A Beginner's Guide to Bayes' Theorem, Naive Bayes Classifiers and Bayesian Networks

Bayes’ Theorem is formula that converts human belief, based on evidence, into predictions. It was conceived by the Reverend Thomas Bayes, an 18th-century British statistician who sought to explain how humans make predictions based on their changing beliefs. To understand his theorem, we will start by learning its notation.

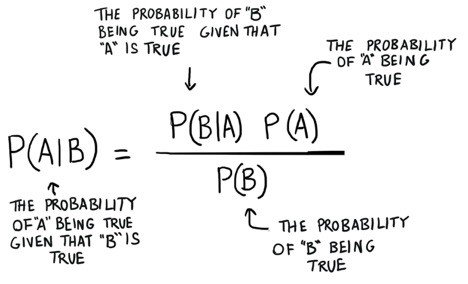
Bayesian Notation

Here’s how to read Bayesian notation:

* P(A) means “the probability that A is true.”
* P(A|B) means “the probability that A is true *given that B is true*.”

In this case, it’s easiest to think of B as the symptom and A as the disease; i.e. B is a skin rash that includes tiny white spots, and A is the probability of the measles. So we use phenomena or evidence that is easily visible to calculate the probability of phenomena that are hidden. What you can see enables you to predict what you can’t see.

It turns out that the the probabilities of A and B are related to each other in the following manner:



That is Bayes Theorem: that you can use the probability of one thing to predict the probability of another thing. But Bayes Theorem is not a static thing. It’s a machine that you crank to make better and better predictions as new evidence surfaces.

|  |  |
| --- | --- |
| An interesting exercise is to twiddle the variables by assigning different speculative values to P(B) or P(A) and consider their logical impact on P(A | B). |

For example, if you increase the denominator P(B) on the right, then P(A|B) goes down. Concrete example: A runny nose is a [symptom of the measles](https://www.mayoclinic.org/diseases-conditions/measles/symptoms-causes/syc-20374857), but runny noses are far more common than skin rashes with tiny white spots. That is, if you choose P(B) where B is a runny nose, then the frequency of runny noses in the general population decreases the chance that runny nose is a sign of measles. The probability of a measles diagnosis goes down with regard to symptoms that become increasingly common; those symptoms are not strong indicators.

Likewise, as measles become more common and P(A) goes up in the numerator on the right, P(A|B) goes up necessarily, because the measles are just generally more likely regardless of the symptom that you consider.

[Learn to build AI apps now »](https://skymind.ai/learn)

Naive Bayes Classifiers: A Playful Example

Maybe you’ve played [a party game called Werewolf](https://www.wikihow.com/Play-Werewolf-(Party-Game)).[1](https://pathmind.com/wiki/bayes-theorem-naive-bayes#one)

It’s what they call a “hidden role” game, because roles are assigned to you and your fellow players, but nobody knows what anyone else is.[2](https://pathmind.com/wiki/bayes-theorem-naive-bayes#two) You take a group of, say, 10 players and divide them into two roles – werewolves and villagers. Everyone draws slips of paper from a hat, closes their eyes, and then the werewolves are instructed to wake up to know who’s on the werewolf team.

Since only the werewolves know that they are werewolves, everyone in the game claims to be a villager. As the game moves through artificial cycles of day and night, each night the werewolves wake up and kill one villager, and each day the villagers try to figure out who the werewolves are, and vote to lynch one player based on their suspicians. Good, clean fun!

So a great way to learn Bayes Theorem, and in particular the implementation called Naive Bayes, is to think about how you track down a werewolf. So A is the variable you are trying to predict: It is a categorical variable that can be either A1 == werewolf or A2 == villager. Your job in this game is to act as a binomial classifier. B would be all the symptoms of werewolfiness, which in this case mean the symptoms of deception, since the werewolves only win by hiding their identity from the villagers as they slowly kill those poor people off.

The moderator of the game tells you how many werewolves there are, that’s P(A1), but doesn’t tell you who.

Over the course of your life, you’ve learned that people do certain things when they’re nervous, lying to you and afraid of getting caught.

So you start looking for symptoms you might use to diagnose a werewolf: Are they shifting in their seat? p(B1=Shifting|A1) Are they avoiding eye contact? p(B2=AvoidEyeContact|A1) Are they touching their face or hair with their fingers? p(B3=ActiveFingers|A1) That is, instead of using the symptoms to diagnose the disease P(Measles|Rash), you’re flipping it around, using an assumed diagnosis to predict the probability of symptoms. This is the generative side of Bayes.

What is the probability that you exhibit a combination of these symptoms if you are a werewolf? It’s P(B1, B2, B3 | A1).

And so your prediction that someone is a werewolf or a villager would be: Prediction = argmax\_A[P(B1, B2, B3| A) \* P(A)] for all A.

At the end of each round, when a person is lynched, you find out whether they were, in fact, a werewolf or a villager. (For our purposes, that means you can update what you know about P(A1), or the probability that one of the remaining players is a werewolf.) And that, in a microcosm, is the magic of Bayes, allowing you to incorporate new knowledge about the world into this machine that updates your beliefs.

The only problem is that, as the number of features you analyze increases (i.e. B grows larger), P(B1, B2, B3, B4 | A) gets very hard to compute without a large sample. You would have to see each, exact combination of features many times to calculate their probilities. That’s where the “naive” part of Naive Bayes comes in: Naive Bayes assumes all inputs are conditionally independent.

Instead of trying to find the probability of B1, B2, B3 etc. occurring simultaneously for a certain label, you calculate the probability of each feature occurring individually given that label: P(B1, B2, B3, B4| A1) = P(B1|A1) \* P(B2|A1) \* P(B3|A1) \* P(B4|A1)

We assume their combinations don’t matter. Each symptom has a relationship with the probability of werewolf, but combinations of those symptoms do not increase that probability. Why do we call that assumption “naive”?

The features avoiding eye contact and usually friendly on their own might not tell you a lot about whether the person in question is a werewolf. P(B1|A1) \* P(B2|A1)

But if you know simultaneously that a person who is usually friendly has suddenly fallen quiet and is avoiding all eye contact, common sense dictates that your expectation of that person being a werewolf should increase. P(B1, B2|A1).

So Naive Bayes discards information in favor of computational efficiency, a tradeoff we’re forced to make with other algorithms like [convolutional networks](https://pathmind.com/wiki/convolutional-network) as well.

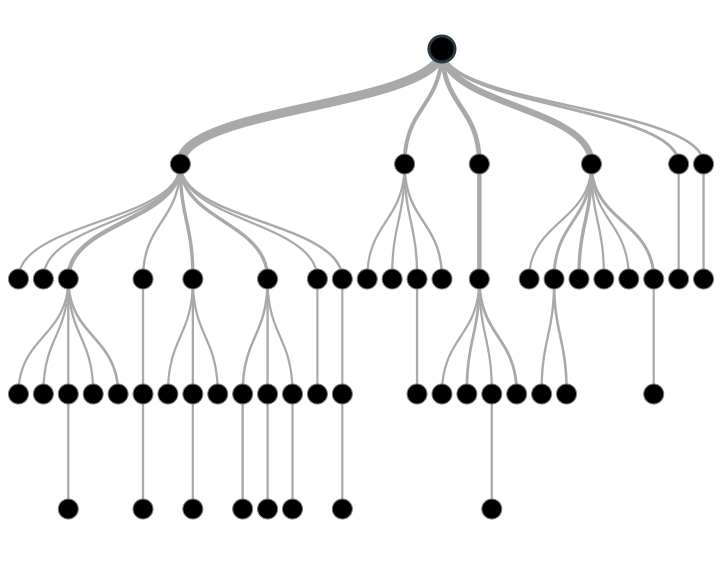
**Bayesian Networks**

Bayesian networks are [graphical models](https://pathmind.com/wiki/graph-analysis) that use Bayesian inference to compute probability. They model conditional dependence and causation. In a Baysian Network, each edge represents a conditional dependency, while each node is a unique variable (an event or condition). Bayesian networks were invented by Judea Pearl in 1985. They were a particularly popular approach to machine learning problems in the 1990s, and remain a powerful tool for thinking about causality.

Bayesian Network can be used to model any number of causal relationships. Instead of tackling werewolves and the symptoms of deceipt, you could try to model all the possible causes of your front lawn being wet, which would include rain and sprinklers and leaky fire hydrants and kids with squirt guns (this scenario is, in fact, the hello world of Bayesian Networks).

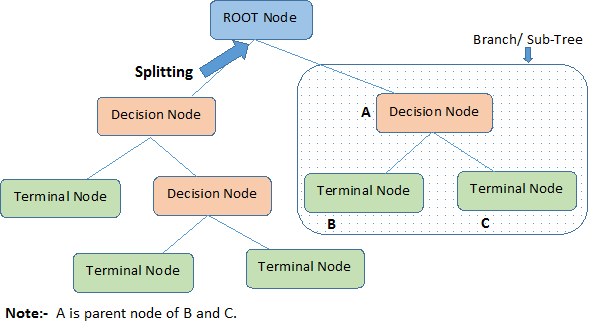
Decision Tree

A decision tree is a series of nodes, a directional graph that starts at the base with a single node and extends to the many leaf nodes that represent the categories that the tree can classify. Another way to think of a decision tree is as a flow chart, where the flow starts at the root node and ends with a decision made at the leaves. It is a decision-support tool. It uses a tree-like graph to show the predictions that result from a series of feature-based splits.



Here are some useful terms for describing a decision tree:

* Root Node: A root node is at the beginning of a tree. It represents entire population being analyzed. From the root node, the population is divided according to various features, and those sub-groups are split in turn at each decision node under the root node.
* Splitting: It is a process of dividing a node into two or more sub-nodes.
* Decision Node: When a sub-node splits into further sub-nodes, it’s a decision node.
* Leaf Node or Terminal Node: Nodes that do not split are called leaf or terminal nodes.
* Pruning: Removing the sub-nodes of a parent node is called pruning. A tree is grown through splitting and shrunk through pruning.
* Branch or Sub-Tree: A sub-section of decision tree is called branch or a sub-tree, just as a portion of a graph is called a sub-graph.
* Parent Node and Child Node: These are relative terms. Any node that falls under another node is a child node or sub-node, and any node which precedes those child nodes is called a parent node.



Decision trees are a popular algorithm for several reasons:

* Explanatory Power: The output of decision trees is interpretable. It can be understood by people without analytical or mathematical backgrounds. It does not require any statistical knowledge to interpret them.
* Exploratory data analysis: Decision trees can enable analysts to identify significant variables and important relations between two or more variables, helping to surface the signal contained by many input variables.
* Minimal data cleaning: Because decision trees are resilient to outliers and missing values, they require less data cleaning than some other algorithms.
* Any data type: Decision trees can make classifications based on both numerical and categorical variables.
* Non-parametric: A decision tree is a non-parametric algorithm, as opposed to neural networks, which process input data transformed into a tensor, via tensor multiplication using large number of coefficients, known as parameters.

[Learn to build AI apps now »](https://skymind.ai/learn)

**Disadvantages**

* Overfitting: Over fitting is a common flaw of decision trees. Setting constraints on model parameters (depth limitation) and making the model simpler through pruning are two ways to regularize a decision tree and improve its ability to generalize onto the test set.
* Predicting continuous variables: While decision trees can ingest continuous numerical input, they are not a practical way to predict such values, since decision-tree predictions must be separated into discrete categories, which results in a loss of information when applying the model to continuous values.
* Heavy feature engineering: The flip side of a decision tree’s explanatory power is that it requires heavy feature engineering. When dealing with unstructured data or data with latent factors, this makes decision trees sub-optimal. Neural networks are clearly superior in this regard.

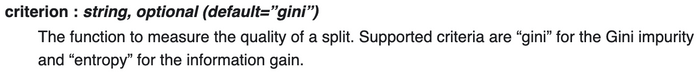
One weird thing about decision trees (or random forests) is how conceptually simple they are, while in terms of implementation they’re non-trivial. How do you find the optimal split/feature based on entropy? Naively implemented, they require something on the order of O(kNlogN) for each split. Multiply that by the number of leaves (2^depth), and multiply that by the number of trees in your forest.

A Simple Explanation of Gini Impurity

What Gini Impurity is (with examples) and how it's used to train Decision Trees.

**MARCH 29, 2019**

If you look at the documentation for the [DecisionTreeClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html" \t "_blank) class in [scikit-learn](https://scikit-learn.org/" \t "_blank), you’ll see something like this for the criterion parameter:



The [RandomForestClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html" \t "_blank) documentation says the same thing. Both mention that the default criterion is “gini” for the **Gini Impurity**. What is that?!

*TLDR: Read the*[*Recap*](https://victorzhou.com/blog/gini-impurity/#recap)*.*

**Decision Trees 🌲**

Training a decision tree consists of iteratively splitting the current data into two branches. Say we had the following datapoints:

The Dataset

Right now, we have 1 branch with 5 blues and 5 greens.

Let’s make a split at x = 2*x*=2:

A Perfect Split

This is a **perfect** split! It breaks our dataset perfectly into two branches:

* Left branch, with 5 blues.
* Right branch, with 5 greens.

What if we’d made a split at x = 1.5*x*=1.5 instead?

An Imperfect Split

This imperfect split breaks our dataset into these branches:

* Left branch, with 4 blues.
* Right branch, with 1 blue and 5 greens.

It’s obvious that this split is worse, but **how can we quantify that?**

Being able to measure the quality of a split becomes even more important if we add a third class, reds . Imagine the following split:

* Branch 1, with 3 blues, 1 green, and 1 red.
* Branch 2, with 3 greens and 1 red.

Compare that against this split:

* Branch 1, with 3 blues, 1 green, and 2 reds.
* Branch 2, with 3 greens.

Which split is better? It’s no longer immediately obvious. We need a way to **quantitatively evaluate** how good a split is.

**Gini Impurity**

This is where the Gini Impurity metric comes in.

Suppose we

1. Randomly pick a datapoint in our dataset, then
2. **Randomly classify it according to the class distribution in the dataset**. For our dataset, we’d classify it as blue \frac{5}{10}105​ of the time and as green \frac{5}{10}105​ of the time, since we have 5 datapoints of each color.

**What’s the probability we classify the datapoint incorrectly?** The answer to that question is the Gini Impurity.

**Example 1: The Whole Dataset**

Let’s calculate the Gini Impurity of our entire dataset. If we randomly pick a datapoint, it’s either blue (50%) or green (50%).

Now, we randomly classify our datapoint according to the class distribution. Since we have 5 of each color, we classify it as blue 50% of the time and as green 50% of the time.

What’s the probability we classify our datapoint **incorrectly**?

| **Event** | **Probability** |
| --- | --- |
| Pick Blue, Classify Blue **✓** | 25% |
| Pick Blue, Classify Green ❌ | 25% |
| Pick Green, Classify Blue ❌ | 25% |
| Pick Green, Classify Green **✓** | 25% |

We only classify it incorrectly in 2 of the events above. Thus, our total probability is 25% + 25% = 50%, so the Gini Impurity is \boxed{0.5}0.5​.

**The Formula**

If we have C*C* total classes and p(i)*p*(*i*) is the probability of picking a datapoint with class i*i*, then the Gini Impurity is calculated as

G = \sum\_{i=1}^C p(i) \* (1 - p(i))*G*=*i*=1∑*C*​*p*(*i*)∗(1−*p*(*i*))

For the example above, we have C = 2*C*=2 and p(1) = p(2) = 0.5*p*(1)=*p*(2)=0.5, so

\begin{aligned} G &= p(1) \* (1 - p(1)) + p(2) \* (1 - p(2)) \\ &= 0.5 \* (1 - 0.5) + 0.5 \* (1 - 0.5) \\ &= \boxed{0.5} \\ \end{aligned}*G*​=*p*(1)∗(1−*p*(1))+*p*(2)∗(1−*p*(2))=0.5∗(1−0.5)+0.5∗(1−0.5)=0.5​​

which matches what we calculated!

**Example 2: A Perfect Split**

Let’s go back to the perfect split we had. What are the Gini Impurities of the two branches after the split?

A Perfect Split

Left Branch has only blues, so its Gini Impurity is

G\_{left} = 1 \* (1 - 1) + 0 \* (1 - 0) = \boxed{0}*Gleft*​=1∗(1−1)+0∗(1−0)=0​

Right Branch has only greens, so its Gini Impurity is

G\_{right} = 0 \* (1 - 0) + 1 \* (1 - 1) = \boxed{0}*Gright*​=0∗(1−0)+1∗(1−1)=0​

Both branches have 00 impurity! The perfect split turned a dataset with 0.50.5 impurity into 2 branches with 00 impurity.

**A Gini Impurity of 0 is the lowest and best possible impurity**. It can only be achieved when everything is the same class (e.g. only blues or only greens).

**Example 3: An Imperfect Split**

Finally, let’s return to our imperfect split.

An Imperfect Split

Left Branch has only blues, so we know that G\_{left} = \boxed{0}*Gleft*​=0​.

Right Branch has 1 blue and 5 greens, so

\begin{aligned} G\_{right} &= \frac{1}{6} \* (1 - \frac{1}{6}) + \frac{5}{6} \* (1 - \frac{5}{6}) \\ &= \frac{5}{18} \\ &= \boxed{0.278} \\ \end{aligned}*Gright*​​=61​∗(1−61​)+65​∗(1−65​)=185​=0.278​​

**Picking The Best Split**

It’s finally time to answer the question we posed earlier: **how can we quantitatively evaluate the quality of a split?**

Here’s the imperfect split yet again:

An Imperfect Split

We’ve already calculated the Gini Impurities for:

* Before the split (the entire dataset): 0.50.5
* Left Branch: 00
* Right Branch: 0.2780.278

We’ll determine the quality of the split by **weighting the impurity of each branch by how many elements it has**. Since Left Branch has 4 elements and Right Branch has 6, we get:

(0.4 \* 0) + (0.6 \* 0.278) = 0.167(0.4∗0)+(0.6∗0.278)=0.167

Thus, the amount of impurity we’ve “removed” with this split is

0.5 - 0.167 = \boxed{0.333}0.5−0.167=0.333​

I’ll call this value the Gini Gain. This is what’s used to pick the best split in a decision tree! **Higher Gini Gain = Better Split**. For example, it’s easy to verify that the Gini Gain of the perfect split on our dataset is 0.5 > 0.3330.5>0.333.

Random Forests for Complete Beginners

The definitive guide to Random Forests and Decision Trees.

**APRIL 10, 2019**

In my opinion, most Machine Learning tutorials aren’t beginner-friendly enough.

Last month, I wrote an [introduction to Neural Networks **for complete beginners**](https://victorzhou.com/blog/intro-to-neural-networks/). This post will adopt the same strategy, meaning it again **assumes ZERO prior knowledge of machine learning**. We’ll learn what Random Forests are and how they work from the ground up.

Ready? Let’s dive in.

**1. Decision Trees 🌲**

A Random Forest 🌲🌲🌲 is actually just a bunch of Decision Trees 🌲 bundled together (ohhhhh that’s why it’s called a *forest*). We need to talk about trees before we can get into forests.

Look at the following dataset:

The Dataset

If I told you that there was a new point with an x*x* coordinate of 11, what color do you think it’d be?

Blue, right?

You just evaluated a decision tree in your head:

That’s a simple decision tree with one **decision node** that **tests** x < 2*x*<2. If the test passes (x < 2*x*<2), we take the left **branch** and pick Blue. If the test fails (x \geq 2*x*≥2), we take the right **branch** and pick Green.

The Dataset, split at x=2

Decision Trees are often used to answer that kind of question: given a **labelled** dataset, how should we **classify** new samples?

***Labelled****: Our dataset is labelled because each point has a****class****(color): blue or green.*

***Classify****: To classify a new datapoint is to assign a class (color) to it.*

Here’s a dataset that has 3 classes now instead of 2:

The Dataset v2

Our old decision tree doesn’t work so well anymore. Given a new point (x, y)(*x*,*y*),

* If x \geq 2*x*≥2, we can still confidently classify it as green.
* If x < 2*x*<2, we can’t immediately classify it as blue - it could be red, too.

We need to add another **decision node** to our decision tree:

Pretty simple, right? That’s the basic idea behind decision trees.

**2. Training a Decision Tree**

Let’s start training a decision tree! We’ll use the 3 class dataset again:

The Dataset v2

**2.1 Training a Decision Tree: The Root Node**

Our first task is to determine the root decision node in our tree. Which feature (x*x* or y*y*) will it test on, and what will the test threshold be? For example, the root node in our tree from earlier used the x*x* feature with a test threshold of 22:

Intuitively, we want a decision node that makes a “good” split, where “good” can be loosely defined as **separating different classes as much as possible**. The root node above makes a “good” split: *all* the greens are on the right, and *no* greens are on the left.

Thus, our goal is now to pick a root node that gives us the “best” split possible. **But how do we quantify how good a split is?** It’s complicated. I wrote [an entire blog post about one way to do this using a metric called Gini Impurity](https://victorzhou.com/blog/gini-impurity/). **← I recommend reading it right now** before you continue - we’ll be using those concepts later in this post.

Welcome back!

*Hopefully, you just read*[*my Gini Impurity post*](https://victorzhou.com/blog/gini-impurity/)*. If you didn’t, here’s a very short TL;DR: We can use Gini Impurity to calculate a value called****Gini Gain****for any split.****A better split has higher Gini Gain****.*

Back to the problem of determining our root decision node. Now that we have a way to evaluate splits, all we have to do to is find the best split possible! For the sake of simplicity, we’re just going to **try every possible split** and use the best one (the one with the highest Gini Gain). **This is not the fastest way to find the best split**, but it is the easiest to understand.

Trying every split means trying

* Every feature (x*x* or y*y*).
* All “unique” thresholds. **We only need to try thresholds that produce different splits.**

For example, here are the thresholds we might select if we wanted to use the x*x* coordinate:

x Thresholds

Let’s do an example Gini Gain calculation for the x = 0.4*x*=0.4 split.

| **Split** | **Left Branch** | **Right Branch** |
| --- | --- | --- |
| x = 0.4*x*=0.4 |  |  |

First, we calculate the Gini Impurity of the whole dataset:

\begin{aligned} G\_{initial} &= \sum\_{i=1}^3 p(i) \* (1 - p(i)) \\ &= 3 \* (\frac{1}{3} \* \frac{2}{3}) \\ &= \boxed{\frac{2}{3}} \\ \end{aligned}*Ginitial*​​=*i*=1∑3​*p*(*i*)∗(1−*p*(*i*))=3∗(31​∗32​)=32​​​

Then, we calculate the Gini Impurities of the two branches:

G\_{left} = 0 \* 1 + 1 \* 0 + 0 \* 1 = \boxed{0}*Gleft*​=0∗1+1∗0+0∗1=0​\begin{aligned} G\_{right} &= \frac{3}{8} \* \frac{5}{8} + \frac{2}{8} \* \frac{6}{8} + \frac{3}{8} \* \frac{5}{8} \\ &= \boxed{\frac{21}{32}} \\ \end{aligned}*Gright*​​=83​∗85​+82​∗86​+83​∗85​=3221​​​

Finally, we calculate Gini Gain by subtracting the weighted branch impurities from the original impurity:

\begin{aligned} \text{Gain} &= G\_{initial} - \frac{1}{9} G\_{left} - \frac{8}{9} G\_{right} \\ &= \frac{2}{3} - \frac{1}{9} \* 0 - \frac{8}{9} \* \frac{21}{32} \\ &= \boxed{0.083} \\ \end{aligned}Gain​=*Ginitial*​−91​*Gleft*​−98​*Gright*​=32​−91​∗0−98​∗3221​=0.083​​

*Confused about what just happened? I told you you should’ve read*[*my Gini Impurity post*](https://victorzhou.com/blog/gini-impurity/)*. It’ll explain all of this Gini stuff.*

We can calculate Gini Gain for every possible split in the same way:

| **Split** | **Left Branch** | **Right Branch** | **Gini Gain** |
| --- | --- | --- | --- |
| x = 0.4*x*=0.4 |  |  | 0.0830.083 |
| x = 0.8*x*=0.8 |  |  | 0.0480.048 |
| x = 1.1*x*=1.1 |  |  | 0.1330.133 |
| x = 1.3*x*=1.3 |  |  | 0.2330.233 |
| x = 2*x*=2 |  |  | 0.3330.333 |
| x = 2.4*x*=2.4 |  |  | 0.1910.191 |
| x = 2.8*x*=2.8 |  |  | 0.0830.083 |
| y = 0.8*y*=0.8 |  |  | 0.0830.083 |
| y = 1.2*y*=1.2 |  |  | 0.1110.111 |
| y = 1.8*y*=1.8 |  |  | 0.2330.233 |
| y = 2.1*y*=2.1 |  |  | 0.2330.233 |
| y = 2.4*y*=2.4 |  |  | 0.1110.111 |
| y = 2.7*y*=2.7 |  |  | 0.0480.048 |
| y = 2.9*y*=2.9 |  |  | 0.0830.083 |

All Thresholds

After trying all thresholds for both x*x* and y*y*, we’ve found that the x = 2*x*=2 split has the highest Gini Gain, so we’ll make our root decision node use the x*x* feature with a threshold of 22. Here’s what we’ve got so far:

Making progress!

**2.2: Training a Decision Tree: The Second Node**

Time to make our second decision node. Let’s (arbitrarily) go to the left branch. **We’re now only using the datapoints that would take the left branch** (i.e. the datapoints satisfying x < 2*x*<2), specifically the 3 blues and 3 reds.

To build our second decision node, **we just do the same thing!** We try every possible split for the 6 datapoints we have and realize that y = 2*y*=2 is the best split. We make that into a decision node and now have this:

Our decision tree is almost done…

**2.3 Training a Decision Tree: When to Stop?**

Let’s keep it going and try to make a third decision node. We’ll use the right branch from the root node this time. The only datapoints in that branch are the 3 greens.

Again, we try all the possible splits, but they all

* Are equally good.
* Have a Gini Gain of 0 (the Gini Impurity was already 0 and can’t go any lower).

It doesn’t makes sense to add a decision node here because doing so wouldn’t improve our decision tree. Thus, we’ll make this node a **leaf node** and slap the Green label on it. This means that **we’ll classify any datapoint that reaches this node as Green**.

If we continue to the 2 remaining nodes, the same thing will happen: we’ll make the bottom left node our Blue leaf node, and we’ll make the bottom right node our Red leaf node. That brings us to the final result:

**Once all possible branches in our decision tree end in leaf nodes, we’re done.** We’ve trained a decision tree!

**3. Random Forests 🌲🌳🌲🌳🌲**

We’re finally ready to talk about Random Forests. Remember what I said earlier?

*A Random Forest is actually just a bunch of Decision Trees bundled together.*

That’s true, but is a bit of a simplification.

**3.1 Bagging**

Consider the following algorithm to train a bundle of decision trees given a dataset of n*n* points:

1. Sample, **with replacement**, n*n* training examples from the dataset.
2. Train a decision tree on the n*n* samples.
3. Repeat t*t* times, for some t*t*.

To make a prediction using this model with t*t* trees, we aggregate the predictions from the individual decision trees and either

* Take the **majority vote** if our trees produce class labels (like colors).
* Take the **average** if our trees produce numerical values (e.g. when predicting temperature, price, etc).

This technique is called **bagging**, or [**b**ootstrap **agg**regating](https://en.wikipedia.org/wiki/Bootstrap_aggregating). The sampling with replacement we did is known as a [bootstrap](https://en.wikipedia.org/wiki/Bootstrapping_(statistics)) sample.

Bagged Decision Trees predicting color

Bagged decision trees are very close to Random Forests - they’re just missing one thing…

**3.2 Bagging → Random Forest**

Bagged decision trees have only one parameter: t*t*, the number of trees.

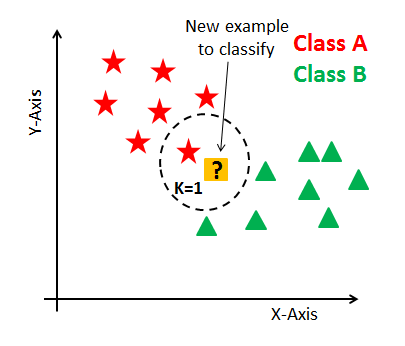
Random Forests have a second parameter that controls **how many features to try when finding the best split**. Our simple dataset for this tutorial only had 22 features (x*x* and y*y*), but most datasets will have far more (hundreds or thousands).

Suppose we had a dataset with p*p* features. Instead of trying all features every time we make a new decision node, we **only try a subset of the features**, usually of size \sqrt{p}*p*​ or \frac{p}{3}3*p*​. We do this primarily to inject randomness that makes individual trees more unique and **reduces correlation between trees**, which improves the forest’s performance overall. This technique is sometimes referred to as **feature bagging**.

KNN is a non-parametric and lazy learning algorithm. Non-parametric means there is no assumption for underlying data distribution. In other words, the model structure determined from the dataset. This will be very helpful in practice where most of the real world datasets do not follow mathematical theoretical assumptions. Lazy algorithm means it does not need any training data points for model generation. All training data used in the testing phase. This makes training faster and testing phase slower and costlier. Costly testing phase means time and memory. In the worst case, KNN needs more time to scan all data points and scanning all data points will require more memory for storing training data.

**How does the KNN algorithm work?**

In KNN, K is the number of nearest neighbors. The number of neighbors is the core deciding factor. K is generally an odd number if the number of classes is 2. When K=1, then the algorithm is known as the nearest neighbor algorithm. This is the simplest case. Suppose P1 is the point, for which label needs to predict. First, you find the one closest point to P1 and then the label of the nearest point assigned to P1.



Suppose P1 is the point, for which label needs to predict. First, you find the k closest point to P1 and then classify points by majority vote of its k neighbors. Each object votes for their class and the class with the most votes is taken as the prediction. For finding closest similar points, you find the distance between points using distance measures such as Euclidean distance, Hamming distance, Manhattan distance and Minkowski distance. KNN has the following basic steps:

1. Calculate distance
2. Find closest neighbors
3. Vote for labels



-------------------------------------------------------------------------------------------------------------------------------------------