Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2023

Contents

CHAPTER 14

Non-parametric Methods

14.0.1 Parametric Methods

We've spent a large part of the course on models that rely on **parameters**:

- Linear regression/classification models, with parameters $\Theta = (\theta, \theta_0)$:

$$h(x; \Theta) = \theta^T x + \theta_0 \tag{14.1}$$

- Neural networks, with weights W^ℓ and W_0^ℓ:

$$A^\ell = f(Z^\ell) \qquad Z^\ell = (W^\ell)^T A^{\ell-1} + W_0^\ell \tag{14.2}$$

These can be thought of as "parametric" methods:

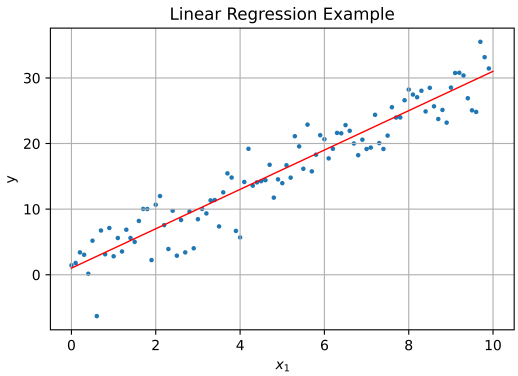
Definition 1

Parametric methods are those with a **fixed number of parameters**.

- Typically, we optimize these models by **changing** those parameters.

We can also phrase these as "models with a particular, known mathematical form".

Example: The equation $y = mx + b$ has the **fixed** parameters m and b. It assumes our data will follow the "mathematical form" of a line.



This kind of model works best if the "true distribution" follows a similar shape.

Neural networks have the advantage of being very **expressive**: they can express **many different** kinds and distributions of data.

- The same NN architecture can be re-used to make tools for multiple different tasks.

These methods have been our primary focus, because they're very configurable, and have a wide range of applications.

But they aren't our only option.

And if our problem is too complex, we can systematically increase expressiveness: increase layer size and depth.

We should be careful adding layers blindly, though.

14.0.2 Non-parametric methods

We can consider models that are more *flexible*. Rather than assuming a particular structure, we can discover patterns, directly from our data.

Definition 2

Non-parametric methods *exclude* models with a **fixed number** of parameters.

Instead, it includes:

- Models with **no parameters**
- Models with a **variable number of parameters**, depending on the data.

The name can be misleading: non-parametric models *can* have parameters!

As suggested in our definition, these methods often base their structure on the data they receive.

Here are some **examples**, each with a unique, **data-based** model structure:

- **k-means clustering**: We already discussed this in the **Clustering** chapter!

Note that k isn't a parameter: it's a *hyperparameter*.

  - We want to cluster our training data as tightly (low-variance) as possible.

- **Nearest neighbor**: We predict the output $g^{(i)}$ of a data point $x^{(i)}$, based on the **nearest** points of training data.

  - In this case, we **don't have** a model at all: we just directly use our data.

Reminder: $y^{(i)}$ is the true value, $g^{(i)}$ is our prediction.

- **Tree models**: We **split** up our space into smaller pieces. Each region of inputs is assigned an output y.

  - We divide up space to get good accuracy on training data.

And here are some examples that **combine** many simpler models, in a non-parametric way:

- **Ensembles**: We train multiple models on the whole data set, and we **average** them.

  - By combining multiple models, they'll (hopefully) average out to being more accurate, reducing estimation error.

We'll specifically refer to "bagging" in this chapter.

- **Boosting**: in boosting, we use multiple models **consecutively**: we train one model, and then we use our second model to try to improve on the mistakes of the first.

  - If an earlier model struggled with a data point, that data point is **weighted more heavily**: future models will focus more on that mistake.

  - We will not discuss boosting further.

14.0.3 Why learn about non-parametric methods?

Of course, neural networks are incredibly popular for a reason: they're **effective** at what they do.

So, why do we need non-parametric methods? Well, they come with several major benefits:

Concept 3

Non-parametric methods can have genuine benefits over parametric, **neural network** models:

- Fast to **implement**, few hyperparameters to tune.
- Often **human-interpretable**, easier to understand than a neural network computation.
- In some circumstances, **just as well or better** than neural networks, despite their relative simplicity.

14.1 Nearest Neighbor

Suppose that you're trying to figure out how to approach a problem. Maybe a medical problem, or just a personal situation with a friend.

- One question you can ask yourself is, "what's the most **similar** situation I can think of?"
- If that's not enough, you could think of 2, or 3, similar examples, and try to **guess** from that, what the best course of action is.

This is the basic idea of the **nearest neighbor** approach: we have a new data point. We want to use the most similar examples from our past **training data** to make a judgment.

- We judge the most relevant/"similar" data based on lowest **distance**: we call this the "nearest neighbor".

What's interesting is that we're **not** developing a model: we're directly using our training data to make predictions.

In the simplest variation, we only choose the **single closest data point**.

Definition 4

Nearest neighbor method is a **non-parametric** method where we predict output $g^{(i)}$ based on the **nearest** data point (**nearest neighbor**) in our training set.

- Our assumption is that nearby data points are **similar** to the situation we're currently dealing with.
- So, they should have similar **outcomes**.

This method works **exactly the same** for regression and classification:

- Whatever output $g^{(i)}$ we find for the nearest neighbor, is our **prediction** for our data point.

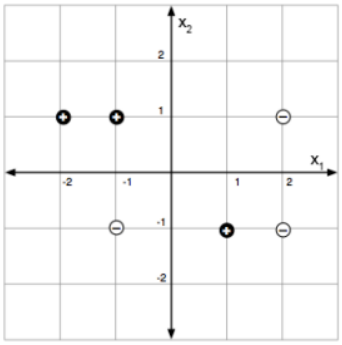
One nice benefit of nearest neighbor is that we require no training: we just check the training data, to make a judgment.

Concept 5

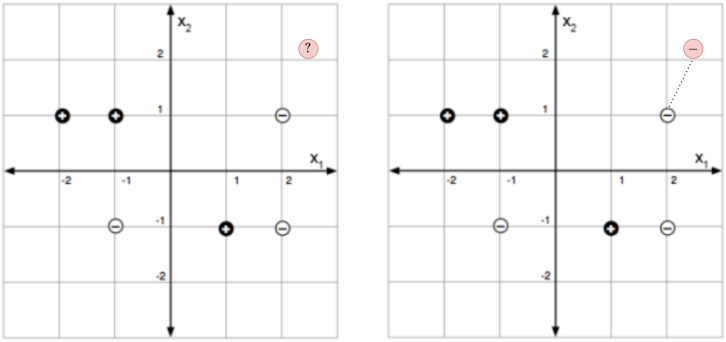
Nearest-neighbor models don't require training: you directly reference the training data to come to answers.

14.1.1 Nearest Neighbors: An example

Consider this classification problem.

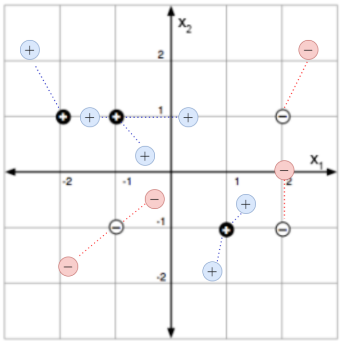


Rather than use a linear model, we'll assign any data point to the **same class** as whichever data point it's closest to.



This red data point is closest to a negative data point. So, we'll assign it negative.

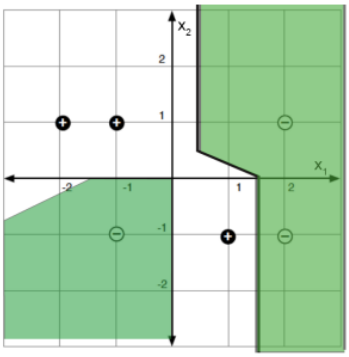
We could apply this to as many data points as we want:



We can see all of the data points that we assign our nearest neighbor to.

We're starting to fill the space: we see that some whole "**regions**" are positive or negative.

- We can even depict that: we'll highlight the regions which are labelled positive vs. negative.



We can see all of the data points that we assign our nearest neighbor to. Negative regions are green.

14.1.2 Simplified Voronoi Diagram (Optiona)

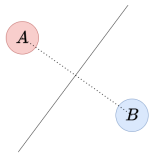
This kind of diagram lets you classify data very quickly. It would be useful to be able to draw:

Concept 6

To help with finding **nearest-neighbor classification boundaries**, it can be helpful to draw a line **halfway** between opposite-label data points, A and B.

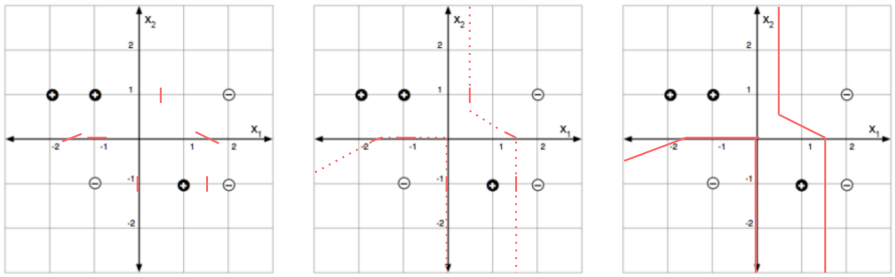
- This is where the distance to either data point is **equal**.
- We decide output based on closeness to A or B, so this a **decision boundary**.

In order to split the space between "closer to point A" and "closer to point B", we'll draw a boundary line **perpendicular to** the line from A to B.



We drew a solid line, where the entire line is equally far from points A and B. Notice that it is 1. halfway between them and 2. perpendicular to the line from A to B.

If we apply this to our previous problem:



We just have to draw our perpendiculars, and extend them.

The resulting diagram is a simplification of a **Voronoi diagram**.

Clarification 7

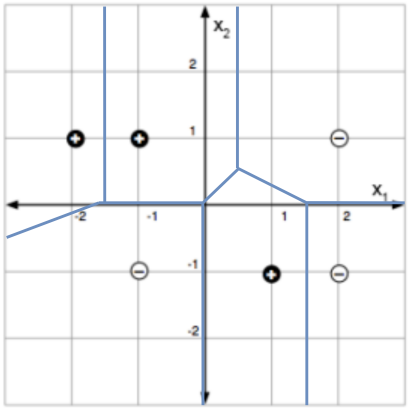
This system is for **nearest neighbors**.

You **CANNOT** use it for k **nearest neighbors** (discussed below).

If you try to use it this way, I will be sad.

Why is this a "simplified" voronoi diagram?

- In a real voronoi diagram, every single data point gets its own tile of "closer to this point than every other.
- In this one, we've simplified, so all of the + and − classifications join into one region.



Here's a "complete" voronoi diagram: each region represents all data which are closest to that particular data point.

14.1.3 k-Nearest Neighbors

If one neighbor is too simplified, we can also use **multiple** neighbors: our number of neighbors is k.

Definition 8

In the **k-nearest neighbors** (kNN), we use k ($k \in \mathbb{N}$) to give us the "size" of our neighborhood:

- When making a prediction, we only focus the k nearest data points.
- We ignore all data points beyond the k^{th} one.

This same approach can be applied to both **regression** and **classification**:

- In **regression**, you would **average** the output of those k nearest neighbors.
- In **classification**, you would pick the **majority**: the most common label of your k nearest neighbors.

Example: Suppose your $k = 4$ nearest neighbors had output values, $y = (3, 4, 5, 6)$. You would predict your output by simply averaging them:

$$\frac{3+4+5+6}{4} = 4.5 \tag{14.3}$$

One reason to use k-nearest neighbors is that "nearest neighbor" ($k = 1$) might **overfit**.

Concept 9

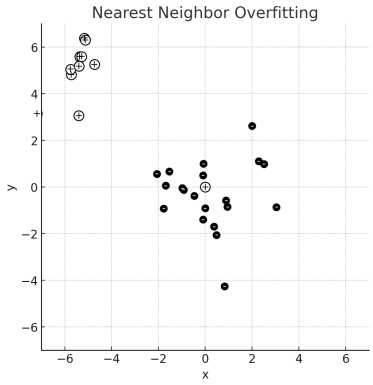
Having a low k-value could be seen as **overfitting**:

- If you're closest to an **outlier** data point, you'll **choose** that value, rather than the general trend you see with more data.

So, you're sensitive to **noise** in the data, if you happen to be too close to that noise.

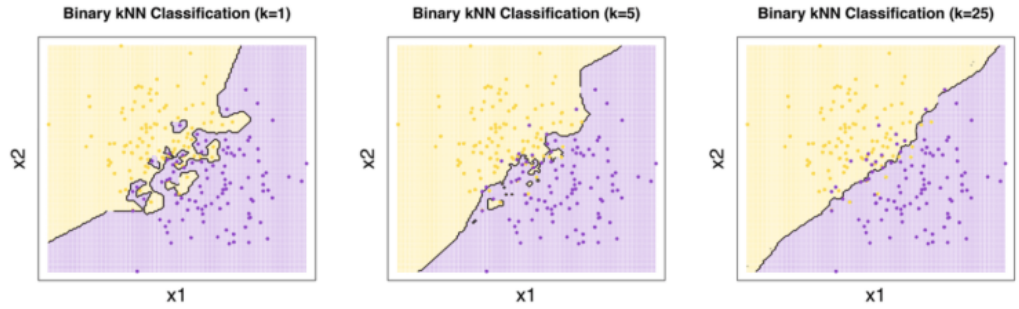
- With a larger k-value, you're **averaging** over more data points: this can reduce the effect of noise.

In other words: you're not create a general pattern based on the data structure: you're closely matching the training data.



Suppose we happened to have a data point right next to (0,0). Based on the whole region, the result is probably a negative (-). But with $k = 1$, we would assume it was positive.

What happens with different k-values? Here's a different dataset to test this out on:



We can see that we get a more "general" pattern as k increases. (Credit to blog.eduonix.com)

Concept 10

On the opposite end, having a **high k-value** makes it more difficult to see more detailed trends, or fit well to **complex** data.

- If the data really does have a complex shape, but our k is **too low**, we won't be able to match it.

In other words, we can "underfit" as well.

Example: In the extreme case: imagine that k equals the size of our training data. That means, you always include **all of your training data** to make a judgement.

- No matter what input you give, you'll **always** get the same output: the average of all data.
- That's not going to be very predictive.

14.1.4 Locally weighted regression

Another technique is *locally weighted regression*, where we use the local region to create a small regression model:

Definition 11

In **locally weighted regression**, we take the k nearest data points, and **fit** a regression model to only those data points.

We might **weigh** closer data points more heavily than those which are further away:

- Meaning, they're more important to the fitting process.

14.1.5 Tradeoffs

One major concern for k-nearest neighbors is that it's a pain to compare a new data point to the **entire training set**, to find the closest data points.

- So, it's important to use data structures (e.g., ball trees) that make it **easier** to quickly find possible "nearest neighbors".

On the other hand, it's relatively easy to **interpret** nearest neighbors:

- If you want to know why you got the answer that you did, you can directly view your "**nearest neighbors**": these exact data points gave you your answer.
- This makes it easier to check for strange **outliers**, or interesting patterns.

14.1.6 Distance metrics (Optional)

What do we mean when we say that two data points are "close" or "far away"?

- This requires us to have some kind of way to measure **distance**: a **distance metric**.
- Above, we were using the **euclidean** distance metric.

Definition 12

A **distance metric** d is one way of measuring the total distance between two data points.

It must follow three basic rules:

- The distance from x to **itself** is 0.

$$d(x, x) = 0$$

- Distance is always **positive**.

$$d(a, b) > 0$$

- The distance from a to b is the **same** as the distance from b to a.

$$d(a, b) = d(b, a)$$

- **Triangle Inequality**: A **direct path** from a to c is always the **shortest** – taking a detour from a to b cannot be faster.

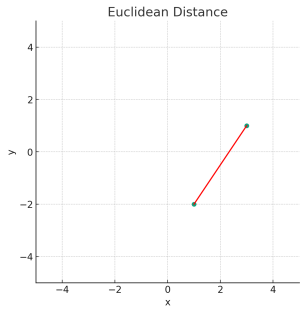
$$d(a, c) \leqslant d(a, b) + d(b, c)$$

Different metrics may be useful for different situations, or different data types.

Here's a few common distance metrics:

- **Euclidean** distance: the "shortest path" distance in space.

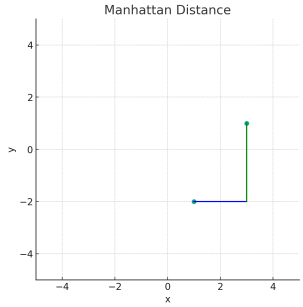
$$d(a, b) = \sqrt{\sum_i (a_i - b_i)^2} \tag{14.4}$$



Here, we have $\sqrt{(3-1)^2+(1-(-2))^2} = \sqrt{13}$

- **Manhattan** distance: the "shortest path", while only moving along one axis at a time.

$$d(a,b) = \sum_i \left(|a_i - b_i|\right) \tag{14.5}$$



Here, we have $|3-1| + |1-(-2)| = 5$

- **Hamming distance**: if we have two binary codes, how many digits are different between them?

$$d(a,b) = \sum_i \left(\mathbb{1}(a_i \neq b_i)\right) \tag{14.6}$$

$$a = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \Longrightarrow \quad d(a,b) = 2 \tag{14.7}$$

And as a bonus:

- **Minkowsky** distance: a generalized version of the euclidean/manhattan distance:

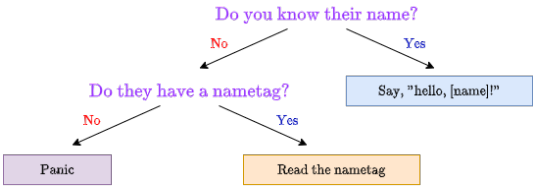
Notice that, for $p = 1$, it's equal to manhattan. For $p = 2$, it's equal to euclidean.

$$d(a, b) = \Big(\sum_i (a_i - b_i)^p \Big)^{1/p} \tag{14.8}$$

14.2 Tree Models

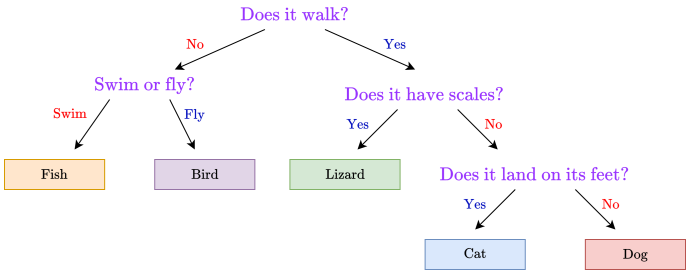
Here, we'll create another algorithm, based on a common way that humans solve problems.

- You might ask a series of **questions**, to narrow your situation down to a more specific example, that you know how to solve.



This tree answers the question, "how to start a conversation?". Panicking may not be the optimal strategy, but it's what this tree advises.

Each question is simple: a **binary**, "yes or no" question. But with enough questions, you can get pretty specific classifications:



A tree for classifying household pets. A bit too simple to be fully accurate.

Definition 13

A **binary tree** is a structure where:

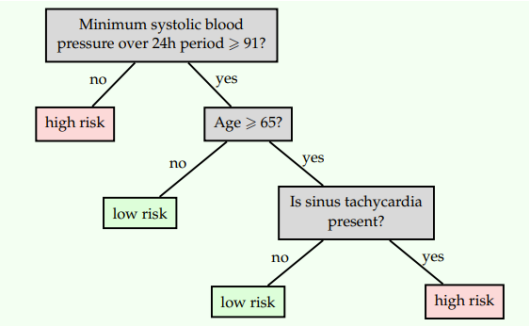
- You start at a **root node**: the **top** of the tree.
- At each node, you **split** your data into two "branches", based on a **binary** (yes-or-no) question.
- You **terminate** at a **leaf node**: each leaf node stops branching, and returns an output y.

The above diagrams show a strength of tree diagrams: they are **very interpretable**.

Concept 14

Tree models tend to be more interpretable than most other types of models.

They're so interpretable, that they can even be used for day-to-day problems, like medical analysis:



Reproduced from Breiman, Friedman, Olshen, Stone (1984).

However, they tend to work best on data with a small number of input dimensions, or at least a small number that matter.

Concept 15

Tree models tend to work best on datasets with:

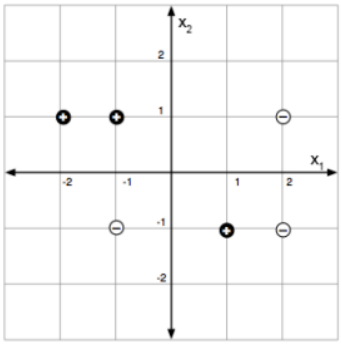
- Low dimensionality
- A small number of individual dimensions contain a lot of information

Otherwise, it can take a huge number of splits to get a productive result.

14.2.1 An example in 2D space

The examples we've shown so far have all been abstract, categorical questions. Now, let's consider an example where we have a **continuous** input space, 2D space.

We'll re-use the plot from the nearest-neighbor section:



Our goal is to split the space up, so that we can classify more easily.

This is the type of problem we'll deal with for the rest of the chapter: **n-dimensional, real-valued** space.

How do we want to split up our space? For simplicity, we'll only use **one axis** for each split.

Definition 16

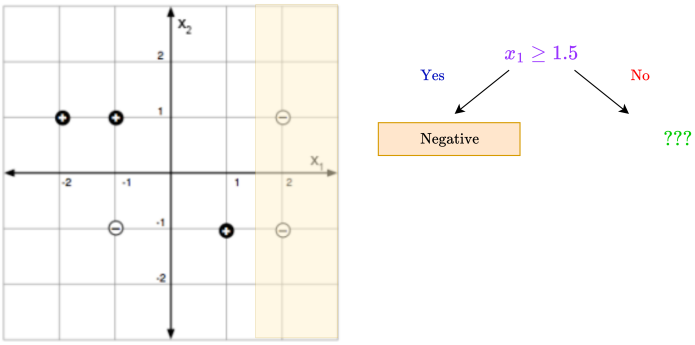
Our **tree-generating algorithm** will split the data along **one axis at a time**:

$$x_i \geqslant C \qquad \text{or} \qquad x_i < C$$

Let's give an example: what split would help you classify data?

- The two rightmost data points are both **negative**. So, let's separate them from the rest of the data:

$$x_1 \geqslant 1.5 \tag{14.9}$$

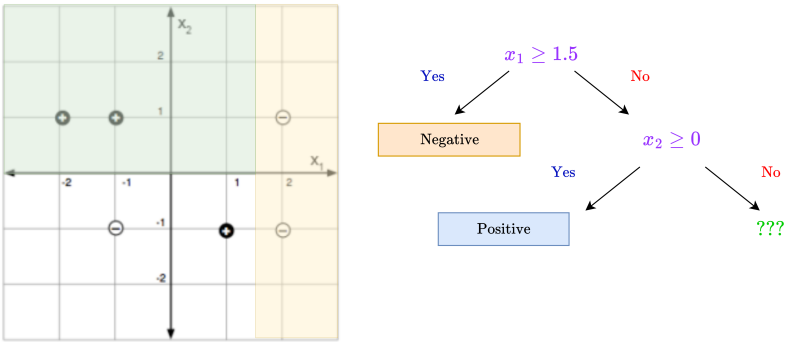


We've split our dataset once.

The right side is taken care of: everything is **negative**. Let's take our first attempt at splitting the left side.

- The top two data points are both **positive**.

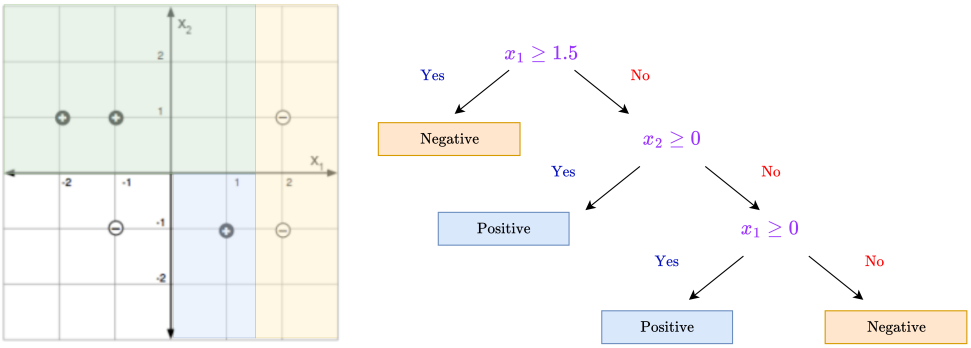
$$x_2 \geqslant 0 \tag{14.10}$$



Another split.

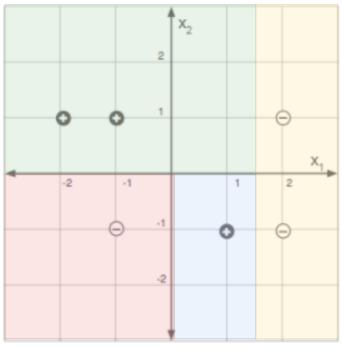
We only have to make one more split: between the positive and negative data point.

$$x_1 \geqslant 0 \tag{14.11}$$



And we're finished!

- Let's color in our last region.



14.2.2 Partitioning: Formalizing our Tree

In the above example, we've broken up our input space into chunks, or **partitions**.

Definition 17

Our tree is used to "**partition**" our data into multiple different chunks.

- Thus, we call one of these chunks, a single **partition**. Partitions are the "**leaf nodes**" of our tree.
- If we gather all of our partitions together, they should cover the **whole space**.

Each of these partitions is assigned a single **constant** O_m: every data point in that partition is given this constant as an output.

$$g^{(i)} = O_m$$

Example: Each of the differently colored regions above is one partition. The red and yellow partitions are assigned "negative", while the green and blue ones are assigned "positive".

Now that we understand how trees work, we can **formalize** our process, mathematically.

Our tree does two things:

- Put each data point in **one partition** R_m.
- Give each partition an **output value** O_m.
  - This is the output we'll use for any data point in this partition.

This is one of our regions, after splitting up the space.

These two parts make up our **predictor**.

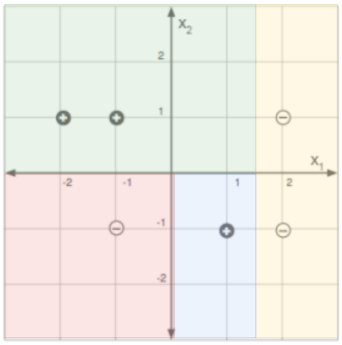
Definition 18

Suppose our tree creates M distinct partitions. The **predictor** generated by our **tree** has two main parts:

- A **partition function** π: this function assigns each point x in the **input space** to a **partition** R_m.
- A **collection of outputs** O: the m^{th} **output**, O_m, is assigned to **all points** x in region R_m.

Each of our training data is assigned to one partition, by the partition function.

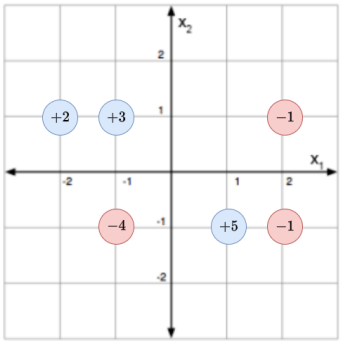
$$\pi\left(x^{(i)}\right) = R_m \qquad \Longrightarrow \qquad x^{(i)} \in R_m \tag{14.12}$$



The data point at $(2, 1)$ is assigned to the yellow partition. Since we created it first, we could call the yellow partition R_1. If we classify either -1 or +1, we should choose $O_1 = -1$.

14.2.3 Regression

Now, we want to show how to measure, and create this tree. We'll start with the problem of **regression**.



For regression, our values aren't categorical: they're real numbers. We used integers here, but we could've used 2.5 or $-\sqrt{2}$.

What output value do we give for O_m? Typically, the most accurate option would be the **average** of all outputs $y^{(i)}$ for data points in region R_m.

- But we only want to include the data points $(x^{(i)}, y^{(i)})$ where $x^{(i)} \in R_m$.
- To simplify things, we'll refer to each data point by its index.

Notation 19

Each training data point is referenced by its **index** i. So, rather than **partitioning** $x^{(i)}$, we'll partition indices $i \in I$.

- The training data which are **included** in R_m, will have their indices included in I_m.

$$x^{(i)} \in R_m \iff i \in I_m$$

We can use this to take our average.

Definition 20

In **regression**, O_m is the **average** of all outputs $y^{(i)}$ for training data in **partition** R_m.

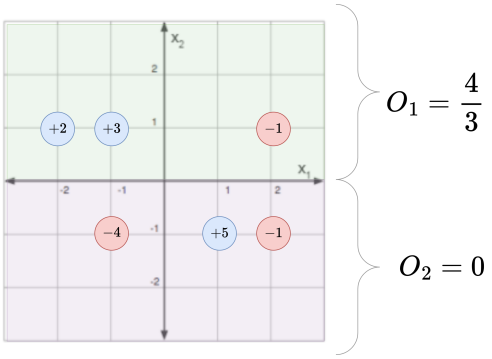
- So, we only include the data points $x^{(i)}$ relevant to R_m.

$$O_m = \text{Average}_{i \in I_m} \left(y^{(i)} \right)$$

or,

$$O_m = \text{Average} \left(\quad \left\{ y^{(i)} \quad \text{if} \quad x^{(i)} \in R_m \right\} \quad \right)$$

Example: Consider the following split.



We create a split at $x_2 = 0$. For each region, we average the value of all elements.

14.2.4 Regression Loss

How do we measure our loss? Same as usual for regression: we use **squared error**.

Definition 21

The **regression loss** for our region R_m is given by the **squared error** between our guess and the answer.

- We guess O_m for every data point in region R_m.

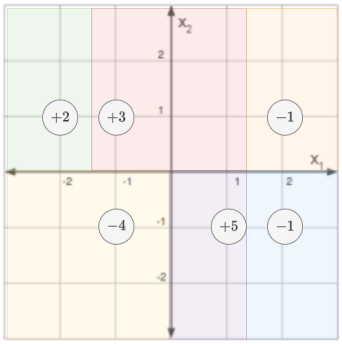
$$E_m = \sum_{i \in I_m} \left(O_m - y^{(i)} \right)^2$$

To get our loss, we could add up this loss over all of our regions.

If you want to practice with the above example, the error is 152/3.

$$\mathcal{L}_{err} = \sum_{m=1}^{M} E_m \tag{14.13}$$

But we have a concern: **overfitting**. What if we create too many regions? That wouldn't be very helpful.



If we wanted to, we could create a partition for every **single data point**. 100% accuracy.

This especially becomes a problem as we get a **larger dataset**: having a partition for every piece of training data will make you incredibly **sensitive to noise**.

So, we want to *discourage* this kind of behavior.

Concept 22

To reduce **overfitting**, we'll include a **regularization term** that discourages having too many regions.

- Our number of regions is M, so we want to **penalize** this: we'll add it to our loss function.

$$\mathcal{L}_{reg} = \lambda M$$

λ, similar to ridge regression, is used to indicate how strongly we want to **regularize**:

- Too high λ can result in **underfitting**: we get structural error, not splitting enough to accurately represent our data.
- Too low λ can result in **overfitting**: we split more than we need, focusing on noise in the data.

Combining these, we find our loss function for tree-based regression.

Key Equation 23

The **objective function** for **tree**-based regression has two parts:

- Loss $\mathcal{L}_{err}$, telling us how **inaccurate** our predictions are.
- Regularization $\mathcal{L}_{reg}$, penalizing us for having **too many splits**, and overfitting.

$$J = \mathcal{L}_{err} + \mathcal{L}_{reg} \quad = \quad \sum_{m=1}^{M} E_m + \lambda M$$

Where

$$E_m = \sum_{i \in I_m} \left(O_m - y^{(i)}\right)^2 \qquad O_m = \text{Average}_{i \in I_m} \left(y^{(i)}\right)$$

It's possible to search all partitions of our training data, and find the best one directly.

- But this is NP-hard. All you need to know is that it's incredibly time-consuming.

Concept 24

For large training sets, it's often **too expensive** to try all possible partitions of our training data.

Since partitions also aren't **smooth**, it'll be difficult to find the **optimal** solution via gradient descent.

So, instead, we'll come up with a (greedy) algorithm to find a pretty good solution.

14.2.5 Greedy algorithms

We need to design a tree-building algorithm. Our easiest bet is to be **greedy**:

Definition 25

A **greedy algorithm** is one that takes the choice that looks best, **immediately**.

- It doesn't factor in the **future** effects of our decision.

Example: In an MDP problem, this would be like looking for the best immediate reward $R(s, a)$, without at all consider what our future rewards look like.

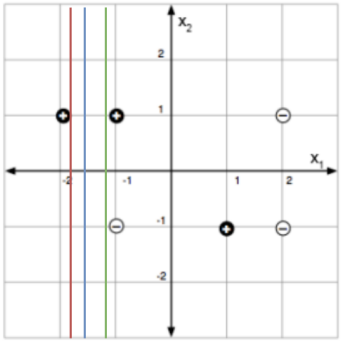
This is a very rough approach, but it's much faster than trying every possibility.

14.2.6 How to be greedy

So, we does our "greedy" choice look like?

- Our first thought might to be to try every possible split. But our input space is made up of **real numbers**: there are an infinite number of possible splits?

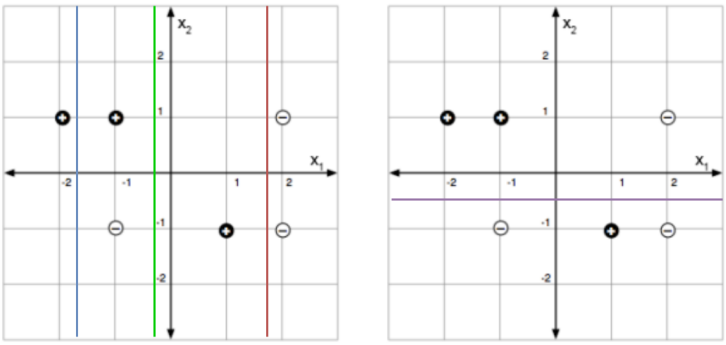
But are there an infinite number of splits that *matter*?



(Using classification ex., for visual clarity) All three of these splits are the same, as far as our training data is concerned.

This is useful: we don't have to try every possible split, because some splits are equivalent.

- We only create new splits each time we **move past** a new data point.
- So, we can iterate through our splits by going **one data point** at a time.



Here are all of the distinct splits between data points, on our two axes. We only create one split each time we cross data points on an axis.

This is the complete set of all possible "first splits".

Notice that we don't need to include splits that have no data on one side of our dataset: these splits do nothing.

- In order to be greedy, we just try all of these splits, and see which one works best.

If many splits are equivalent to the ones shown above, how do we know which ones to use?

Our answer: for this class, we don't really care. If you need a mental image, you can place each split halfway between the closest data points on either side.

Definition 26

In our **greedy algorithm** for tree generation, we:

- List every **distinct** split on each axis.
- Try every one of these splits, and measure their **error**.
- Choose the split with the **lowest error**.

After splitting once, we repeat this algorithm in **both halves** of the tree.

- And, once we've split a second time, we split all quarters.
- We repeat this process **recursively**.

One thing left: **termination**.

- We need a stopping point: if we don't terminate, then we'll continue until **every data point** has its own region.

Definition 27

We **terminate** one region of our tree-building algorithm if that region has **fewer than** k data points in it.

- Suppose that I_m gives us all of the **indices** in a particular region. We terminate when:

$$\left|I_m\right| \leqslant k$$

k is a **hyperparameter**.

Now, our algorithm is complete.

14.2.7 Tree regression pseudocode

We can also write this as pseudocode. But, let's establish some notation:

Notation 28

For this section, each split occurs on **dimension** $j \in J$, at **position** $s \in S_j$.

$$x_j \geqslant s$$

We're ready to go.

Below, we use $i \in I^+[j,s]$ to filter for the data points **above** the split, and $i \in I^-[j,s]$ to filter for the data points below the split.

BUILDTREE(I, k)

```
if |I| ⩽ k            # If fewer than k data points: no splitting

    ŷ = Average_{i∈I}(g^(i))          # Final output
    return LEAF(output = ŷ)           # Leaf node: no more splits

else                                  # Try every possible split
    for dim j in J                    # Check each dimension
        for value s in S_j            # Check each split on dim j

            I⁺[j,s] = { i ∈ I | x_j^(i) ⩾ s }     # Data points "above" the split (j,s)

            I⁻[j,s] = { i ∈ I | x_j^(i) < s }     # Data points "below" the split (j,s)

            ŷ⁺ = Average_{i∈I⁺[j,s]} (g^(i))      # Output "above" the split
            ŷ⁻ = Average_{i∈I⁻[j,s]} (g^(i))      # Output "below" the split

            E⁺ = Σ_{i∈I⁺[j,s]} ( ŷ⁺ − y^(i) )²
            E⁻ = Σ_{i∈I⁻[j,s]} ( ŷ⁻ − y^(i) )²
            E[j,s] = E⁺ + E⁻                      # Error for this split

    (j*, s*) = arg min_{j,s} ( E[j,s] )           # Pick split (j,s) with lowest error

# Recursion step
left_branch  = BUILDTREE( I⁻[j*, s*], k )         # Split the left/lower half of data
right_branch = BUILDTREE( I⁺[j*, s*], k )         # Split the right/upper half of data

return NODE(j*, s*, left_branch, right_branch)    # Our node contains the split, and the two halves after the s
```

14.2.8 Pruning

It's possible that our tree has more branches than it needs. There are a couple ways we might try to avoid this:

- Set k relatively **high**,
- Stopping when splitting doesn't improve the **error** very much.

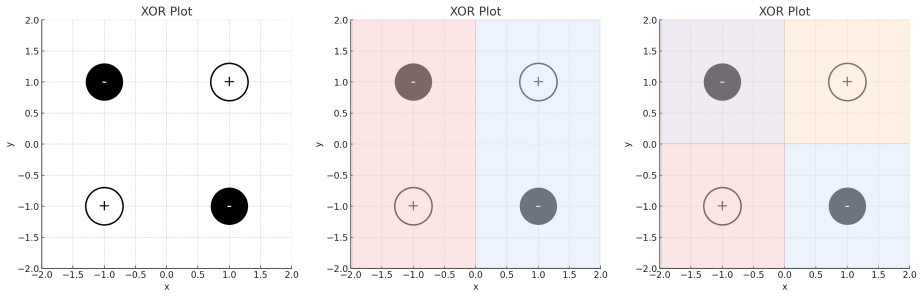
But stopping too early can be a problem:

Concept 29

One problem with **early stopping** in tree-building, is that some splits aren't obviously, immediately beneficial.

- But with one or two more splits after, they become very useful.

Consider the XOR problem.



With only one split, the accuracy isn't any better. But with two splits, we've fixed our problem!

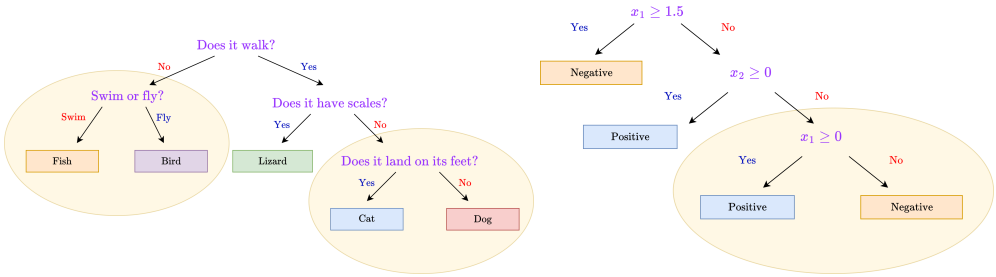
So, rather than stopping early, it's better to make too many splits, and then **prune**.

Definition 30

Pruning is to remove **branches** from your **tree**.

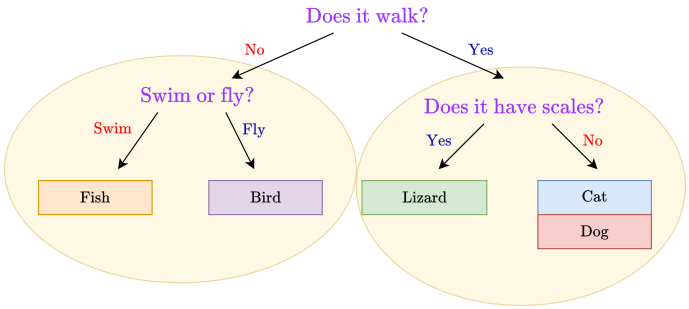
- We remove the "**lowest level**" branches first: splits that create two leaves.

Example: Consider our examples from earlier in the chapter.



The circled branches on each tree are the only ones we can prune.

Let's prune one from the pet tree:



Now that we've pruned the lowest branch, we can prune a new, higher branch.

How do we decide which branches to prune? With our **objective**:

$$J = \sum_{m=1}^{M} E_m + \lambda M \tag{14.14}$$

Here, we use a slightly different notation, so that we can express this as a **function**:

ML notation is a fickle thing.

Notation 31

Rather than our objective/loss, we have a **cost complexity function**, for our tree T, with some key notational changes:

$$\lambda \to \alpha \qquad M \to |T|$$

Giving us

$$C_\alpha(T) = \sum_{m=1}^{|T|} E_m + \alpha|T|$$

$|T|$ implies that our tree T's size is the number of partitions it has, M.

From here, we can **greedily** prune, until we have nothing left to prune.

- We'll return the tree that has the lowest cost complexity.

Concept 32

To **prune** our tree, we:

- Try to prune each of our **bottom-level** branches.
  - Actually prune the one with the **lowest cost complexity** (greedy algorithm).
- Repeat until we reach the **root note** (we've removed all of our splits).
- **Return** the pruned tree that has the lowest cost complexity.

Note that, just like how we keep **building** our tree past when it seems beneficial, we also **prune** past when it seems beneficial.

- For the same reason: it's possible for 1 prune to be worse, and 2 prunes to actually be much better.

How do we decide our "regularization term", α?

Concept 33

α, much like λ in **ridge regression**, can be selected via **cross-validation**.

Reviewing cross-validation:

- Break our training data into disjoint **chunks**.
- For each α value, train the model with a **different chunk**.
- Whichever model that performs best on the **held-out** data (the data outside the chunk), used the best α (we hope).

This is the α we use for future training.

14.2.9 Classification

We can re-apply this process for **classification**. Thankfully, most of the steps are the same.

- Our first major difference is that you can't **average** different categories.
- Instead, we'll pick the **most common** category: the *majority*.

How would you average cat, toaster, and forklift? It's, at best, ambiguous.

Definition 34

In **classification**, every point in a region is assigned O_m. O_m is the **most common** output for training data in **partition** R_m.

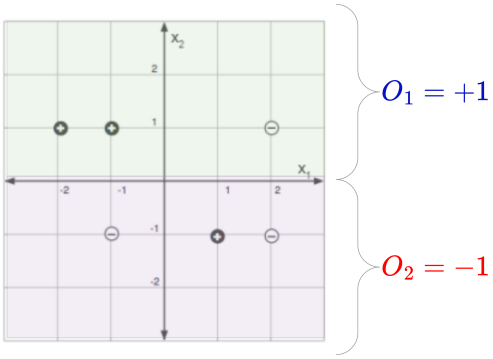
- In other words we want the **majority**.

$$O_m = \text{Majority}_{i \in I_m}\left(y^{(i)}\right)$$

or,

$$O_m = \text{Majority}\left(\left\{ y^{(i)} \quad \text{if} \quad x^{(i)} \in R_m \right\} \right)$$

Technically, we're willing to take the *plurality*, if there is no majority. But we can just say we want the "most common".



We create a split at $x_2 = 0$. For each region, we choose the most common class.

14.2.10 Classification Loss: Misclassification Error

How do we measure our performance?

- We'll need this kind of tool for choosing the **best split**, **pruning**, and **evaluating** our tree.

The simplest way would be, to simply count how many data points are misclassified.

Key Equation 35

One way to measure loss of our classification tree in **region** m is **misclassification error**: the fraction of data points misclassified.

$$Q_m(T) = \frac{\#(\text{Incorrectly Labelled, region } m)}{\#(\text{All data points, region } m)}$$

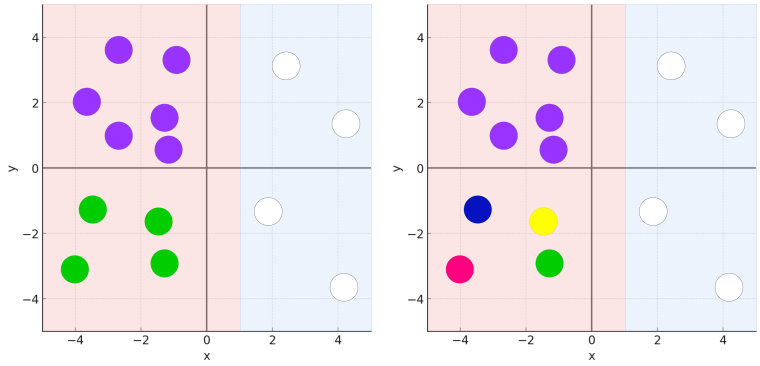
$Q_m(T)$ can also be seen as a **probability**: if you pull a random data point, what's the chance that it was predicted incorrectly?

This is a pretty simple metric, but there's something else we can try.

14.2.11 "Purity" of child nodes: Empirical Probability

One possible problem with "misclassification error" is that it could be **too simple**.

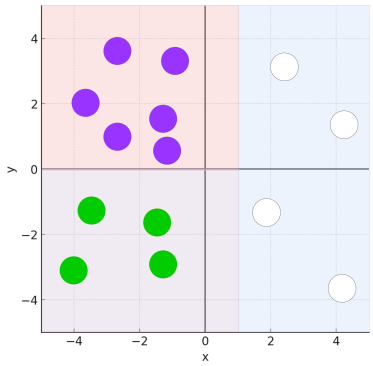
- **Example:** Suppose that we have 6 different classes, and one of our regions has a 60% accuracy rate.
- Our "incorrect" data points could be from **5 different classes**.



Let's focus on the left region. These are two very different situations!

For our more "pure" split on the left, we get **perfect accuracy** with only one more split:

This is an exaggerated, "lucky" example: having more of the same category just makes it **more likely** that we can find a good split.



While, this isn't true for the more "impure" split, with four different categories in the bottom-left.

- We can see that, despite having the same accuracy, the left split is generally "better".

Concept 36

Our **tree classification** problem is generally *easier* if our data is more **concentrated**, in fewer categories.

- If there are **fewer** categories, it'll be easier to find a large group of data points in the **same category**, to split from the rest.

So, we don't just prefer splits that have higher accuracy: we also prefer ones that create **more pure** regions ("child nodes").

So, we want to measure how "pure" a region is.

- A region that is more pure will have a **larger percentage** of data points in the same category.
- So, we want to measure the *fraction* of our data from a given category.

Definition 37

The **empirical probability** $\widehat{P}_{m,k}$ tells us what fraction of data points in **region** m, are in **category** k.

$$\widehat{P}_{m,k} = \frac{\#(\textbf{Category } k, \textbf{ region } m)}{\#(\textbf{Region } m)}$$

- We call it "empirical probability", because we have the **probability** of getting a data point in category k, in region m.

High $\widehat{P}_{m,k}$ means that **category** k is very **common** in **region** m.

14.2.12 Classification Loss 2: The Gini Index

Generally, we want a **high** empirical probability in a few categories: that'll mean that **most** of our data is in those few categories.

Naturally, this means low empirical probability in all the other categories.

- We have **two** different metrics for measuring this property, of having our data "**concentrated**" in a few categories.
- These are loss functions, so if they're large, then we have very "**diluted**" data, with lots of categories.

Concept 38

For our data to have high **purity**, we want our **empirical probabilities** $\widehat{P}_{m,k}$ to all be either high, or low.

- In other words: a **few** classes have a lot of data, **the rest** have very little.
- The fewer the number of "popular" classes, the **more pure**.

Our first measure asks the question, "if we randomly select a data point, and then select **another** (with replacement), what's the chance that we get a **different class** the second time"?

"With replacement" means that it's possible the select the same data point twice: after we select the data point the first time, we don't remove it.

- If we're likely to get a different category, each time we select one, that suggests that our data is "spread out" across a several categories.

Conversely, if our data was only in one class, then you'd always get the same category, every time.

There are two (equivalent) expression that compute this.

- First version: we focus on the chance of them being the **same** class.

$$\mathbf{P}\Big\{\text{2 different classes}\Big\} \quad = \quad 1 - \mathbf{P}\Big\{\text{Both same class}\Big\} \tag{14.15}$$

or,

$$\overbrace{Q_m(t)}^{\text{Region } m} \quad = \quad 1 - \overbrace{\sum_k \underbrace{\left(\widehat{P}_{m,k}\right)^2}_{\text{Same class twice}}}^{\text{Sum over all classes}} \tag{14.16}$$

- Second version: We select a first class k, and then, make sure our second class is **different**.

$$\mathbf{P}\Big\{\text{2 different classes}\Big\} \quad = \quad \sum_k \mathbf{P}\Big\{\text{Class } k\Big\} \cdot \mathbf{P}\Big\{\text{Not class } k\Big\} \tag{14.17}$$

or,

$$\overbrace{Q_m(t)}^{\text{Region } m} = \overbrace{\sum_k \underbrace{\left(\widehat{P}_{m,k}\right) \cdot \left(1 - \widehat{P}_{m,k}\right)}_{\text{Class } k\text{, then different class}}}^{\text{Sum over all classes}} \tag{14.18}$$

Key Equation 39

Another way to measure the loss of our split is the **Gini index**:

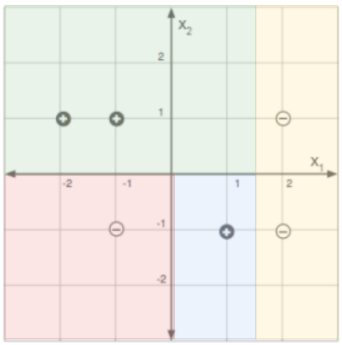
- "If we randomly sample **two** data points (with replacement), what's the probability they have **different classes**?"

$$Q_m(t) = \sum_k \left(\widehat{P}_{m,k}\right) \cdot \left(1 - \widehat{P}_{m,k}\right) = 1 - \sum_k \left(\widehat{P}_{m,k}\right)^2$$

14.2.13 Information 1: Uncertainty (Optional)

Entropy is another measure we can use. Entropy comes from **information theory**: we want each of our splits to give us the **most** information possible.

- But what do we mean by "information"?
- Let's consider our very "**informative**" tree example from the beginning of this section:



Based on which region you're in, you know the exact class of your data.

The tree provides perfect information, for **classifying** the training data: if we know the region of our data point, we know its classification.

- In other words, we have **no uncertainty** left.

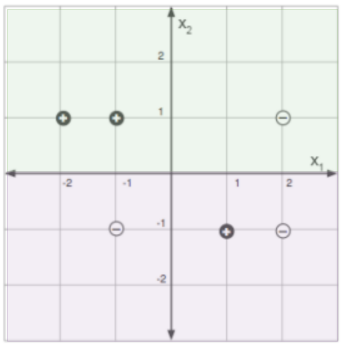
Definition 40

Uncertainty describes "how much" randomness we have in our outcome.

The more information we gain, the **less uncertainty** remains.

This is the kind of information we're discussing.

- Let's investigate further: what does "incomplete" information look like?



Even if we know the region we're in, we're still uncertain about the classification of our data.

This tree is less "informative", because we **don't know** exactly what class we're in.

- Still, it's better than no splits: in the top half, we have more +1 data than -1: we have **less uncertainty** than before.

If you had to guess the classification, and you guessed +1, you'd be right 2/3 of the time, rather than half the time.

Concept 41

Our tree is designed to provide **information** about our data:

- If you learn that a data point is in region R_m, you are **more certain** in guessing which class it's in.

The **less variation** we have in a region, the more **informative** it is:

- If a region R_m only contained a **single class**, you would know **exactly** the class of any data point you find there.
- But if there are many classes that are all likely, then you're pretty **uncertain** about which class to choose.

Example: You want to figure out if someone is sick. If I say, "she yawned earlier", that doesn't help: plenty of people (sick and non-sick yawn.

- But, if I say, "she's sneezing and coughing", that narrows things down: a lot more

people who are sneezing and coughing, are sick.

14.2.14 Information 2: Entropy (Optional)

Now, we have an idea of what we want. Next, we need to figure out how to **quantify** information.

- Let's use **complete certainty** as a baseline: the situation where we know exactly what class we're in.
- How much more work do we need to do, before we know our class with **100% confidence**?

Concept 42

Our goal is to measure our "**distance**" from being **completely certain** in our classification.

Suppose we want to identify a random data point. We use our **tree structure** to narrow it down to R_m. We've gained some information.

But how far are we from **complete certainty**?

- Our binary tree recursively split our data into two parts, so that we could "**narrow down**" our classification.
- We'll re-use that idea here: we'll ask a **binary** yes-or-no question, to narrow it down further.

We'll assume the **best case**:

Concept 43

We'll **measure uncertainty** based on how many **binary questions** we have to ask about our data point, before we're completely certain of its class:

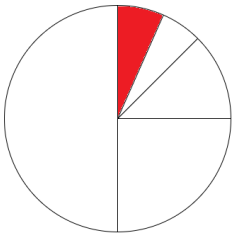
- Each binary question eliminates **half** of the data.
- We're careful with our split: we only remove data that is from a **different class** from our true data.

In other words: "how much work (how many questions) do you need to do, to get complete certainty?"

- If it takes more work, you're **more uncertain** of your answer.

Example: We select a data point, which happens to be in class A. Class A makes up 1/8 of the data.

- First question: we've remove half, Class A is 1/4 of the remaining data.
- Second question: we remove half again, Class A is 1/2.
- After 3 questions, all of our data is class A: our data point must be in class A.



We have to split our data in half 3 times, to get down to our 1/8.

So, each time, we **double** the proportion of class A.

- If $p = 1/8$, then we double 3 times. If $p = 1/16$, we double 4 times, and so on.
- This matches the behavior of the **logarithm** with **base 2**.

$$\log_2\left(\frac{1}{p_i}\right) = -\log_2(p) \tag{14.19}$$

This can be a bit weird when our probability isn't a power of 2: for example, 1/3. In which case, we could round up: we have to ask 2 questions.

But when we're computing uncertainty, we still consider 1/4 "more uncertain" than 1/3. So, we'll allow non-integer values.

Key Equation 44

Each class c_i contributes to the **uncertainty** within our region R_m:

$$\log_2\left(\frac{1}{p_i}\right) = -\log_2(p_i)$$

We also sometimes call this...

- **Information**: if you need to ask n questions to narrow down your data, then you need n binary answers: n **bits of information**.
- **Surprisal**: if an outcome is less common/likely (lower p), then we're **more surprised** when it happens.

But we're not quite done: this is just our uncertainty associated with one of our outcomes.

- How do we aggregate this, over all of our possible classes?

We'll get the **expected value**: if we pull a random data point, what's the **average number** of **binary questions** until we've identified it?

$$E[X] = \sum_i p(x_i) \cdot x_i \quad \Longrightarrow \quad -\sum_i p_i \log_2(p_i) \tag{14.20}$$

This is called **Entropy**.

Definition 45

Entropy $H(X)$ tells us how **spread-out** our data is, across different outcomes.

- More entropy means that our data is more spread-out, and **uncertain**.

$$H(X) \quad = \quad -\sum_i p_i \log_2(p_i)$$

In the simplified case, we could think of this as answering the following question:

- If we pull a random data point, what's the **average number** of **binary questions** until we've identified it?

What do we do if we have a non-integer entropy? This still measures how "spread-out" or uncertain our data is:

- We're closer/further from having to add one more binary question.

14.2.15 Classification Loss 3: Entropy

Now, we move back to our tree. We need to make one notational change: *empirical probability*.

Key Equation 46

The entropy of a **region** is computed with our **empirical probabilities**:

$$H(I_m) \quad = \quad -\sum_i \widehat{P}_{m,k} \cdot \log_2\left(\widehat{P}_{m,k}\right)$$

This is one way to measure the **purity** of our data.

One important caveat:

Clarification 47

$0 \log_2(0)$ is **not defined**. However, for calculations, we usually assume:

$$0 \log_2(0) = 0$$

- The **limit** supports this choice:

$$\lim_{n \to 0} n \log_2(n) = 0$$

- Additionally, we use entropy to indicate "**uncertainty**". With no data, there's no uncertainty.

Now that we have entropy, we can use it to determine the best splits.

- The lower the entropy, the less **spread-out** our data is.
- So, we want splits that **reduce entropy** the most.

Concept 48

In our greedy algorithm, we choose the splits with the **lowest entropy**:

- These are the splits which give us the most "**information**" about our data.

Because we have fewer classes in each region, our next split may give better accuracy, as well.

We discussed the possible benefits of having more "pure" data in 12.2.11.

A region with only 2 classes of data, might be more easily split, than a region with 5 classes.

How do we compute the "**change** in entropy" after we've split? We could just add, or **average**, the entropy of both regions.

- But, this could be **misleading**: if we split our data into a region of 2 data points, and a region of 20 data points, we probably care more about the region with 20.
- So, we'll do a **weighted average**: the region with more data points, contributes more to entropy.

$$\frac{\#(\text{Data in region } m)}{\#(\text{Total data})} \cdot \overbrace{H(I_m)}^{\text{Entropy in region}} \tag{14.21}$$

Or, using more dense notation:

$$\left(\frac{\#I_m}{\#I}\right) \cdot \overbrace{H(I_m)}^{\text{Entropy in region}} \tag{14.22}$$

Remember that I represents all of your data points, while I_m represents data in a region.

Key Equation 49

The **entropy** $\widehat{H}$ *after* a split is the **weighted average** of the two splits:

$$\widehat{H} = \left(\frac{\#I^+}{\#I}\right) \cdot H(I^+) + \left(\frac{\#I^-}{\#I}\right) \cdot H(I^-)$$

Our data I is broken into I^+ (data points above split), and I^- (data points below the split).

If we want to be pedantic, we could create separate notation for splitting at value s, on axis j: we use $I_{j,s}$ notation.

$$\widehat{H} = \left(\frac{\#I^+_{j,s}}{\#I}\right) \cdot H(I^+_{j,s}) + \left(\frac{\#I^-_{j,s}}{\#I}\right) \cdot H(I^-_{j,s}) \tag{14.23}$$

We can also say that we "**gain information**" when we reduce entropy: it takes **less** additional information to completely know our classification.

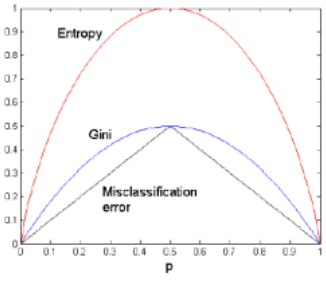
Key Equation 50

The **information gained** from a split comes from comparing the entropy, before and after the split:

$$\text{InfoGain} = \overbrace{H(I_m)}^{\text{Pre-split}} - \overbrace{\hat{H}}^{\text{Post-split}}$$

14.2.16 Which Loss function to use?

In the past, there's been a lot of debate over which loss function works best.



All three capture the basic idea of accuracy/purity:

- If a class matches **none** of our data ($\widehat{P}_{m,k} = 0$), it doesn't contribute to the loss.

- If a class matches **all** of our data ($\widehat{P}_{m,k} = 1$), it *also* doesn't contribute to the loss.

For our purposes, we'll consider the following.

Concept 51

Traditionally, we use:

- **Entropy** for tree-builiding
- **Misclassification error** for pruning

14.2.17 Bagging: General Concept

One major weakness of our tree model is their **sensitivity** to the data they receive.

- Suppose some noise in our data makes our first split **different**. That will affect **every split** that comes after.
- The second split **depends** on the regions created by the first split. The same is true for the third split.

So, a small change early in our tree can create a dramatically different overall structure.

Concept 52

Trees are **sensitive to noise**.

- If you have **slightly different** training data, you can end up with a **very different** tree structure.

This means that our trees are very vulnerable to **estimation error**:

- Even if it's *possible* to get a good tree (low structural error), random chance can often give you a **much worse** tree.

Our solution? Create **several different trees**, with modified training data.

- We **combine** the "opinion" from each tree, and give an answer based on that. This is a type of **ensemble method**.

Definition 53

When using an **ensemble method**, we use **multiple models** together, to solve a problem.

- There are multiple different kinds of ensembles: **boosting** and **bagging** are popular examples.

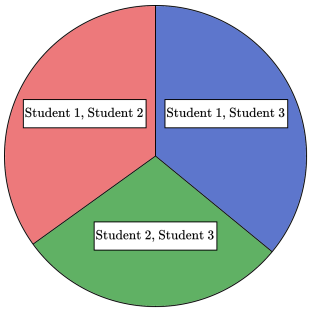
Here, we'll discuss **bagging**.

We hope that having multiple trees allows us to "**average out**" the randomness from each one.

- Each tree might have some estimation error problems, but we hope that most trees don't make the **same mistakes**.
- If they make different mistakes, then each tree can **cover** for the weaknesses of other trees!

Example: Suppose that you have 3 students, trying to do 3 homework problems together. For each question, they take the majority vote.

- Each student gets 1 in 3 questions wrong, but it's a different question.



Every student gets one question wrong, but by majority, they get all three right!

Despite each student having a $\tilde{66}$% accuracy, they have a 100% accuracy together!

This is a very optimistic version of why bagging could be beneficial. But the idea holds true in general.

We call this "bootstrap aggregation", or "bagging".

Concept 54

Bagging is a particular kind of **ensemble method**, combining several models to compute an answer.

- In bagging, we train several models **separately**, and then combine all of their answers, to hopefully find a more **accurate** result.

14.2.18 Bagging: Bootstrapping (Optional)

So, now, we need to hammer out the details of this process:

- Create **multiple** datasets
- Train a **tree** on each of those datasets
- **Combine** the results of those trees

First: how do we create multiple datasets from our training data?

- We could try breaking our data into **chunks**, like we do in cross-validation.

But that's not what we want here:

Concept 55

In **bagging**, we don't want to break our data into **chunks** (partitioning).

This is because these chunks of data aren't independent: they're **correlated** with each other.

- If you include data point i in chunk k, you **know** that data point i **isn't** in chunk $k+1$.
- Knowing about one chunk, provides information about the other chunks: that's how you know they're correlated.

We want each of our models to make its decisions, **independent** of our other models.

Remark (Optional)

If our chunks of data are correlated, then why do we use **partitioning** for our **cross-validation**?

- In cross-validation, we want to see how our model performs on data it **hasn't seen before**.
- So, it's important to make sure that the chunk we **train** on, is different from the chunk we **test** on.

How do we create our datasets, then?

- We **sample with replacement**: after you sample a data point, you **put it back** in the dataset.
- So, you can sample the same data point **multiple times**, or not at all.
- Each dataset we make is called a **bootstrap sample**.

Concept 56

When creating data for **bagging**, we use **bootstrapping**:

- Each dataset is created by **sampling with replacement**. This dataset is a **bootstrap sample**.

This creates two major benefits:

- Each bootstrap sample is **uncorrelated**: knowing the data in one dataset, tells you nothing about the others.
  - That means our trees will be **fully separate** from one another.
- When bootstrapping our dataset, we randomly **modify** it, which helps us account for **estimation error**:
  - Each dataset can end up with multiple of a single data point, or missing several data points.
  - So, each tree uses a different "**variation**" of our dataset.

This bootstrapping process, along with "aggregating" opinions across multiple models, is why we call this **bootstrap aggregating** (shortened to bagging).

But why do we call it "bootstrapping"?

Remark (Optional)

Our training data comes from our true distribution. We could say that it was "**sampled**" from it.

- Meanwhile, our "**bootstrap sample**" comes from sampling our **training data**.
- In other words, we're **sampling a sample**.

This is kind of **circular**: we're getting "more data", without actually getting **new** data.

Thus, we call it **bootstrapping**, referencing the phrase "pull yourself up by your bootstraps".

- Because, it's circular to try to "pull yourself up".

Bootstrap sampling is also used for statistical analysis, when our data is limited.

- If we bootstrap from our sample, multiple times, we can compute the *mean* of those bootstraps.

- So, rather than just having the single mean of our whole sample, we find multiple possible means we could get from our data.

This can be useful: for example, suppose that your bootstrap means vary wildly. We might not be able to trust our sample mean, if it changes so easily.

14.2.19 Bagging: Completed

We have a procedure for bagging now.

Definition 57

Bagging (bootstrap aggregation) is an **ensemble method** for reducing **estimation error**, by combining the answers from B independent models.

- First, we create a **bootstrap sample** for each tree, sampling with replacement from our dataset $\mathcal{D}$.
- We train the b^{th} tree using the b^{th} bootstrap. The **predictor** we get from this is written as

$$\hat{f}^b(x)$$

When we're evaluating a particular data point x, we use each of our models, and then **aggregate** the results.

- Regression and classification use different methods of aggregation.

Our "aggregation" method is the same as it was for determining the output of one tree region R_m.

Key Equation 58

In a **regression** problem, we aggregate our B trees by **averaging** their results.

$$\hat{y}_{bag}(x) = \frac{1}{B}\sum_b \hat{f}_b(x) = \text{Average}_b\left(\hat{f}_b(x)\right)$$

In a **classification** problem, we aggregate our B trees by taking the **majority** (most common) vote.

$$\hat{y}_{bag}(x) = \text{Majority}_b\left(\hat{f}_b(x)\right)$$

We *hope* that bagging reduces estimation, but does it really? It turns out it does!

- But as a tradeoff, it's not easily interpretable, like a single tree is.

Imagine having to reference 20 different trees every time you need to make a decision... it's unwieldly.

Concept 59

Bagging can be shown to **reduce** estimation error.

- So, **bagged trees** will generally perform **better** than an individual tree.

But, we it's much more difficult to interpret a bagged tree, than a single one.

- You have to look at **several** different trees to understand a decision.

This isn't just saying that, in practice, bagging seems to reduce estimation error. We actually have theoretical results that suggest it should!

14.2.20 Random Forests

In bagging, we created several different "opinions" on our data, by training each tree on a modified version of our dataset.

Here, we'll consider a different approach:

- Instead of modifying our dataset, we'll modify which **dimensions** we split along.
- Each time we make a split, we'll **randomly select** a few dimensions, and only choose the best of those to split along.

So, if the original "best" splitting dimension is omitted, then the tree will choose the best one it has access to.

This time, all of our trees have the **same dataset**, but they split in **randomly different** ways.

- Many trees, modified randomly: we call this a **random forest**.

Definition 60

The **random forest** approach is an **ensemble method** where, instead of modifying the data for each tree, we modify the **splitting algorithm** for each tree.

- Normally, when training a tree, it selects the best split, on the best dimension.
- But in random forests, we **randomly restrict** our tree to only split along **some dimensions**.

So, our tree splits the "best dimension", among the ones those that are randomly selected.

Restricting our tree seems counter-intuitive: why would we deliberately prevent it from choosing the best split?

- This forces some of our trees to "**explore**" other splits, which might be worse short-term, but better long-term.
- Hopefully, this avoids the **estimation error** problem: by trying lots of trees, we can avoid getting "unlucky" with one tree.

Random forests often perform remarkably well, compared to many, much fancier methods.

14.2.21 Other types of tree models

There are tons of tree variations:

- We could split along **any hyperplane**, rather than restricting ourselves to only one axis.
  - In which case, we have to figure out how to limit our options: there are many more hyperplanes, than ways to split on only one axis.

For example, we could try $2x_1 + 3x_2 \geqslant 0$, rather than $x_1 \geqslant 10$.

- We could expand beyond hyperplanes: we could use **polynomial curves**, like paraboloids.

Similar to our "polynomial features" method.

- We could use **linear regression** for each region R_m, rather than just averaging all of our data points.
- We could use a **probabilistic** split: rather than a data point belonging in exactly one region, R_m, it could *partly* belong to all of them.
  - **Example:** Our data point could 90% belong to R_1, 10% belong to R_2.
  - Because this is more continuous than our previous method, we can often use **gradient descent** to train.

This is called a "hierarchical mixture of experts".

14.2.22 Benefits of Trees

We've shown lots of reasons you might want to use a tree:

- Easy to interpret, fast to train.
- Very flexible with different loss functions, problem types.
- Often surprisingly effective, despite their simplicity.

Concept 61

It's often good practice to use **trees** as a **baseline**, to compare more **complex** models against:

- If your complex model doesn't perform much better than a tree, it may not be worth using.

14.3 Terms

- Parametric Methods
- Expressive (Review)
- Non-parametric methods
- k-means clustering (Review)
- Nearest neighbor
- Tree models
- Ensembles
- Boosting (Optional)
- Voronoi Diagram (Optional)
- k-nearest neighbors (kNN)
- Locally weighted regression (Optional)
- Distance metric
- Euclidean distance (Review)
- Manhattan distance (Optional)
- Hamming distance (Optional)
- Minkowsky distance (Optional)
- Binary tree
- Root node
- Leaf node
- Branch
- tree model
- Partition
- Partition function π
- Collection of outputs O
- Index
- Tree model objective function
- Greedy algorithm

- Early stopping (Review)
- Pruning
- Low-level/high-level branch
- Cost complexity function
- Majority function
- Misclassification Error
- Empirical Probability
- Purity
- Gini Index
- Information (Optional)
- Uncertainty (Optional)
- Surprisal (Optional)
- Entropy
- Information Gain
- Ensemble Method
- Bagging
- Correlated
- Bootstrapping
- Sampling with Replacement
- Random forest
- Hierarchical Mixture of Experts