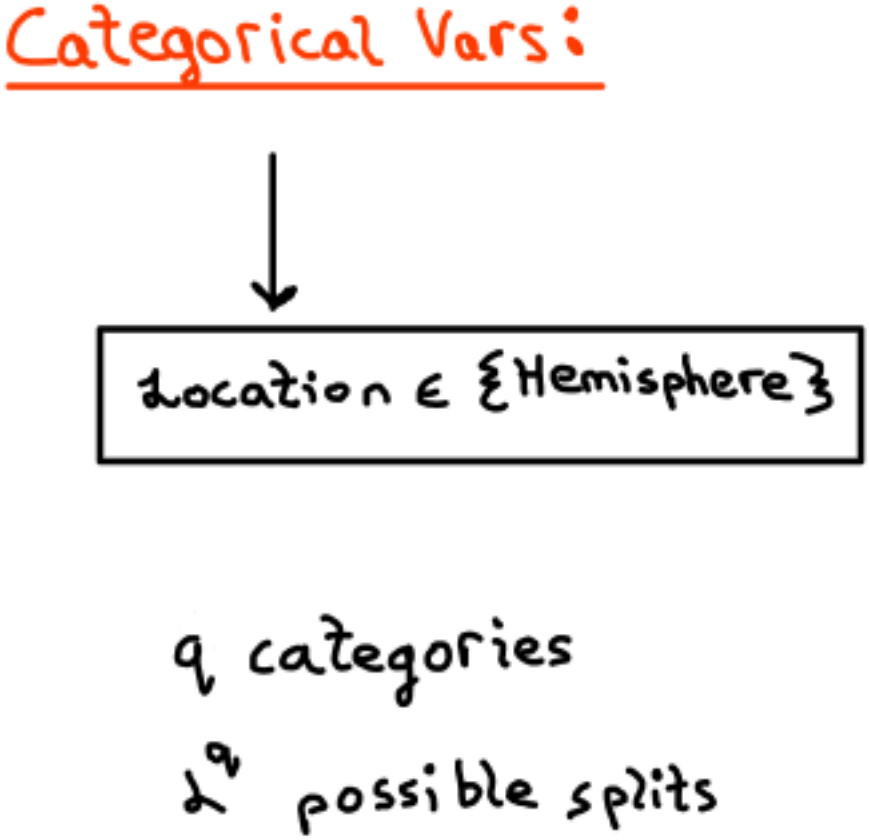
For ***Regression Trees***, another useful extension that you don't really get for other learning algorithms is that you can also deal with ***categorical variables*** fairly easily



Basically for this case, you could imagine that instead of having your **latitude** in **degrees**, you could just have three categories. You could have something like the above. And then you could instead ask questions such as "Is Location in (North/Equator/South) Hemisphere

So you could ask a question about any sort of subset of the categories you're looking at

One thing to be careful about is that if you have **q** categories, then you have $\mathbf{2^{q}}$ possible splits:

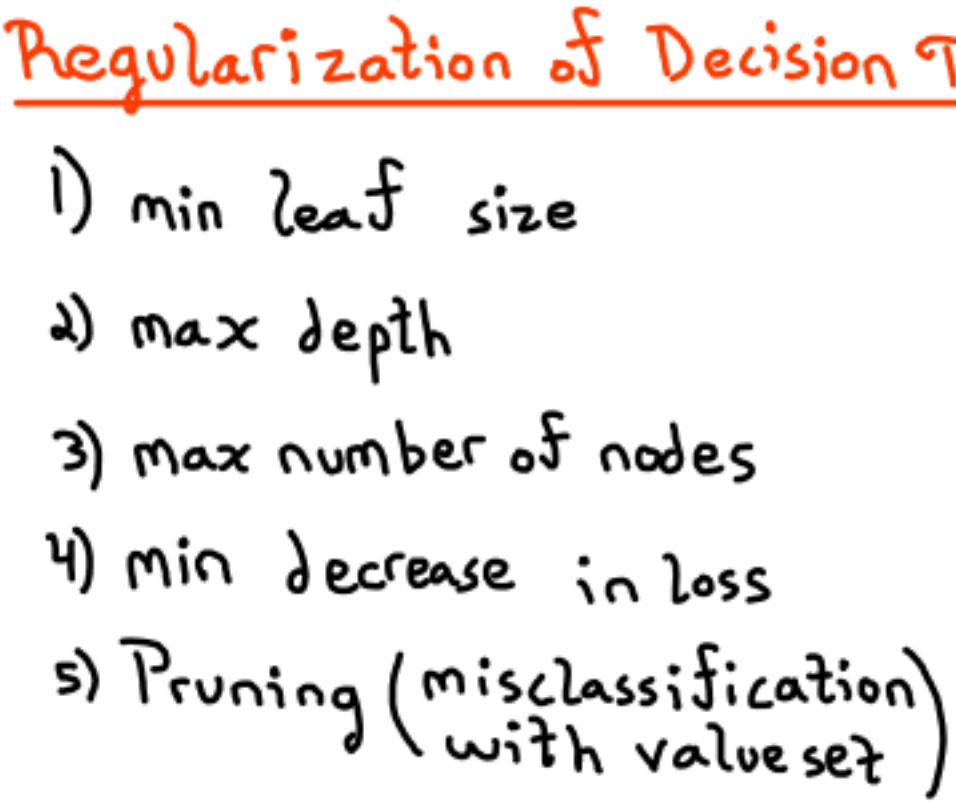


And so in general, you don't want to deal with too many categories because this will become quickly intractable to look through that many possible examples

In certain, very specific cases, you can still deal with a lot of categories. One such case is for ***binary classification*** where you can basically sort your categories by how many positive examples are in each category, and then just take that as a sorted order, then search through it linearly, which will yield you an optimal solution

We can use ***Decision Trees*** for ***Regression***, we can also use them for ***Categorical Variables***

You can imagine that in the limit, if you grew your tree without ever stopping, you could end up just having a separate region for every single data point that you have. You could consider that probably overfitting if you ran it all the way to that completion. So you can see that ***decision trees*** are fairly ***high variance*** models. So one thing that we're interested in doing is ***regularizing*** these ***high variance*** models



One such heuristic is that if you hit a certain ***minimum leaf size***, you stop splitting that leaf. So for example, if you only have four examples left in a leaf (***region***), then you just stop

Another one is you can enforce a ***maximum depth***, and a related one is a ***maximum number of nodes***.

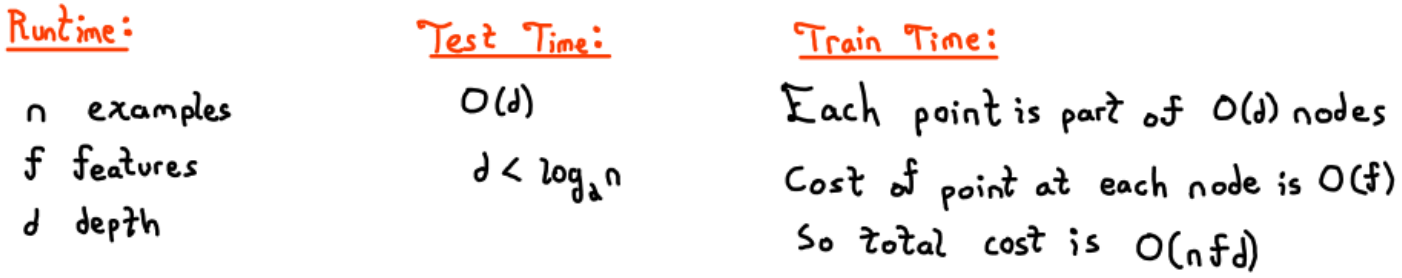
A fourth, very tempting one, to use is a ***minimum decrease in loss***. This one is tempting because it's generally not actually a good idea to use this minimum decrease in loss 1. You can think about that by thinking that if you have any sort of higher-order interactions between your variables, you might have to ask one question that is not very optimal or doesn't give you that much of an increase in loss, and then your follow-up question combined with that first question might give you a much better increase. In the case of the previous example, the initial latitude question doesn't really give us that much of a gain, we sort of split some positive and negatives, but the combination of the latitude question plus the time question, really nails down what we want. And if we were looking at it purely from the ***minimum decrease in loss*** perspective, we might stop too early and miss that entirely

And so a better way to do this kind of loss decrease is instead, you grow out your full tree and then you ***prune*** it backwards instead. So you grow out the whole thing and then you check which nodes to ***prune*** out. How you generally do this is you have a ***validation set*** that you use this with, and you evaluate what your ***misclassification error*** is on your ***validation set*** for each example that you might remove (for each leaf that you might remove)

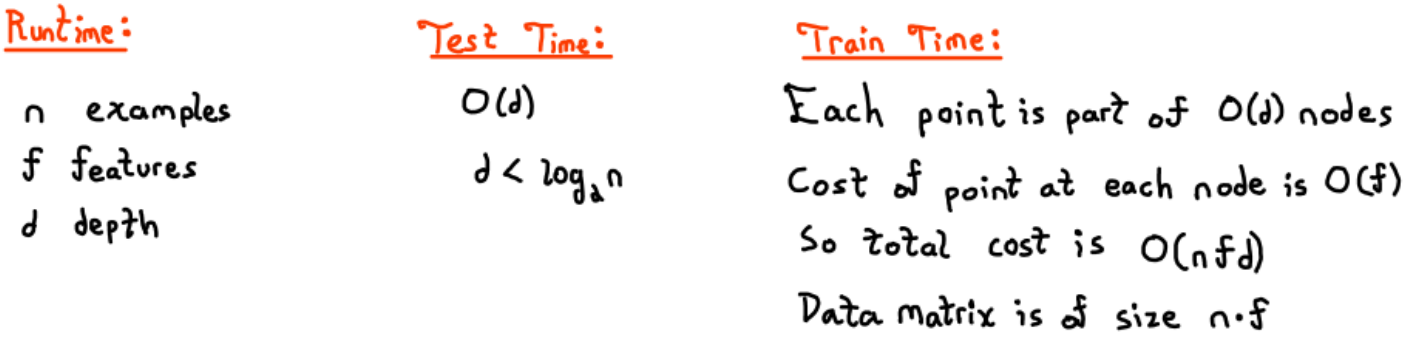
For ***Runtime***, say you have **n** examples, **f** features, and the depth **d** of your tree. You have **n** examples that you trained on, each with **f** features, and your resulting tree has depth **d**. At ***Test time***, your ***runtime*** is basically just your depth **d**. Typically, though not in all cases, **d** is less than the **log** of your number of examples. You can think about this as if you have a failry balanced tree, you'll end up sort of evenly splitting out all the examples, recursively doing these binary splits, and so you'll be splitting it at the **log** of that **n**. So at test time, you've generally got it pretty quick

At ***Train time***, if your tree is of depth **d**, each point is part of **O(d)** nodes and then at each node, you can actually work out that the cost of evaluating that point at ***train time*** is actually just proportional to the number of features **f**. You can consider that if you're doing binary features, for example, where each feature is just yes or no of some sort, then you only have to consider, if you have **f** features total, you only have to consider **f** possible splits, and that's why the cost in that case would be **f**. And then if it was instead a quantitative feature, you could sort the overall features and then scan through them linearly, and that also ends up being asymptotically **O(f)** to do that

So, each point is at most **O(d)** nodes, and then the cost of the point at each node is **O(f)** and you have **n** points total, so the total cost is **O(nfd)**



It turns out that this is actually surprisingly fast, especially if you consider that **nf** is just the size of your ***original design matrix*** (***data matrix***)

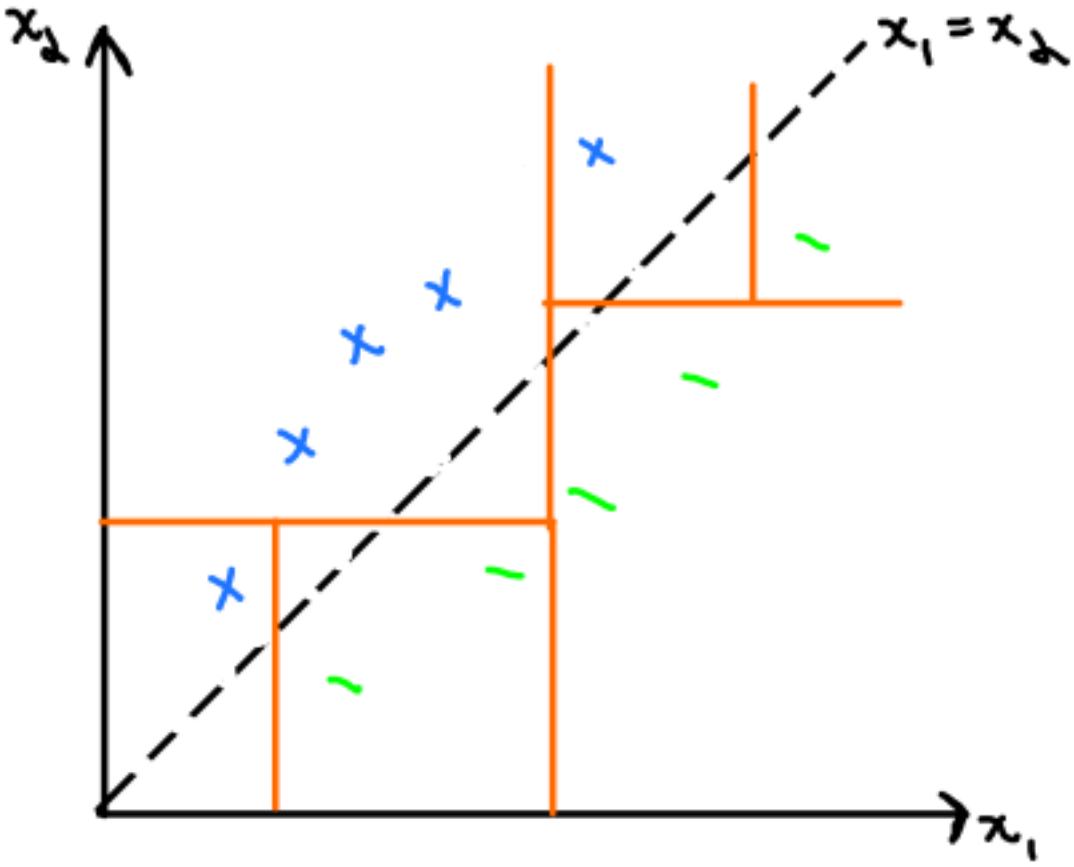


And then your ***runtime*** is going through the ***data matrix*** at most, depth times, and since depth is **log n**, that turns out to be generally bounded by **log n**; you have, generally, a fairly fast ***training time***

So far we've talked about the good sides of decision trees, however there are a number of downsides too. One big one is that it doesn't have ***additive structure*** to it

Let's say we have an example and you have just two features again, and say you define a line running through the middle defined by $x_1 = x_2$, and all the points above this line are positive and all the points below it are negative. Now if you had a simple ***linear model***, like ***logistic regression***, it'd have no issue with this kind of setup. But for a ***decision tree***, basically you'd have to ask a lot of questions to even somewhat approximate this line

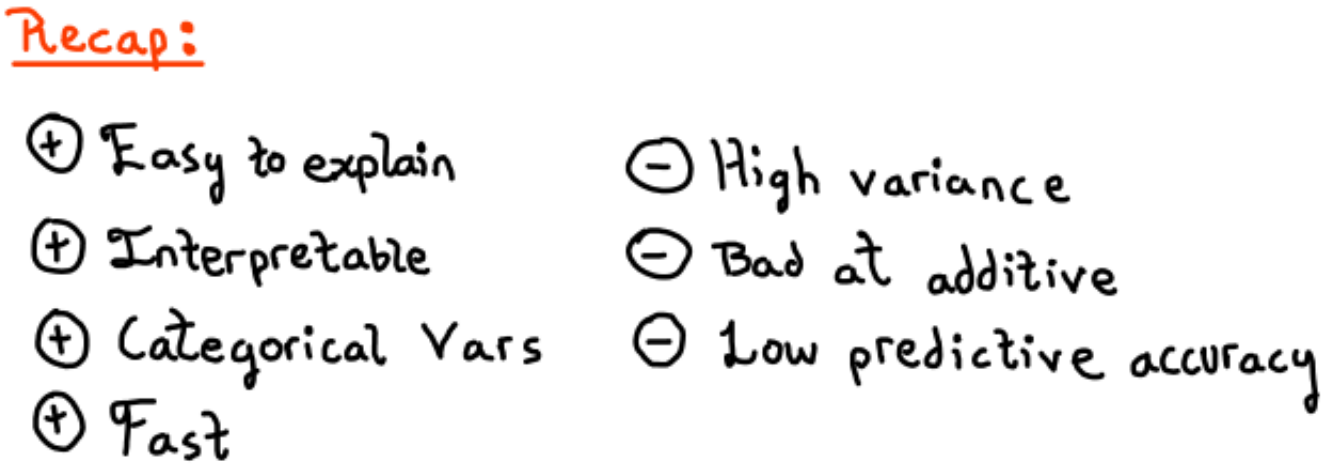
No additive structure:



You've asked a lot of questions and you've only gotten a very rough approximation of the actual line that you've drawn in this case

So ***decision trees*** do have a lot of issues with these kinds of structures where the features are interacting additively with one another

Recap of ***Decision Trees***:

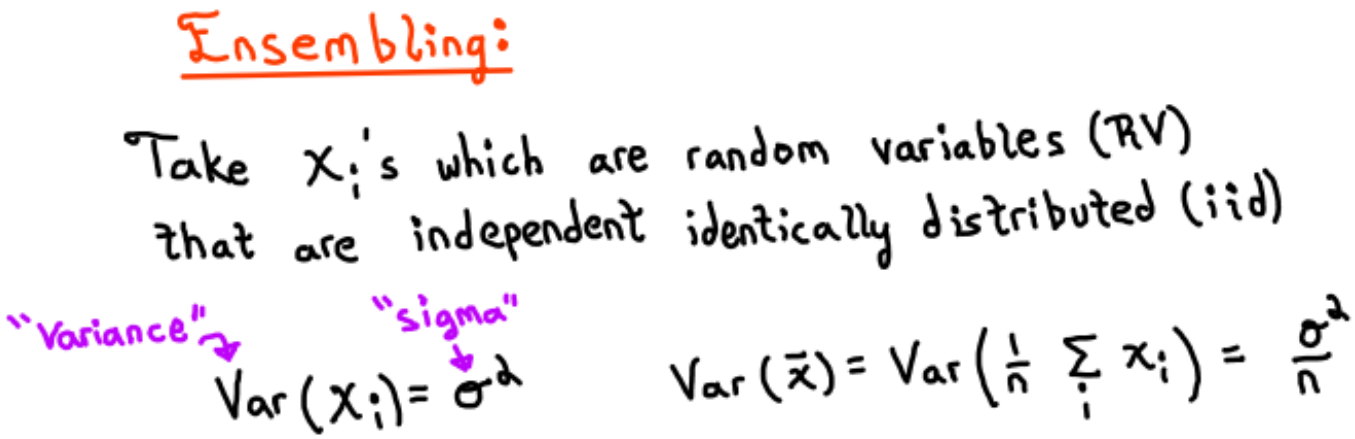


You can make ***decision trees*** a lot better through **ensembling**. A lot of the methods, for example, the leading methods in ***Kaggle*** these days are actually built on ***ensembles*** of ***decision trees*** and they really provide an ideal sort of ***model framework*** to look at, through which we can examine a lot of these different ***ensembling methods***

For the ***cross-entropy loss***, does the **log** need to be **base 2**?
It's not very relevant in this case. ***Cross-entropy loss*** actually initially came out of ***information theory*** where you have computer bits and you're transferring bits, so it's useful to think in terms of bits of information that you can transmit, which is why it came up as log base 2 in the initial formulation

Ensemble Methods:

Why does ***ensembling*** help? At some level, you can sort of think back to your basic statistics. So say you have:

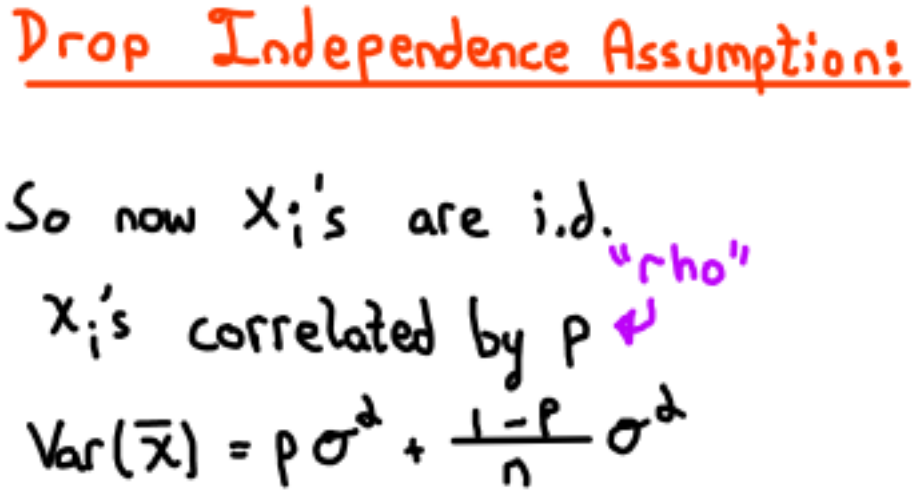


And so, each ***independent variable*** you factor in is decreasing the ***variance*** of your model. The thought is that if you can factor in a number of independent sources, you can slowly decrease your ***variance***

Though this is a little bit simplistic of a way of looking at this because really, all these different things are factoring together, have some

amount of correlatio with each other, and so this independence assumption is often times not correct

So if instead, you drop the independence assumption and say you characterize what the correlation between any two $\mathbf{X_i}$'s is, we can write that down as:



You can see that if they're fully correlated, then the **term on the right** will drop to **0** and you'll just have **sigma squared** (σ^2) again because adding a bunch of fully correlated variables is just gonna give you the original variable's **variance** versus if they're completely decorrelated, then the **term on the left** drops to **0** and you just end up with **sigma squared** (σ^2) over **n**, which gives you the initial ***independent identically distributed equation***

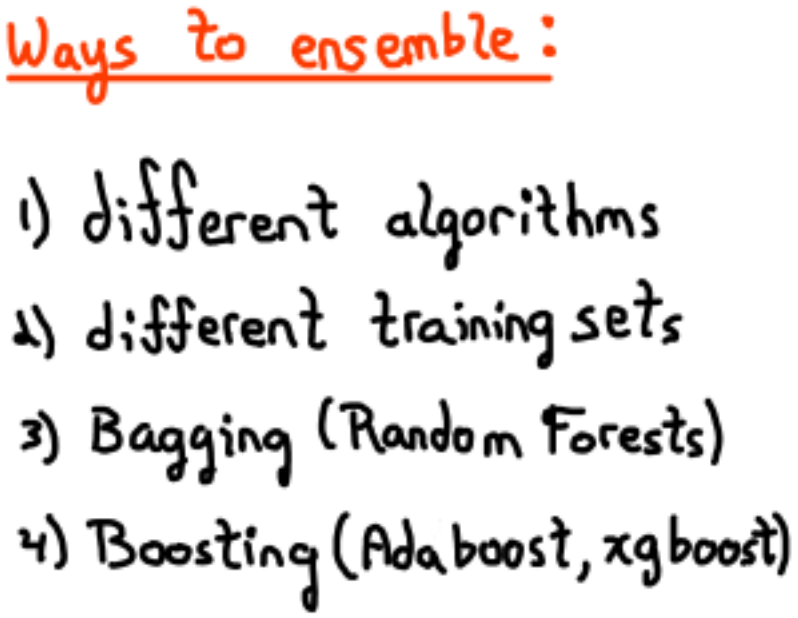
In this case, what you really want to do is you want to have as many different models that you're factoring as possible to increase the **n**, which drives the ***term on the right*** down, and then on the other hand, you also want to make sure those models are as decorrelated as possible so that your **rho** (**P**) goes down and the ***term on the left*** goes down as well

This gives rise to a number of different ways to **ensemble**

One way you could think about doing this is you just use different algorithms. This is actually what a lot of people in **Kaggle**, for example, will do is they'll just take a ***neural network*** or ***random forest***, an **SVM**, average them all together and generally that actually works pretty well, but then you sort of have to spend your time implementing all these separate algorithms, which is oftentimes not the most efficient use of your time

Another one that people would like to do is just use different ***training sets***, and again in this case, you probably spent a lot of effort collecting your initial ***training set***. You don't want your machine learning person to just come and recommend to you that you should just go and collect a whole second ***training set*** or something like that to improve your performance. That's generally not the most helpful recommendation

One of the other two methods is called ***Bagging***, which is sort of trying to approximate having different ***training sets***, and then you also have **Boosting**. For context, with ***bagging***, you might have heard of ***random forests***, which is a variant of ***bagging*** for ***decision trees***, and then for ***boosting***, you might have heard of things like ***AdaBoost*** or ***xgboost***, which are variants of ***boosting*** for ***decision trees***

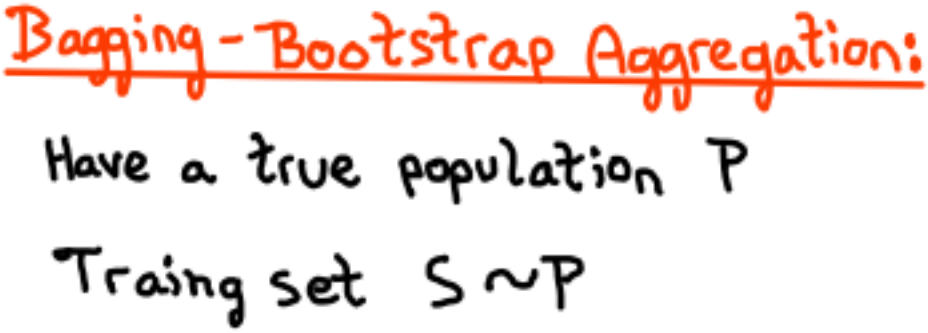


These first two are very nice because they sort of would give us less correlated variables, but generally we end up doing these latter two because we don't want to collect new ***training sets*** or train entirely new algorithms

Bagging:

Bagging stands for this thing called ***Bootstrap Aggregation***

To break down this term, ***Bootstrap*** is typically this method used in statistics to measure the uncertainty of your estimate, and so, what is useful to define in this case for when you're talking about ***bagging***, is that you can say that you:



Ideally, for example, with the different ***training sets*** approach from ***enembling***, what you do is you just draw $\mathbf{S_1}$, $\mathbf{S_2}$, **...**, and then train your model in each one of those separately. Unfortunately, you generally don't have the time to do that. And so, what ***bootstrapping*** does is you assume that your ***population*** is your ***training sample***. Now that you have this **S** approximating **P**, then you can draw new samples from your population by just drawing samples from **S** instead

Bagging - Bootstrap Aggregation:

Have a true population P

Traing set $S \sim P$

Assume $P = S$

Bootstrap samples $Z \sim S$

How that works is you basically just take your ***training sample***, say it's of cardinality **n** or something, and you just sample **n** times from **S** and, this is important, you do it with replacement. Because you're pretending that this (**S**) is the population and so, doing it with replacement sort of makes that assumption hold, that you're sampling from it as a population

That's ***Bootstrapping***, you generate all these different ***bootstrap samples* Z** from your ***training set*** and what you can do is you can take your model and train it on all these separate ***bootstrap samples***, and then you can look at the variability in the predictions that your model ends up making based on these different ***bootstrap samples***, and that gives you a measure of uncertainty

What we want to use ***bootstrapping*** for is we want to ***aggregate*** these two ***bootstrap samples***, and so at a very high-level, what that means is we're going to take a bunch of ***bootstrap samples***, train separate models on each, and then average their outputs

To make this a bit more formal:

Bootstrap Samples $Z_1, \ldots, Z_M$

Train model G_m on Z_m

$$G(m) = \frac{\sum_{m=1}^{M} G_m(x)}{M}$$

M is how many bootstrap samples you're going to take

G(m) is your meta model

You're taking these ***bootstrap samples*** and then you're training separate models, and then you're just ***aggregating*** them all together to get this ***bagging*** approach

If we just do a little bit of analysis from the ***bias-variance perspective*** on this, we can sort of see why this kind of thing might work

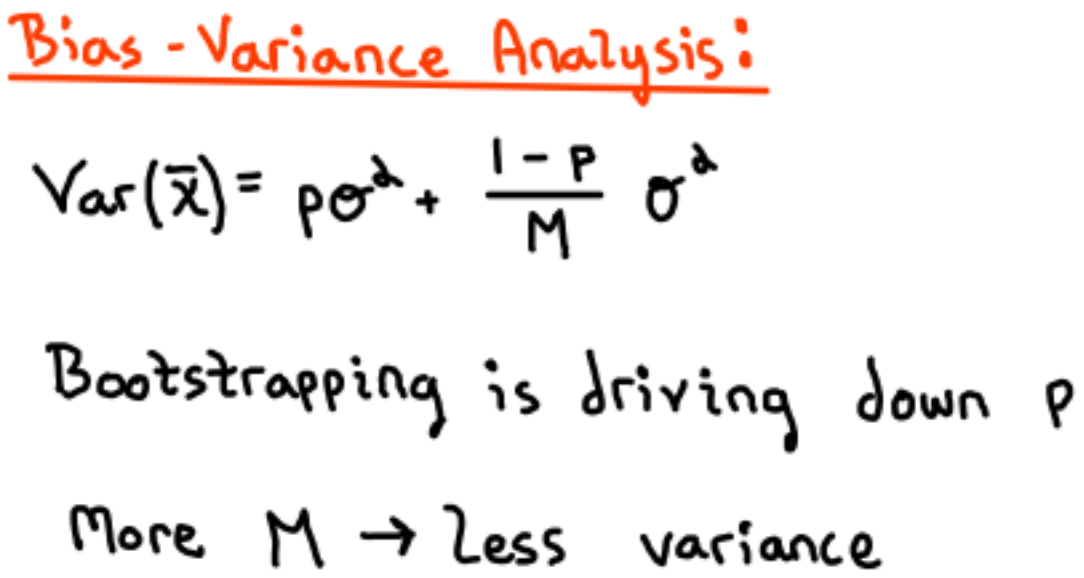
M is the number of ***bootstrap samples***

What you're doing is by taking these ***bootstrap samples***, you're sort of decorrelating the models you're training

By driving down ***Rho***(**P**), you're making the ***term on the left*** get smaller and smaller

It turns out that basically you can take as many ***bootstrap samples*** as you want and that'll increases **M** and drive down the ***term on the right***

One nice thing about ***bootstrapping*** is that increasing the number of ***bootstrap models*** you're training, doesn't actually cause you to ***overfit*** anymore than you were beforehand because all you're doing is you're driving down the ***term on the right***, so more **M** is just less ***variance***. All you're doing is driving down the ***term on the right*** as much as possible when you're getting more and more ***bootstrap samples***. So generally, it only improves performance and so generally what people will do, is they'll sample more and more models until they see that their error stops going down, because that means they basically eliminated the ***term on the right***



 # Can you define a bound on how much you decrease Rho(P) by?

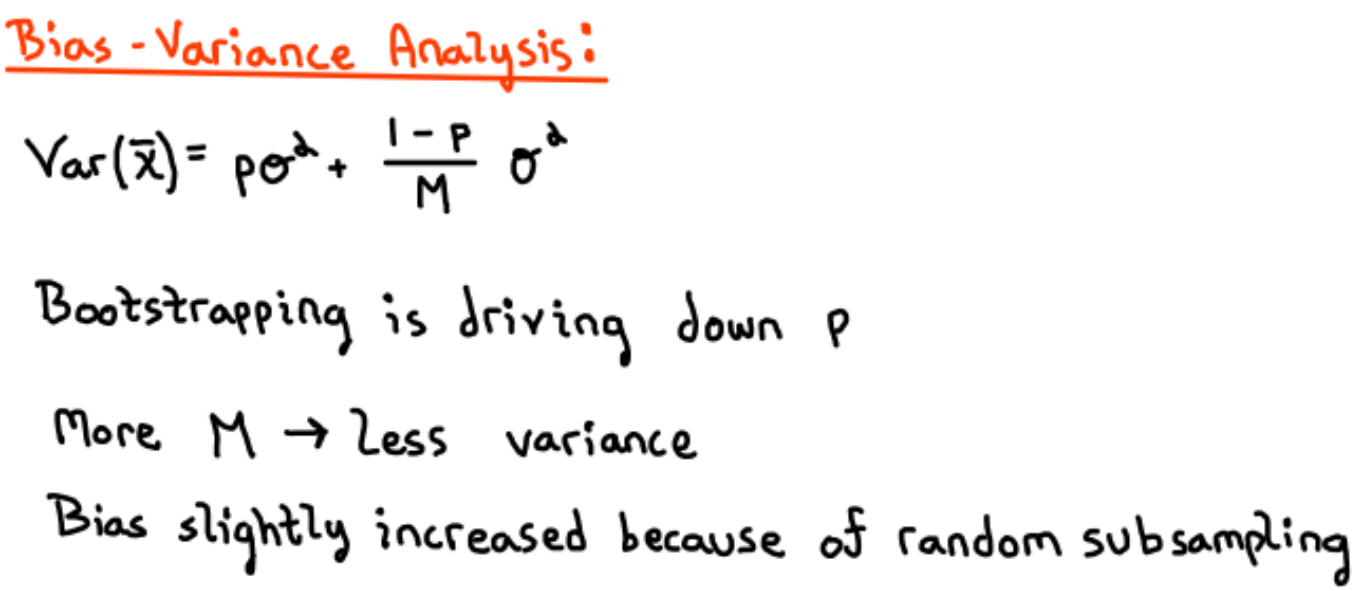
There's definitely a lower bound on how far you can decrease rho(**P**). Basically it comes down to, your bootstrap samples are still fairly highly correlated with one another because you're still just drawing it from the same ***sample set* S**. Really, each **Z** is going to end up containing about two-thirds of **S**, and so your **Z**s are still going to be fairly highly correlated with each other. You can intuitively see that there is a bound there and that you can't just magically decrease rho (**P**) all the way down to **0** and achieve **0** ***variance***

 # Can you explain the difference between a random variable and an algorithm in this case?

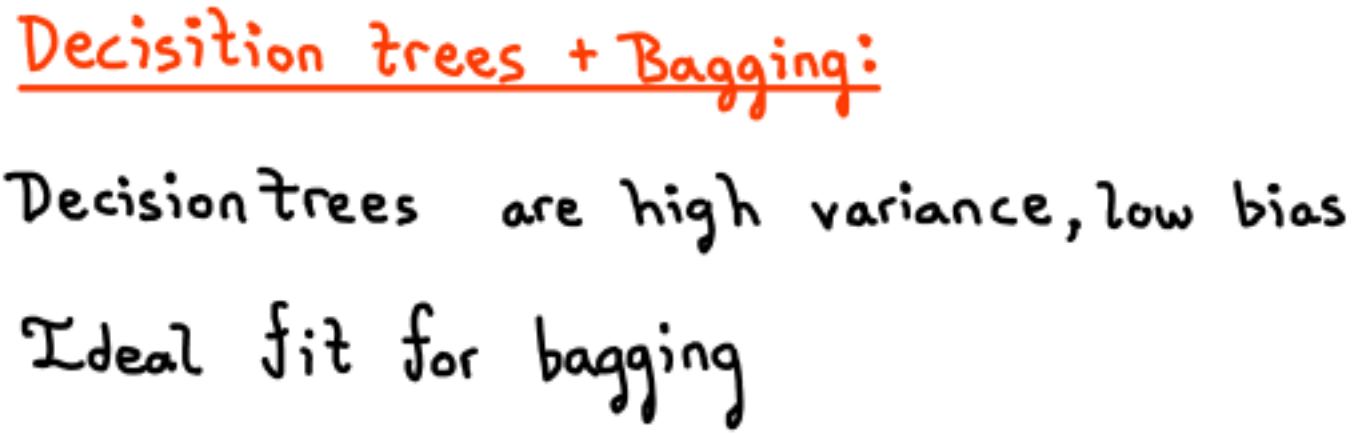
At a very high-level, you can think of an algorithm as a function that's taking in some data and making a prediction. If you sort of see that whole setup as sort of like probability, the algorithm is giving some sort of output in the probabilistic perspective, you can sort of see the algorithm as a random variable in this case. You're basically considering the space of possible predictions that your algorithm can make, and that you can see as a distribution of possible predictions, and that you can approximate that as a random variable. It is a random variable at some level because it's based on what training sample you end up with, your predictions of your output model are going to change. Since you're sampling these random samples from your population set, you can consider your algorithm as sort of based on that random sample and therefor a random variable itself

This seems kind of nice, you're decreasing the variance, but where is the trade-off coming in?

One issue that comes up with ***bootstrapping*** is that in fact, you're actually slightly increasing the ***bias*** of your models when you're doing this. The reasoning for that is because of this ***sub-sampling***. Each one of your **Z**s is now about two-thirds of the original **S**. So you're training on less data and so your models are becoming slightly less complex and so that increases your bias in this case because of random ***sub-sampling***, but generally the decrease in ***variance*** that you get from doing this, is much larger than the slight increase in ***bias*** you get from doing this ***randomized sub-sampling***, so in a lot of cases, ***bagging*** is quite nice



Recall that ***Decision Trees*** are ***high variance*** and ***low bias***. This right here sort of explains why they're a pretty good fit for ***bagging*** because with ***bagging***, what you're doing is you're decreasing the ***variance*** of your models for a slight increase in ***bias***, and since most of your error from your ***decision trees*** is coming from the ***high variance*** side of things, by driving down that ***variance***, you get a lot more benefit than for a model that would be on the reverse ***high bias*** and ***low variance***. So this makes this like an ideal fit for ***bagging***



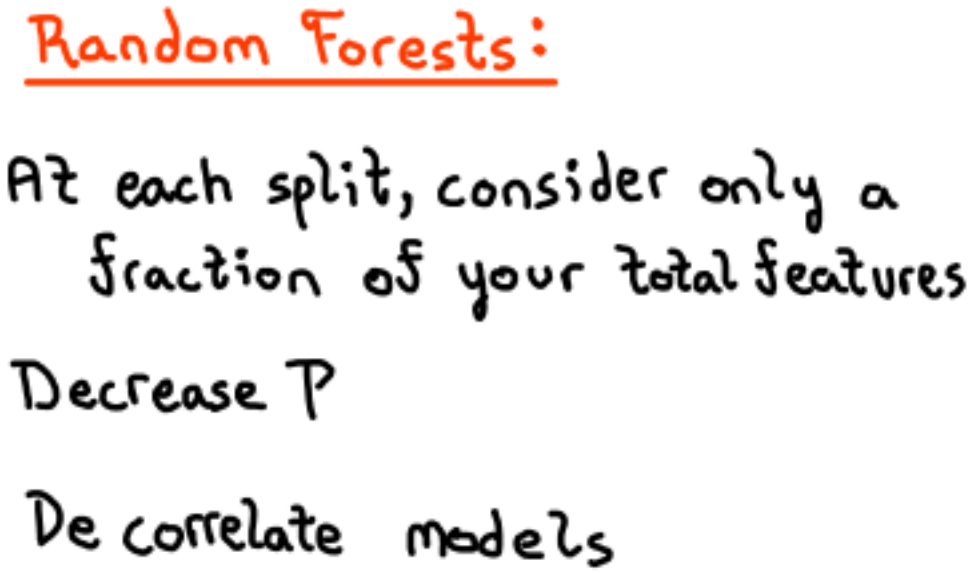
Random Forests:

Random forests are a version of ***decision trees + bagging***. What was described above is almost a ***random forest*** at this point. The one key point we're still missing is that ***random forests*** actually introduce even more randomization into each individual ***decision tree***. The idea behind that is that you can only drive rho (**P**) down so far through just pure ***bootstrapping***, but if you can further decorrelate your different ***random variables***, then you can drive down that ***variance*** even further

The idea is that basically at each split for ***random forests***, you consider only a fraction of your total features

It's sort of like, for the ski example, maybe for the first split you only let it look at latitude, and then for the second split you only let it look at the time of the year

This might seem a little bit unintuitive at first, but you can sort of get the intuition from two ways. One is that you're decreasing Rho (**P**) and then the other one is you can think that, say you have a classification example where you have one very strong predictor that gets you very good performance on its own and regardless of what bootstrap sample you select, your model is probably going to use that predictor as its first split. That's going to cause all your models to be very highly correlated right at that first split, for example, and by instead forcing it to sample from different features instead, that's going to decrease the correlation between your models. So it's all about decorrelating your models in this case



Boosting:

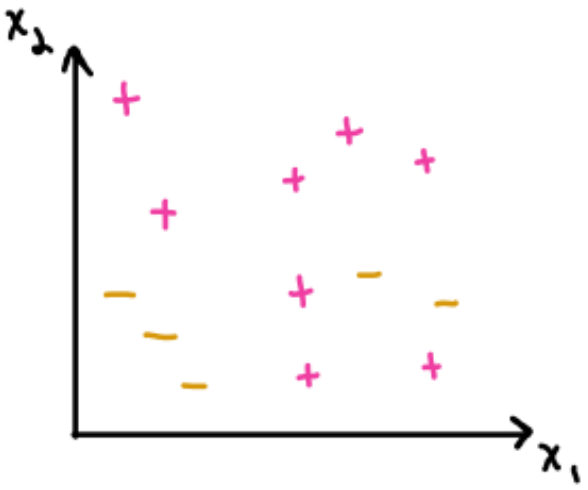
Basically, whereas ***bagging***, we sort of saw in the intuition that we were decreasing **variance**, ***boosting*** is sort of actually more of the opposite where you're decreasing the ***bias*** of your models and also it's basically more ***additive*** in how it's doing things

You'll recall that for ***bagging***, you were taking the average of a number of variables. In ***boosting***, what happens is that you train one model and then you add that prediction in to your ***ensemble***, and then when you train a new model, you just add that in as a prediction

Say you have a dataset and some data points

Boosting:

Decreasing bias
Additive

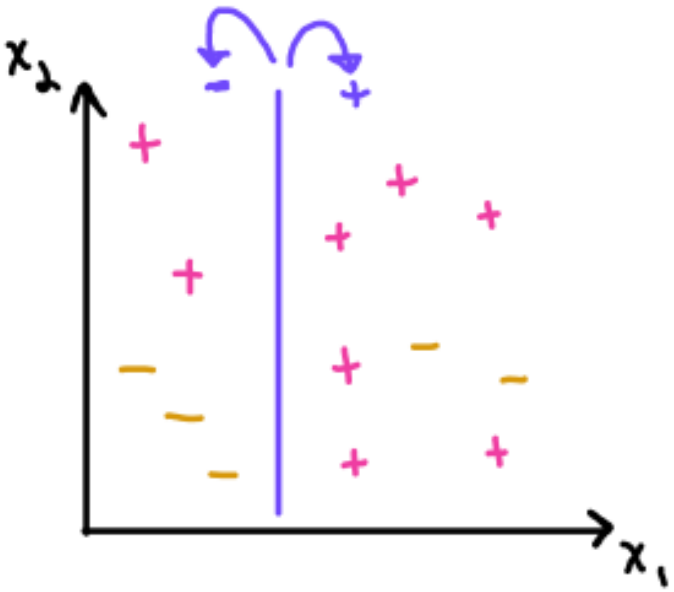


Say you're training a size **1** ***decision tree*** (called ***decision stumps***), so you only get to ask one question at a time, and the reason behind this is that because you're decreasing ***bias*** by restricting your trees to be only depth **1**, you basically are increasing their amount of ***bias*** and decreasing their amount of ***variance***, which makes them a better fit for ***boosting*** kind of methods

Say that you come up with a decision boundary and on each side you're going to predict either positive or negative. This is a reasonable line that you could draw here but it's not perfect, you've made some mistakes

Boosting:

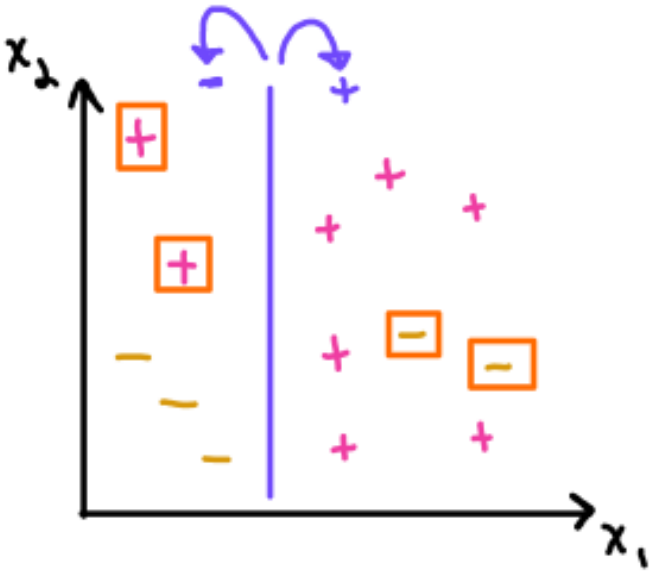
Decreasing bias
Additive



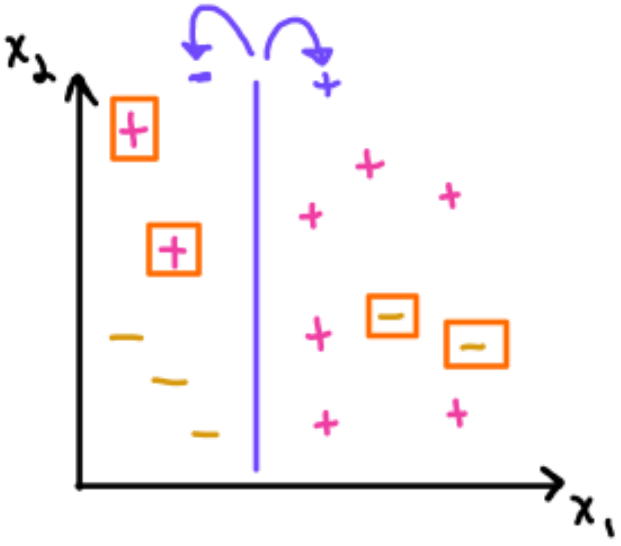
What you can do is you can sort of identify these mistakes

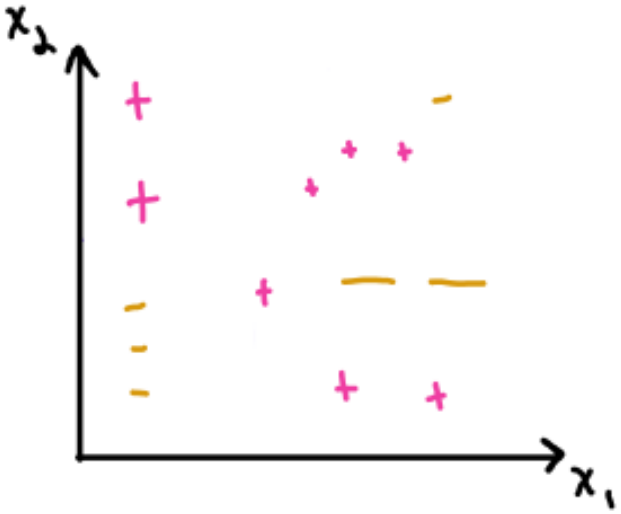
Boosting:

Decreasing bias
Additive

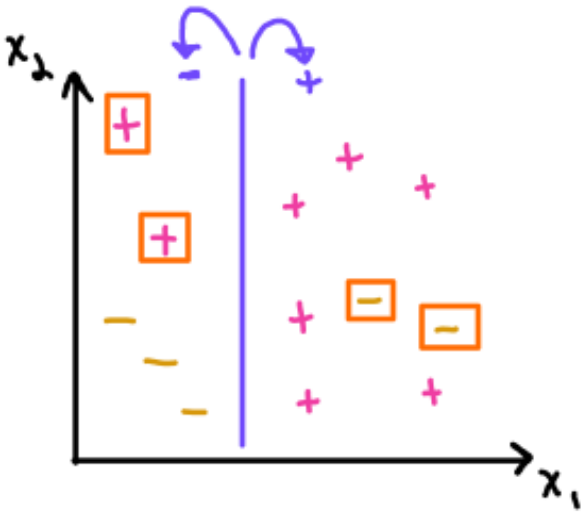


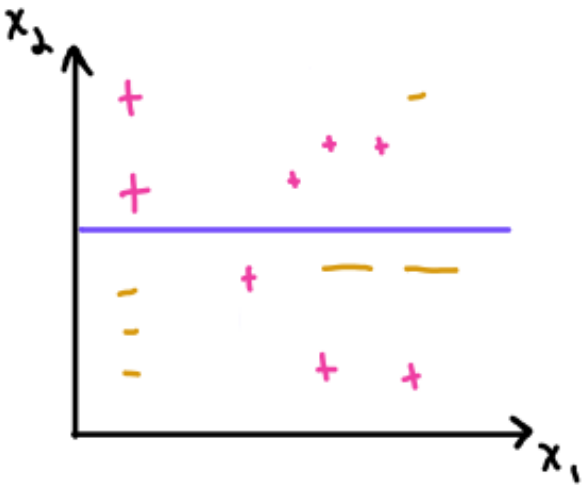
What ***boosting*** does is basically it increases the weights of the mistakes you've made, and then for the next ***decision stump*** that you train, it's now trained on this modified set





And so now your model, to try and get these right, might pick a decision boundary like this:

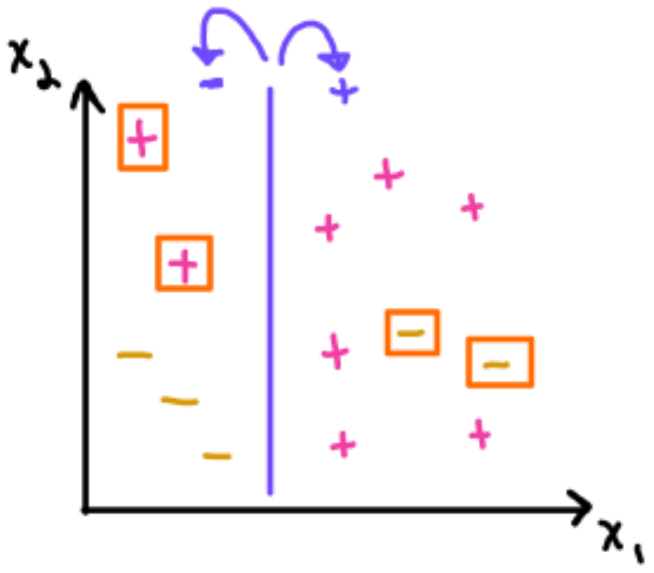


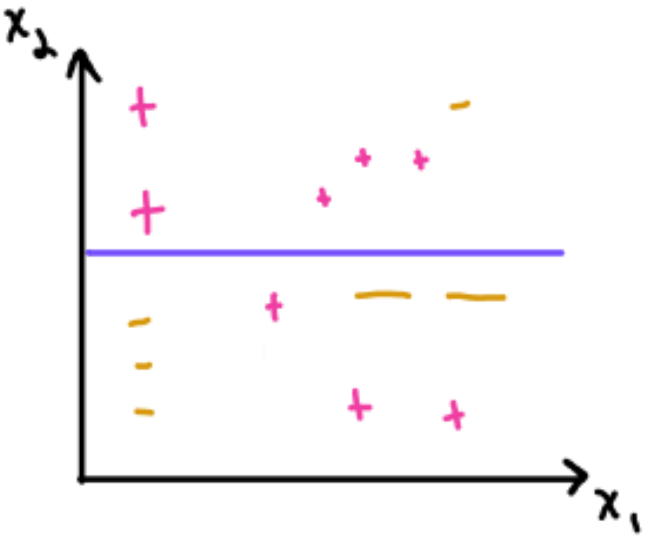


This is also basically recursive in that each step, you're going to be reweighting each of the examples based on how many of your previous ones have gotten it wrong or right in the past

Basically what you're doing is you can sort of weight each one of these classifiers, you can determine for classifier G_m a weight α_m which is proportional to how many examples you got wrong or right. So a better classifier, you want to give it more weight and a bad classifier you'll want to give it less weight

The exact equation used in ***AdaBoost*** is:

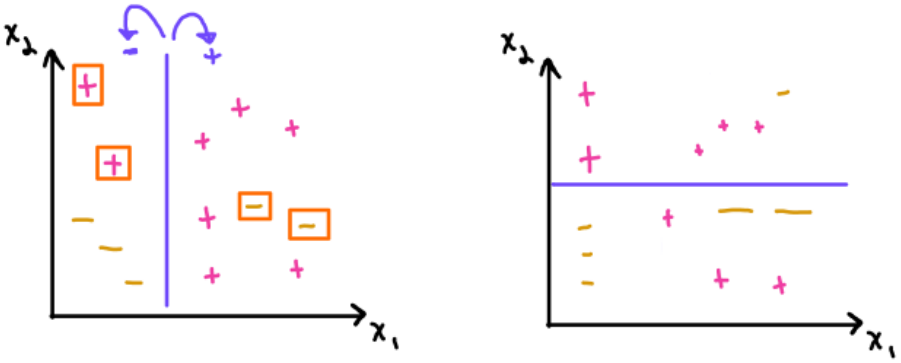




Determine for classifier G_m a weight α_m

proportional $\log\left(\frac{1-err_m}{err_m}\right)$

And then your total classifier is just:



This algorithm is actually known as ***AdaBoost***

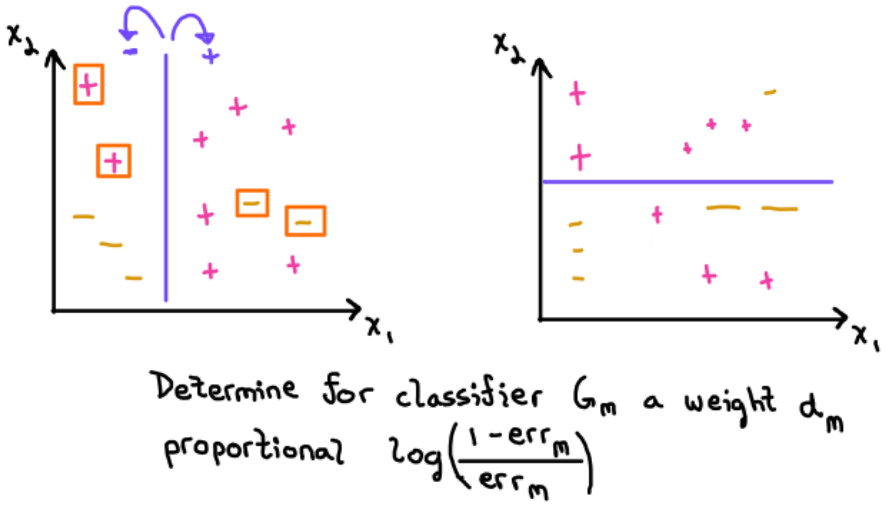
Bias - Variance Analysis:

$$Var(\bar{x}) = \rho\sigma^2 + \frac{1-\rho}{M}\sigma^2$$

Bootstrapping is driving down ρ

More $M \rightarrow$ less variance

Bias slightly increased because of random subsampling



Basically through similar techniques, you can derive algorithms such as ***XGBoost*** or ***Gradient Boosting machines*** that also allow you to basically reweight the examples you're getting right or wrong in this sort of dynamic fashion, and slowly adding them in this additive fashion to your composite model