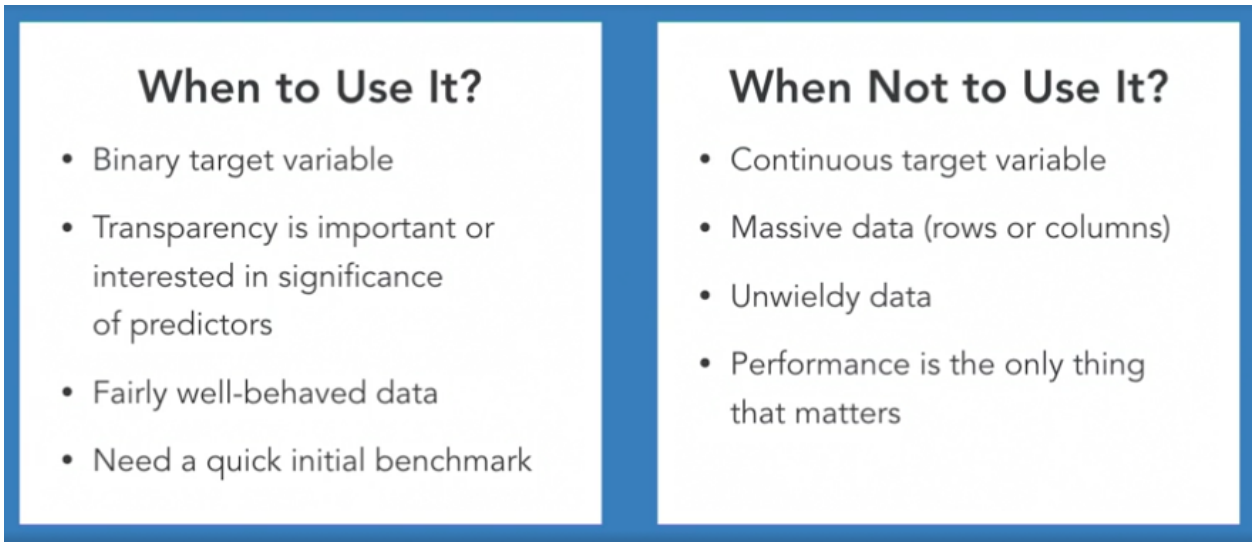
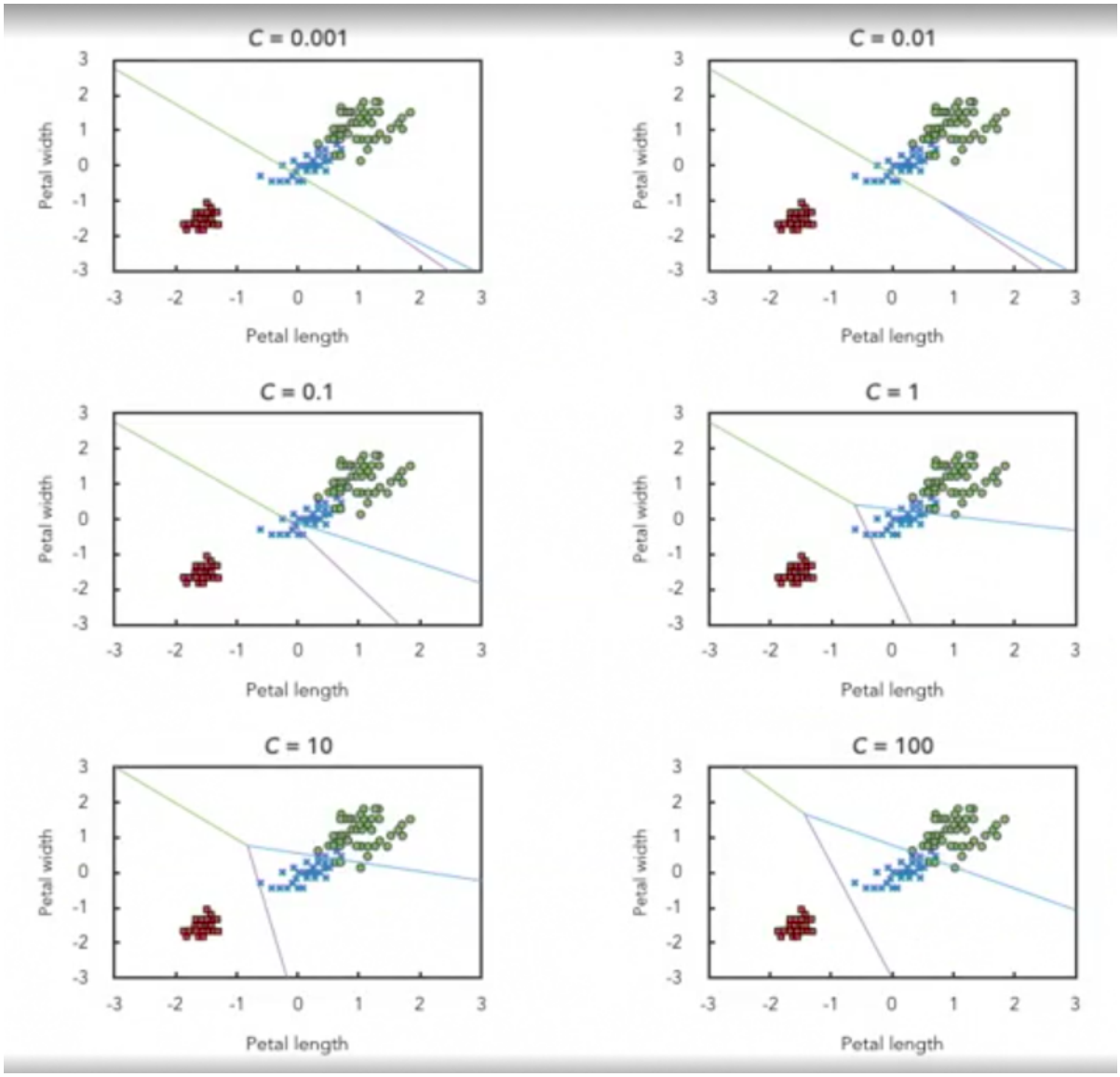
LOGISTIC REGRESSION



What logistic regression is great for is an initial benchmark model on a binary classification problem with fairly well-behaved data. It's relatively flexible and it's very fast to train. While it doesn't usually generate the greatest performance, it does set a very nice benchmark to start. So we talked about when you should use this algorithm, now when should you not use this algorithm? You should not use this algorithm when you have a continuous target variable. Remember that S shape between zero and one? That would not work very well for the umbrella rainfall example that we saw before, or really any other modeling of continuous variables. Logistic regression also shouldn't be a first choice if you have a massive amount of data. There are other algorithms that really shine when you have a ton of data, logistic regression just isn't one of them. I also want to call out that there are many different shapes of data. So there's what's called short and fat data, that's when you have lots of features but very few rows, or you can have long and skinny data, where you have a lot of rows, but very few features. I would probably avoid logistic regression in either of these extreme cases. As I mentioned before, logistic regression shouldn't really be considered when you have data with a lot of outliers, missing values, skewed features, or really complex relationships. Logistic regression is a relatively simple algorithm, and typically won't do a great job on these types of problems. Lastly, there are things that logistic regression are good for but generally speaking, it's not going to be the best performing algorithm on any given use case. It will usually do pretty well on any

given problem, but it will rarely be the best. So just to recap, logistic regression is a great tool for a fast, transparent baseline model for binary classification problems. It doesn't do well with a lot of data or particularly messy data. It's also unlikely to give you the best possible performance on any given problem.



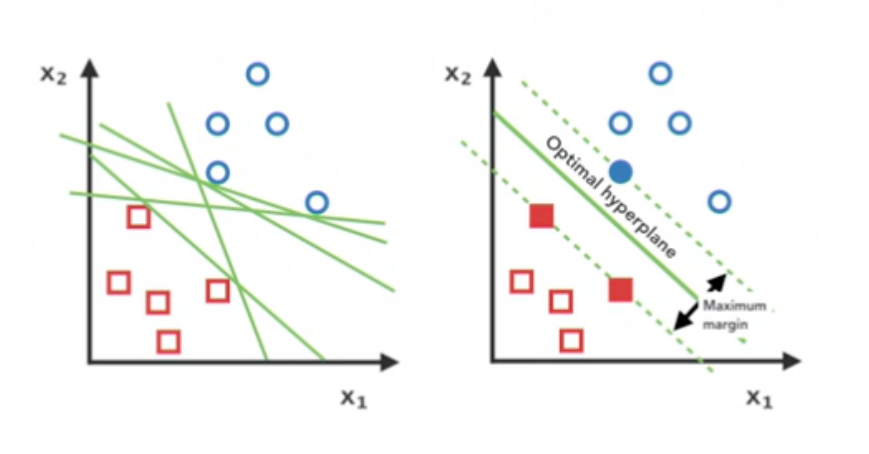
based on the pedal length and the pedal width. So ideally what we would want to see is down here in the bottom right. This model is properly drawing a boundary between each of these three groups. So the green points kind of have their own segment. The blue points have their own segments and lastly the red points have their own portion of the plot. Now it's achieving that with a high value of C of 100, which means low Lambda and low regularization. Now, as we kind of cycle through these plots and we make C smaller and smaller, in other words that means we're increasing the regularization, this given models having a harder and harder time correctly classifying these flower types. If you look all the way to the top left, that's the smallest C parameter that we have, which is going to mean the highest regularization that we have. The models having a very hard time splitting these into three groups. Essentially all it does is it splits the plot in half, and it really doesn't understand how to split these points into different segments. So again, very low C value means high regularization and means more likely to under fit. Now recall the default setting is at one but we can see based on this plot that the model actually does a better job grouping these when C is equal to a hundred. So that is why hyperparameter tuning is necessary here. So we'll dive into testing logistic regression on our data in the next lesson.

SUPPORT VECTOR MACHINE (SVM)

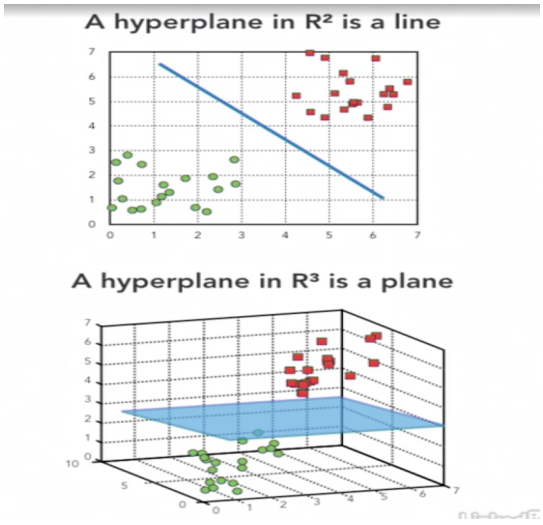
A support vector machine is a classifier that finds an optimal hyperplane that maximizes the margin between two classes.

What line gives us the best chance to classify additional examples in each of these two classes. You would probably want a line that is evenly spaced between the two classifications to give a little bit of buffer.

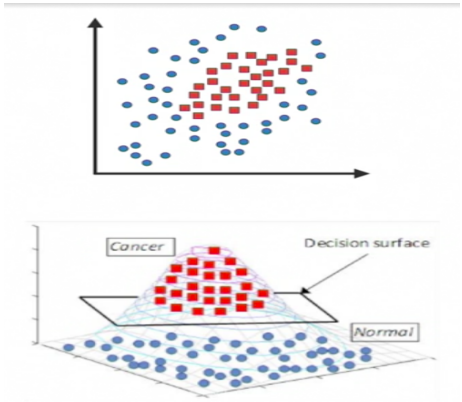
SVM seeks to maximize the distance between the decision boundary and the nearest points.



A hyperplane is a generalized term to identify your decision boundary in a N dimensional space.



The kernel trick (or kernel method) transforms data that is not linearly separable in n-dimensional space to a higher dimension where it is linearly separable. We are using a straight line in 2d, and a flat hyperplane in 3d. We are not using any curved lines. So what happens when we have data that can not be separated by a straight line or hyperplane?
This leads us to the kernel trick. Below we have data where we are trying to identify cancer based on gene X and gene Y. It clearly is not linearly separable in two dimensions.
When we project into 3d we can now use a flat hyper-plane to separate our data nicely and identify cancer.



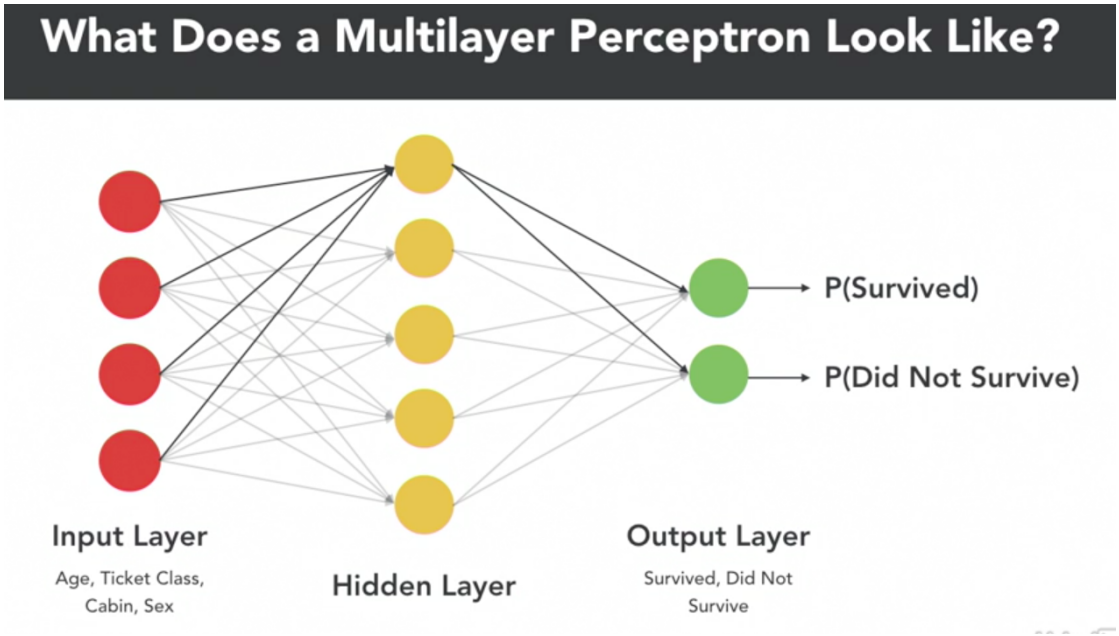
SVM

In summary, SVM should be used when you have a lot of features, but few rows, or when you have a lot of outliers, or if you just have a lot of complex relationships that you're trying to untangle. But it is quite slow to train. So if you have a ton of data or limited compute power, you might want to look somewhere else.

WHAT IS MULTILAYER PERCEPTRON

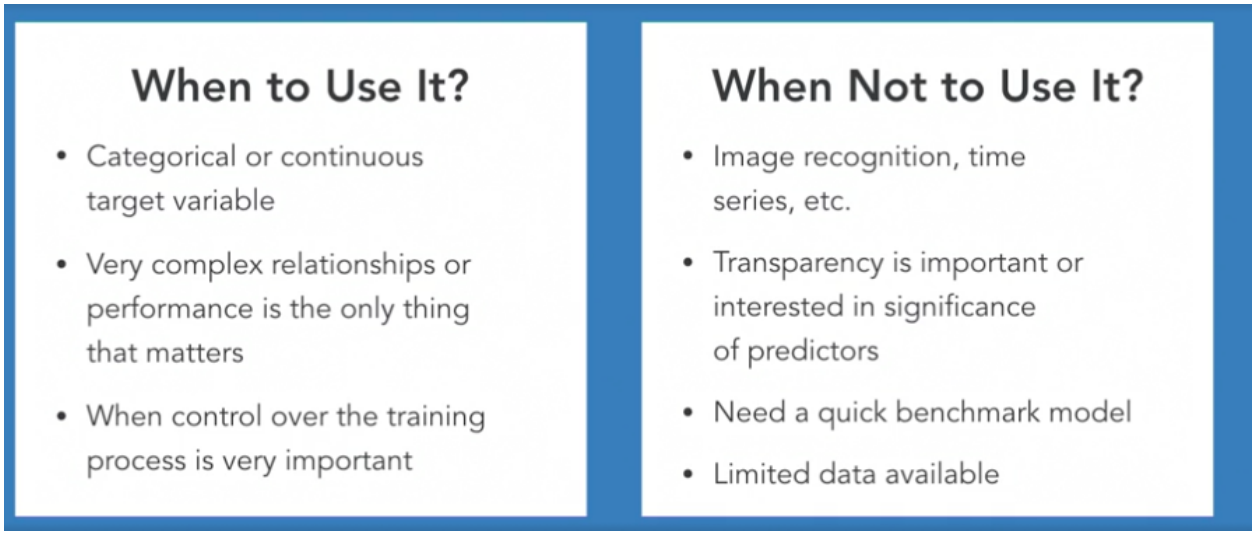
A multilayer perceptron is a classic feed-forward artificial neural network, the core component of deep learning.
Alternatively: A multilayer perceptron is a connected series of nodes (in the form of a directed acyclic graph), where each node represents a function or a model.
Another way to look at this is that a multi-layer perceptron is a connected series of nodes where each node represents a function. This connected series of nodes creates what's called a directed acyclic graph, meaning that there's directionality between the nodes and no node will ever be revisited.

We pipe those into each node in the hidden layer. So what's happening here is each node or function in the hidden layer is getting each input featured. So age, ticket class, cabin, and sex are passed into each node in the hidden layer. So you could view each of these nodes as something like logistic regression. So more or less a logistic regression model would be fit for each node, maybe with slightly different weights. Then we have an output layer and here we're going to have two nodes and they represent the two possible outcomes



So you can see it's receiving the four inputs. Then this one node in the hidden layer will basically fit some model or learn some aspect of the data. Then based on that model that it learns, it'll make some predictions about whether somebody survived or not. So maybe it's 70% likelihood that they survived, 30% likelihood that they did not. So that's one individual node in the hidden layer, but we could repeat this slide for each of the five nodes in the hidden layer. So then at the end, you put this all together and you get a final prediction of the likelihood of a given person surviving. As you can imagine, these can be incredibly powerful. As you see in this example, you're essentially aggregating the predictions of five different models altogether. This allows the overall model to learn some really powerful relationships in the data.

MULTILAYER PERCEPTRON



To summarize, multi-layer perceptrons are often the best tool for the job when you have a lot of data and you really only care about performance, but they're not ideal if you have limited data, if you don't have much time or compute power, or you really care about the transparency of the model in understanding its predictions.

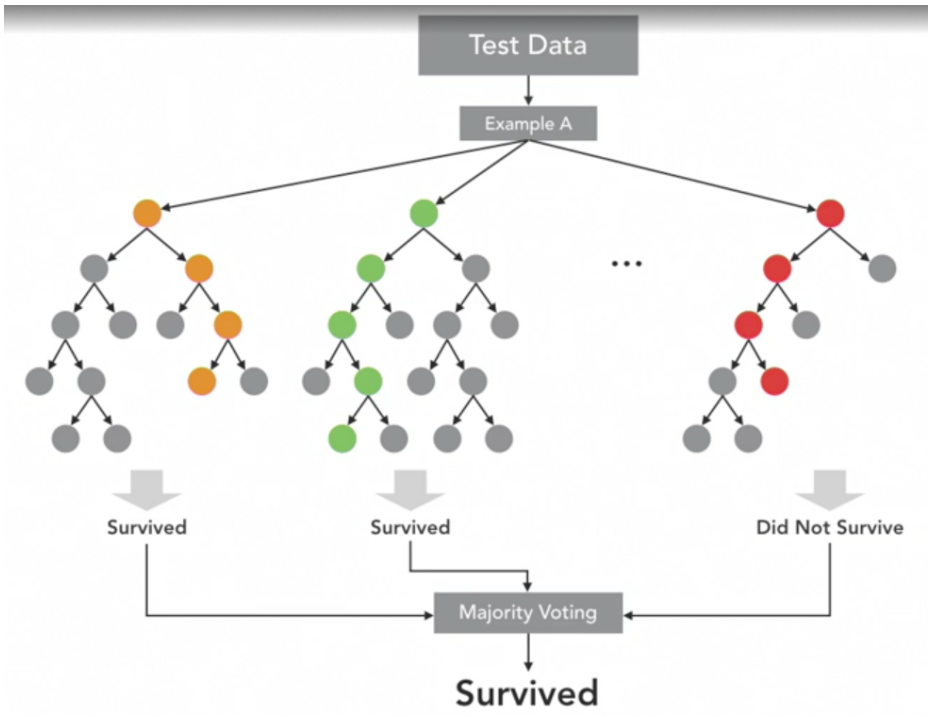
RANDOM FOREST

A random forest merges a collection of independent decision trees to get a more accurate and stable prediction.
Now, keep that word independent in the back of your mind. It's actually quite important. And I'll explain why in a couple minutes. So, random forest is a type of ensemble method, and ensemble combined several machine learning models to decrease both bias and variance.

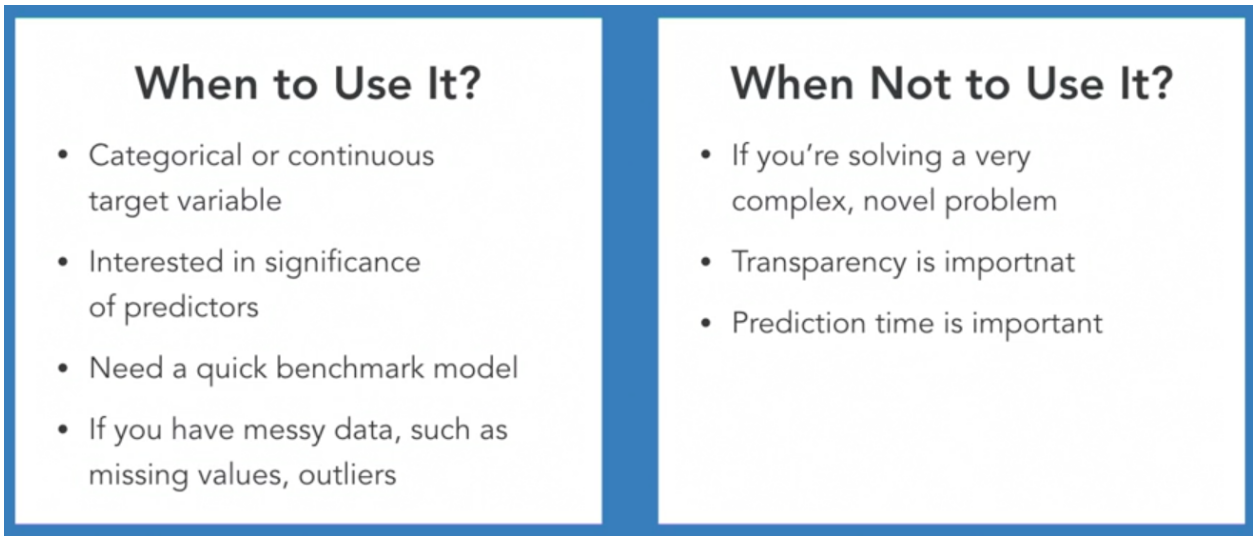
The overall idea here is that multiple models are fit and each key in on different aspects of the data. And combining all of these models ultimately generates better predictions together than any of those single models on their own.

In training, this algorithm will take N samples from the training data. I just want to mention that this is sampling with replacement. So, that means one single example is likely to appear in multiple different samples. Then the next step is you take a sample of features to be used for each of the data samples. So, in our Titanic dataset, if we have seven features in each of these data samples, maybe we'll sample four or five of

those features to be used. So, now we have N subsets of our overall data, and those subsets contain both a subset of the rows and also a subset of the columns. Then for each feature and data subset, the algorithm will build a decision tree to try to generate the most accurate results. And it's important to realize that these decision trees are all developed on their own. They do not know what any of the other trees are doing.



So, perhaps looking at that left-hand most tree, maybe that's saying the example A has a cabin, then move on to the second layer of the tree. Maybe it says this person paid over $50, and then maybe we move on to the third layer and say, "Okay, this person is a female." Based on those features, this model will predict that this person would survive. Now, the splits and the features on the second tree will be totally different. So, it's possible to get a different prediction. So, you'll now have N predictions for example A, one from each tree. Then the random forest algorithm will basically just aggregate all of those predictions together. And based on voting with majority rules, it will make a final prediction. So, in this case, it would just say that the person survived.



While it's nice to understand the features and the decision trees on their own can be fairly easy to understand, when you have a random forest of hundreds of decision trees, it can be hard to interpret what's really happening across that entire model. So if you need to see the details within the model, random forest might not be the right choice for you. So random forest is relatively quick to train, but it's not quite as quick to make predictions. Overall, random forest is a tremendously flexible, relatively fast tool that plays well with different data and could give you relatively good performance. It doesn't always give you the best performance, but it is a really nice Swiss Army Knife to have in your tool set.

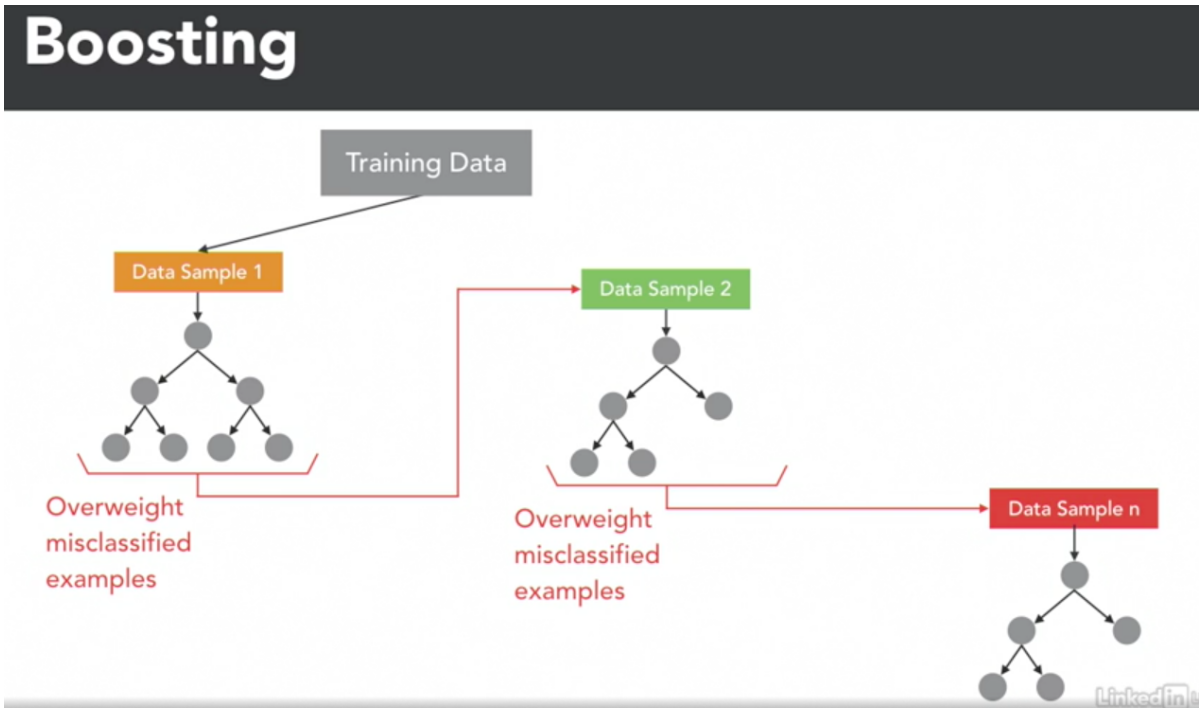
WHAT IS BOOSTING?

Boosting is an ensemble method that aggregates a number of weak models to create one strong model.

A weak model is one that is only slightly better than random guessing. (Roughly 50% accuracy). A strong model is one that is strongly correlated with true classification. Boosting effectively learns from its mistakes with each iteration.

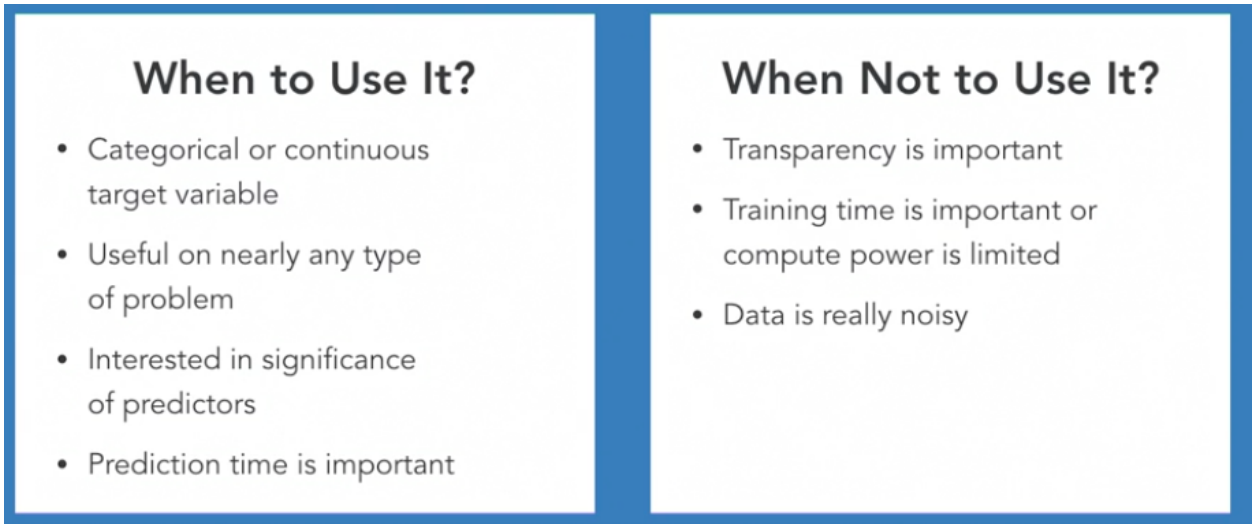
The big difference between random forest and boosting is that each individual model or decision tree in random forest was built independently. Each tree did not know what any of the other trees were doing. That is not the case with boosting. In boosting, each successive model learns from the mistakes of the ones before it so they're not independent.

Boosting is an umbrella term that covers many different variants. Were going to explore a specific type of boosting called gradient boosted trees.



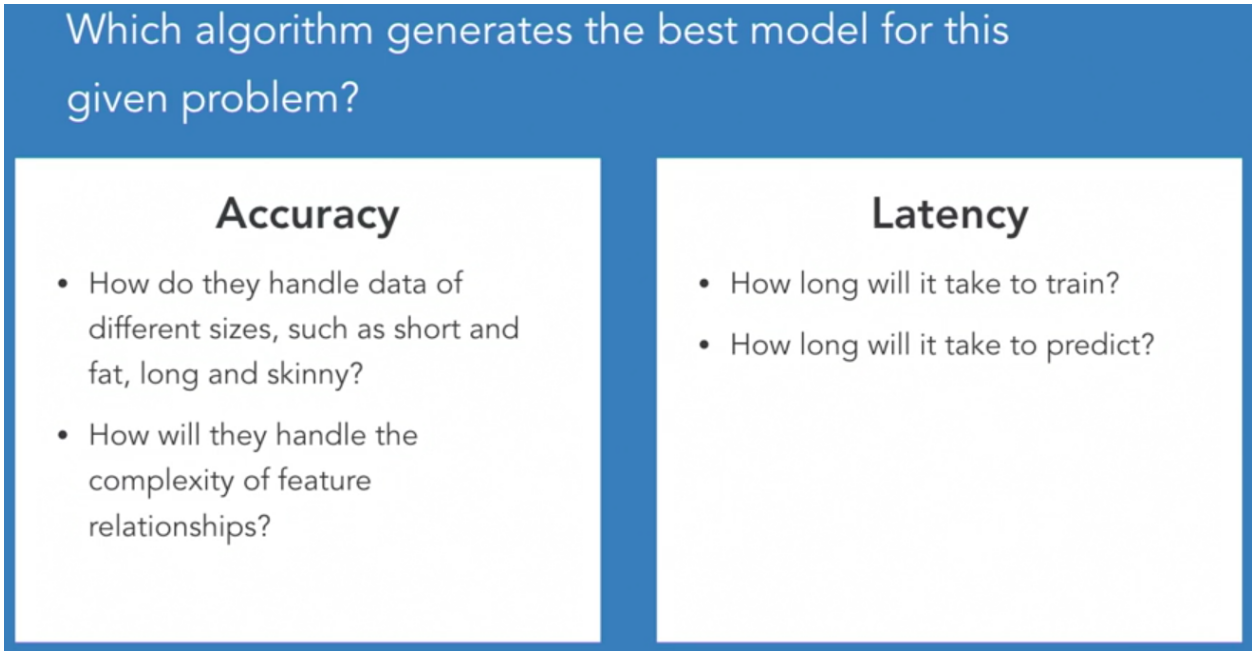
So you would start with training data and you would take one data sample from it, just like we saw in random forest. With that data sample, you would build a simple decision tree. I do want to call out that this tree is much more shallow than the trees in our random forest. Again, this is because we want each individual model to be a weak model and a shallow tree is more likely to be a weak model. And this is where the big difference between boosting and random forest comes in. At this stage, the algorithm will evaluate the performance of that weak model, and then it will re-sample while over-weighting the examples that were misclassified by that first model. Essentially it assumes that that first model has a good handle on any examples that it correctly classified. So this next model should focus on the ones that the first model couldn't quite figure out. So once the next sample is set with a higher proportion of examples that were misclassified by the prior model, now it builds a new weak model that hopefully keys in on what the prior model missed. And then I would repeat this evaluation and sampling process over and over. So each model will attempt to learn from the mistakes of the previous model. So by the end, you have N relatively weak models, but as a whole, they have learned from the mistakes of all prior iterations. So together they represent a very strong battle.

So what we would do is we would feed in the example from the test set and each model would end up generating a prediction, just like we saw with random forest. Now at this stage, remember in random forest, we just did a majority vote. In boosting, it's a little different. It's a weighted voting based on how well each individual model performed in training. Remember the after each week model was built in training, we evaluated it to figure out which examples it got wrong. Then this performance is used for weighting, these final votes, so that the best individual models get more say in the final prediction. Ultimately, the weighted vote ends up generating a final prediction. This model's ability to learn from its own mistakes, and then calibrate the weighted voting based on performance of each weak model makes this a tremendously powerful algorithm. There's one of the most used algorithms in machine learning.



So in summary, gradient boosting is one of the most flexible, powerful tools out there, and honestly you should really consider it for any problem. With that said, you do need to consider that it will take a long time to fit and you should be careful of its tendency to overfit.

WHY DO YOU NEED TO CONSIDER SO MANY ALGORITHMS?



Generally, these questions fall into two broad categories, accuracy and latency. So we'll start with latency. How long will this model take to train? We talked about how logistic regression is really fast to train, while gradient boosting and SVM are quite slow. So if you have limited compute power or limited time, then you'll want to lean towards logistic regression. The next question is how long will it take to make predictions? Again, we talked about how gradient boosting is quite fast in making predictions, so maybe if you have a lot of real-time data coming in that you need to make really fast real-time predictions on, then you might want to consider a gradient boosting. Now moving over to accuracy, what does our data look like? Is it short and fat? Is it long and skinny? For instance, if it's short and fat, in other words, it has a lot of features, but very few rows, we talked about how SVM does quite well in this space. The next question is just how complex is this data set? Are there clear connections that should be easy to pick up on? If so, then we should start thinking about prioritizing time to train or predict. However, if the relationships are really complex, then we'll probably start thinking about something like gradient boosting, multilayer perceptrons or SVM. Where they take a little longer to train, but they're much better at picking up on these really complex relationships. Lastly, how messy is our data? If there are a ton of outliers, then SVM is probably a good choice while gradient boosting probably is not, because it's likely to overfit to those outliers. So where does this leave us? Well, using the tendencies and strengths of the algorithms, we can kind of narrow down the algorithms considered for a given problem, but the bottom line is, sometimes we just don't know which algorithm will perform best. So we just need to test a few and see which actually does perform best on the data that we have. Generally, you'll start with maybe six or seven algorithms to consider.

	Problem Type	Train Speed	Predict Speed	Interpretability	Performance	Performance with Limited Data
Logistic Regression	Classification	Fast	Fast	Medium	Lower	Higher
Support Vector Machines	Classification	Slow	Moderate	Low	Medium	Higher
Multilayer Perception	Both	Slow	Moderate	Low	High	Lower
Random Forest	Both	Moderate	Moderate	Low	Medium	Lower
Boosted Trees	Both	Slow	Fast	Low	High	Lower