Machine Learning and AI with Python - Notes

CS109xa, Harvard

Alessio Santoro

January 15, 2025

Contents

Chapter 1

Decision Trees

1.1 Decision trees for classification

Logistic regression is a fundamental statistical method used in machine learning for binary classification tasks. It predicts the probability of an instance belonging to a particular class.

Part A: Classification using trees You may have learned in previous courses that **logistic regression** is most effective for constructing classification boundaries when:

- The classes are well-separated in the feature space
- The classification boundary possesses a simple geometry

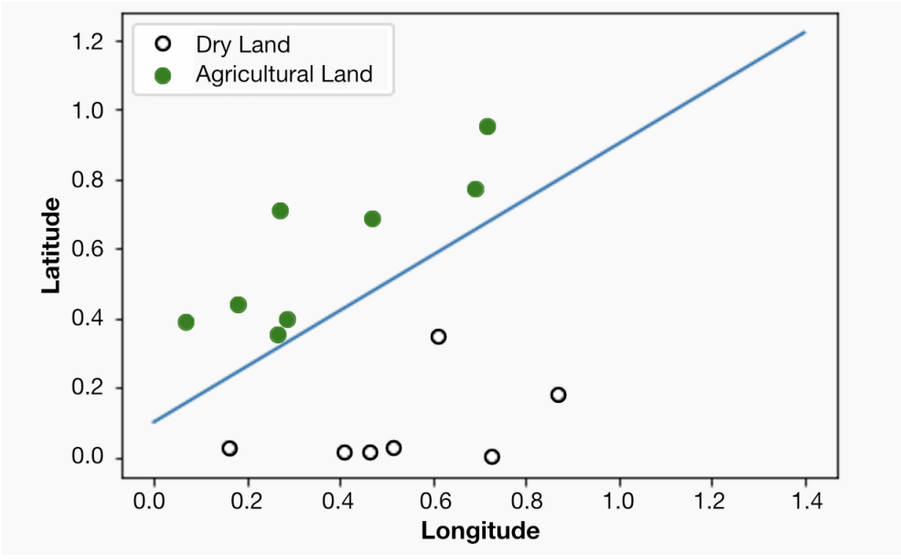
The **decision boundary** is determined at the point where the probability of belonging to class 1 is equal to that of class 0.

$$P(Y = 1) = 1 - P(Y = 0)$$
$$\rightarrow P(Y = 1) = 0.5$$

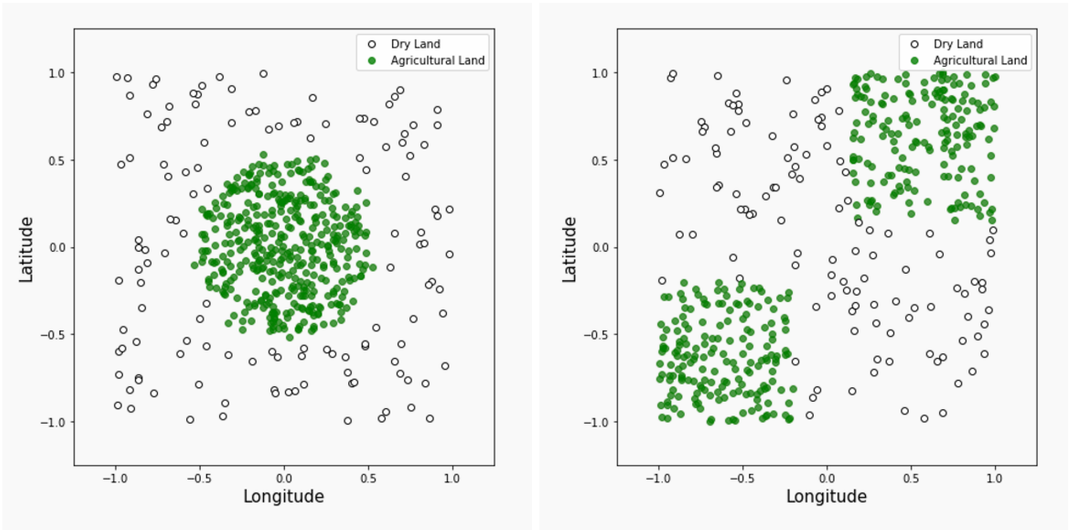
This is equivalent to the scenario where the log-odds are zero. The log-odds are defined as:

$$\log\left[\frac{P(Y = 1)}{1 - P(Y = 1)}\right] = X\beta = 0$$

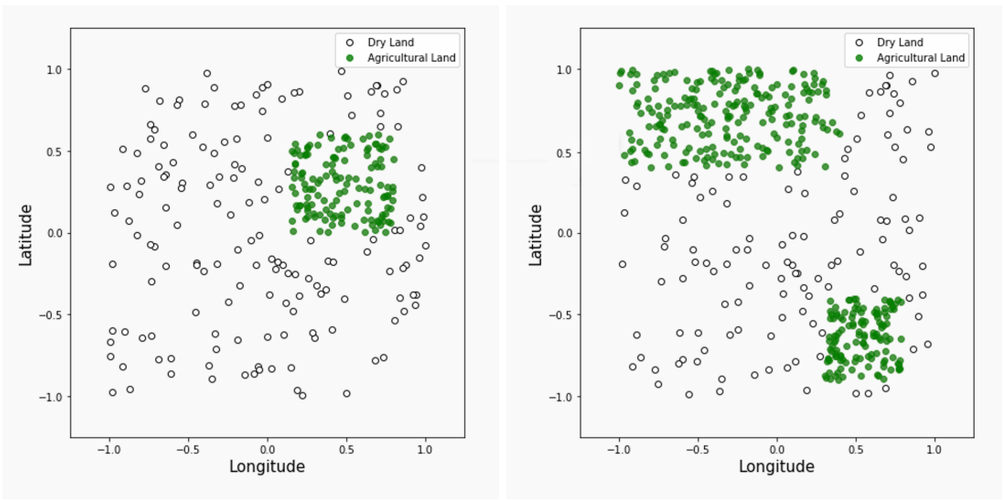
The equation $X\beta = 0$ deine an hyperplane, but can be **generalized** using higher order polynomial terms to specify a non-linear boundary hyperplane.



In this case the blue line can be easily described as $(y = 0.8x + 0.1)$ or $-0.8x + y = 0.1$ (assuming that, given their position as coordinates we call *longitude* as x and *latitude* as y).



In the first figure, we see that we can make a circular bounding box whereas in the second figure it is likely that we can make two square bounding boxes. However, as the geometric shapes in the figures become more complicated, determining the appropriate bounding boxes becomes increasingly less straightforward and more complex.



Observe that in all the datasets, the classes remain well-separated in the feature space, **yet the decision boundaries cannot be easily described using a single equation**.

While logistic regression models with linear boundaries are intuitive to interpret by examining the impact of each predictor on the log-odds of a positive classification, it is **less straightforward to interpret nonlinear decision boundaries** in the same context.

1.1.1 Interpretable Models

People in every walk of life have long been using interpretable models for differentiating between classes of objects and phenomena.
Simple flow charts can be formulated as mathematical models for classification and these models have the properties we desire. Those properties are:

1. They possess **sufficiently complex** decision boundaries, and
2. They remain **interpretable** by humans.

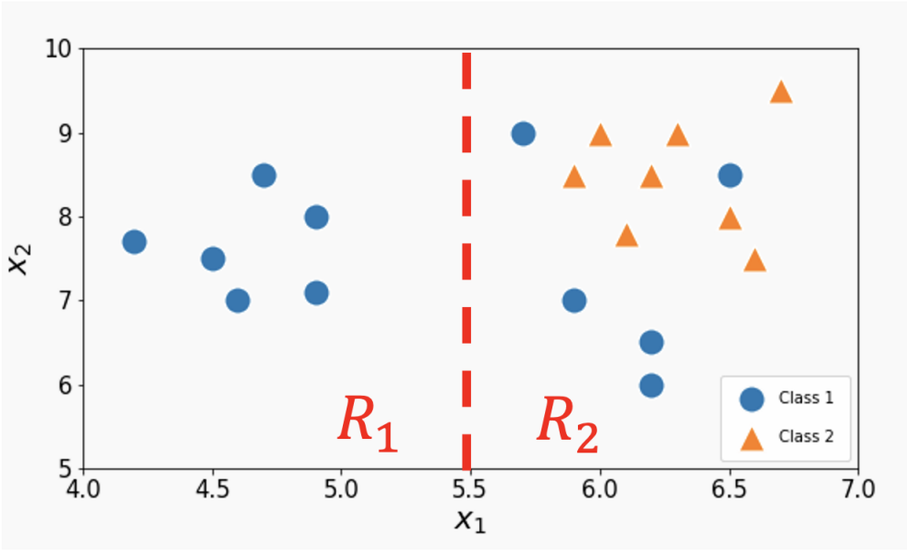
In addition, an important characteristic of these models is that their decision boundaries are locally linear. This means each segment of the decision boundary can be simply described in mathematical terms.

Geometry of Flow Charts Each flow chart node corresponds to a partitioning of the feature space — specifically, the height and width of the fruit — by axis-aligned lines or (hyper)planes.

Prediction Now, when presented with a data point, we can utilize its values to navigate or traverse the flow chart. This process will guide us to the model's predicted classification outcome - in other words, it leads us to land on a leaf node.

1.1.2 Classification Error

To understand the concept of classification error as a splitting criterion, consider the following figure.



Consider the region R_2.
We define Classification error as $\frac{error}{total}$,

$$= \frac{\text{number of minority data points}}{\text{total number of data points}}$$

$$= \frac{\text{number of majority data points}}{\text{total number of data points}}$$

$$= 1 - \frac{\text{number of majority data points}}{\text{total number of data points}}$$

In addition, when considering the region R_2,

$$1 - \frac{\text{number of majority data points}}{\text{total number of data points}}$$

$$= 1 - \Psi(\triangle | R_r) = 1 - max_k(\Psi(k|R_k))$$

Where $\Psi(k|R_r)$ is the portion of training points in R_2 that are labeled class k.

The table below shows how the classification error for each region is calculated.

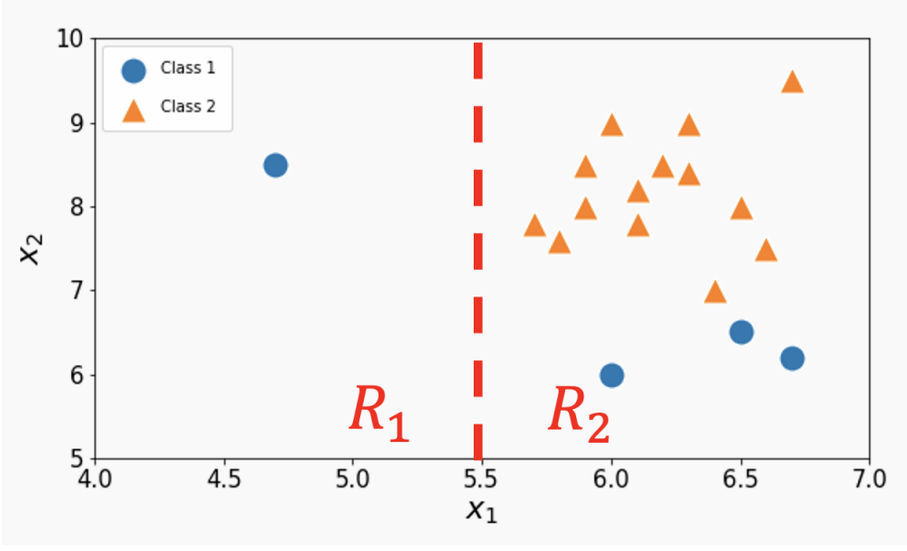
	◯	△	Error=$1 - max_k(\Psi(k\|R_r))$
R_1	6	0	$1 - max(\frac{6}{6}, \frac{0}{6}) = 1 - 1 = 0$
R_2	5	8	$1 - max(\frac{5}{13}, \frac{8}{13}) = \frac{5}{13} \approx 0.38$

In general: Assume we have P predictors and K classes. Suppose we select the pth predictor and split a region along the threshold t_p.

We can assess the quality of this split by measuring the classification error made by each newly created region by calculating:

$$\text{Error}(R_r|p, t_p) = 1 - max_k(\Psi(k|R_r))$$

Where $\Psi(k|R_r)$ is the proportion of training points in R_r that are labeled class k. Here's another example:



	◯	△	Error $= 1 - max_k(\Psi(k\|R_r))$
R_1	1	0	$1 - max(\frac{1}{1}, \frac{0}{1}) = 1 - 1 = 0$
R_2	3	14	$1 - max(\frac{3}{17}, \frac{14}{17}) = \frac{3}{17} \approx 0.18$

We need to calculate the **weighted average** over both regions, taking into consideration the number of points in each region, and then minimize this weighted average over the parameters p and t_p:

$$min_{p,t_p}\left[\frac{N_1}{N}Error(R_1|p,t_p) + \frac{N_1}{N}Error(R_2|p,t_p)\right]$$

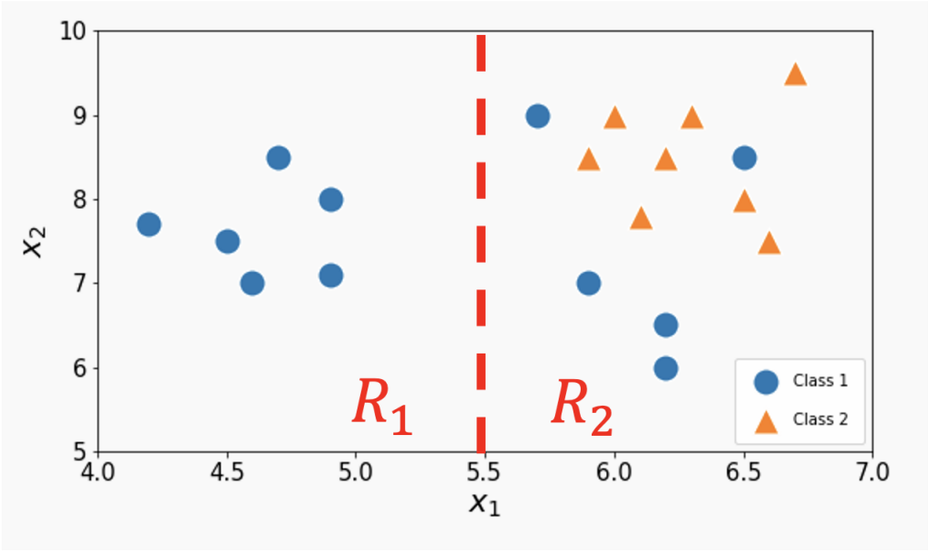
Where N_r is the number of training points inside of the region R_r.

1.1.3 Gini index

Assume we have P **predictors** and K **classes**. Suppose we select the pth predictor and split a rgion along the **threshold** $t_p \in \mathbb{R}$.

We can assess the quality of this split by measuring the **Gini Index** made by each newly created region by calculating:

$$Gini(R_r|p,t_p) = 1 - \sum_k \Psi(k|R_r)^2$$



In the image above, we see that we have a certain number of each type of point in the specified regions based on our deliminator. We can then use these point tallies to deterime the index values.

	◯	△	Gini $= 1 - \sum_k \Psi(k\|R_r)^2$
R_1	6	0	$1 - \left[\left(\frac{6}{6}\right)^2 + \left(\frac{0}{6}\right)^2\right] = 1 - 1 = 0$
R_2	5	8	$1 - \left[\left(\frac{5}{13}\right)^2 + \left(\frac{8}{13}\right)^2\right] = \frac{80}{169} \simeq 0.47$

We can now try to find the predictor p and the threshold t_p minimizes the weighted average Gini Index over the two regions:

$$min_{p,t_p}\left[\frac{N_1}{N}Gini(R_1|p,t_p) + \frac{N_2}{N}Gini(R_1|p,t_p)\right]$$

Where N_r is the number of training points inside of the region R_r.

1.1.4 Information Theory

The last metric for evaluating the quality of a split is motivated by metrics of uncertainty in information theory.

One way to quantify the strength of a signal in a particular region is to analyze the distribution of classes within the region. We compute the **entropy** of this distribution.

What is entropy? In this course we are using the information theory definition. Entropy is a fundamental concept in information theory that quantifies the average amount of information or uncertainty associated with a random variable or message.
For a random variable with a discrete distribution, the entropy is computed by:

$$H(x) = -\sum_{x \in X} \psi(x) \log_2 \psi(x)$$

Entropy is therefore a value that describes how "well-mixed" or "well-separated" a data distribution is. Highly mixed distributions will have entropy near 1. Distributions with well-separated categories will have entropy near 0.

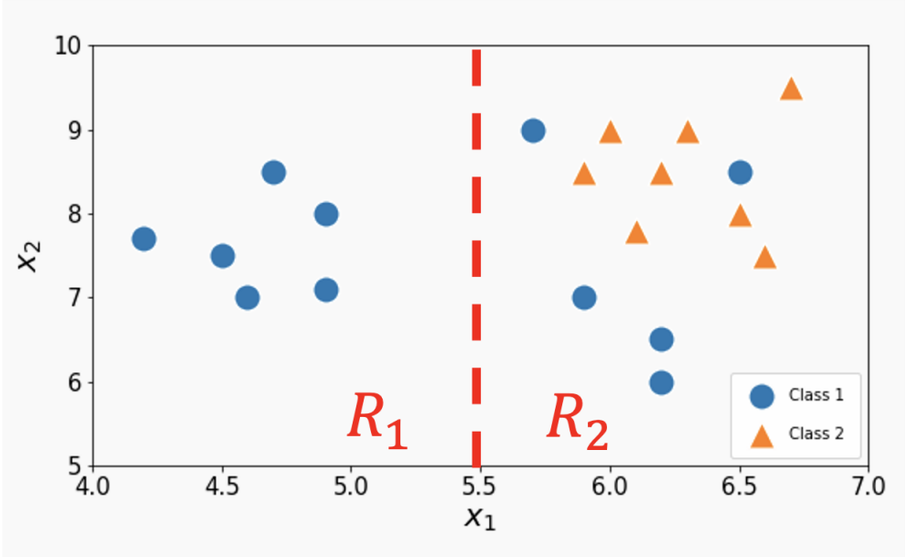
1.1.5 Entropy

Assume we have P **predictors** and K **classes**. Suppose we select the k-th predictor and split a region along the threshold $t_p \in \mathbb{R}$.

We can assess the quality of this split by measuring the **entropy of the class distribution** in each newly created region by calculating:

$$\text{Entropy}(R_r|p,t_p) = -\sum_k \psi(k|R_r)\log_2\psi(k|R_r)$$

> **Note:** We are actually computing the conditional entropy of the distribution of training points amongst the K classes given that the point is in region r.



	◯	△	Entropy $= -\sum_k \Psi(k\|R_r)\log_2\Psi(k\|R_r)$
R_1	6	0	$\left[\frac{6}{6}\log_2\frac{6}{6} + \frac{0}{6}\log_2\frac{0}{6}\right] = 0$
R_2	5	8	$\left[\frac{5}{13}\log_2\frac{5}{13} + \frac{8}{13}\log_2\frac{8}{13}\right] \approx 1.38$

The entropy calculation here yields a value of 1.38, compared to a misclassification rate of 0.38 and a Gini index of 0.47.

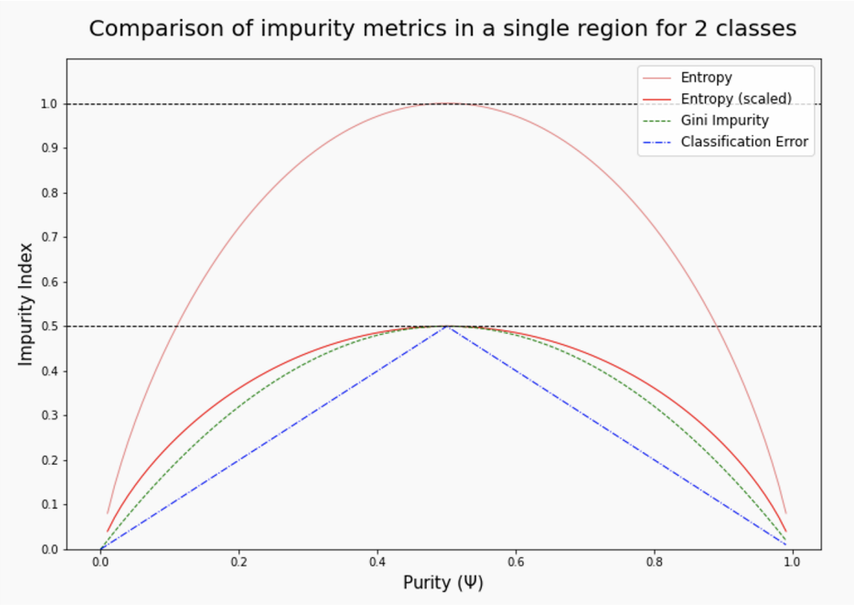
We can now try to find the predictor p and the threshold t_p minimizes the weighted average entropy over the two regions:

$$min_{p,t_p}\left[\frac{N_1}{N}Entropy(R_1|p,t_p) + \frac{N_2}{N}Entropy(R_2|p,t_p)\right]$$

Where N_r is the number of training points inside of region R_r.

1.1.6 Comparison of Criteria

The graph below shows how our various impurity metrics compare for a fairly simple region with two classes. On the bottom is the purity Ψ. You can see that all three measures reach a maximum around purity 0.5, but they take different paths to get there. For entropy and Gini impurity, they are curved (concave downward), while the classification error is a straight line from zero at both ends of the axis to a maximum in the center. None of these are "better", they just give different weights to different values of Ψ.



Learning the "optimal" decision tree for any given set of data is **NP complete** (intractable) for numerous simple definitions of "optimal". Instead, we will use a greedy algorithm that works as follows:

1: Start with an empty decision tree (undivided feature space)
2: Choose the "optimal" predictor on which to split, and choose the "optimal" threshold value for splitting.
3: Recurse on each new node until stopping condition is met.
4: For the case of classification, predict each region to have a class label based on the largest class of the training points in that region (Bayes' classifier).

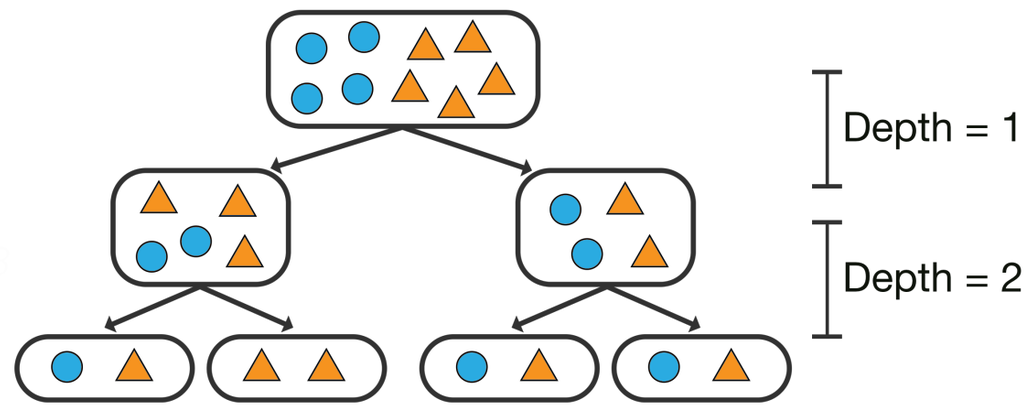
This is not guaranteed to give us the best decision tree (for any particular definition of "best"), but it is very likely to give us a good one.

1.2 Stopping Conditions

1.2.1 Common Stopping Conditions

The most common stopping criterion involves restricting the **maximum depth** (`max_depth`) of the tree.

The following diagram illustrates a decision tree trained on the same dataset as the previous one. However, a `max_depth` of 2 is employed to mitigate overfitting.

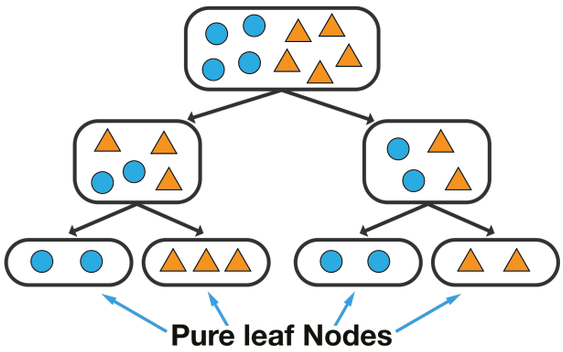


Four other common simple stopping conditions are:

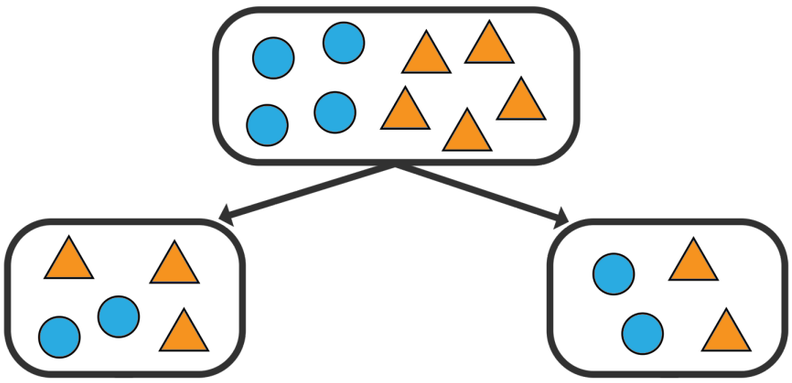
1. Do not split a region if all instances in the region **belong** to the same class.
2. Do not split a region if it would cause the number of instances in any sub-region to go below a pre-defined threshold (`min_samples_leaf`).
3. Do not split a region if it would cause the total number of leaves in the tree to exceed a pre-defined threshold (`max_leaf_nodes`).
4. Do not split if the gain is less than some predefined threshold (`min_impurity_decrease`).

Let's look at each one individually.

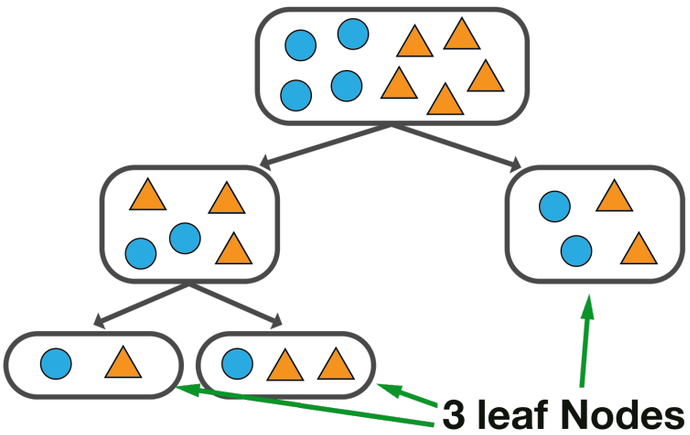
1. Do not split a region if all instances in the region belong to the same class. The diagram below displays a tree where each of the end leaf nodes are clearly of the same class, therefore we can stop at this point in growing the tree further.



2. Do not split a region if it would cause the number of instances in any sub-region to go below a pre-defined threshold (`min_samples_leaf`). In the following diagram, we set `min_samples_leaf` to be 4, and we can observe that we can't split the tree further because each leaf node already meets the minimum requirement.



3. Do not split a region if it would cause the total number of leaves in the tree to exceed a pre-defined threshold (`max_leaf_nodes`). In the diagram below, we observe a tree with a total of 3 leaf nodes, as specified as the maximum.



4. Do not split if the gain is less than some pre-defined threshold (`min_impurity_decrease`). Compute the gain in purity of splitting a region R into R_1 and R_2:

$$Gain(R) = \Delta(R) - m(R) - \frac{N_1}{N}M(R_1) - \frac{N_2}{N}M(R_2)$$

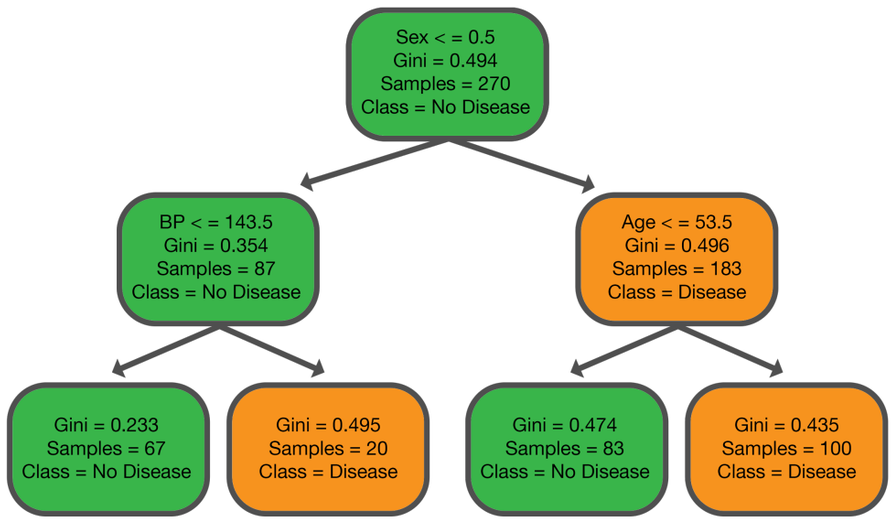
Where M is the classification error, Gini index or Entropy, depending on which is chosen to use.

Scikit-learn (`sklearn`) grows trees in **depth-first** fashion by defaul, but when `max_leaf_nodes` is specified it is grown in a best-first fashion.

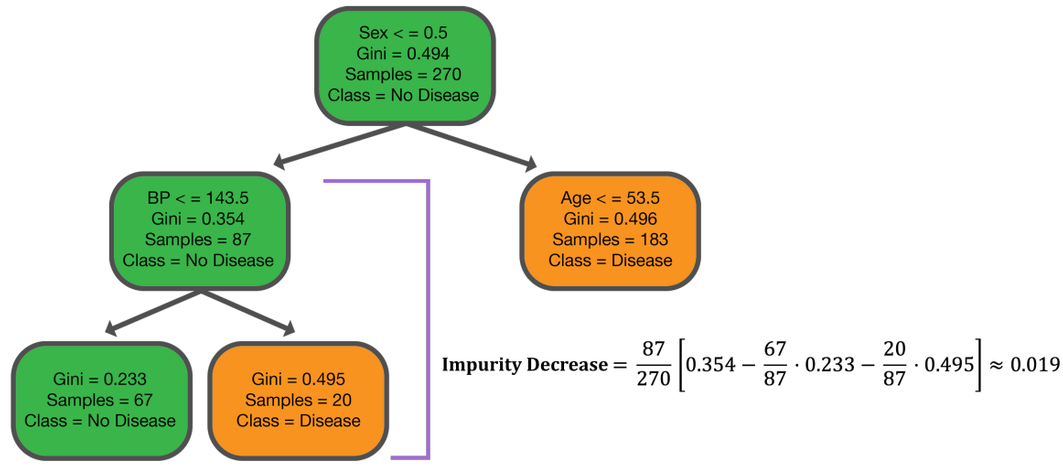
1.2.2 Depth-first and Best-first

To illustrate the difference between depth-first and best-first, we'll consider the following decision trees that predicts if a person has heart disease based on age, sex, BP and cholesterol.

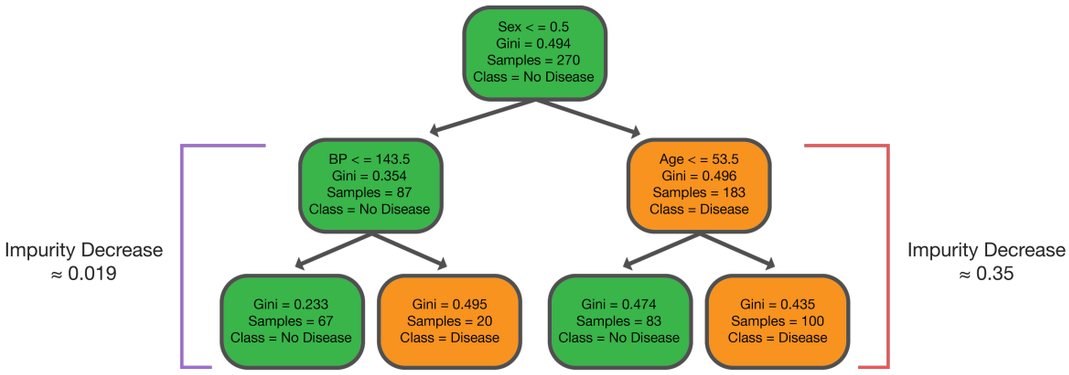
Depth-first Growth For depth-first, sklearn determines the best split based on impurity decrease, with the greatest amount of decrease being the best choice. Here we see a tree with `max_depth=2`.



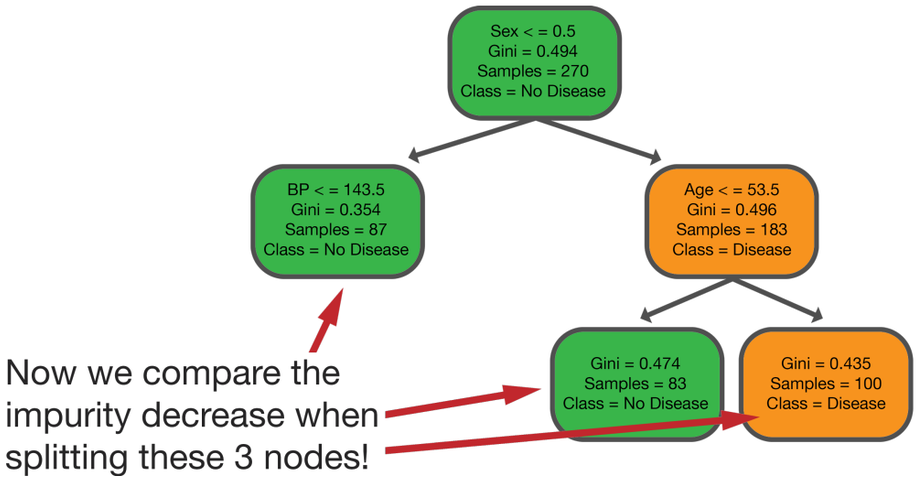
Best-first Growth Here we see a series of images showing the growth of the best first tree. Each time we calculate the impurity decrease for the threshold to determine when to stop. For the first diagram we see that the impurity decrease is minimal.



In our second split, the impurity decrease is slightly larger as compared to the split from before. We want to split where the decrease is greater; therefore, we choose the second split.

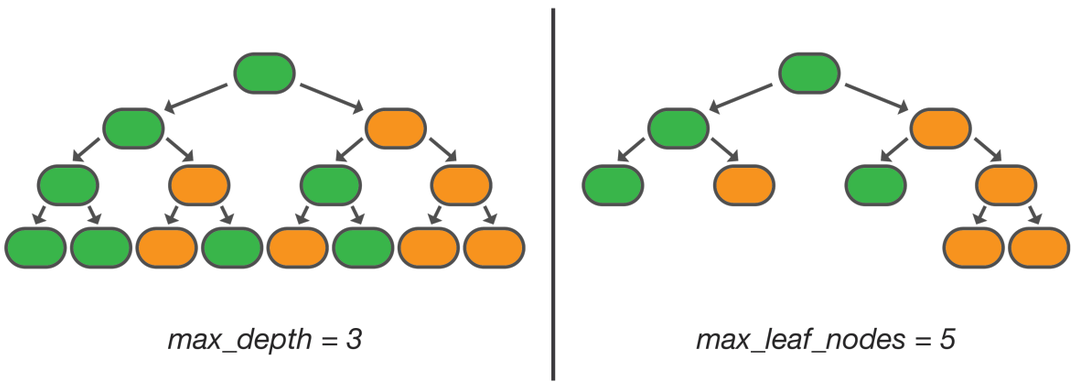


Once we have chosen the appropriate split, we can now delibrate the next split given the remaining nodes to compare. In the diagram below we see the next nodes we will compare the splits for.



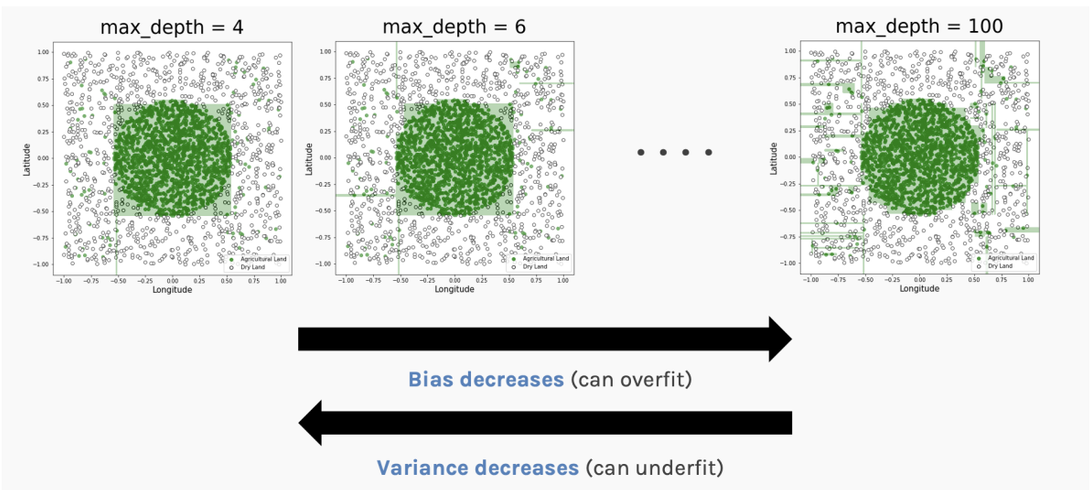
This process is repeated until `max_leaf_nodes` is reached.

Depth-first vs. Best-first The following is a comparison between depth-first or best-first growth. The image on the left shows a stopping condition for the depth. The right image has a constraint on the number of leaf nodes, which is a breadth-first stopping condition.



1.2.3 Variance vs. Bias

How do we decide what is the appropriate stopping condition?



Complex trees are also harder to interpret and more computationally expensive to train. This gives us a bias-variance tradeoff, like you saw in our previous course. Let us revisit these concepts in the context of decision trees.

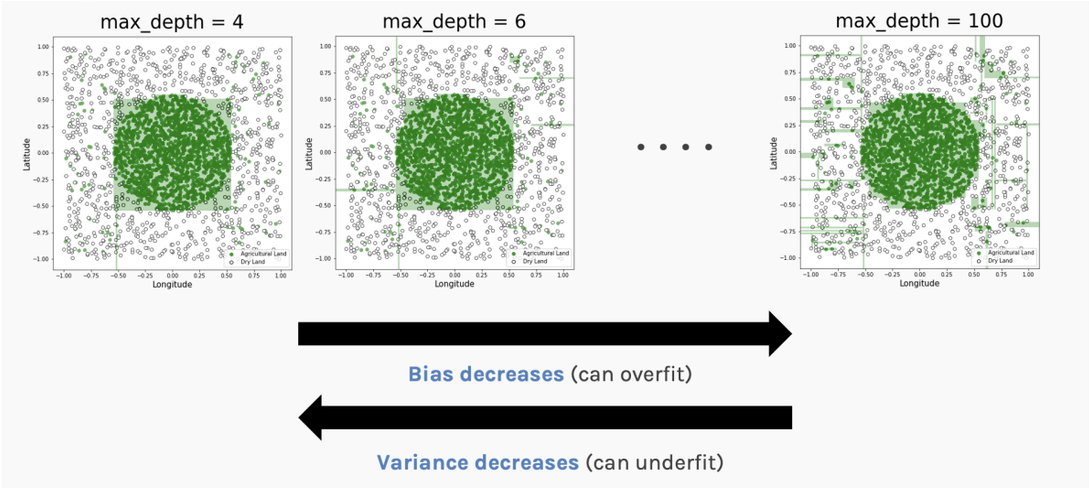
High Bias: Trees of low depth are not a good fit for the training data - it's unable to capture the nonlinear boundary separating the two classes.

Low Variance: Trees of low depth are robust to slight perturbations in the training data - the square carved out by the model is stable if you move the boundary points a bit.

Low Bias: With a high depth, we can obtain a model that correctly classifies all points on the boundary (by zig-zagging around each point).

High Variance: Trees of high depth are sensitive to perturbations in the training data, especially to changes in the boundary points.

The image below gives us a visual example. On the left we have a circular area of green data points that we're trying to model with a tree of depth 4. It's not going to be very good - it's essentially fitting a round peg in a square hole. As the depth increases, we get a better fit, but you can see more and more places where the model predicts there should be green data points for no discernible reason. As we get to depth 100, the prediction is very messy, with "false positives" everywhere. We've committed the classic mistake of over-fitting our data, capturing every single point in the training set while making our future predictions worthless.



How Can We Determine the Appropriate Hyperparameters?

Cross-Validation is the key As always we rely on CV to find the optimal set of hyper-parameters.

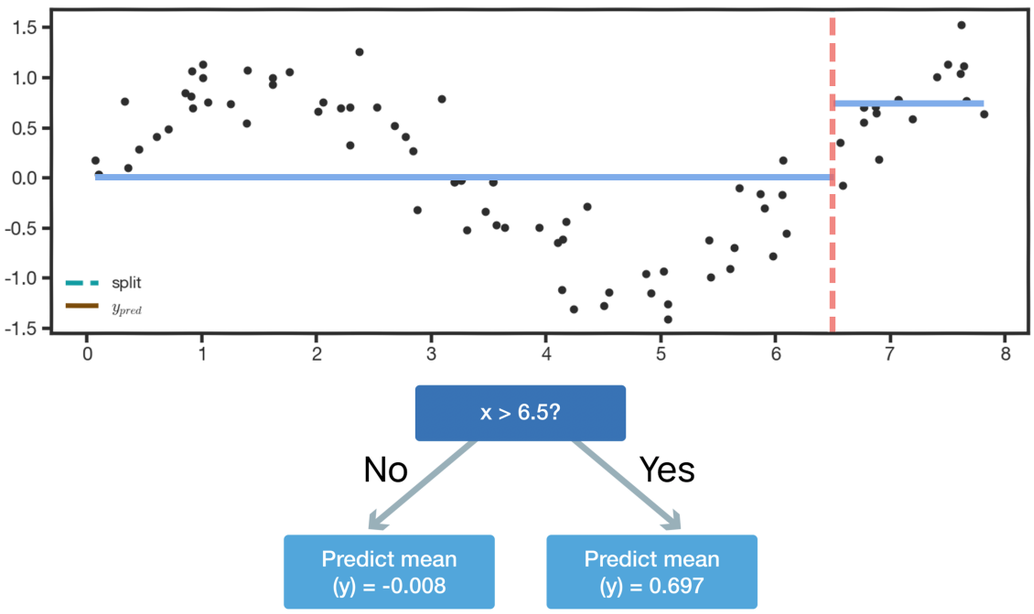
Chapter 2

Decision trees II

2.1 Decision trees in regression

2.1.1 Splitting Criteria

To illustrate how we determine the splitting criteria for a regression tree, consider the following regression tree diagram:



Here the data has been split based on whether x is greater than or less than 6.5. Our predicted means are -0.008 and 0.697. You can tell by looking at the graph that this is not a great split.

The quality of a split in the regression tree can be assessed by calculating the **mean squared error** (MSE) of the predicted outcomes for each newly generated region. We do this by subtracting the average of the observed outcomes from each observation within the region and squaring the results, then taking the mean of these squared differences:

$$\text{MSE}(R_r) = \frac{1}{n} \sum_{i \in R_r} (y_i - \overline{y}_{R_r})^2$$

However, this raw MSE doesn't account for the size of each region. To correct this, we also compute a weighted average of the MSEs over both regions, so that the quantity of observations in each region is factored in:

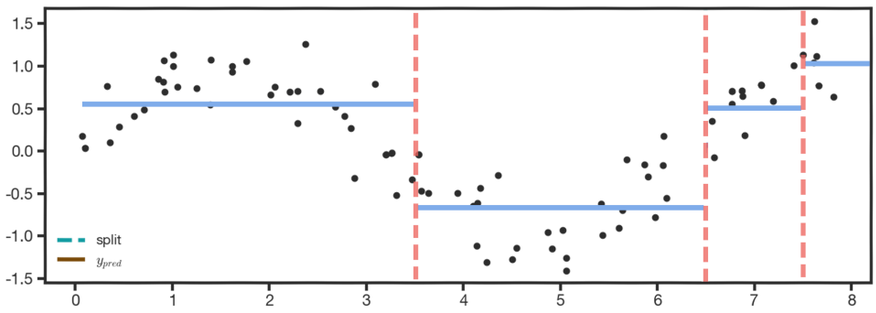
$$\text{MSE}(R_1, R_2) = \left[\frac{N_1}{N} \text{MSE}(R_1) + \frac{N_2}{N} \text{MSE}(R_2) \right]$$

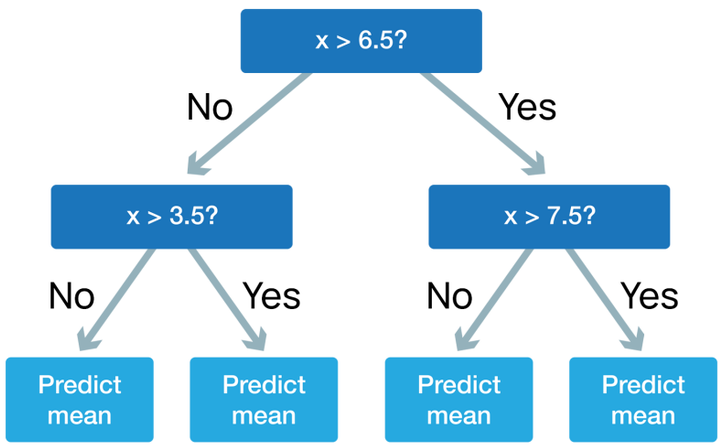
In this formula, N represents the total number of observations, while N_1 and N_2 indicate the number of observations in each of the two regions respectively.

To find the appropritate split we minimize the weighted MSE from above with respect to the predictor p and the treshold t_p:

$$\min_{p,t_p} \left[\frac{N_1}{N} \text{MSE}(R_1) + \frac{N_2}{N} \text{MSE}(R_2) \right]$$

Below, we present a decision tree featuring three split points to provide another example. You can see that the averages for each region are closer to the data points in that region, which means this is a better split.





2.1.2 Stopping Conditions

The stopping criteria used for classification trees, such as **maximum depth** (`max_depth`), **minimum number of samples per leaf** (`min_samples_leaf`), and **maximum number of leaf nodes** (`max_leaf_nodes`), are also applicable to regression trees in `scikit-learn`.

Similar to classification trees using **purity gain**, regression trees leverage **information gain** (or **reduction in variance**) to assess the suitability of splitting a region. This metric quantifies the decrease in mean squared error (MSE) achieved by a split, guiding the tree's growth. The splitting process stops when the information gain falls below a predefined threshold.

$$\text{Gain}(R) = \Delta(R) = \text{MSE}(R) - \frac{N_1}{N}\text{MSE}(R_1) - \frac{N_2}{N}\text{MSE}(R_2)$$

2.1.3 Prediction with regression trees

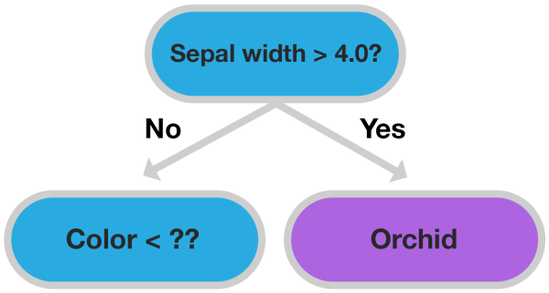
In order to make a prediction for a continuous, quantitative outcome given a regression tree, we can follow these steps:

1. Traverse the decision tree, starting from the root and moving towards a leaf node, based on the attribute values of the data point x_i.
2. Once at a leaf node, the prediction is calculated as the average of the observed outcomes y within that leaf node, denoted as $\overline{y}_t$. These averages are derived from the training dataset. These averages are computed from the training dataset.

2.1.4 Handling Numerical and Categorical Attributes

Consider the following data for our discussion. An important question arises: How do we construct a decision tree with this data, which includes both numerical and categorical attributes?

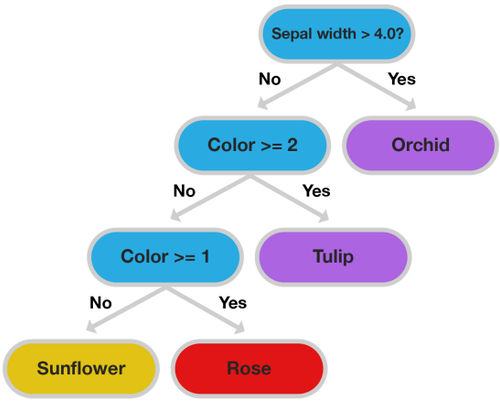
Sepal width	Color	Flower
3.0 mm	0	Sunflower
3.5 mm	1	Rose
4.5 mm	2	Orchid
3.7 mm	2	Tulip



As we can see, the '*compare and branch*' mechanism we used in the classification tree works well for numerical attributes. However, with a categorical feature (with more than two possible values), comparisons like 'feature ¡ threshold' are meaningless.

One straightforward solution is to encode the values of a categorical feature as numerical values, treating this feature as if it were numerical. This process is known as ordinal encoding. For instance, if we encode **Yellow**=0, **Red**=1, and **Purple**=2, our decision tree could look like this:

Sepal width	Color	Flower
3.0 mm	0	Sunflower
3.5 mm	1	Rose
4.5 mm	2	Orchid
3.7 mm	2	Tulip



However, the specific numerical encoding of categories can affect the potential splits in the decision tree. For example if we consider the order as above the possible non-trivial splits on color are:

{{Yellow}, {Red, Purple}} and {{Yellow, Red},{Purple}}

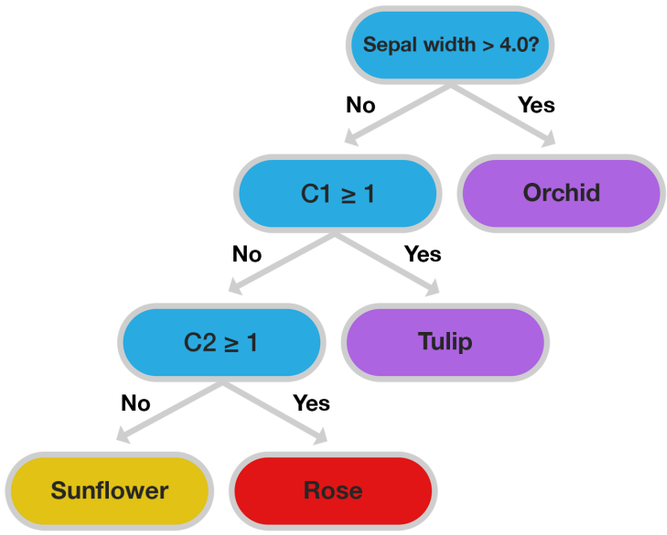
On the other hand, if we adopt the following different numerical encoding (Yellow = 2, Red = 0, Purple = 1):

{{Red}, {Yellow, Purple}} and {{Red, Purple}, {Yellow}}

This shows that depending on the ordinal encoding, the splits we optimize over can be different! Nonetheless, in practice, the impact of our choice of naive encoding of categorical variables is often negligible - models resulting from **different choices of encoding will perform comparably**.

What if your categorical data is not ordinal? In such cases, ordinal encoding could lead to nonsensical splits. The solution is **one-hot encoding** or **dummy encoding**. This technique, while computationally more demanding, is implemented in several computational libraries like R's `randomForest`, `H2O`, and `XGBoost`.

Sepal width	C1	C2	C3	Flower
3.0 mm	0	0	1	Sunflower
3.5 mm	0	1	0	Rose
4.5 mm	1	0	0	Orchid
3.7 mm	1	0	0	Tulip

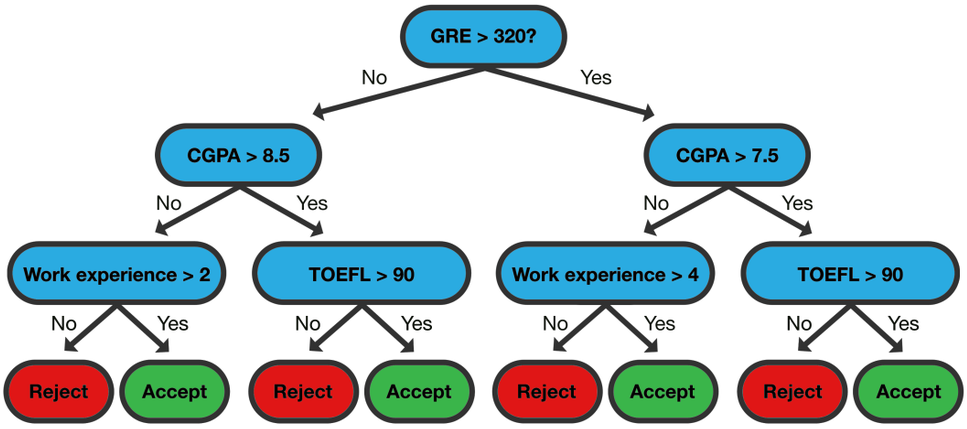


While decision trees in general can handle categorical data, the scikit-learn implementation currently requires converting it before use. For the most current information, always consult the scikit-learn documentation.

2.2 Pruning

2.2.1 Example: Master's application

Before delving into the pruning technique, let's consider an example scenario to better understand its practical application.



In the given example, we have a decision tree for assessing master's applications. Each node represents a question or criterion used for evaluating applicants, such as GPA, letters of recommendation, and GRE scores. The tree's structure guides the decision-making process, ultimately leading to an admission decision.

However, this tree appears quite complex, potentially indicating overfitting. Instead of halting the tree's growth prematurely, we can employ pruning techniques to obtain a simpler, yet effective, decision tree.

2.2.2 Pruning techniques

Pruning refers to the process of selectively reducing the size of a decision tree by removing branches or subtrees. Among various pruning methods, one commonly used approach is **cost complexity pruning**.

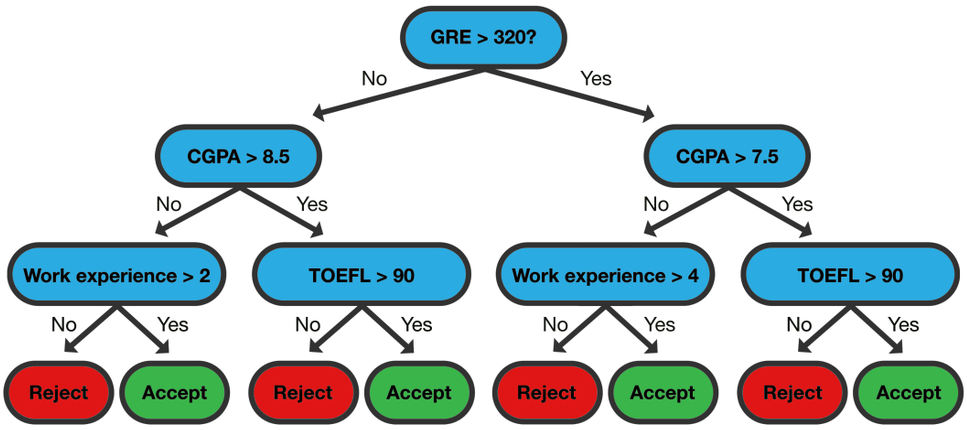
The cost complexity pruning process involves assigning a cost to each node in the tree based on its complexity and accuracy. The cost is defined as follows:

$$C(T) = \text{Error}(T) + \alpha|T|$$

where T is the Decision tree, $\text{Error}(T)$ is the classification error, $|T|$ is the number of leaves in the tree and α is a hyperparameter that balances the trade-off between tree complexity and accuracy. On this page, we're working with an example that has $\alpha = 0.2$.

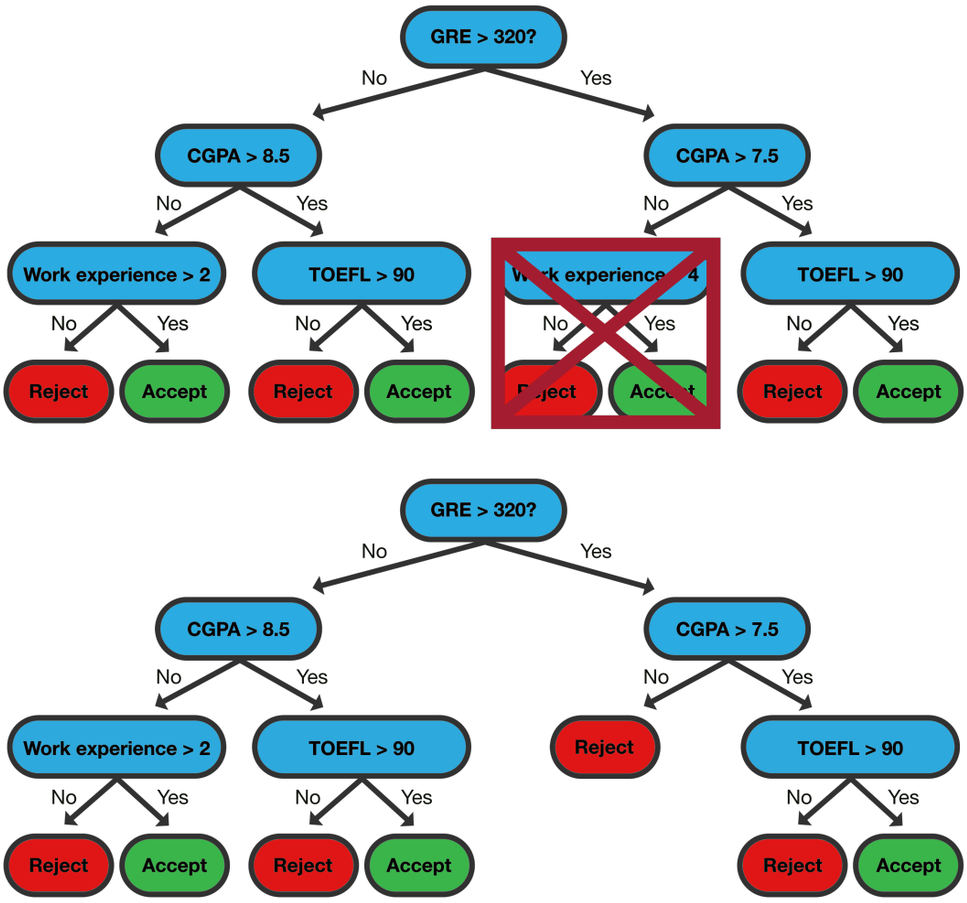
The pruning process can be visualized as follows. Initially, we start with a full-grown decision tree, represented as T_0. It splits on GRE score first, then on CGPA on both sides. Below that split, it examines years of work experience and TOEFL scores. (Remember, this tree is just a mockup - it's not how the

Computer Science department actually decides who gets in).



Tree	Error(T)	$\|T\|$	$\text{Error}(T) + \alpha\|T\|$
T_0	0.32	8	$0.32 + 8 \cdot 0.2 = 1.92$

We then choose subtrees, replace them with leaf nodes, and calculate the cost complexity. Below, we have one such example.



Tree	Error(T)	$\|T\|$	$\text{Error}(T) + \alpha\|T\|$
T_0	0.32	8	$0.32 + 8 \cdot 0.2 = 1.92$
T_1	0.33	7	$0.33 + 7 \cdot 0.2 = 1.73$

In the example above we notice that the smaller tree has a slightly larger error $\text{Error}(T)$ but smaller complexity score $C(T)$.

From there, we iteratively evaluate different pruning options to create a sequence of pruned trees $T_1, T_2, \ldots, T_L$, where T_L is the tree consisting of only the root node. This process follows a bottom-up approach, considering subtrees and their corresponding cost complexity scores.

Algorithm 1 The pruning algorithm can be summarized as follows:

1: Start with a full tree T_0, where each leaf node is pure.
2: Replace a subtree in T_0 with a leaf node to obtain a pruned tree T_1. The selection of the subtree aims to minimize the expression:

$$\frac{\text{Error}(T_1) - \text{Error}(T_0)}{\alpha(|T_0| - |T_1|)}$$

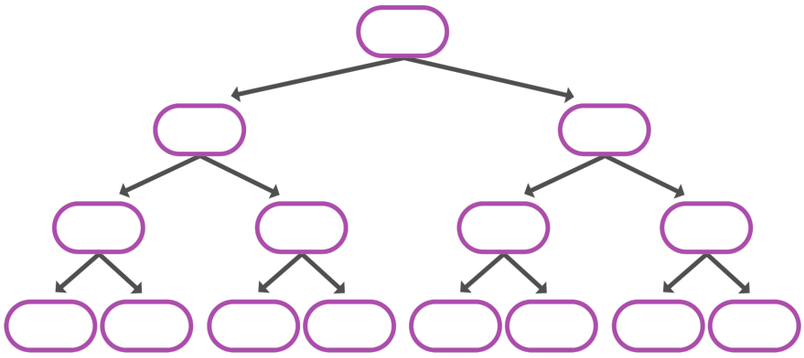
3: Iterate this pruning process to obtain a sequence of pruned trees $T_0, T_1, \ldots, T_L$, where T_L consists of only the root node. This process follows a bottom-up approach, considering subtrees at each iteration.
4: Select the optimal tree T_i using a validation set.

It's worth noting that the pruning process implicitly optimizes the cost complexity score defined earlier at each step, even though we don't explicitly compute it.

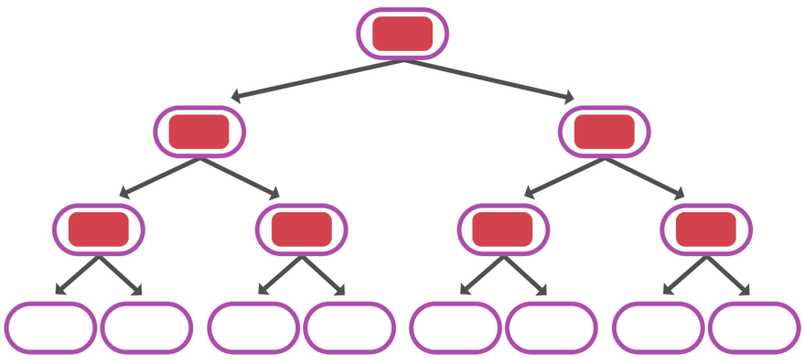
To determine the optimal tree, we need to find the best value for the hyperparameter α. This is typically achieved through cross-validation, where we assess the performance of each pruned tree using a validation set. More on this below.

2.2.3 Evaluating Pruning Options

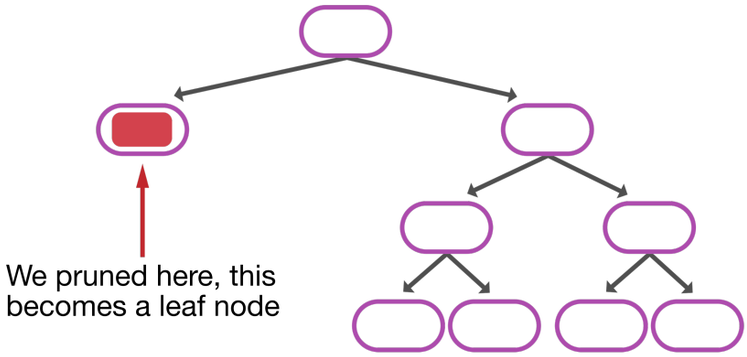
Here we demonstrate this process in more details. Let's consider an example where we have a **full tree** T_0:



To prune this tree, we have several possible options. We can prune individual branches or subtrees, resulting in different pruned trees denoted as T^*. Note that in general we do not have to prune only the last layer.



For each potential pruning, we obtain a set of pruned trees.



Now, the question arises: How do we determine the best pruned tree?

The selection process involves maximizing the difference in the cost complexity scores between the full tree and each pruned tree. We want to find the pruned tree that maximizes $C(T) - C(T^*)$, using the same definition of $C(T)$ as we did before:

$$C(T) = \text{Error}(T) + \alpha|T|$$

In other words, we aim to maximize $C(T) - C(T^*)$ by comparing the error measure $E(T)$ and $E(T^*)$, as well as the tree sizes $|T|$ and $|T^*|$. Our goal is to maximize such that:

$$\begin{aligned}
\text{argmax}\,[C(T) - C(T^*)] &= \text{argmax}\,[E(T) - E(T^*) + \alpha|T| - \alpha|T^*|] \\
&= \text{argmax}\left[\frac{E(T) - E(T^*)}{\alpha|T^*| - \alpha|T|} - 1\right] \\
&= \text{argmax}\left[\frac{E(T) - E(T^*)}{\alpha|T^*| - \alpha|T|} - 1\right] \\
&= \text{argmax}\left[\frac{E(T) - E(T^*)}{\alpha|T^*| - \alpha|T|}\right] \\
&= \text{argmax}\left[\frac{E(T) - E(T^*)}{\alpha|T| - \alpha|T^*|}\right] \\
&= \text{argmax}\left[\frac{E(T^*) - E(T)}{\alpha|T| - \alpha|T^*|}\right]
\end{aligned}$$

This iterative procedure of selecting the optimal pruned tree from a given pruned tree is repeated until we obtain the final pruned tree $T^{(L)}$. Therefore, we can instead, at each step, minimize:

$$\frac{E(T^*) - E(T)}{\alpha|T| - \alpha|T^*|}$$

This iterative pruning process considers multiple levels of pruning, leading to a sequence of pruned trees $T^{(1)}, T^{(2)}, \ldots, T^{(L)}$,.

Finally, we select the optimal tree $T^{(i)}$ using validation. This tree represents the pruned version that achieves the best performance on the validation set.

2.2.4 Choosing the Optimal Hyperparameter

Algorithm 2 To fins the best α, follow this steps:

1: Fix a specific value for α.
2: Find the best tree, T^*, for the given α by evaluating the cost complexity score $C(T)$ on the validation fold.
3: Use cross-validation to determine the best α by assessing the average error measure on all validation sets.

This iterative process allows us to identify the hyperparameter α that yields the most suitable pruned tree.

By employing pruning techniques and optimizing the cost complexity score, we can effectively simplify decision trees and mitigate overfitting, resulting in more interpretable and generalized models.

Chapter 3

Bagging

3.1 Bagging

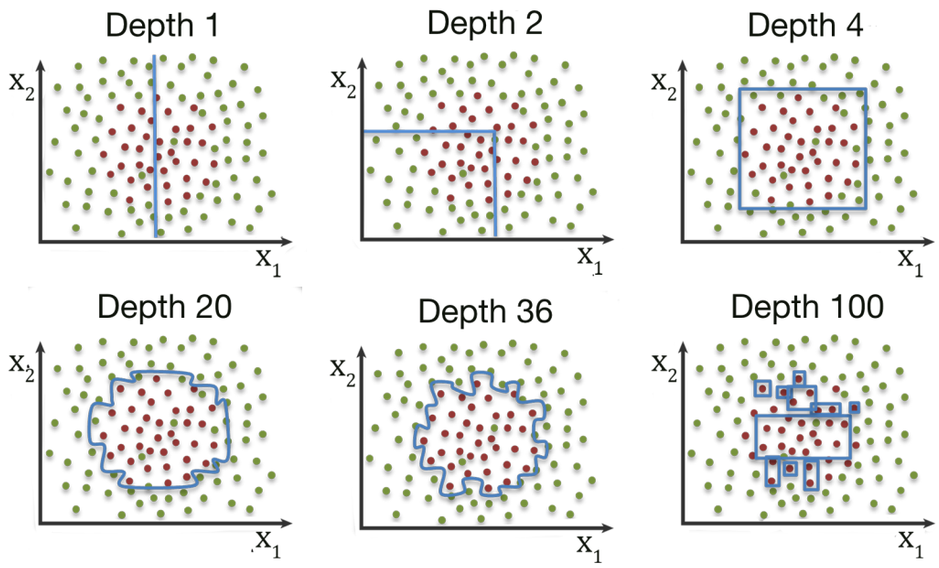
3.1.1 Overfitting

"Underfitting" is when a model is not detailed enough to describe its data accurately.

"Overfitting" is when it is too detailed, fitting the test data very accurately but giving bad predictions about future data.

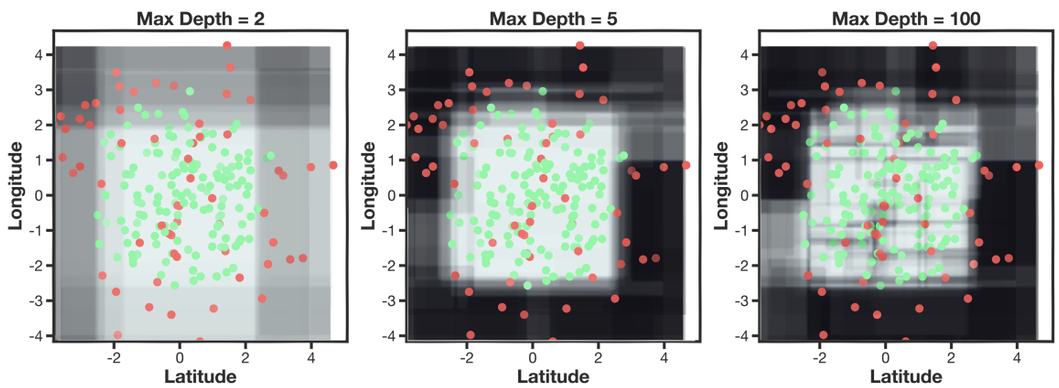
In the case of decision trees:

- When a tree is too shallow, it cannot divide the input data into enough regions. This results in a decision boundary which does not envelop the data points well. Thus, the model underfits.
- When the tree is too deep it cuts the input space into too many regions and fit to the noise of the data so it overfits. Look at the data set below, with tree depths of 1, 2, 4, 20, 36, and 100.

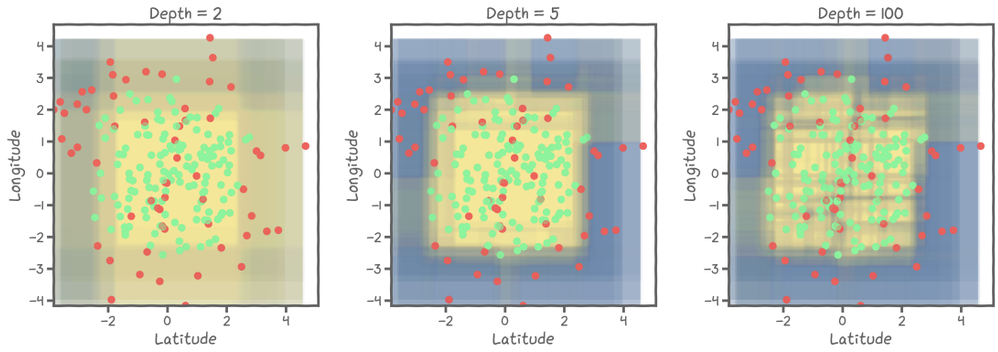


You can see that the trees of depth 36 and depth 100 are near-perfect fits, but their predictions for future data are probably nonsense. *Data has noise in it* - there's no point in trying to draw such a complicated boundary when a simpler one will be easier to use and interpret, and just as accurate.

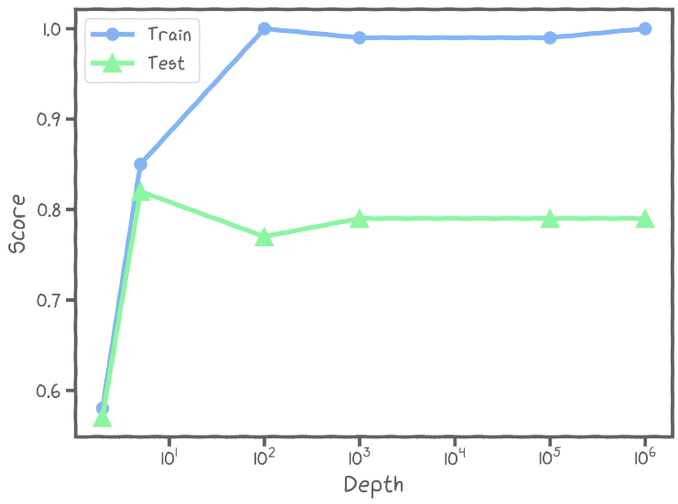
Here's another view. The following figure shows three plots of increasing decision boundary complexity, from left to right.



And another example: in the plots below, regardless of increase in tree depth, the decision boundaries and accuracy scores are limited.



The plot on the right displays depth vs accuracy score for these kinds of classification. It shows that as the depth of the tree increases, the gap between the train and test scores increases.



There are two techniques you can use to avoid overfitting:

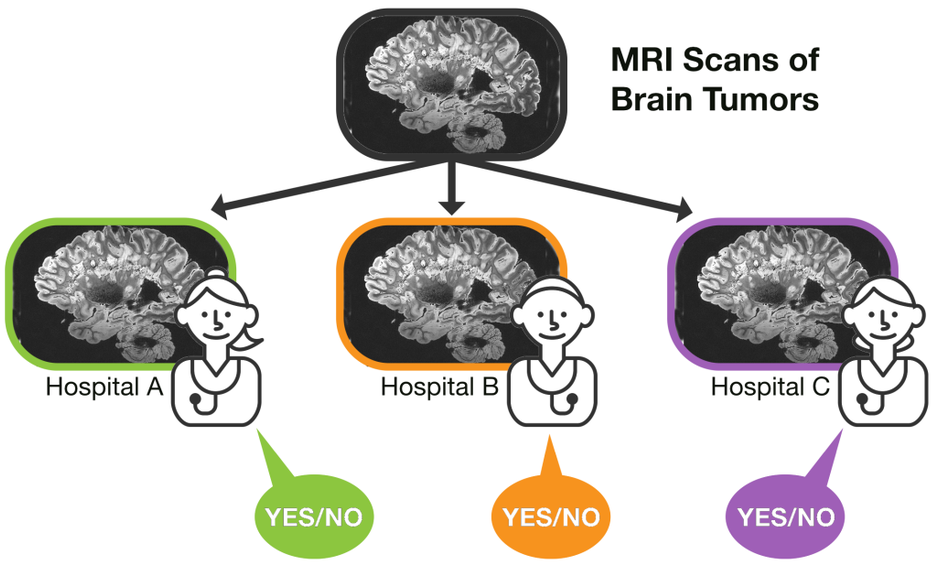
- Pruning the tree
- Limiting the maximum depth of the tree

3.1.2 Ensemble Learning

Ensemble learning is a machine learning technique that combines several base models in order to produce one optimal predictive model.

- For classification, we return a *plurality* of the models.
- For regression, we return the *average* of the output of the model.

A real-world example of ensemble learning is shown in the figure below. Doctors are trained on varying samples of MRI in their respective Med school. In the hospital they are currently working at, they are shown an MRI that none have seen before. Each one of them makes a decision on the presence of brain tumor. The final decision is an aggregation of all their votes.



Advantages

- A group of models gives a better accuracy as compared to the individual models
- Using a single model causes overfitting, whereas the group is less likely to overfit.
- It helps reduce bias and variance

Types of Ensemble methods Kinds of ensemble methods include...

- Bagging
- Boosting
- Stacking
- Blending

Bagging and boosting are going to be our main focus as we move forward in this course.

3.1.3 Bootstrapping

Motivation: In practice, we do not have different or unlimited datasets to train various models. However, we still want multiple models trained on different datasets. Bootstrapping is the way we accomplish this.

Bootstrapping is the process of **sampling with replacement** from a dataset, and performing calculations on multiple such bootstrapped datasets to get multiple different predictions.

Consider an example of a dataset consisting of the MRI scans of 5 patients, namely scan A, scan B, scan C, scan D and scan E. Thus, the following can be different versions of the bootstrapped data:

- scans A, C, A (again), E, and B
- scans E, B, A, C, and another copy of C
- scans D, B, A, E, and A again

It is important to note that the bootstrapped dataset consists of the same amount of data as the original dataset, and that they are constructed via random sampling with replacement.

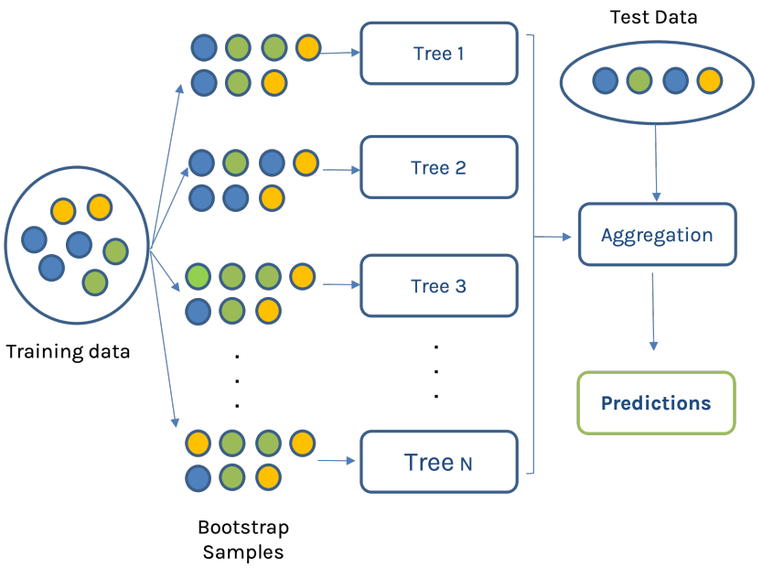
This leads us to an even more powerful approach:

3.1.4 Bagging

Bagging is short for **B**ootstrap **Agg**regating.

- **Bootstrapping** involves generating multiple samples of training data, via bootstrapping. Then, a deeper decision tree is trained on each sample of data.
- Once the trees are trained, we **Aggregate**: For a given input, the final output is the averaged outputs of all the models for that input.

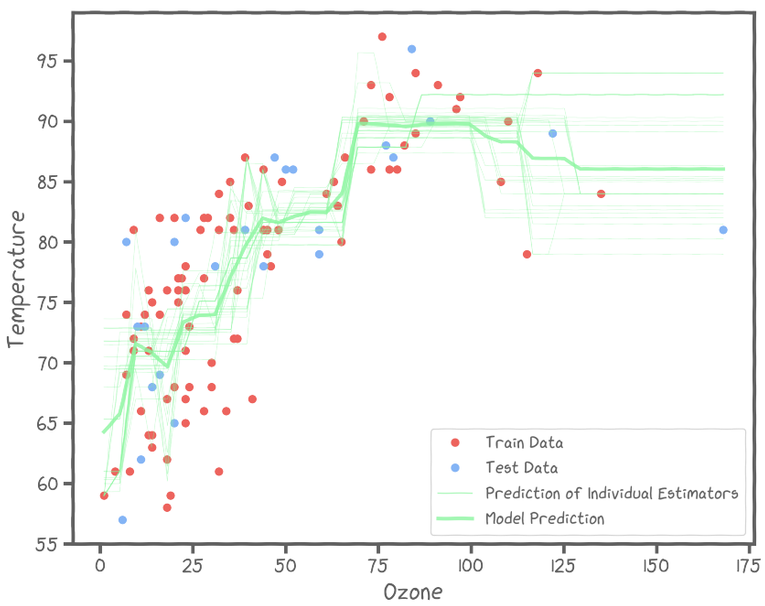
Here's a graphical representation of that process:



Advantages of Bagging Bagging has the following advantages:

1. High expressiveness - by using deeper trees each model is able to approximate complex functions and decision boundaries.
2. Low variance - averaging the prediction of all the models reduces the variance in the final prediction, assuming that we choose a sufficiently large number of trees.

Bagging Types For each bootstrap, we build a decision tree. The result is a combination (majority) of the predictions from all trees for classification tasks. While for regression, we take the average of all individual predictions of the trees or estimators.
An example of regression bagging is shown in the following plot. Each of the thin green lines is a prediction from an individual tree from a different bootstrapped sample. The thick green line is the average of all the individual predictions. It gives us a stronger predictor overall than any one of the individual trees.



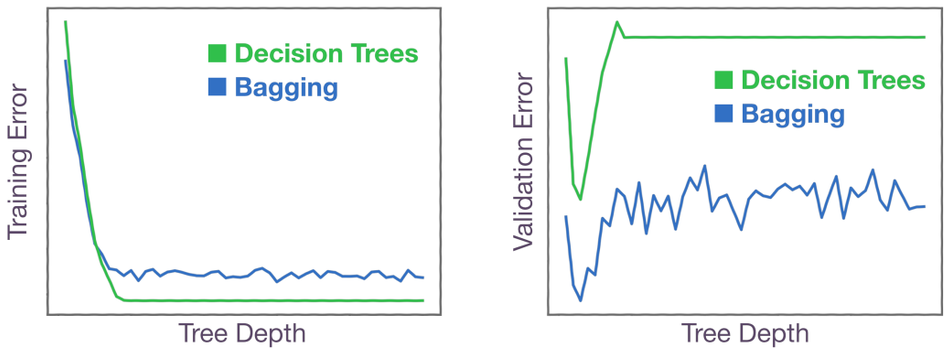
Drawbacks of Bagging

- If the trees are too shallow it can still underfit.
- If the trees are too large, there is a high chance of overfitting.
- An averaged model is no longer easily interpretable. One can no longer trace the 'logic' of an output through a series of decisions based on predictor values.

3.1.5 Decision Tree vs Bagging

In the graph below we see that the **more depth we add to decision trees, the more quickly the model overfits**.

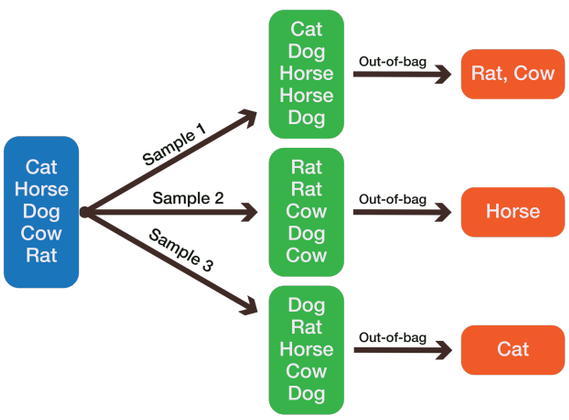
However, for Bagging, we see that there is still some form of **better model performance maintained after increase of depth of trees**, although eventually it does lead to **no improvement of performance after a certain point**.



3.2 Out-of-bag error

Out-of-bag error (**or Out-of-bag estimate**) is a method of determining the prediction error that allows the trees to be fit and validated whilst being trained. It's called out-of-bag because you use the points that were not included in an aggregated bootstrap training set (the "bag"). Out-of-bag is usually abbreviated as OOB.

The diagram below shows items that are "out-of-bag" for a bag. For instance, "Rat" and "Cow" do not appear in the first sample, so they're in the "out-of-bag" group.



Uses of OOB OOB techniques can be used to:

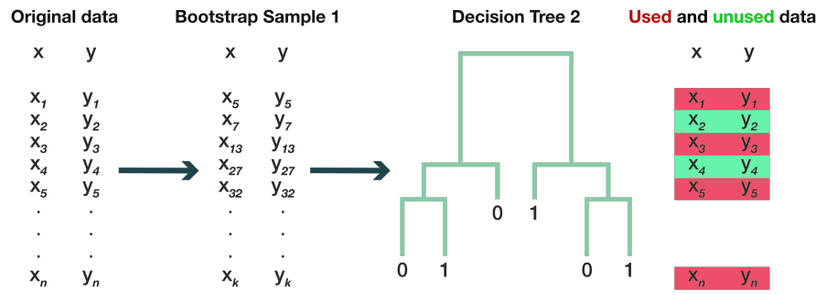
- Measure the generalizability of a model
- Replace the need for a separate measurement of performance for a validation set

3.2.1 OOB Steps

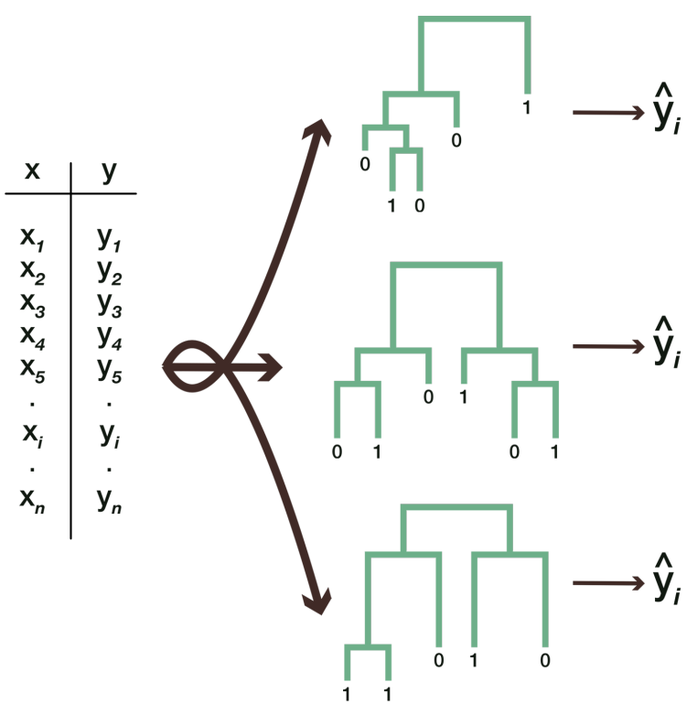
Algorithm 3 Calculating OOB Error

1: For a given dataset, create multiple bootstrapped versions of the data.
2: Fit a decision tree on each of the boostrapped data versions. This would lead to having B trees that did not see point X_i, y_i.
3: Compute the point-wise prediction for point X_i, y_i from the B trees
   - For classification: $\hat{y}_{y,pw} = \text{majority}(\hat{y}_i)$
   - For regression: $\hat{y}_{y,pw} = \frac{1}{B}\sum_{j \in B} \hat{y}_{i,j}$
4: Given the prediction, compute the point-wise out-of-bag error
   - For classification; $e_i = \mathbb{I}(\hat{y}_{i,pw} \neq y_i)$
   - For regression; $e_i = (y_i - \hat{y}_{i,pw})^2$
5: Average the point-wise out-of-bag error over the full training set
   - For classification; $\text{Error}_{OOB} = \frac{1}{N}\sum_i^N e_i = \frac{1}{N}(\hat{y}_{i,pw} \neq y_i)$
   - For regression; $\text{Error}_{OOB} = \frac{1}{N}\sum_i^N e_i = \frac{1}{N}(\hat{y}_i - \hat{y}_{i,pw})$

Graphically, the approach looks like this, starting from the original data on the left and generating a decision tree from the bootstrapped sample.



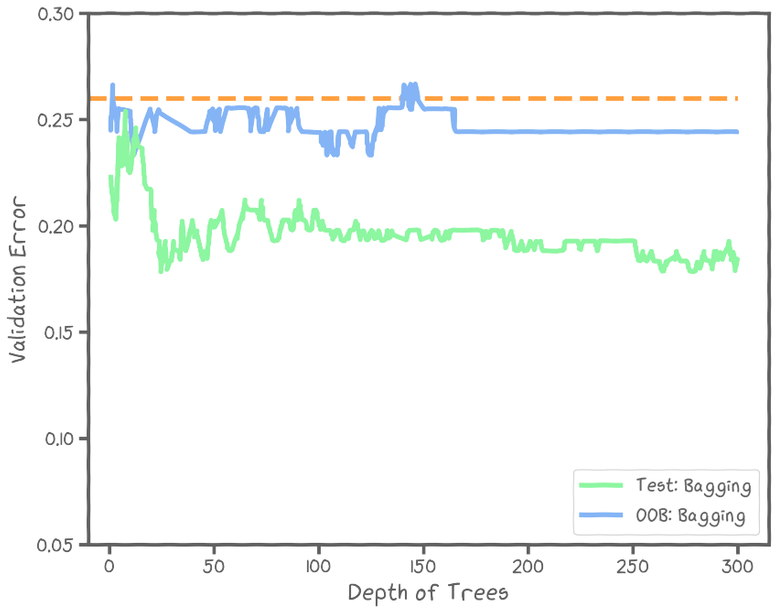
One set of data can get us a large number of different trees to work with, even when we are working just with models that have not seen one particular data point.



3.2.2 Benefits and Drawbacks

Why use OOB error?

- While using the cross-validation technique, every validation set has already been seen or used in training by a few decision trees. This results in a leakage of data, and therefore more variance. OOB Error prevents leakage and gives a better model with low variance.
- Computation cost for OOB error is lower as compared to cross-validation for bagging. You can see in the graph below that the OOB error for bagging is lower, and it decreases quickly.



Bagging drawbacks One of the major drawbacks of bagging (and other ensemble methods that we will study) is that the averaged model is not easily interpretable. It's not like a simple decision tree where you can clearly see why and where each choice is made. With bagging it can be harder to determine why the model makes a particular prediction. This issue is common across many AI techniques.

Improving Bagging In practice, the ensembles of trees in Bagging tend to be highly correlated. Suppose we have an extremely strong predictor,x_j , in the training set amongst moderate predictors. Then the greedy learning algorithm ensures that most of the models in the ensemble will choose to split on x_j in early iterations. A lot of our trees will start off exactly the same.
However, we assumed that each tree in the ensemble is independently and identically distributed, with the expected output of the averaged model the same as the expected output of any one of the trees. Since our assumption was wrong, we're in trouble.
Fortunately, Random Forests, our next topic, will help us de-corrolate our trees and give us a stronger method overall.

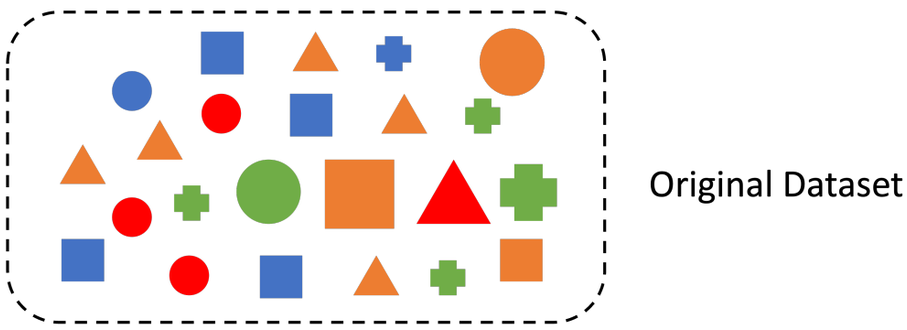
Chapter 4

Random Forest

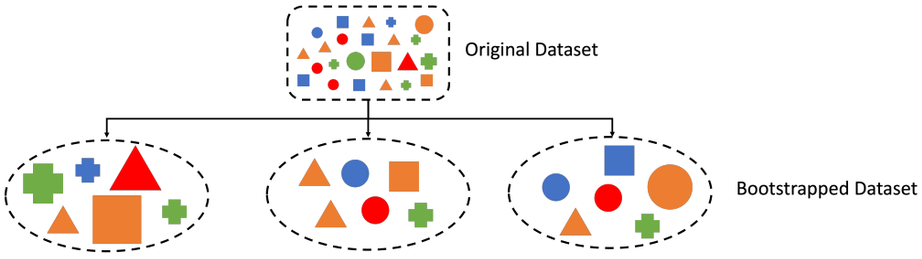
Why Random Forests? In our last section, we explored several typical hurdles encountered in classification tasks, including high variance and overfitting. Additionally, we also learned the concept of bagging, a robust ensemble learning technique aimed at mitigating these challenges. However, one question that arose at the end of the last session is whether bagging can be impacted when a single feature dominates. It's a valid concern, and one that we should consider. If a single feature outweighs the others, the models created may all end up making similar predictions, resulting in a lack of diversity in the final predictions. In summary, although bagging is an effective approach for mitigating high variance and overfitting, we need to be cautious about the potential impact of a dominant feature. It's important to consider employing techniques to mitigate this issue, which will be the focus of upcoming sessions.

4.1 Introduction to Random forest

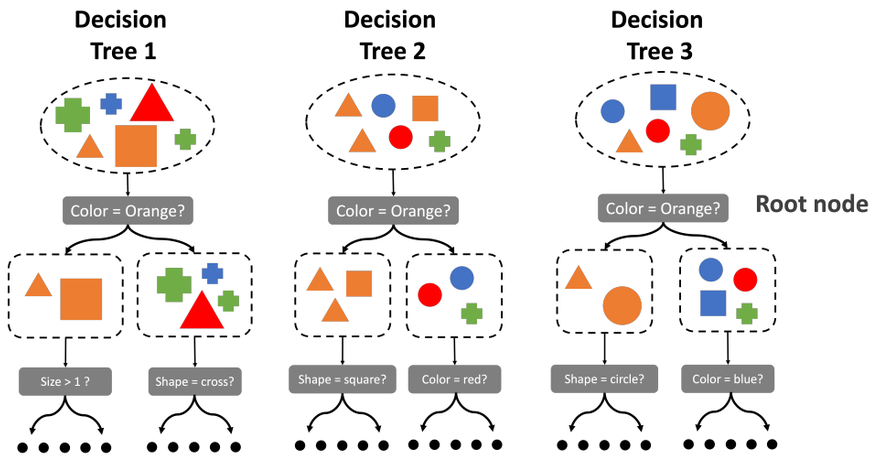
Consider a dataset of geometric shapes with features of Color, Shape and Size. We want to use the ensemble learning method, Bagging (Bootstrap Aggregation).



As a first step of Bagging, multiple subsets of dataset are created as below. We use replacement during the sampling process, so some elements might appear more than once in the same set.



Then, multiple decision trees are fitted from each of the bootstrapped datasets.



This is where we run into the problem we mentioned above. We can notice that each decision tree looks similar. In particular, every root node has the same predictor, Color.

This is because, regardless of the bootstrapped dataset, each tree chooses the same predictor that provides the best split for the corresponding criterion at the top of the tree.

In summary, the ensembles of trees in bagging tend to be highly correlated.

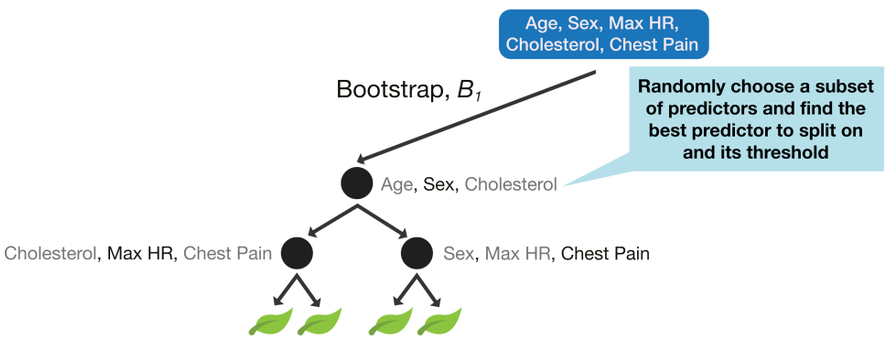
4.1.1 What is a Random Forest?

The Random Forest method is a modified form of bagging that creates **ensembles of independent decision trees**.

We're going to use the heart rate dataset as our example. As a reminder, it has five features: Age, Sex , Max HR, Cholesterol, and Chest Pain.

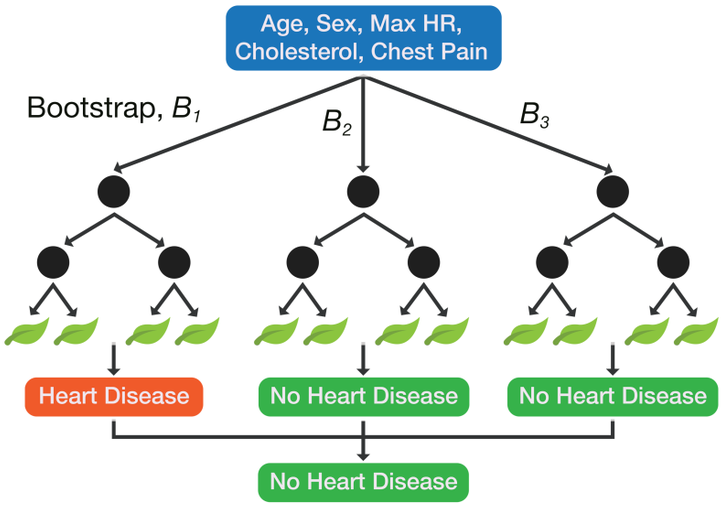
Let's take a look at the first node. Starting from a bootstrapped dataset B_1, instead of using all 5 predictors, a sub-set of the predictors are randomly selected: Age, Sex, and Cholesterol.

Of those 3 predictors, the best predictor and threshold is Sex. Based on on that criteria, we would choose to split on Sex first. However, remember that our trick here is that we're going to select our predictors randomly, so on each split we may or may not have Sex as an available variable.



Following the same procedure above, the tree randomly chooses a subset of predictors for the rest of the nodes and finds the best predictor to split on and its threshold.

We repeat the same steps on each tree and finally aggregate the prediction results by taking the majority of the decisions from each tree.



Algorithm 4 In summary:

1: Create bootstrapped datasets with all J predictors (same as in bagging).
2: Initialize a random forest with T decision trees.
3: For each tree, at each split, randomly select a sub-set, j, of all the possible J predictors.
4: From amongst the j predictors, select the optimal predictor and the optimal corresponding threshold for the split.
5: Repeat until the stopping criterion is met.

4.1.2 Tuning Random Forests

Random forest models have multiple hyper-parameters to tune:

1. The **number of predictors**, to randomly select at each split.
2. The total **number of trees** in the ensemble.
3. The **stopping criteria** - maximum depth, minimum leaf node size, etc.

There are standard (default) values for each of the random forest hyper-parameters recommended by long-time practitioners. However, because they are dependent on exactly what data you have and what problem you are solving, these parameters should generally be tuned through **Out of Bag (OOB)** methods. This will make them better for your particular data and problem.

For the number of predictors to randomly select at each split, we normally start with (if N is the total number of predictors):

- $\sqrt{N}$ for classification
- $\frac{N}{3}$ for regression

For the number of trees, we use OOB errors. With OOB, training and validation can be done in a single sequence and we cease training once the out-of-bag error stabilizes.

4.1.3 Variable Importance for Random Forest

In both linear and logistic regression, we assess the importance of features by analyzing their respective coefficients, where a large coefficient value signifies a high level of importance for the corresponding feature. However, in Random Forest, feature importance is typically determined based on metrics such as Gini impurity or information gain, which measure the extent to which each feature contributes to the overall purity or homogeneity of the decision trees in the forest. Features that lead to greater reductions in impurity when used in the decision trees are considered more important.

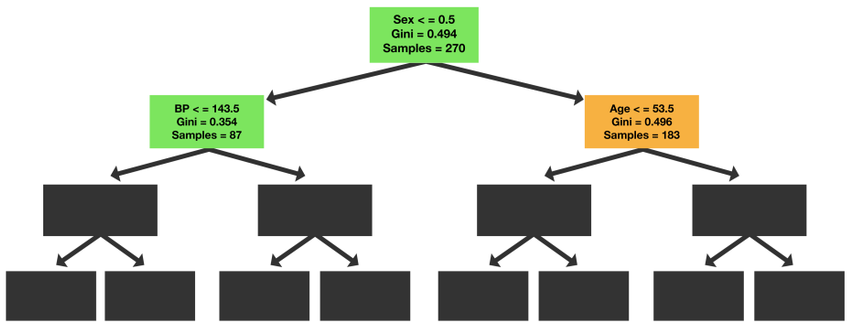
There are four main methods for determining variable importance in Random Forest. While we won't delve into methods 3 and 4 here, there are links below that you can follow for further information.

- Mean Decrease in Impurity (up next).
- Permutation Importance (later in this section).
- LIME (Local Interpretable Model-agnostic Explanations).
- SHAP (SHapley Additive exPlanations).

4.1.4 Mean Decrease in Impurity (MDI)

Decision trees make splits that maximize the decrease in impurity. By calculating the mean decrease in impurity for each feature across all trees, we arrive at the variable importance for a random forest model. Note that this approach works the same for bagging models.

Consider the following decision tree, once again trained on the heart rate dataset, which comprises 5 features: age, sex, maximum heart rate, cholesterol, and chest pain.



1: Calculate the mean decrease in impurity for each node n in the decision tree as:

$$\Delta I_n = (\frac{n}{N}) \left[Gini_n - \sum_{m \in \text{Child}(n)} \frac{m}{n} Gini_m \right]$$

where ΔI_n represents the mean decrease in impurity for each node n, $(\frac{n}{N})$ is the fraction of n samples from the nade out of the whole dataset, $\sum_{m \in \text{Child}(n)}$ is the sum over all children of none n.

2: Calculate the importance of the feature by summing up the impurity decrease calculated in step 1 for each node in which that feature occurs. In this equation, FI is the feature importance, and we use Ch for cholesterol.

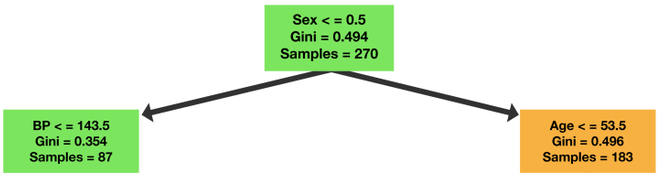
$$FI_{Ch} = \sum_{n \in \text{nodes split on } Ch} \Delta I_n = 0.009 + 0.008 = 0.017$$

3: Normalize this value between 0 and 1 by dividing by the sum of all feature importance values. This ensures the sum of all feature importance in a decision tree adds up to 1.

$$\text{Norm}FI_{Ch} = \frac{FI_{Ch}}{\sum_{j \in \text{all features}} FI_j}$$

4: Up to step 3, we only consider the feature importance at a single tree level. To calculate the feature importance at the **Random Forest or Bagging level**, we average the normalized feature importance of the given predictor over all the trees.

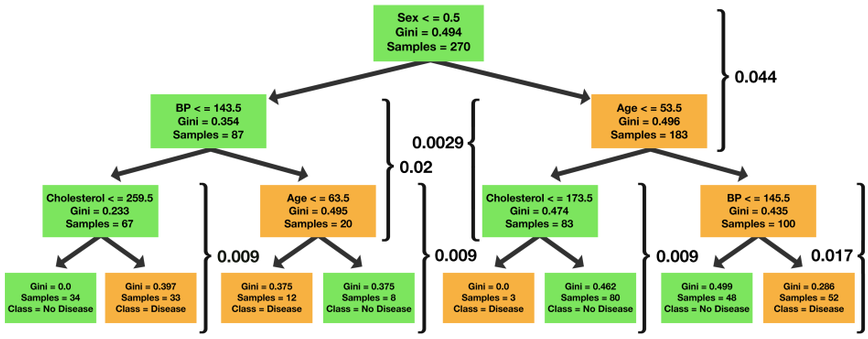
Application of the algorithm For example, the decrease in impurity for the first node:



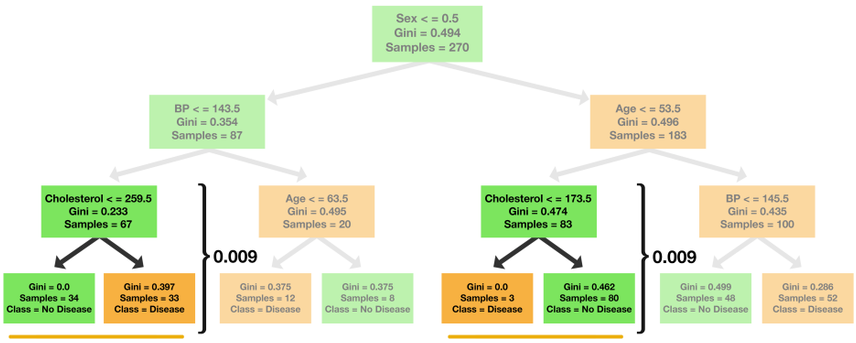
$$\frac{270}{270} \left[0.494 - \frac{87}{270} 0.354 - \frac{183}{270} 0.496 \right] = 0.0044$$

We then proceed to the next step. In this example, we split based on the "age" feature. Everything remains the same as before, but it's important to note that the fraction of the parent node out of the entire dataset changes as we continue the split.

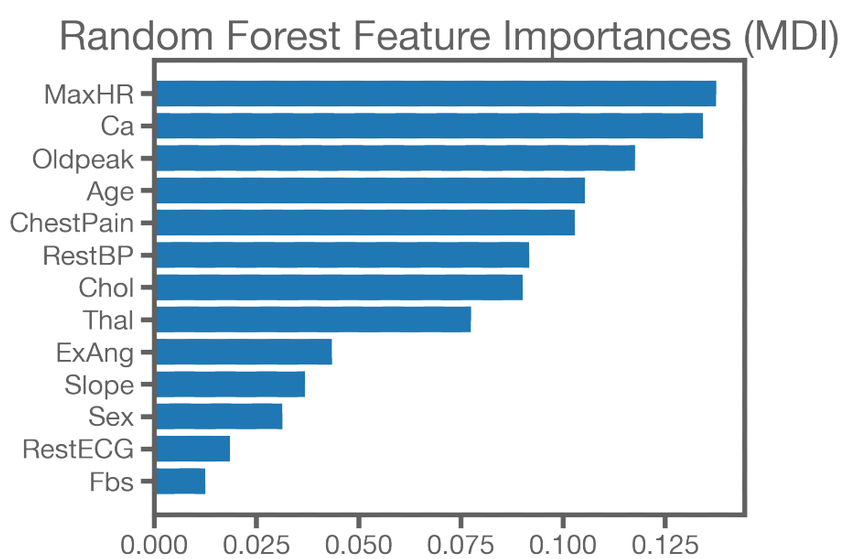
Again, we follow the same procedure as before, with the target predictor to split being "cholesterol" in this case. We continue this process for all features, and finally, we obtain the mean decrease in impurity for each node.



Now, let us try to compute the importance of the feature *Cholesterol*. To do so we look at the impurity decrease for every node that involves *Cholesterol*:



Finally, the overall feature importance for each feature would look like this:



4.1.5 Permutation Importance

Suppose we have a trained model which relies heavily on a certain predictor. When the values of this predictor are randomly shuffled within the dataset and the modified dataset is provided to the model for prediction, the prediction error will increase. On the other hand, if the impact of the feature on the model prediction is not significant, the prediction error will remain unchanged.

This is basic intuition for **Permutation Importance**.

Consider the following dataset:

Height (cm)	**Weight (Kg)**	**...**	**Fitness Level (1-5)**
150	165	...	2
140	50	...	3
...	...	...	...
170	70	...	4
160	80	...	1

1. Record the validation/OOB accuracy of the RF model: Accuracy = 0.88
2. Randomly permute the data for the column j (in this case Weight). Permutation, in case you are unfamiliar with it, is swapping around values within the same column as shown in the image below.

Record the validation/OOB accuracy of the RF model on the modified dataset: in this case, Permuted Accuracy = 0.87

Height (cm)	Weight (kg)	...	Fitness Level (1 – 5)
150	65	...	2
140	50	...	3
...	...	...	...
170	70	...	4
160	80	...	1

3. Repeat the preceding step K times and calculate the average of all the accuracies.
Assuming we permute the feature 3 times, we obtain:

$$PermutedAccuracy_1 = 0.87$$
$$PermutedAccuracy_2 = 0.86 \rightarrow AvgPermutedAccuracy = \frac{0.87 + 0.86 + 0.82}{3} = 0.85$$
$$PermutedAccuracy_3 = 0.82$$

4. Calculate the difference between average permuted and unpermuted accuracy to get the importance of the feature in the random forest: in this case, Difference $= 0.88 - 0.85 = 0.03$

Algorithm 5 – Summary: Permutation Importance For each feature j in the dataset:

1: Record the validation/OOB accuracy of the RF model.
2: For each repetition k in $1 \dots K$: Randomly permute the data for column j. Record the validation/OOB accuracy $s_{k,j}$ of the RF model on the modified dataset.
3: Compute the average of all the permuted datasets' accuracies.

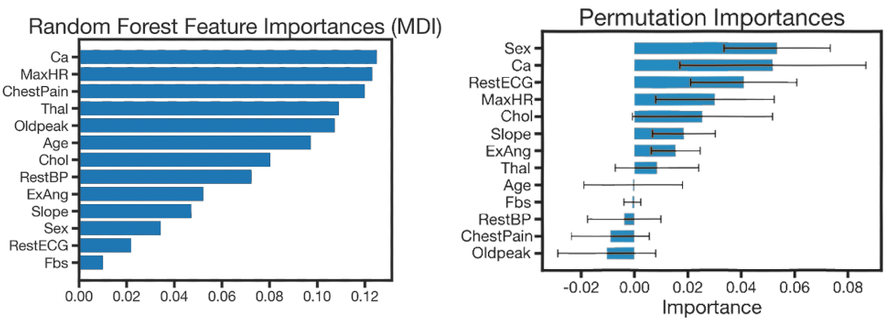
$$s_j = \frac{1}{K} \sum_{k=1}^{K} s_{k,j}$$

4: Calculate the difference between average permuted and unpermuted accuracy to get the importance of the feature in the random forest.

$$\text{RF}FeatureImportance_j = s - s_j$$

4.1.6 MDI vs Permutation Importance

Advantages and drawbacks Random forest (MDI) feature importances and permutation importances look very different in practice. Here's an example:



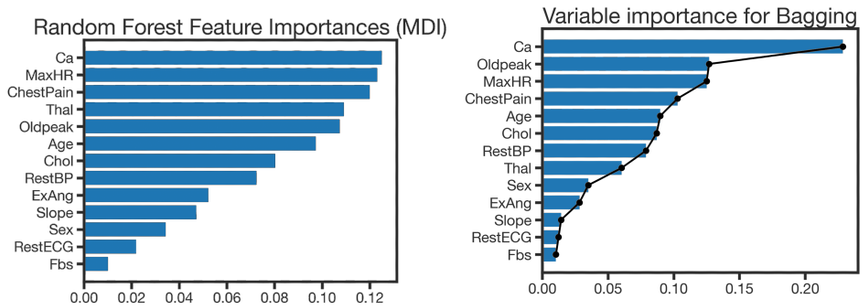
The biggest advantage of the MDI is a **speed of computation**. All needed values are computed during the Random Forest training.

The drawback of the MDI is its tendency to prefer numerical features and categorical features with **high cardinality**. In other words, it selects features with a high number of unique values as important. This is because random forests tend to select predictors that can lead to a one-to-one relationship with the target variable.

In the example shown, `Max_HR` is selected as an important feature in the MDI because of the high cardinality. Therefore, permutation importance might be preferred if the dataset contains features with high cardinality in order to avoid bias.

4.1.7 Variable Importance for Random Forest vs Bagging

Now, let's compare the importance of variables for bagging and random forest. We'll see if the results match what we expect. Note that the methods we're using do not always come up with the same rankings when it comes to which feature is most important. In fact, if they did, there would be no reason to have more than one method! Here's an example of how they might rate things differently:



In bagging we can easily see that a certain predictor appears to be dominant across all predictors. However, the random forest approach demonstrates a more evenly distributed variable importance across all predictors. This is due to the correlation between trees in bagging, as we discussed earlier. Bagging selects the same predictor and provides the best split out of all the predictors. In this case, "chest pain" will always appear on the root node in bagging, resulting in the predictor consistently giving the highest gain in impurity. However, in random forest, since it selects the best predictor from a subset of all predictors, "chest pain" may not always be on the root node. This is why random forests show a more evenly distributed variable importance across all predictors.

4.2 Missing data

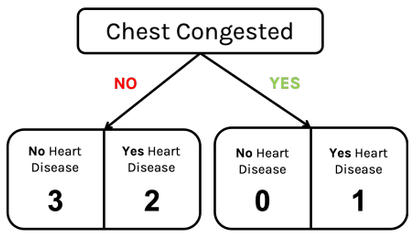
4.2.1 Motivation

Real-world datasets most often have missing values. To deal with missing data, we can impute it (for example, as we did at near end of the first course in this series). However, there is another strategy for dealing with missing data for trees. The basic idea is that during training, we find alternative splits, which we call Surrogate splits, that can be used during prediction.

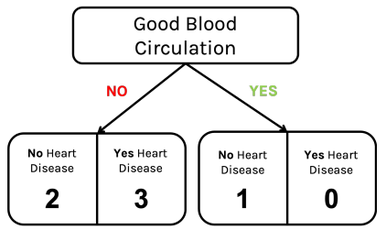
Imagine we have a dataset as shown below. We would like to train a model to predict whether a subject has heart disease or not, given multiple input features from Weight to Arteries Blocked. The goal is to predict the values in the Heart Disease column.

Weight (kg)	Height (cm)	Chest Congested?	Good Blood Circulation?	Arteries Blocked?	Heart Disease
58.3	125.3	No	No	No	No
65.7	193.2	Yes	No	Yes	Yes
75.2	112	No	Yes	No	No
112	165.7	No	No	Yes	Yes
45	135	No	No	No	No
40	120	No	No	No	Yes

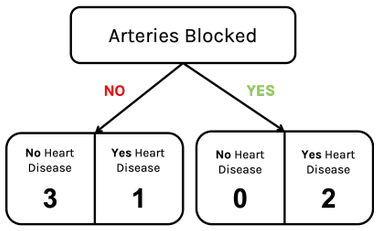
Example 1: Chest Congested As a first step, what we need to do is perform our usual tree analysis. Let's begin with the predictor *Chest Congested* and create a split.



Example 2: Good Blood Circulation For the predictor *Good Blood Circulation*, we follow the same procedure. If *Good Blood Circulation* is TRUE (YES), we have only one "NO" instance. However, if it is FALSE (NO), we have a total of 5 examples, with 3 "YES" instances and 2 "NO" instances.



Example 3: Arteries Blocked For the predictor *Arteries Blocked*, we follow the same procedure. If *Arteries Blocked* is TRUE (YES), we have two "YES" instances and no "NO" instances. However, if it is FALSE (NO), we have one "YES" instance and three "NO" instances.



Comparison Let's compare the split distributions of *Arteries Blocked* with the other two predictors, *Chest Congested* and *Good Blood Circulation*.

- Arteries Blocked
  - FALSE: [3 NOs, 1 YES]
  - TRUE: [0 NOs, 2 YES]
- Chest Congested
  - FALSE: [3 NOs, 2 YES]
  - TRUE: [0 NOs, 1 YES]

The difference in the distribution of instances between FALSE and TRUE for Chest Congested is 2.

- Good Blood Circulation
  - FALSE: [2 NOs, 3 YES]
  - TRUE: [1 NO, 0 YES]

The difference in the distribution of instances between FALSE and TRUE for Good Blood Circulation is 6.

Since *Chest Congested* exhibits **the most similar split distribution** to *Arteries Blocked*, we will take *Chest Congested* value in place of *Arteries blocked* missing value, thereby using *Chest congested* as a "Surrogate"!

What about continuous variable? What do you think is the surrogate for *Weight*? The logical guess would be *Height*!

During training, for every optimal split, we create a rank list of surrogate splits. This ranking is based on the similarity between the split distributions of the optimal predictor and every other predictor.

Some important points about Surrogate Splits

- Surrogates can help us understand the primary splitter.
- Surrogates performs better when there is multi-collinearity.
- There is no guarantee that useful surrogates can be found!

4.3 Imbalanced data

4.3.1 Imbalanced classes

Motivation Training a Random Forest (or any machine learning model) on an imbalanced dataset can introduce unique challenges to the learning problem. For example, if we use a biased dataset to train a model, the resulting model will also be biased. Unfortunately, data imbalance is quite common in the real world. In such cases, how can we effectively evaluate the model's performance?

When evaluating the model's classification accuracy, the most commonly used metric is Accuracy. In brief, Accuracy summarizes the model's performance by calculating the number of correct predictions divided by the total number of predictions

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Number of total predictions}}$$

Limitations of Accuracy score: COVID-19 test example Let's say we have a binary classification problem where we need to predict whether a person has a COVID or not. In our dataset, we have 1,000 samples, out of which only 10 samples belong to the positive class (people with COVID) and the remaining 990 samples belong to the negative class (people without COVID).

Now, let's assume we build a binary classification model. It correctly classifies 990 out of 990 negative samples, but it only correctly identifies 1 out of 10 positive samples. In short,

- True Positives (TP): 1
- False Positives (FP): 0
- True Negatives (TN): 990
- False Negatives (FN): 9

In this case, the Accuracy score would be 991/1000 = 0.991, which might seem impressive at first glance. However, this high Accuracy score is misleading because ***the model failed to correctly identify the majority of positive cases***.

4.3.2 Better Metrics in data imbalance

Recall In this scenario, Recall (also known as Sensitivity or True Positive Rate) is a good indicator. Recall measures the model's ability to identify all positive cases correctly. Given the imbalanced nature of the dataset with only 10 positive cases, a low Recall score would highlight the model's deficiency in accurately detecting the positive instances (COVID cases).

$$\text{Recall} = \frac{\text{True Positive}}{\text{True positive} + \text{False Negative}}$$

In our example above, the Recall would be $1/(1+9) = \mathbf{0.1}$. This metric provides better insight since we are more interested how much our model classifies positive cases correctly.

Precision Precision measures the model's ability to avoid false positives. Ideally, we want our model to maximize the portion of true positive cases out of all instances predicted as positive.

$$\text{Recall} = \frac{\text{True Positive}}{\text{True positive} + \text{False Positive}}$$

Likewise, in our example, the Precision would be $1/(1+0) = \mathbf{1}$. Since there is no false positive case, the Precision score is 1.

F1-score: Harmonic mean of Precision and Recall Both Recall and Precision are insightful metrics, especially in the presence of data imbalance. But how do we consider both simultaneously? The F1-score combines Precision and Recall by taking their harmonic mean External link. The harmonic mean is a useful metric when an average rate is desired.

$$F1 = \frac{2\text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}}$$

It gives equal weight to Precision and Recall, providing a balanced assessment of the model's performance by effectively considering both false positives and false negatives.

ROC Curve An ROC curve (Receiver Operating Characteristic curve) refers to a graph that shows the performance of a classification model at all classification threshold. Then, what does it mean by classification threshold?

Suppose we have a binary classification model that determines whether a subject is COVID positive or not (negative). This classification model will provide the predicted probability of positive test result every time it's applied to a subject.

Here is a hypothetical data example:

Subject (input)	Predicted Probability	Ground truth
Subject 1	0.95	Positive
Subject 2	0.86	Negative
Subject 3	0.75	Negative
Subject 4	0.88	Positive
Subject 5	0.89	Negative
Subject 6	0.80	Negative

The classification model will assign an estimated label based on a given threshold. If the predicted probability exceeds the threshold, the model will classify the subject as COVID positive.

Drawing an ROC curve requires two metrics:

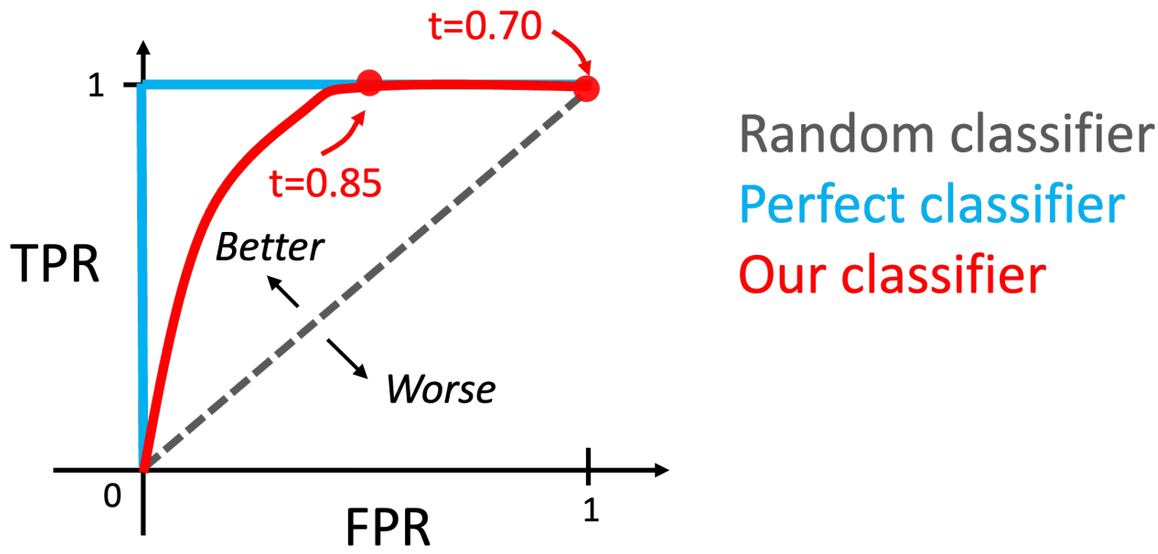
$$\text{True Positive Rate (TPR)} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

$$\text{False Positive Rate (FPR)} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

Let's consider two different threshold as below.

- **Treshold: 0.85:**
  - Predicted results:
    - Positive: Subject 1 (TP), Subject 2 (FP), Subject 4 (TP), Subject 5 (FP)
    - Negative : Subject 3 (TN), Subject 6 (TN)
  - True Positive Rate: 2/(2+0) = 1 (2 true positive out of 2 actual positive)
  - False Positive Rate: 2/(2+2) = 0.5 (2 false positive out of 4 actual negative)
- **Treshold: 0.70:**
  - Predicted results:
    - Positive: Subject 1 (TP), Subject 2 (FP), Subject 3 (FP), Subject 4 (TP), Subject 5 (FP), Subject 6 (FP)
    - Negative : None
  - True Positive Rate: 2/(2+0) = 1 (2 true positive out of 2 actual positive)
  - False Positive Rate: 4/(4+0) = 1 (4 false positive out of 4 actual negative)

The graph below represents the ROC curve with various threshold settings. Intuitively, it makes sense that both the TPR and FPR increase as the threshold decreases. In other words, if our model classifies everything as true, the number of false positive cases will also increase.

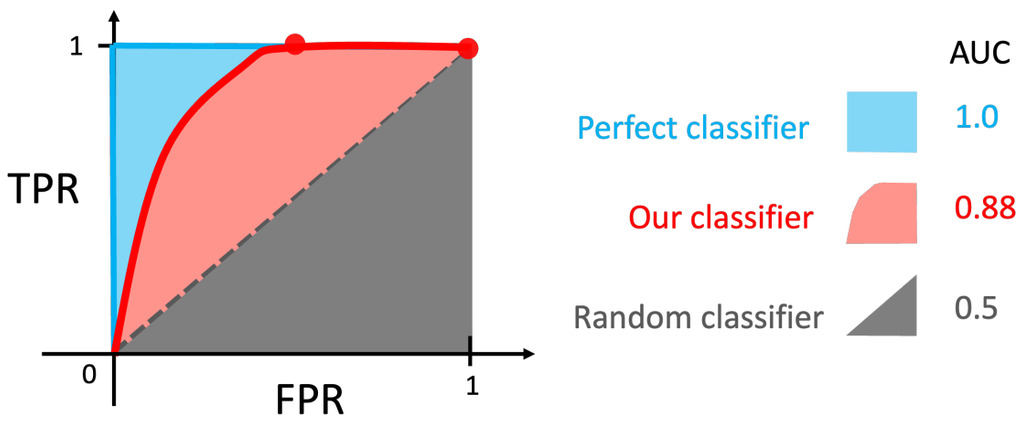


The diagonal line in the ROC curve represents the performance of a random classifier. Regardless of the threshold used, the true positive rate (TPR) and false positive rate (FPR) will remain the same. In other words, the random classifier provides no useful discrimination between the classes.

The goal of a good classifier is to achieve performance metrics that are better than random guessing. Therefore, we strive to position our classifier as far away from the diagonal line as possible. By doing so, we can maximize the TPR (capturing more true positives) while minimizing the FPR (limiting false positives).

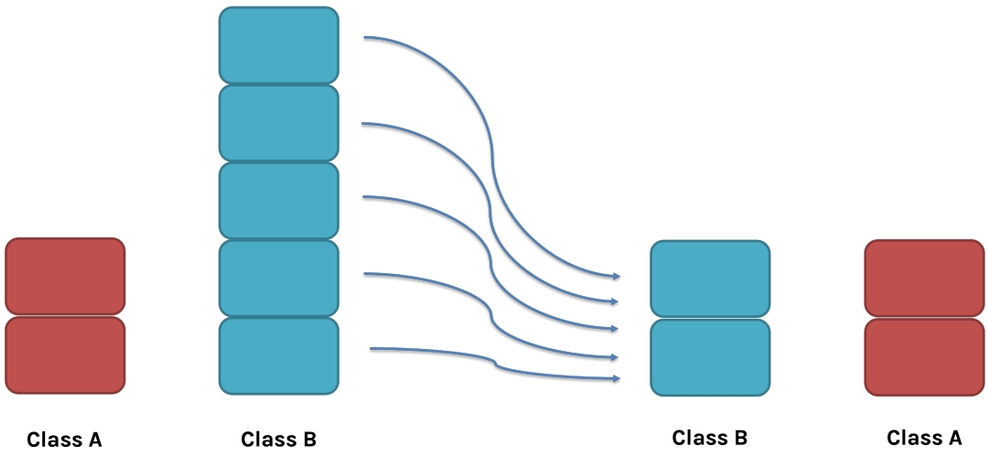
Area under the ROC Curve (AUC) The ROC curve allows us to find the classification threshold that gives the best False Positive Rate (FPR) and True Positive Rate (TPR) trade-off.

We summarize the ROC by computing the Area under the ROC Curve (AUC). For the perfect classifier the AUC is 1, the random classifier has AUC of 0.5.



4.3.3 Dealing with Imbalanced classes

Undersampling With undersampling, we intentionally reduce the number of samples in majority class to match the number of samples in minority class.

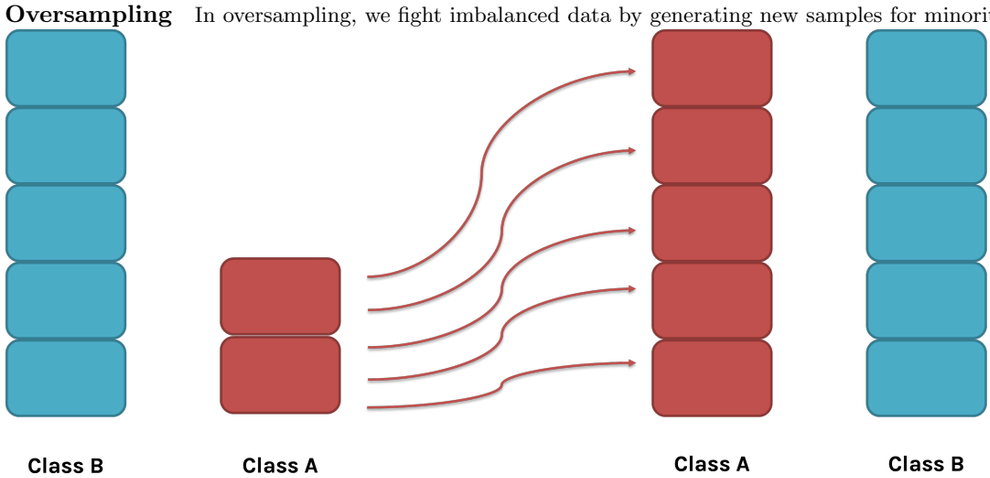


This can be done in two ways:

Random Sampling: Randomly sample from majority class *with* or *without replacement*.

Near Miss: Select data points by using simple heuristics like finding samples from which the average distance to some data points of minority class is smallest. Read more about it at imbalanced-learn.org.

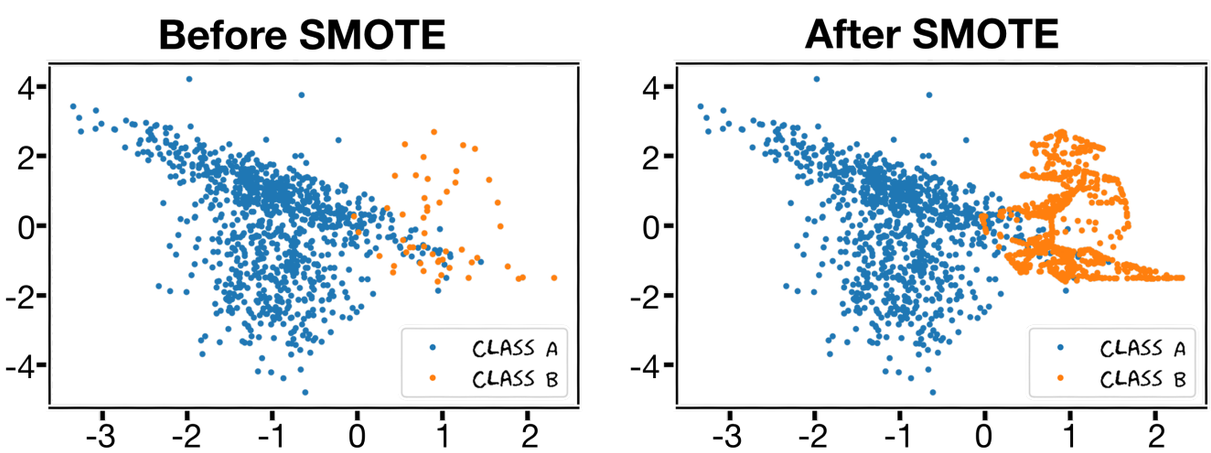
Oversampling In oversampling, we fight imbalanced data by generating new samples for minority class.



This can be done in two ways:

Random Sampling: Randomly sample from minority class *with replacement*.

SMOTE: SMOTE is an improved alternative for oversampling. SMOTE works by finding points that are closer in feature space, drawing a line between these points and generating new data points along this line. You can read more about SMOTE if you're interested.



Class weighting A simple way to address the class imbalance is to provide a **weight for each class** which places more emphasis on the minority classes.

In sklearn we can provide the class weight as a dictionary or use `class_weight = balanced`

Then it automatically adjust weights inversely proportional to class frequencies in the input data as:

- N is the total number of samples
- N_k is the number of samples in class K
- K is the total number of classes.

$$W_k = \frac{N}{K \cdot N_K}$$

Chapter 5

Boosting

Before studying this chapter is suggested to study, from the prerequisite book *"An Introduction to Statistical Learning"*, the section 8.2.3. Boosting - Pages 347 to 350.
Also review the section about **Decision trees** and **Random Forest Issues**.

Recap: Decision Trees

Shallow trees (*trees with a small number of leaves*) suffer from high bias and low variance. As a result, they do not train well.

Deep trees (*trees with a large number of leaves*) have low bias, but suffer from high variance, leading to very low generalizability.

Recap: Random Forest Issues

Variance: Although variance reduction is better in RF than bagging, the generalization error could still be high.

Speed: Large number of trees can make the algorithm very slow and ineffective for real-time predictors.

5.0.1 Motivation

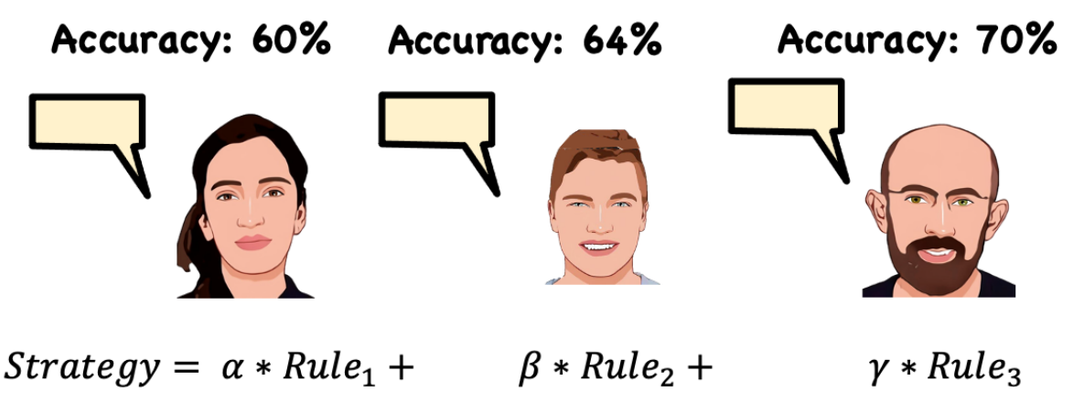
Random Forest and Boosting are two of the most used algorithms when you are working with tabular data (data that is in a spreadsheet format and that is not images, time series, audio, language, etc.).

5.1 Intro to boosting

Let's take an exam scenario. Let's imagine that the only passing grade in the exam is A. How would you go about passing the exam?

1. Go to the library and get previous year question papers.
2. Assemble a set of simple rules from different sources:
   (a) Find a helpful student and ask them to give you a "rule of thumb" to get at least some answers right. The advice that the student gives you is to never choose option "D" on the exam. Testing out the given rule you find out that it works 60% of the time.
   (b) Find a teaching assistant and ask them for a different "rule of thumb". This time the received rule is to always choose an option that has the word "overfitting" in it. You apply this rule only to the questions you got wrong in the previous step, and you get half of those questions correct now.
   (c) Call your professor and ask them for a third "rule of thumb". They tell you that the right answer is almost always "cross-validation". Apply this rule on the ones that you got wrong before.
3. Combine the rules together.

As a result of the first rule you get 60% correct answers, after the second one you get 64% correct, and after the last third rule 70% answers are correct. However, you have to decide which person you trust more. If your trust one person more than others, you put more weight to their rule. In your final strategy each rule will have some weight assigned to it:



Your final step is, of course, taking the test! Use a model consisting of the three simple rules combined appropriately.

Obtaining simple rules and combining them together with respective weights into one strong model is the basic idea of boosting. Boosting methods are general algorithms which combine several "weak learners" to produce a strong rule. In our example above, weak learners were "rules of thumb", in practice usually some sort of simple model is used, like a shallow tree.

5.2 Gradient Boosting

5.2.1 Recap of Boosting

The key intuition behind boosting is that one can take an ensemble of simple models $(T_h, h, \in \mathbb{H})$ and additively combine them into a single, more complex model.

Earlier we combined simple "rules of thumb" like that. Each model T_h might be a poor fit for the data, but a **linear combination** of the ensemble could be expressive/flexible:

$$T = \sum_h \lambda_h T_h$$

5.2.2 Gradient Boosting

Two main ideas of gradient boosting:

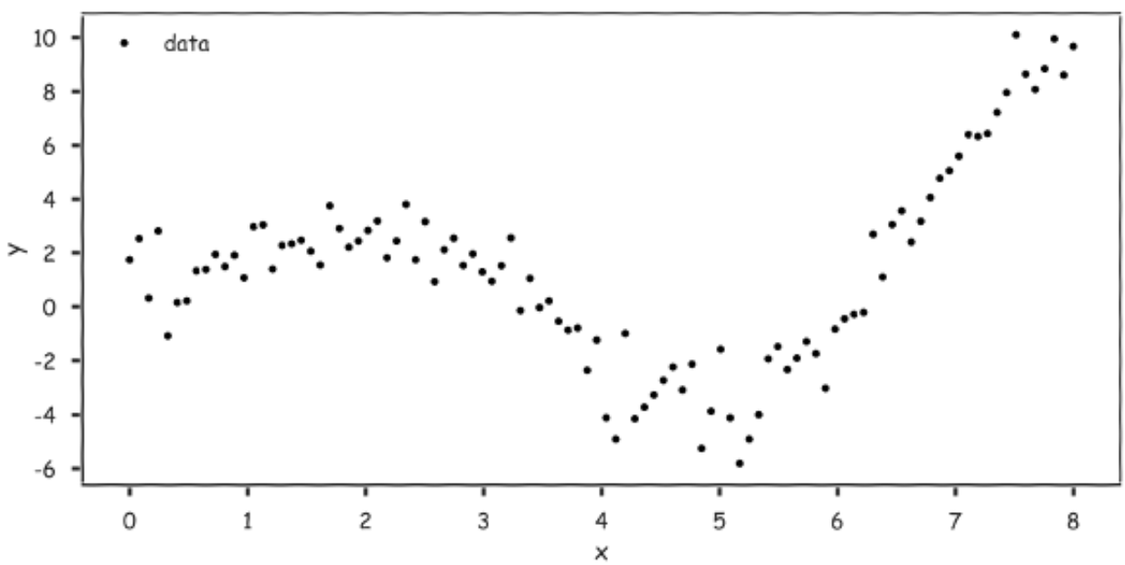
- It is done **iteratively**, in contrast to bagging and random forest. Each new added model would depend on the result of the previous models.
- Each new simple model added to the ensemble compensates for the **weaknesses** of the **current** ensemble.

Gradient boosting is a method for **iteratively** building a complex model T by *adding simple models*.

Algorithm 6 Gradient Boosting algorithm

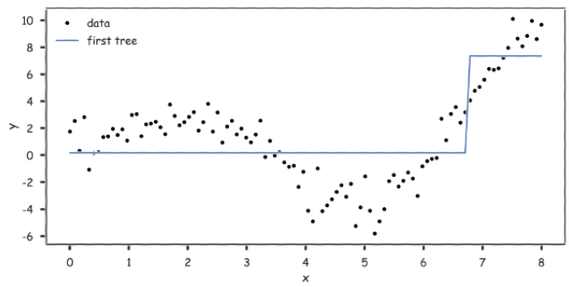
1: Fit a simple model $T^{(0)}$ on the training data $(x_1, y_1), \ldots, (x_N., y_N)$ for T. Set $T \leftarrow T^{(0)}$
2: Compute the residuals $r_1, \ldots r_N$ for T.
For $i = 1$ until stopping condition is met:
3: Fit a simple model, $T^{(i)}$, to the current residuals, that is, train the model using $\{(x_1, r_1), \ldots, (x_N, r_N)\}$.
4: Set the current model T to $T \leftarrow T + \lambda T^{(i)}$.
5: Compute residuals, set $r_n \leftarrow r_n - \lambda T^{(i)}(x_n), n = 1, \ldots, N$ where λ is a constant called the **learning rate**.

Gradient Boosting with an example Consider the dataset in the graph below:

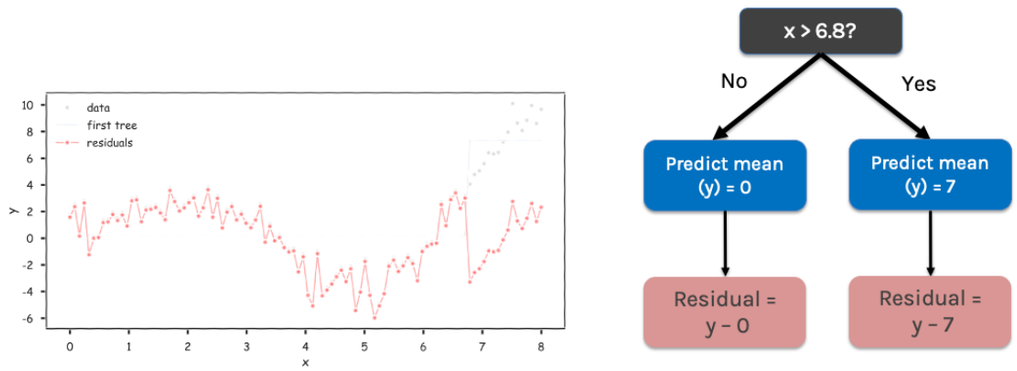


Let us use a very weak learner, a regression tree with only one split: the tree of depth one, aka a "**stump**"

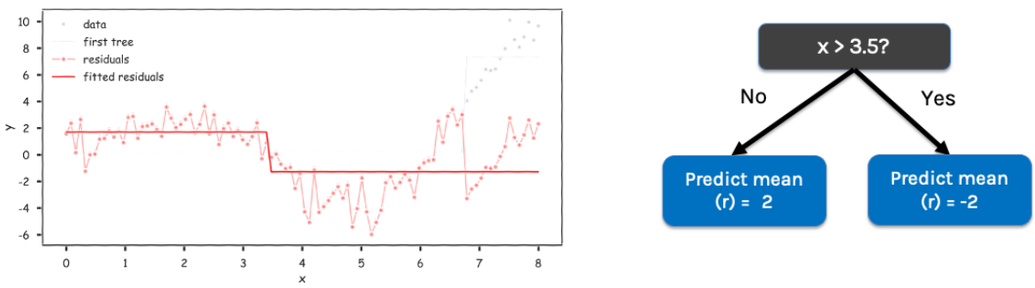
1 Fit a simple model $T^{(0)}$ on the training data $\{(x_1, y_1), \ldots, (x_N, y_N)\}$. The model $T^{(0)}$ has only one split at $x = 6.8$. If a point has x that is above $x = 6.8$ then the tree predicts the mean of the y values of the points that are above $x = 6.8$, which in this case is $\hat{y} = 7$. If a point has x that is below $x = 6.8$, then the tree predicts the mean of the points that are below $x = 6.8$, which would be $\hat{y} = 0$ in this example.



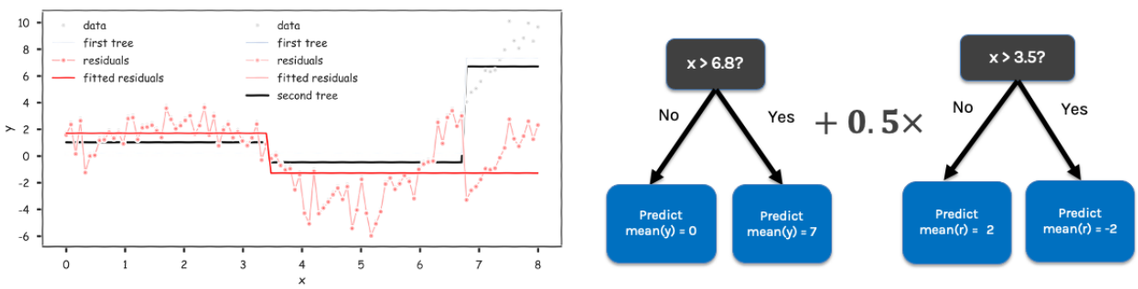
2 We want to learn from our mistakes. How could we express the mistakes of the algorithm here? We could look at the residuals! Compute the residuals $r_1, \ldots, r_N$ for $T^{(0)}$, and set $T \leftarrow T^{(0)}$.



3 Now, we will build a tree of the same depth to predict the residuals. It will be trained on the residuals that we got in Step 2, instead of the original data. Based on the data, $x = 3.5$ is the best split we can perform to minimize the weighted MSE of the residuals.
Fit another model $T^{(1)}$ on $r_1, \cdot, r_N$.



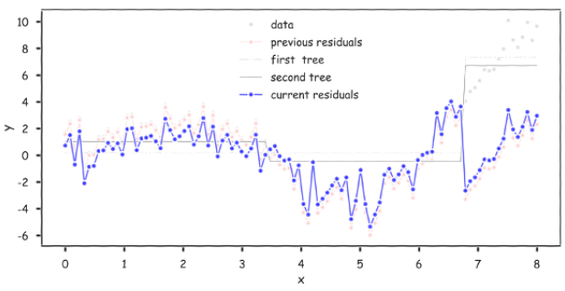
4 If we combine the first tree (from Step 1) and the second tree (from Step 3) we could get better predictions. We can create a third tree that will be a combination of the predictions of the first and second trees. Combine the two trees in Steps 1 and 3 by setting $T \leftarrow T + \lambda T^{(1)}$. Assume $\lambda = 0.5$. We will explain later how to set λ.

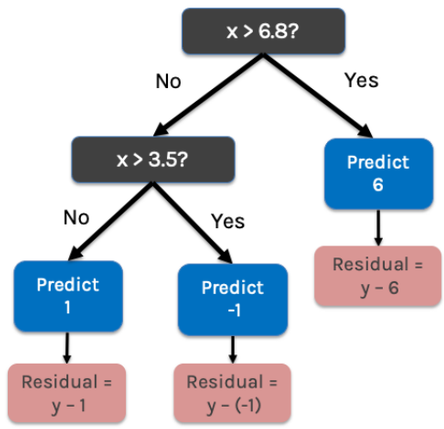


The third tree, which is a combination of the first two trees, predicts: - if an observation has x ¿ 6.8, the predicted value will be 7 + 0.5 • (-2) = 6. - if an observation has x ¿ 6.8 but x is not larger than 6.8, then the predicted value is 0 + 0.5 • -2 = -1 - if the observation's x is not larger than 3.5, the predicted value will be 0.5 • 2 = 1

The predictions could be represented as a deeper tree; however, in practice we do not create the deeper tree, we just keep adding the predictions to the existing ones.

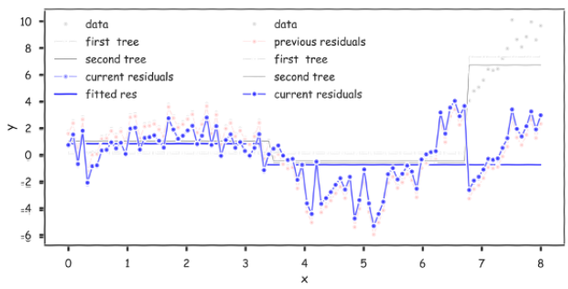
5 Repeat Step 2 on the new model by calculating the residuals $r_1, \cdot, r_N$ for current model, our third tree. This is a combination of the first tree fitted to the data and the second tree fitted to the calculated residuals of the first tree.

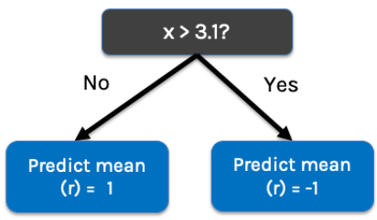




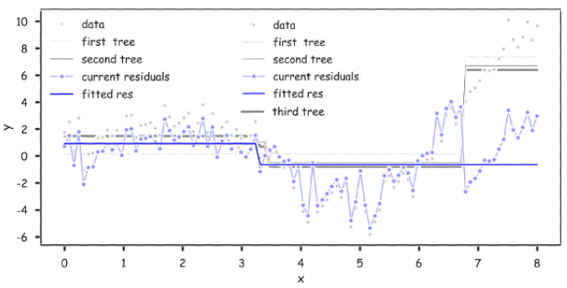
6 Repeat Step 3 and fit another simple tree model of the same depth of one $T^{(2)}$ on the new residuals calculated in Step 5:

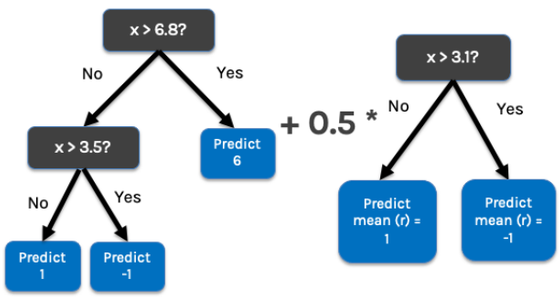
After fitting a new stump on the new residuals, we see the decision rule for the stump tree as follows: if $x > 3.1$ the model predicts -1, else it predicts 1.



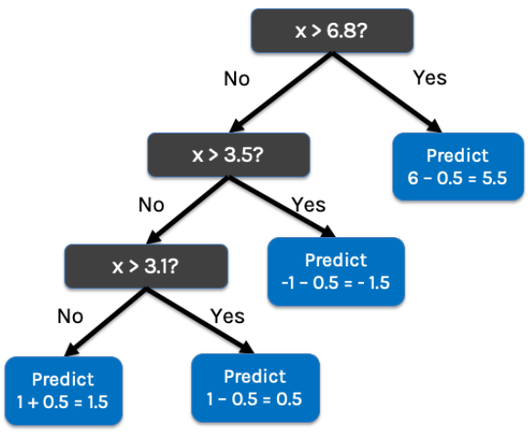


7 As before, we combine the third tree from Step 4 (our first combined tree) with the new stump model fitted to the residuals in Step 6 using some λ. Combine the two trees in Steps 4 and 6 by setting $T \leftarrow T + \lambda T^{(2)}$. Again we assume $\lambda = 0.5$.





We can represent the combination of two trees from Step 7 as one large tree as follows. (For better visualization, in reality, we are not building complex trees, we are just adding predictions).



We repeat the steps again and again until some stopping criteria are met, which we will introduce in a moment.

5.2.3 Termination

Under ideal conditions, Gradient Descent iteratively approximates and converges to the optimum.

When do we terminate Gradient Descent?

- We can **limit the number of iterations** in the descent. But for an arbitrary choice of maximum iterations, we cannot guarantee that we are sufficiently close to the optimum in the end.
- If the descent is stopped when the **updates are sufficiently small** (e.g. the residuals of T are small), we encounter a new problem: the algorithm may never terminate!

Both problems have to do with the magnitude of the learning rate, λ.

5.3 Why does Gradient Boosting work?

Intuitively, it makes sense that we are adding corrections to the model. Each simple model $T^{(i)}$ that we add to our ensemble T, models the residuals of T.

Thus, with each addition of $T^{(i)}$, the residuals are reduced: $r_n \leftarrow r_n - \lambda T^{(i)}(x_n)$.

Intuitively, it makes sense that we are adding corrections to the model. Each simple model $T^{(i)}$ that we add to our ensemble T, models the residuals of T.

Thus, with each addition of $T^{(i)}$, the residuals are reduced: $r_n \leftarrow r_n + -\lambda T^{(i)}(x_n)$.

The claim is that the residuals will continue to reduce with every new iteration of the algorithm. Therefore, the model will improve. Can we prove it?

We started by modeling $T^{(0)}$ that gave us some predictions $T^{(0)}_{pred}$:

$$T^{(0)}_{pred} = T^{(0)}(x)$$

When we train a new tree $T^{(1)}$ on residuals of the previous tree $T^{(0)}$, the new tree $T^{(1)}$ will **approximate** the residuals:

$$T^{(1)}_{pred} \approx T^{(0)}(x)$$

We will call $y - T^{(0)}_{pred}$ "residuals zero" $r^{(0)}$.
Finally, we combine $T^{(0)}$ and $T^{(1)}$ with some lambda:

$$T \leftarrow T^{(0)} + \lambda T^{(1)}$$

and the residuals of this model T would be:

$$y - T_{pred} = y - T^{(0)}_{pred} + \lambda T^{(1)}_{pred}$$

$$y - T_{pred} = r^{(0)} + \lambda T^{(1)}_{pred}$$

Therefore, as we combine the models, the residuals keep decreasing:

$$r_n \leftarrow r_n - \lambda T^{(i)}(x_n)$$

What is λ? And how is λ connected to anything we know? In particular, how can we effectively **descend** through this optimization via an **iterative** algorithm?

We need to formulate gradient boosting as a type of **Gradient Descent**.

Review: A Brief Sketch of Gradient Descent In **optimization**, when we want to minimize a function, called the **objective function (or loss function)**, over a set of parameters, we compute the **partial derivatives** of this function with respect to the parameters. The goal is to find a set of parameters that set these partial derivatives to zero. Computing partial derivatives of a function is called **taking a gradient**, which gives us a vector.

If the partial derivatives are sufficiently **simple**, one could analytically find a common root, i.e. a point at which all the partial derivatives vanish; this is called a **stationary point**.

If the objective function has the property of being **convex**, it will have only one minimum point that we call **global minimum**. A minimum point is the point of a function where its derivative is equal to zero. If the function is not convex, it might have more than one minima, and therefore you might have a "local" minima which is not in fact global.

In practice, if our objective functions are complicated, analytically finding a stationary point is intractable. In this case, we use an iterative method called **Gradient Descent**.

1. Initialize the parameters of the objective/loss function to any value.

   The ω values below are the parameters that we are minimizing over. For example, in logistic regression the parameters would be the betas (β).

   $$\omega = [\omega_1, \omega_2, \ldots, \omega_j]$$

2. Take the gradient of the objective function, f, at the current parameter values.

$$\nabla_\omega f(\omega) = [\frac{\partial f}{\partial \omega_1}(\omega), \frac{\partial f}{\partial \omega_2}(\omega), \ldots, \frac{\partial f}{\partial \omega_j}(\omega),]$$

3. Adjust the parameters by some negative multiple of the gradient. By doing so, we are moving step by step in the opposite direction of the gradient.

$$\omega \leftarrow \omega - \lambda \nabla_\omega f(\omega)$$

where λ is the learning rate.

Gradient descent can be a fairly complex topic. If you find that you need more information, there are good references at IBM or Corenll, or MIT's Open Learning Library.

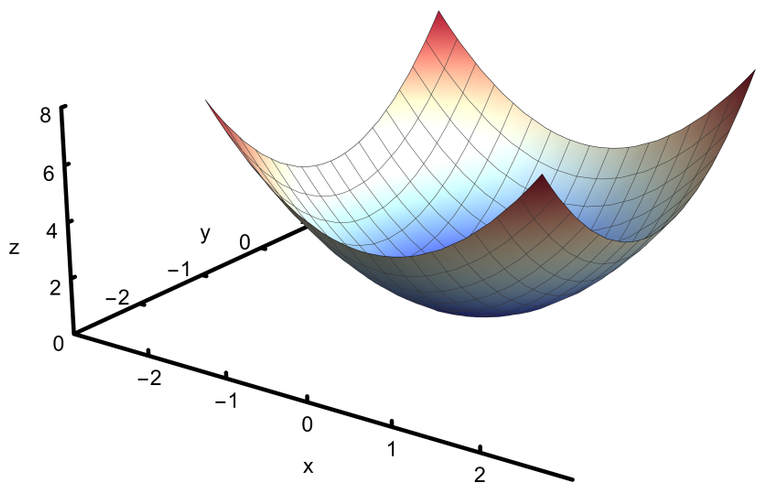
5.3.1 Why does Gradient Descent work?

Claim: If the function is convex, this iterative method will eventually move x close enough to the minimum, for an appropriate choice of λ.

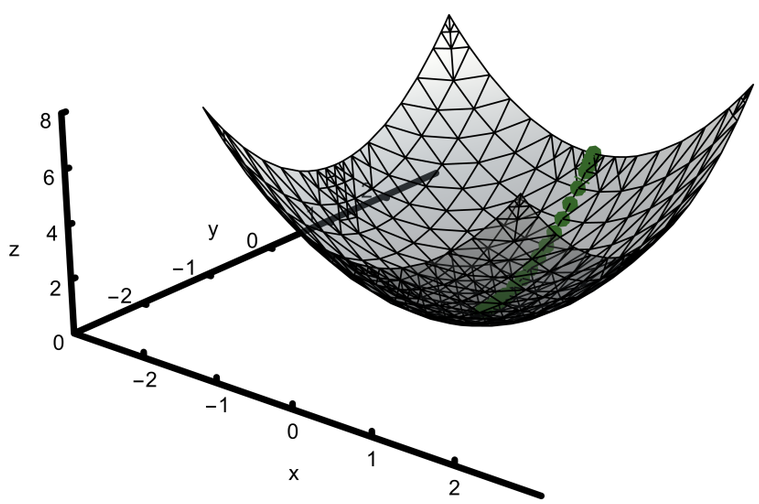
Why does it work? Recall that, as a vector, the gradient at a point gives the direction for the greatest possible rate of increase. In other words, at any moment, the partial derivatives show us the direction of the steepest ascent. The negative of the partial derivatives will give us the direction of the steepest descent.

Subtracting a λ multiple of the gradient from x, moves x in the opposite direction of the gradient (hence towards the steepest decline) by a step of size λ.

So if we start with a parabolic function like this one:

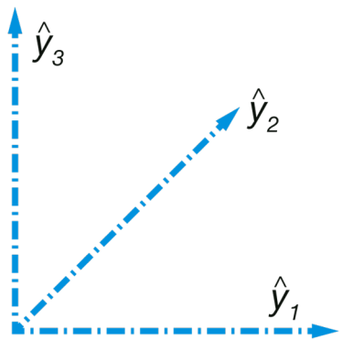


Then we'll be able to use this technique to take steps down the parabola from any point, descending the graph of f and eventually reaching the minimum.



5.3.2 Gradient Boosting as Gradient Descent

Prediction space Assume for now that we have 3 training examples $(x_1, y_1), (x_2, y_2), (x_3, y_3)$. Let us look at the space of predictions.



In this space, each point represents our set of predictions $y = (\hat{y}_1, \hat{y}_2, \hat{y}_3)$ for all data points. Note that here we have not assumed any relation between and whatsoever; we are just optimizing over the space of predictions.

The goal is to find the predictions that will minimize our loss function. The loss for which we are optimizing is the MSE:

$$L(y) = \frac{1}{3}\left[(y_1 - \hat{y}_1)^2 + (y_2 - \hat{y}_2)^2 + (y_3 - \hat{y}_3)^2\right]$$

As we discussed earlier, if we take the partial derivative and equate it to zero, we will find the minimum of the loss. If we change our prediction in some direction, what should $\hat{y}_1$, $\hat{y}_2$, and $\hat{y}_3$?

$$\frac{\partial L}{\partial \hat{y}_1} = -\frac{2}{2}(y_1 - \hat{y}_1)$$

$$\frac{\partial L}{\partial \hat{y}_2} = -\frac{2}{2}(y_2 - \hat{y}_2)$$

$$\frac{\partial L}{\partial \hat{y}_3} = -\frac{2}{2}(y_3 - \hat{y}_3)$$

The stationary point should be at:

$$-\frac{2}{2}(y_1 - \hat{y}_1) = 0$$

$$-\frac{2}{2}(y_2 - \hat{y}_2) = 0$$

$$-\frac{2}{2}(y_3 - \hat{y}_3) = 0$$

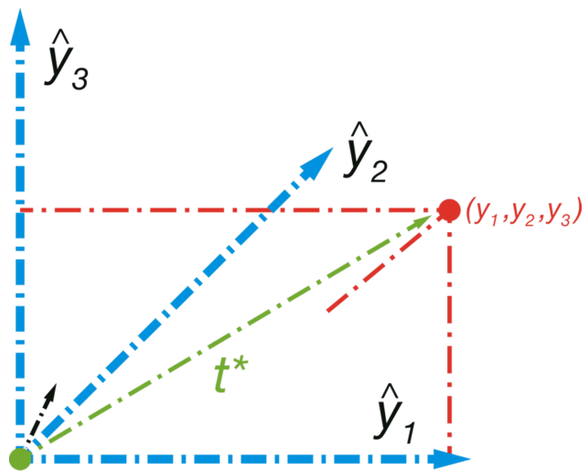
What will be the values $\hat{y}_1$, $\hat{y}_2$, and $\hat{y}_3$ that minimize the equations above?

It will be $\hat{y}_1$, $\hat{y}_2$, and $\hat{y}_3$ themselves. Here we have a very trivial solution, which is that we can set our prediction to be the true values, in this case our loss is zero, and we are done. :)

However, is there any problem with that? What has the model learned?

- Nothing!!

Here, as the loss function is convex, we already know the unique minimizer $y^* = (y_1, y_2, y_3)$. So, if we start at the origin (or anywhere else), we will end at y^* when we optimize using gradient descent. That means that if we walk into a prediction space that has a very trivial solution, where the function is very convex, we do not even need gradient descent, we can just go straight for y_1, y_2, y_3, and that is a problem.



More generally, we have:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Treating this as an optimization problem, we can try to directly minimize the MSE with respect to the predictions:

$$\nabla_{\hat{y}} MSE = \left[\frac{\partial MSE}{\partial \hat{y_1}}, \ldots, \frac{\partial MSE}{\partial \hat{y_N}}\right]$$

$$\nabla_{\hat{y}} MSE = -2\left[y_1 - \hat{y_1}, \ldots, y_N - \hat{y_N}\right]$$

$$\nabla_{\hat{y}} MSE = -2\,[r_1, \ldots, r_N]$$

The updated step for the gradient descent would look like:

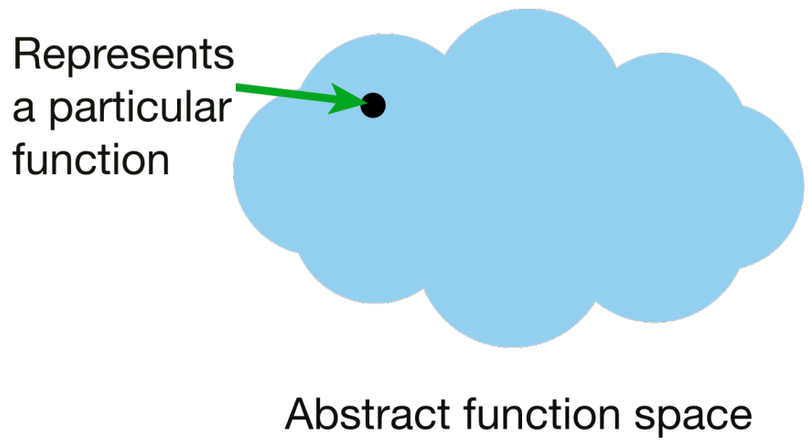
$$\hat{y_n} \leftarrow \hat{y_n} + \lambda r_n, n = 1, \ldots, N$$

There are two reasons why minimizing the MSE with respect to $\hat{y}_n$ is not interesting:

- We know where the minimum MSE occurs: $\hat{y_n} = y_n$, for every n.
- Learning sequences of predictions, $\{\hat{y_n^{(1)}} \ldots \hat{y_n^{(1)}} \ldots\}$ does not produce a model. This is because we have by no means learned to map predictors to predictions.

When trying to view Gradient Boosting as a form of Gradient Descent, we need to look at the **function space**.

Imagine a space where each point represents a function $f : X \rightarrow y$, then ideally we want to optimize over this space to find the best estimator. A point on the picture below represents one function, and in our world a function will be a tree (it could be a logistic regression, but we will talk about trees).



This space represents all possible trees that we can imagine. The point is to find the tree in that space that will give us the best results, and that is impossible. Because this space is huge, it is infinite!

What do we do in this case?

- We go into what is called parametric space.

In case of parametric models, we optimize our model in the weight space (parameters of our model).

$$\hat{f} := [\omega_1, \omega_1, \ldots, \omega_N]$$

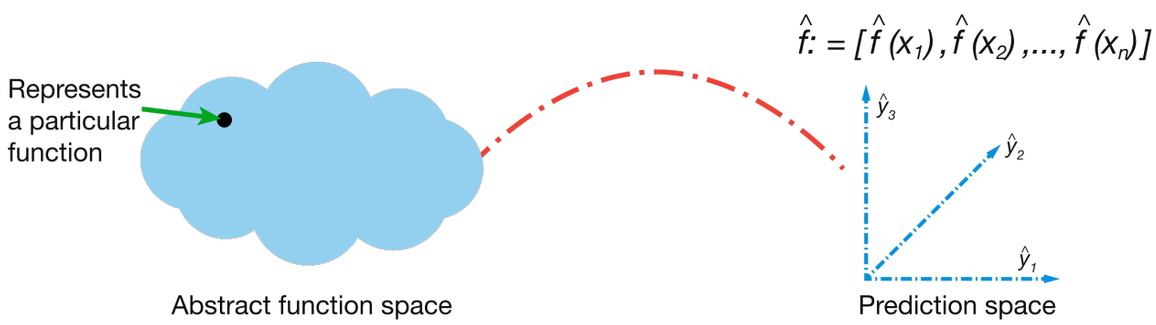
We read := as "f-hat is represented by".

When we go to parametric space, everything will become much easier since f will now depend on some ω-s, and the big scary cloud above becomes something much simpler - it becomes the parameter space. And now we probably can manage the task in that space, like we did when working with linear regression and logistic regression.

What about non-parametric models? We have a problem, we cannot do an optimization in a non-parametric space as such.

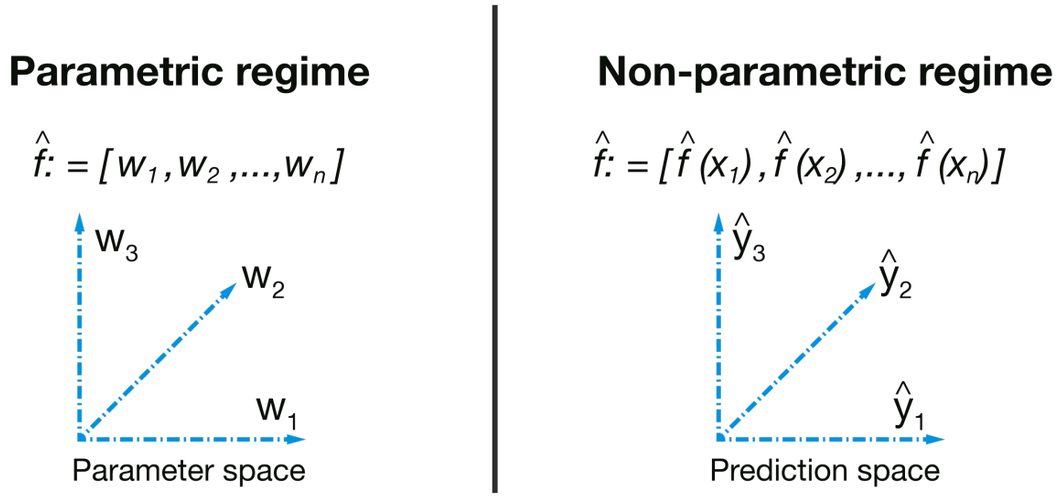
Trees are non-parametric models, because we are working in a non-fixed parametric space, we do not know how many splits and how many leaves the tree will have.

In non-parametric models, like trees, we do not have parameter space, we only have prediction space.



We do not have parameters to optimize on, and if we optimize over the predictions, we get into the trivial solution which was mentioned before.

In summary, for parametric models (say neural nets), gradient descent is done in the parameter space. Boosting can be thought of as doing gradient descent in the prediction space.



How do we deal with working in prediction space?

Let our current estimate be $T^{(n)(x)}$. In the prediction space this function is just a vector of the prediction values. In this case, the optimal direction to take a step using Gradient Descent is along the **negative gradient**.

$$t^* = -\frac{\partial \mathcal{L}}{\partial T^{(n)}(x)}$$

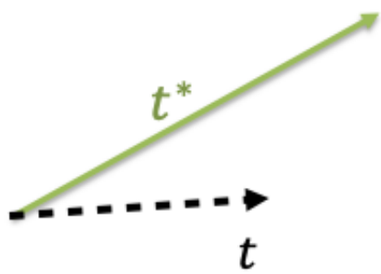
$$t^* = -\left[\frac{\partial \mathcal{L}}{\partial \hat{y}_1}, \frac{\partial \mathcal{L}}{\partial \hat{y}_2}, \ldots, \frac{\partial \mathcal{L}}{\partial \hat{y}_N}\right]$$

$$t^* = -\left[(\hat{y}_1 - y_1). \ldots, (\hat{y}_N - y_N)\right]$$

$$t^* = 2\left[r_1, r_2, \ldots, r_N\right]$$

Hence, the optimal direction to step in along the **residuals**.

If we are restricting to a simple family of functions, we might not be able to always take a step in this optimal direction t^*. But we want to take as close a step to this direction as possible. Let $\mathbb{H}$ be a simple family of models (say Decision Trees of `max_depth` $= 5$).



Then we need to find some $t \in \mathbb{H}$ such that $t = [t(x_1), t(x_2), \ldots, t(x_N),]$ is as close to $t = [r_1, r_2, \ldots, r_N]$ as possible. Well, this is quite easy to do if we train another model in $\mathbb{H}$ on the training set $(x_1, r_1), (x_2, r_2), \ldots, (x_N, r_N)$.

So, we are changing our prediction in a way that we go to as close as possible to the true (optimal) y in small steps, and in that way it allows us to connect the input with the output. We start with $t^{(*)}$ and find a model that approximates that direction (t).

$t^{(*)}$ is our true residual. Our tree is approximating the true residual. We end up going in a similar, but not exactly the same direction as $t^{(*)}$.

In summary, we model the gradients and we follow those gradients in order to get to the minimum, and that is how we connect the input to the output.

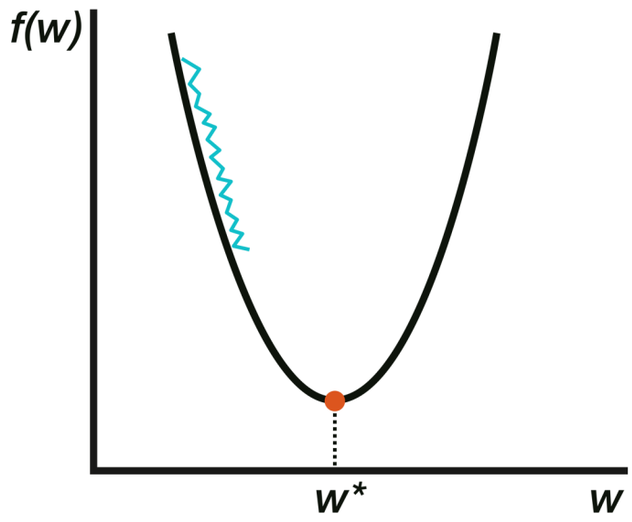
Hence instead of using the gradient (the residuals) we use an **approximation** of the gradient that depends on the predictors:

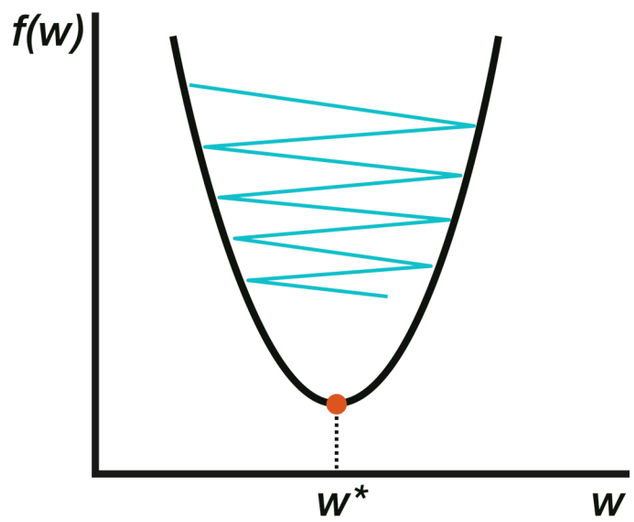
$$\hat{y}_n \leftarrow \hat{y}_n + \lambda \hat{r}_n(x_n), n = 1, 2, \ldots, N$$

In Gradient Boosting, we use a simple model $\hat{r}_n(x_n)$, to approximate the residuals $\hat{r}_n(x_n)$ in each iteration (which is the negative gradient of the loss in the prediction space).

Why do we want to connect gradient boosting to gradient descent? By making this connection, we can import the massive amount of techniques for studying gradient descent to analyze gradient boosting.
For example, we can easily reason about how to choose the learning rate λ in gradient boosting.

For a constant learning rate λ, if λ is too small (upper picture), it takes too many iterations. If λ is too large, the algorithm may "bounce" around the optimum and never get sufficiently close (lower picture).





Briefly, choosing λ:

- If λ is a constant, then it should be tuned through cross validation.
- For better results, use variable λ. That is, let the value of λ depend on the gradient:

$$\lambda = h(||\nabla f(x)||)$$

  where $\nabla f(x)$ is the magnitude of the gradient.

So:

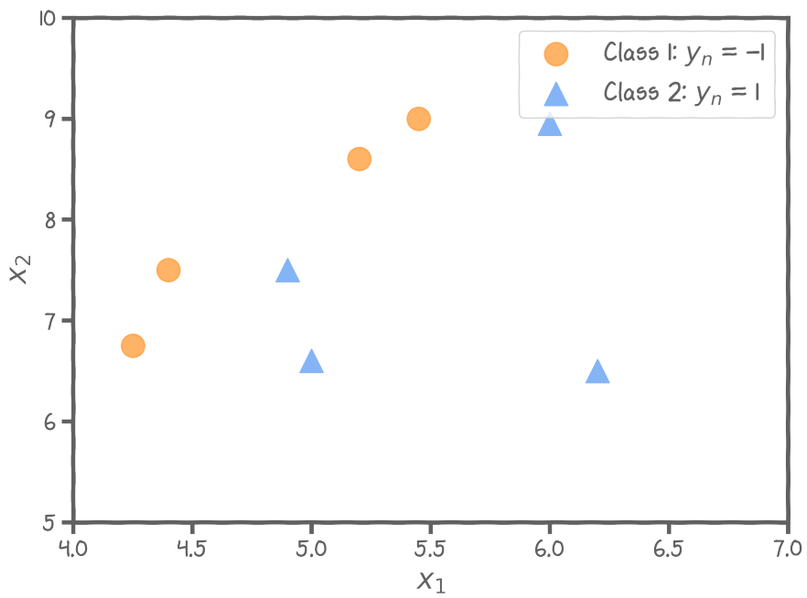
- Around the optimum, when the gradient is small should be small.
- Far from the optimum, when the gradient is large, should be larger.

Chapter 6

AdaBoost Algorithm

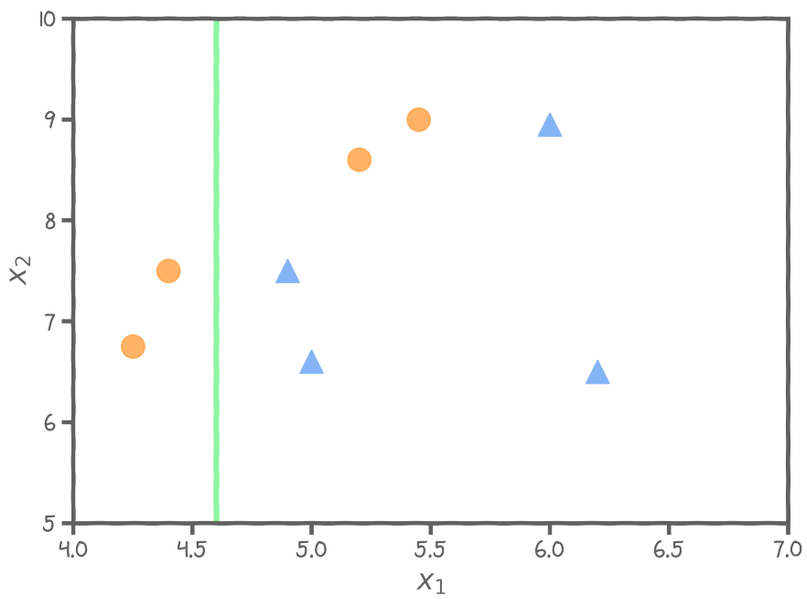
6.1 AdaBoost Algorithm – example

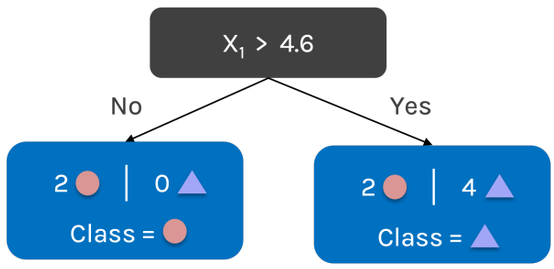
Consider the following dataset, with 2 predictors x_1 and x_2, where all data points are classified into either Class 1 $y_n = 1$ (represented by orange circles) or Class 2 $y_n = -1$ (represented by blue triangles).



Let's work through AdaBoost's iterative process:

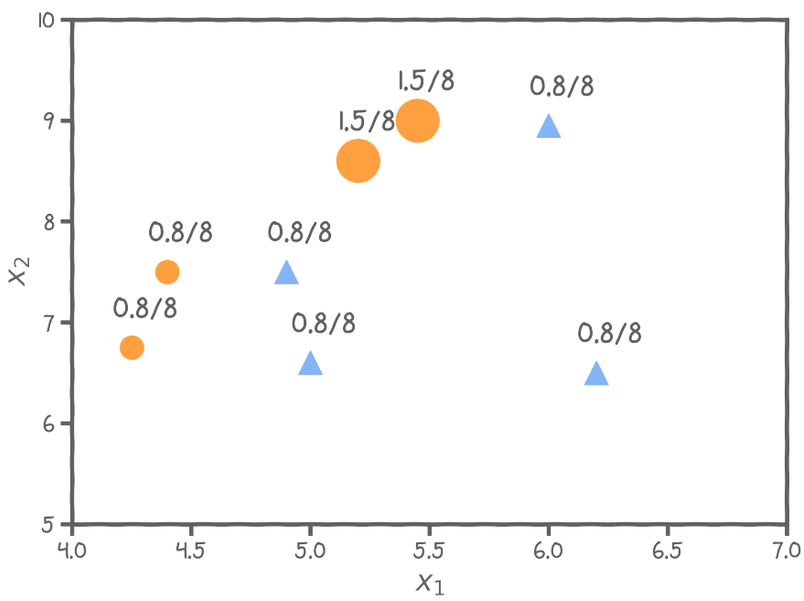
Step 1: Fit a stump $S^{(0)}$ on the dataset.
This will divide the data points into 2 based on the choice of split.





Step 2: Assume we initialize each data point to have equal weight of $\frac{1}{N}$. Then calculate the total error in the stump using:

$$\epsilon^{(0)} = \sum_{n=1}^{N} \omega_n^{(0)} \mathcal{I}(y_n \neq S^{(0)}(x_n))$$

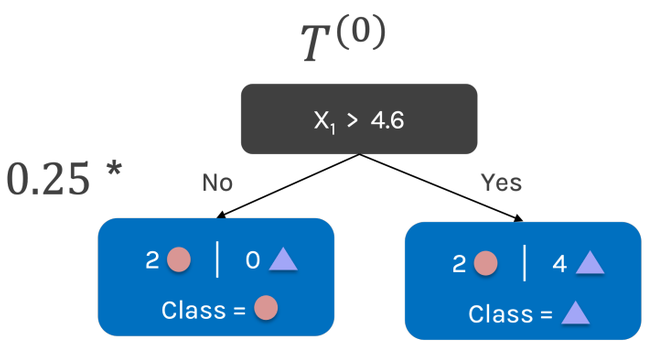


Splitting the data points using $x_1 > 4,6$ results in a total error of Error$(\epsilon) = \frac{1}{8} + \frac{1}{8} = \frac{2}{8}$

Step 3: Now that we have the first weak learner, we will assign the stump a scaling factor, $\lambda^{(0)}$, that indicates how much it **contributes to the entire ensemble**.
We assign λ to each successive stump as it offers some flexibility, and we can give more importance to stumps that perform better. Let this model's scaling factor λ be 0.25.

Step 4: Construct the **ensemble model** $T^{(0)}$ using:

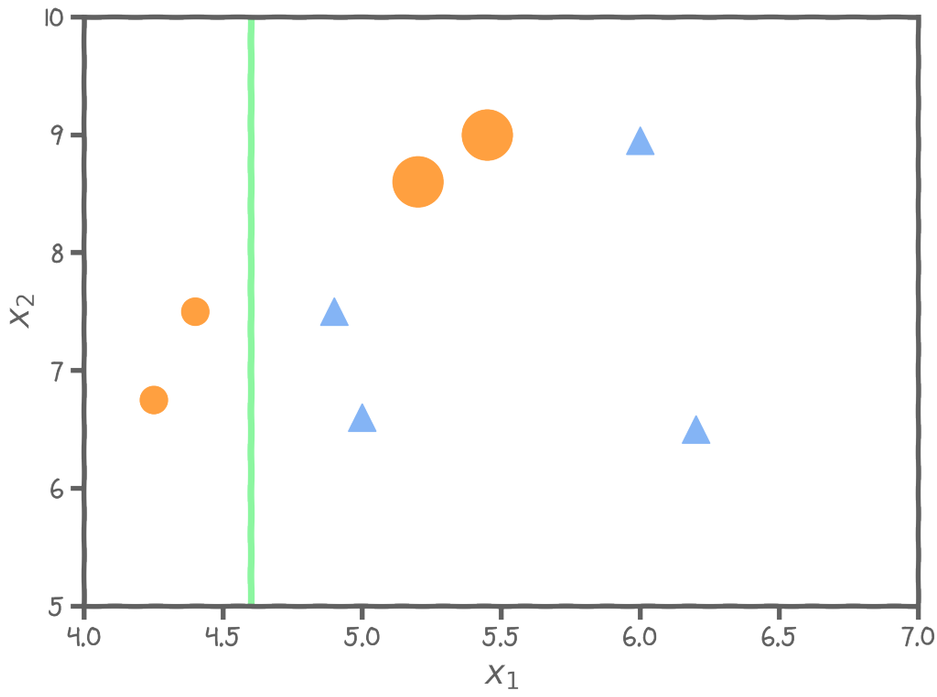
$$T^{(i)} \leftarrow \begin{cases} \lambda^{(i)} S^{(i)}, & i = 0 \\ T^{(i-1)} + \lambda^{(i)} S^{(i)}, & i = 1, 2, \ldots \end{cases}$$



Step 5: Adjust the weights assigned to each data point to ensure the next stump focuses on the points **misclassified by the ensemble** using the following equation:

$$\omega_n^{(1)} \leftarrow \frac{\omega_n^{(0)} e^{-\lambda y_n T^{(0)}}(x_n)}{z} = \frac{\omega_n^{(1')}}{z}$$

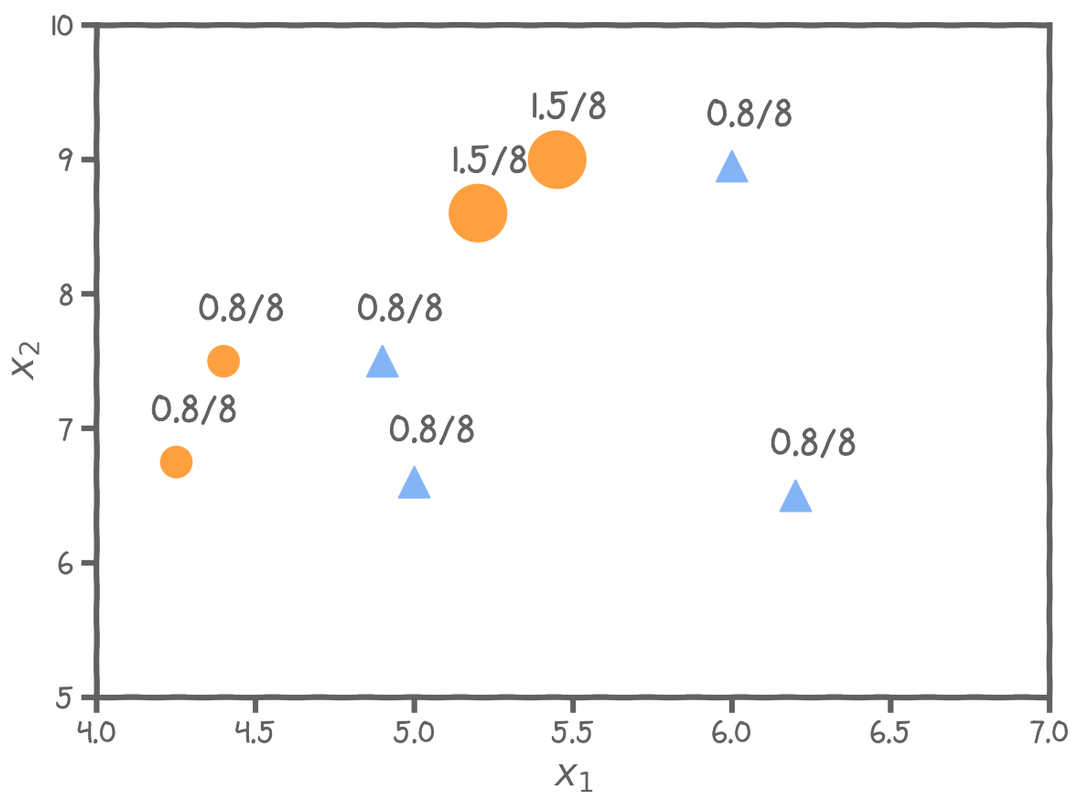
This increases the weights of misclassified samples and decrease the weights of correctly classified samples.



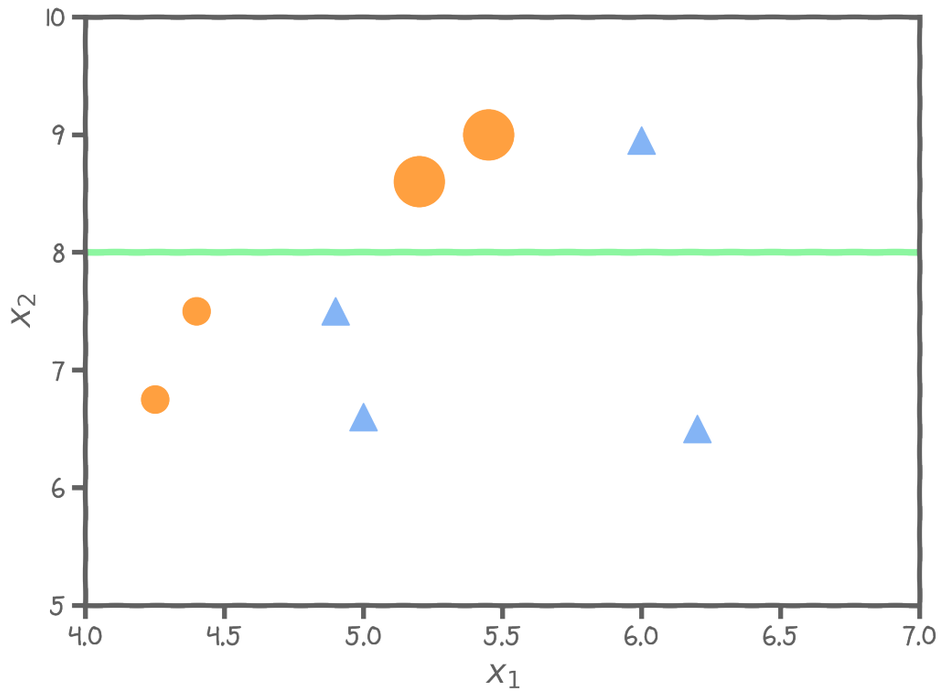
From the previous equation, we have $\omega_n^{(i)}$ as the new weights which is computed from the old weights, $\omega_n^{(0)}$. y_n is the groud truth which is either -1 or 1. The prediction from the previous ensemble is represented by $T^{(0)}(x_n)$.
The new weight is obtained post normalization by a factor z.
Thus, for our example we have;

$$\omega^{(1)} \leftarrow \frac{\omega^{(1')}}{z} = \frac{\frac{1.3}{8}}{2\frac{1.3}{8} + 6\frac{0.7}{8}} \approx \frac{1.5}{8}$$

$$\omega^{(1)} \leftarrow \frac{\omega^{(1')}}{z} = \frac{\frac{0.7}{8}}{2\frac{1.3}{8} + 6\frac{0.7}{8}} \approx \frac{0.8}{8}$$



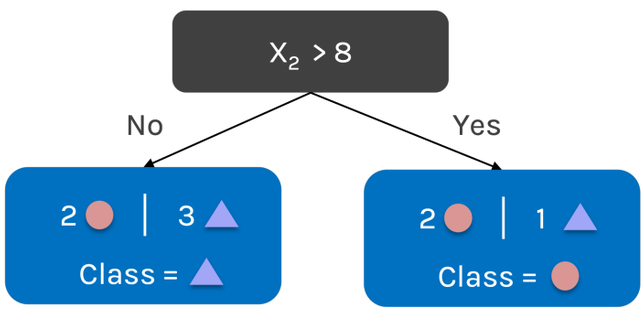
Step 6: Create another stump $S^{(1)}$ on the re-weighted data.



It is important to notice that the new stump has correctly classified the data points that the previous one misclassified.

Step 7: With the new weights, calculate the total error in the stump using:

$$\epsilon^{(1)} = \sum_{n=1}^{N} \omega_n^{(1)} \mathcal{I}(y_n \neq S^{(1)}(x_n))$$

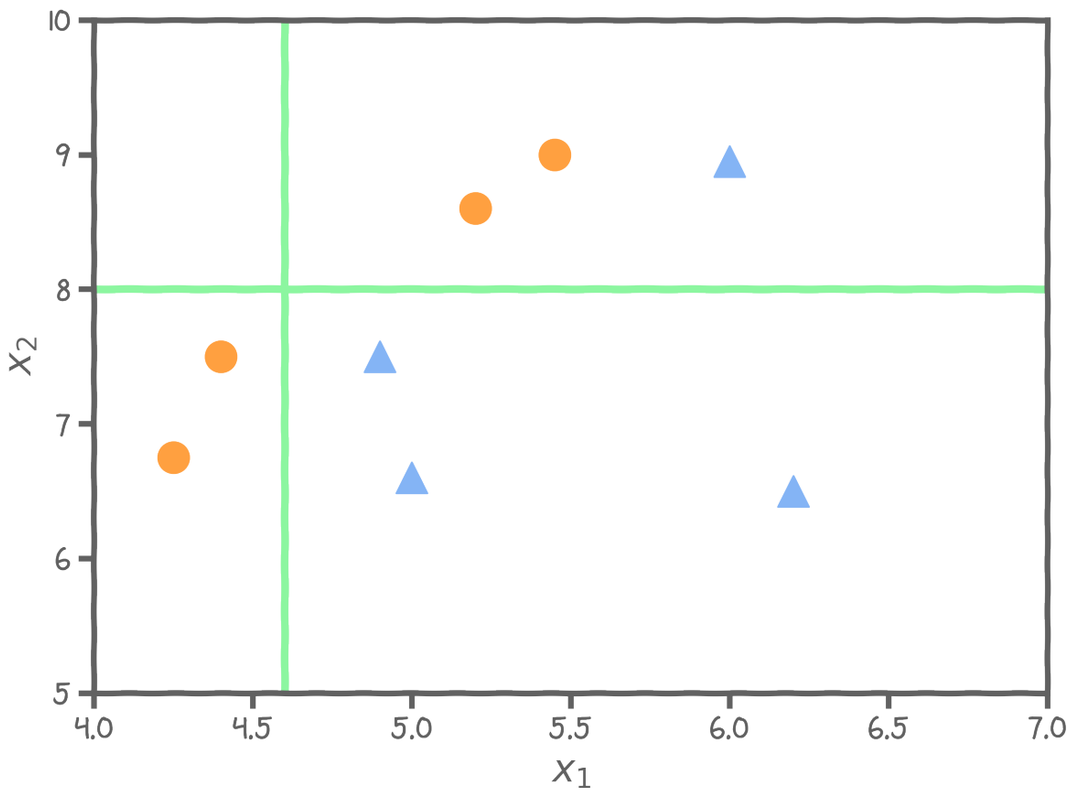


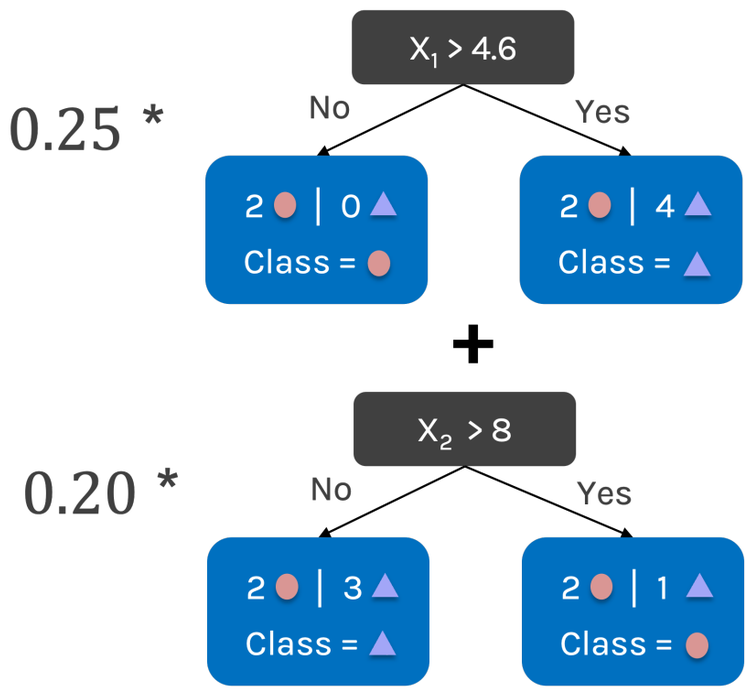
Splitting the data points using x2 ¿ 8 results in a total error of Error(ϵ) = $\frac{0.8}{8} + \frac{0.8}{8} + \frac{0.8}{8} = 0.3$

Step 8: Assign the stump a scale, $\lambda^{(1)}$, that indicates how much it **contributes to the entire ensemble**. Let this model weight $\lambda^{(1)}$ be 0.20.

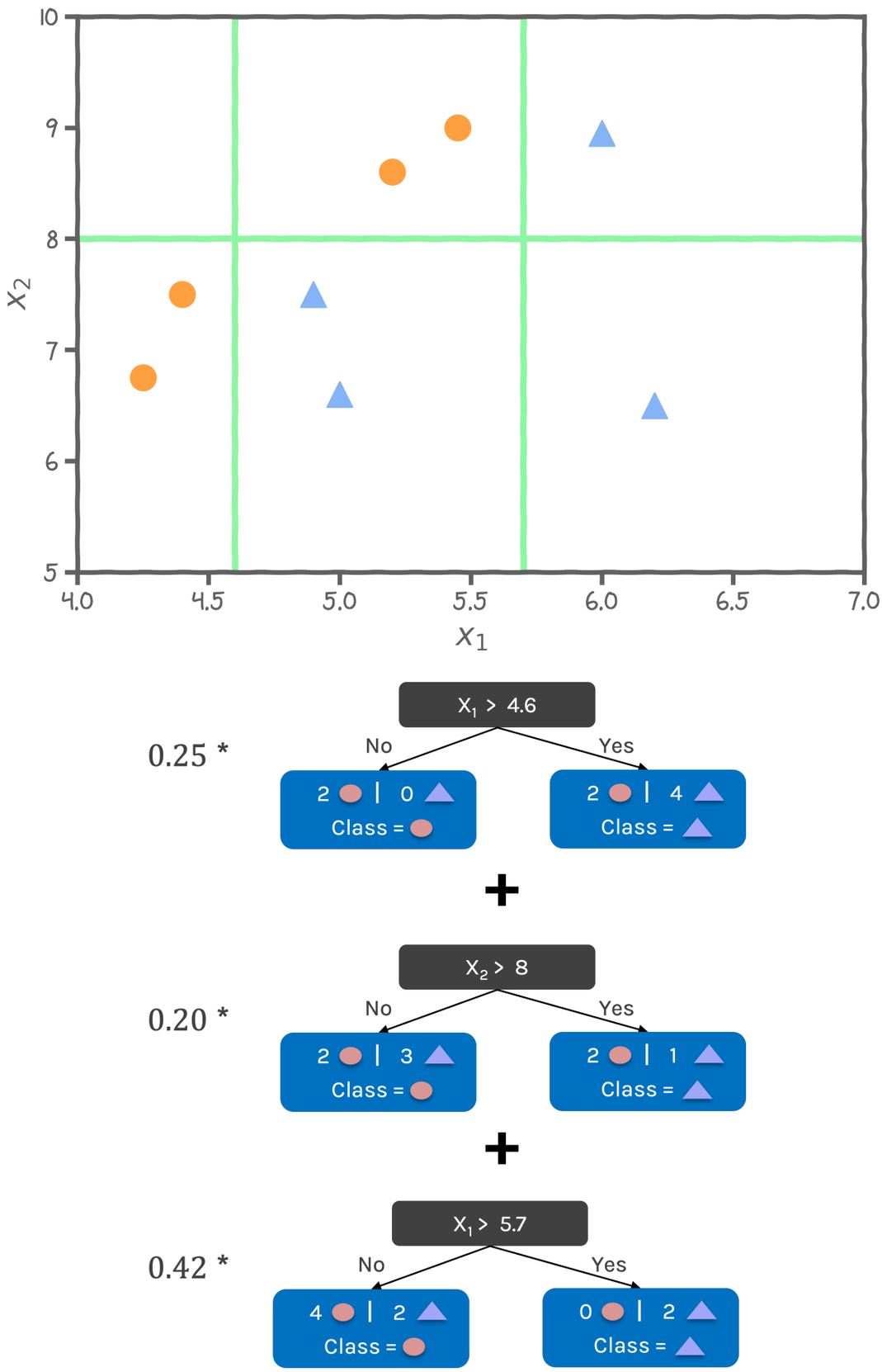
Step 9: Construct the **ensemble model** $T^{(1)}$ using:

$$T^{(i)} \leftarrow \begin{cases} \lambda^{(i)} S^{(i)}, & i = 0 \\ T^{(i-1)} + \lambda^{(i)} S^{(i)}, & i = 1, 2, \ldots \end{cases}$$





Repeating the same process again, we get:



In this case it's clear that we would not want to go further, but you can imagine that having more divisions might be helpful if we had, say, thousands or millions of data points instead of just eight.

6.2 AdaBoost Algorithm – definition

1: Given training data $(x_1, y_1) \dots (x_n, y_n)$, choose an initial distribution $\omega_n^{(0)} = \frac{1}{N}$
2: **for** $i = 0 \dots$ **do**
3: Train a weak learner $S^{(i)}$ using weights $\omega_n^{(i)}$ Calculate the total error of the weak learner using:

$$\epsilon^{(i)} = \sum_{n=1}^{N} \omega_n^{(i)} \mathcal{I}(y_n \neq S^{(i)}(x_n))$$

4: Calculate the importance of each model $\lambda^{(i)}$
5: Construct the ensemble model using:

$$T^{(i)} \leftarrow \begin{cases} \lambda^{(i)} S^{(i)}, & i = 0 \\ T^{(i-1)} + \lambda^{(i)} S^{(i)}, & i = 1, 2, \dots \end{cases}$$

6: Adjust the weights assigned to each data point to ensure the next stump focuses on the points misclassified by the previous stump

$$\omega_n^{(1)} \leftarrow \frac{\omega_n^{(0)} e^{-\lambda y_n T^{(0)}(x_n)}}{z} = \frac{\omega_n^{(1')}}{z}$$

7: **end for**

6.2.1 AdaBoost Q&A

How do I create a stump? In AdaBoost the stumps are created using simple decision tree with `max_depth = 1`

How to use the normalized weights to make stump? Create a new dataset of same size of the original dataset with repetition based on the newly updated sample weight.

How do I calculate the scaling factor $\lambda^{(i)}$ of each stump? In gradient boosting for regression, we minimize the MSE loss. In AdaBoost, the function we choose is called exponential loss:

$$\text{Exp Loss} = \frac{1}{N} \sum_{n=1}^{N} e^{(-y_n \hat{y_n})}$$

$$\text{where} y_n \in -1, 1$$

Exponential loss is differential with respect to $\hat{y_n}$ and it is an upper bound of error.

Choosing the Learning Rate Unlike in the case of gradient boosting for regression, we can analytically solve for the optimal learning rate for AdaBoost, by optimizing:

$$\text{argmin}_\lambda \frac{1}{N} \sum_{n=1}^{N} e^{(-y_n (T + \lambda^{(i)} S^{(i)}(x_n)))}$$

In doing so, we get:

$$\lambda^{(i)} = \frac{1}{2} ln(\frac{1 - \epsilon}{\epsilon})$$

where

$$\epsilon = \sum_{n=1}^{N} \omega_n^{(1)} \mathcal{I}(y_n \neq S^{(1)}(x_n))$$

$$\omega_n^{(i+1)} \leftarrow \frac{\omega_n^{(i)} e^{\lambda^{(i)}}}{Z}$$

6.2.2 Mathematical Formulation - AdaBoost

Gradient Descent with Exponential Loss The exponential loss of AdaBoost is given by:

$$\text{Exp Loss} = \frac{1}{N} \sum_{n=1}^{N} e^{(-y_n \hat{y_n})}$$

Similar to what we did for gradient boosting, compute the gradient for the loss w.r.t the predictions:

$$\Delta_{\hat{y_n}} \text{Exp Loss} = \left[-y_1 e^{(-y_1 \hat{y_1}), \dots} \right]$$

Set $\omega_n = exp(-y_n \hat{y_n})$.
Notice that when $y_n = \hat{y_n}$, the weight ω_n is small; when $y_n \neq \hat{y_n}$, the weight is larger.

This way, we see that the gradient is just a re-weighting applied to the target values.
The update step in the gradient descent is:

$$\hat{y_n} \leftarrow \hat{y_n} + \lambda \omega_n y_n, n = 1, \dots, N$$

Just like in gradient boosting, we approximate the gradient, $\lambda\omega_n y_n$, with a simple model, $S^{(i)}$, that depends on x_n.

This means training $S^{(i)}$ on a re-weighted set of target values:

$$(x_1, \omega_1 y_1), \ldots, (x_n, \omega_N y_N)$$

That is, gradient descent with exponential loss means iteratively training simple models that **focuses on the points misclassified by the previous model**.

6.3 Comparison: Gradient Boosting And Adaboost

Here's a comparison between procedures in Gradient Boosting and AdaBoost. You can find the same information in a table farther down the page if you'd rather see it that way.

- Gradient Boosting we minimize the MSE given by $\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y_n})^2$, while in AdaBoost we minimize the exponential loss given by $\text{Exp Loss} = \frac{1}{N}\sum_{n=1}^{N} e^{(-y_n \hat{y_n})}$
- Gradient Boosting computes the gradient for the loss w.r.t predictions $\Delta_{\hat{y_n}}\text{MSE} = -2[r_1, \ldots, r_n]$ whereas AdaBoost computes the gradient $\Delta_{\hat{y_n}}\text{Exp Loss} = [-y_1\omega_1, \ldots, -y_N\omega_N]$
- The update step in Gadient Boosting is given by $\hat{y_n} \leftarrow \hat{y_n} + \lambda\hat{r_n}$. The update step in the case of AdaBoost is given by $\hat{y_n} \leftarrow \hat{y_n} + \lambda\omega_n y_n$

6.4 Final Thoughts On Boosting

There are several implementations of boosting available. Here are links to a few of them, with descriptions taken from their websites.

XGBoost XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Kubernetes, Hadoop, SGE, Dask, Spark, PySpark) and can solve problems beyond billions of examples.

LGBM: Light Gradient Boosted Machines LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and efficient. ... LightGBM is being widely-used in many winning solutions of machine learning competitions.

CatBoost CatBoost is an algorithm for gradient boosting on decision trees. It is developed by Yandex researchers and engineers, and is used for search, recommendation systems, personal assistant, self-driving cars, weather prediction and many other tasks at Yandex and in other companies, including CERN, Cloudflare, Careem taxi. It is in open-source and can be used by anyone. Annotate