Strong machine learning foundations & traditional approaches

The Academy of AI 2018/19

Session 6

Students for AI

Overview

Strong machine learning foundations

- What are we trying to do?
- Maximum likelihood

Traditional machine learning approaches

- Support vector machines (SVMs)
- Classification and regression trees (CARTs)
- Gaussian Processes (GPs)

What are we trying to do?

Most generally, we want to map some input to some output.

These inputs and outputs can take any form, for example we might map:

- Text to its overall sentiment
- Images to the boxes that bound specific objects within them
- Medical images to classification/prediction of a condition
- Previously purchased items to predicted future purposes

That means: we are trying to find some **function** that performs this mapping.

We will spend the rest of the course exploring how to **learn** these function mappings robustly and effectively

PREDICTION

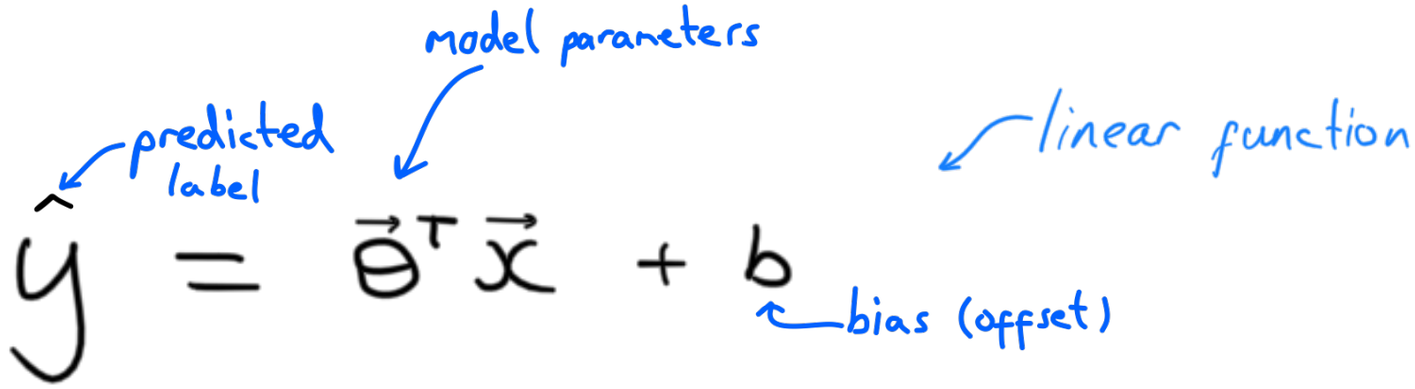
How are we going to do this?

Learning how to perform our desired input to output mapping will require 4 things:

1. The model
   - This is the function that we will pass data forward through, and eventually expect to produce our desired output
2. The data
   - Data that we provide our model will determine what it is that we are trying to learn - that is, what function maps between these inputs and outputs
3. The criterion
   - Our criterion is a function that evaluates how badly our model is performing
4. The optimiser
   - Our optimiser will define how we make adjustments to our model to improve its performance

Linear regression example

Our model:



Parameter estimation

Mathematically, what does it mean to try to find a function that maps our inputs to our outputs in the way that we want?

Almost all of the models that we will be building after today are **parametric models**.

That means that the mappings which they perform are determined by some parameters.

For instance, our univariate linear regression model has parameters **theta** and **b**

$$\hat{y} = \underline{\theta}^T \underline{x} + b$$

parameters

Given the architecture of our model, our main goal is to **estimate** a satisfactory set of model parameters that will make our model perform the function that we want to represent.

But what parameters are these? What qualities do they have?

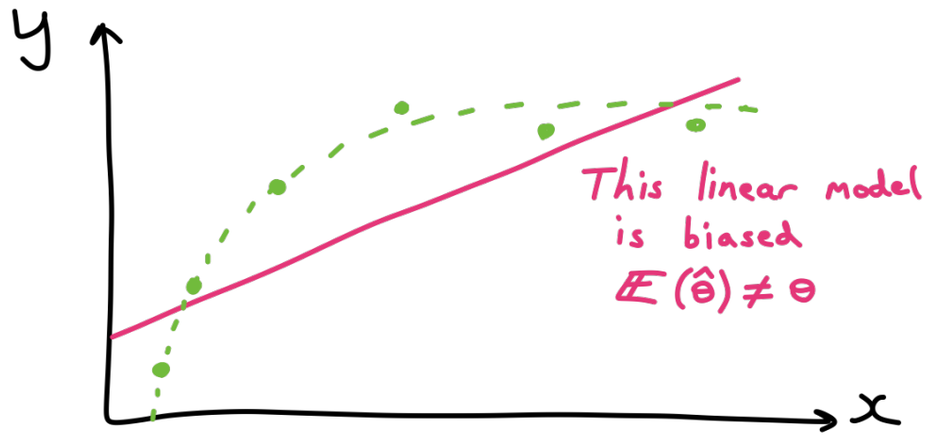
Bias

What is it?

$$bias(\theta) = \mathbb{E}\left[\hat{\theta} - \theta\right]$$

estimated parameter

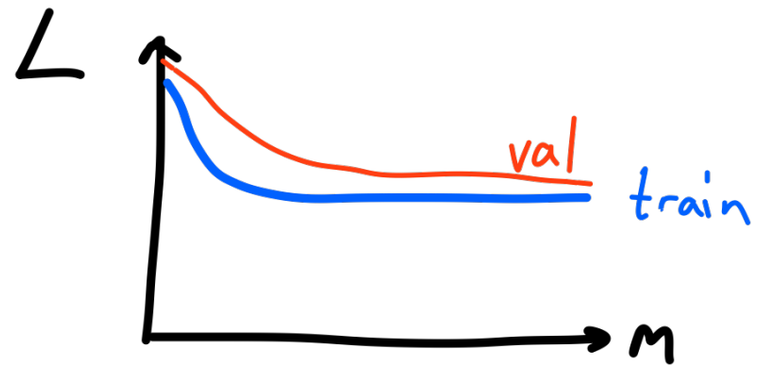
true ideal parameter



- Model is actually estimating something different to what we want

What does this cause?

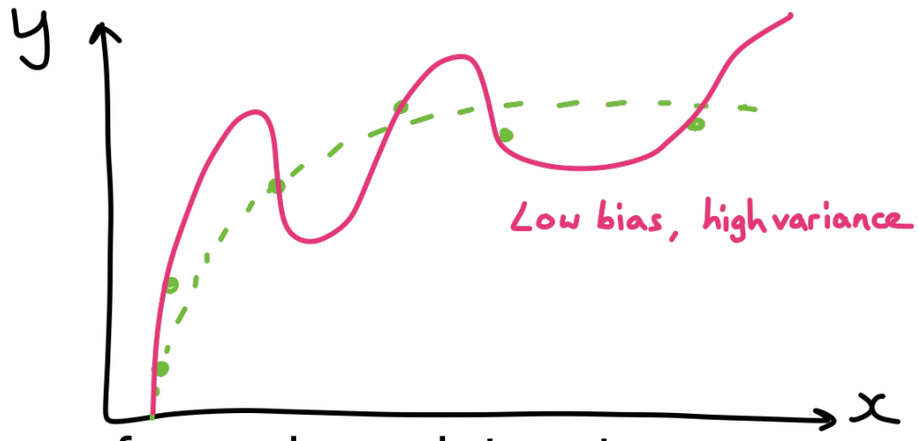
- Both train and validation loss plateau off
- Model fails to capture true input-output relationship
- Underfitting



What can we do about it?

- Increase the capacity of our model (the types of functions that it can represent)
- Change our objective function

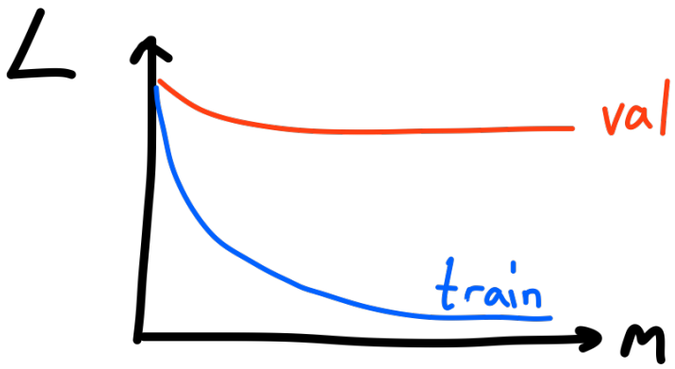
Variance



What is it?

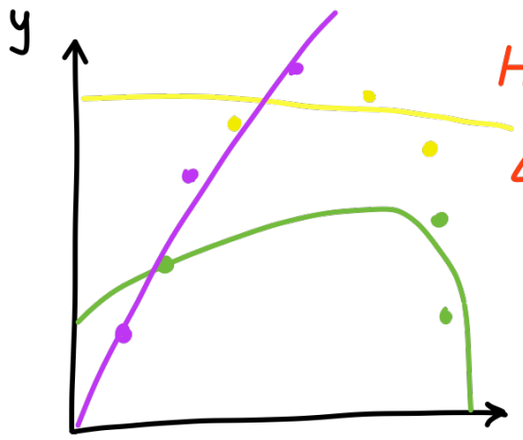
$$Var(\hat{\theta}) = \mathbb{E}\left[(\hat{\theta} - \bar{\theta})^2\right]$$

- There are many different low error parameterisations for a given dataset
- The best parameterisation varies with different data from the same distribution



What does this cause?

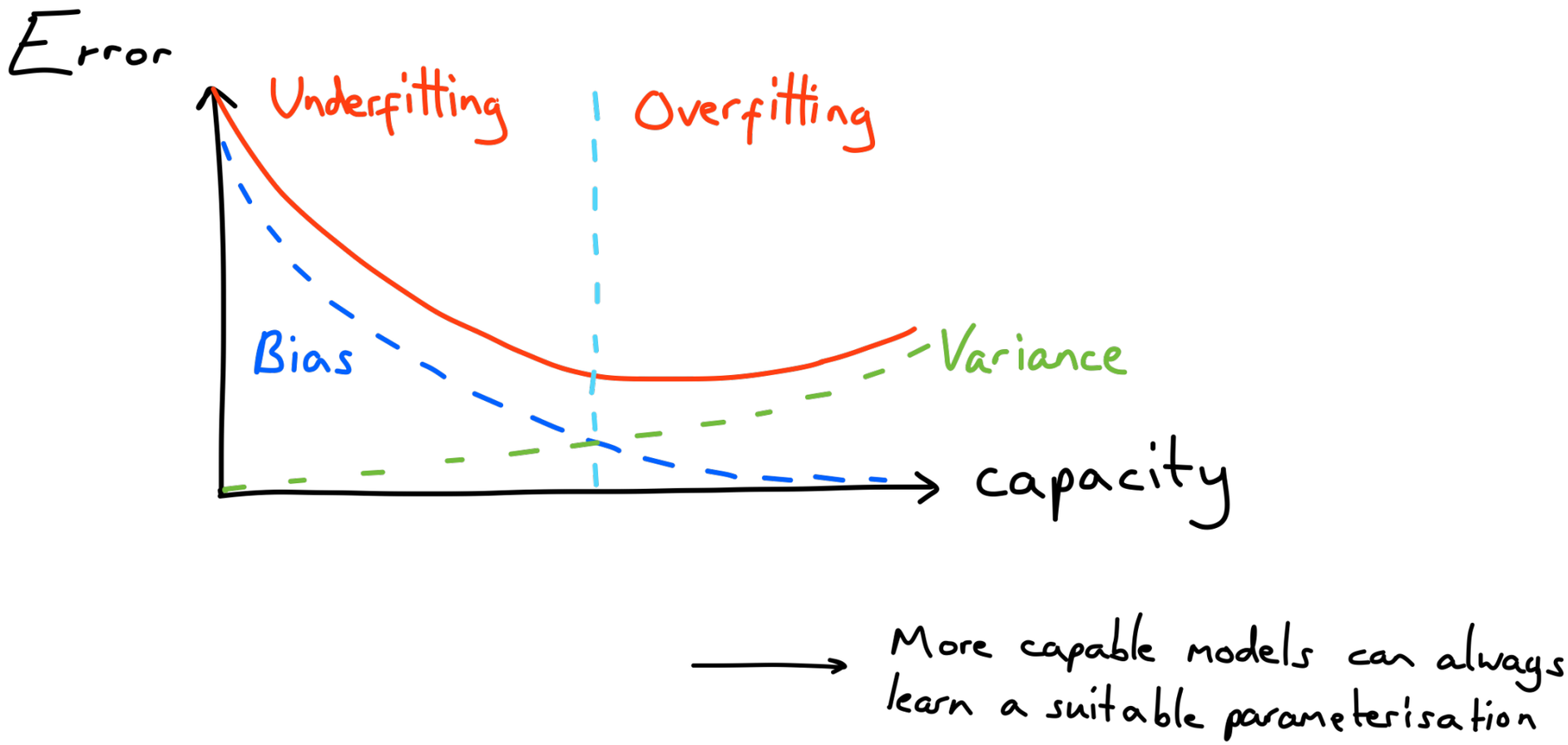
- Great training loss
- Awful validation loss!
- Overfitting
- Highly variable parameter estimates for different datapoints from the same distribution



What can we do about it?

- Reduce the capacity of our model
- Train on more datapoints
- Change our objective

Bias-variance trade off



→ More capable models can always learn a suitable parameterisation

Maximum Likelihood Estimation (MLE)

How do we know what parameters make a good model?

Occam's razor: "the most likely solution is the simplest solution"

That is, the best model (defined by its parameters) is that which predicts observations that we make as being highly likely.

So

Intuitively:

1. We make some observations
2. Our parameterised model predicts how likely those observations were
3. The best parameters for our model are those which maximise the chance of making those observations

The maximum likelihood estimator for our model parameters theta is thus defined by

Formally:

$$p_q = e^{\log p_q} = e^{\log p + \log q}$$

$$\operatorname{argmax} \Pi p = \operatorname{argmax} e^{\log \Pi p} = \operatorname{argmax} e^{\sum \log p} = \operatorname{argmax} \sum \log p$$

1. $\mathbb{X} = \{x^{(0)}, \ldots, x^{(m)}\}$
2. $p_{model}(\mathbb{X}; \theta) = \prod_{i=1}^{m} p_{model}(x^{(i)}; \theta)$

$$\Pi p(x) = e^{\sum \log p(x)}$$

monotonically increases

3. $\theta_{MLE} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(x^{(i)}; \theta)$

this is where "negative log likelihood" comes from

$$= \arg\max_{\theta} \mathbb{E}_{x \sim \hat{p}_{data}} \log p_{model}(x; \theta)$$

this objective is based on the statistics of our training data ← so we better hope that that distribution is representative of the data we care about!

So if we are training with maximum likelihood, where does the mean squared error come from?!

Why not the mean absolute error or the root mean squared error or some other loss function for continuous outputs?

Mean squared error is actually motivated by maximum likelihood!

E.g. predict someone's age from their height

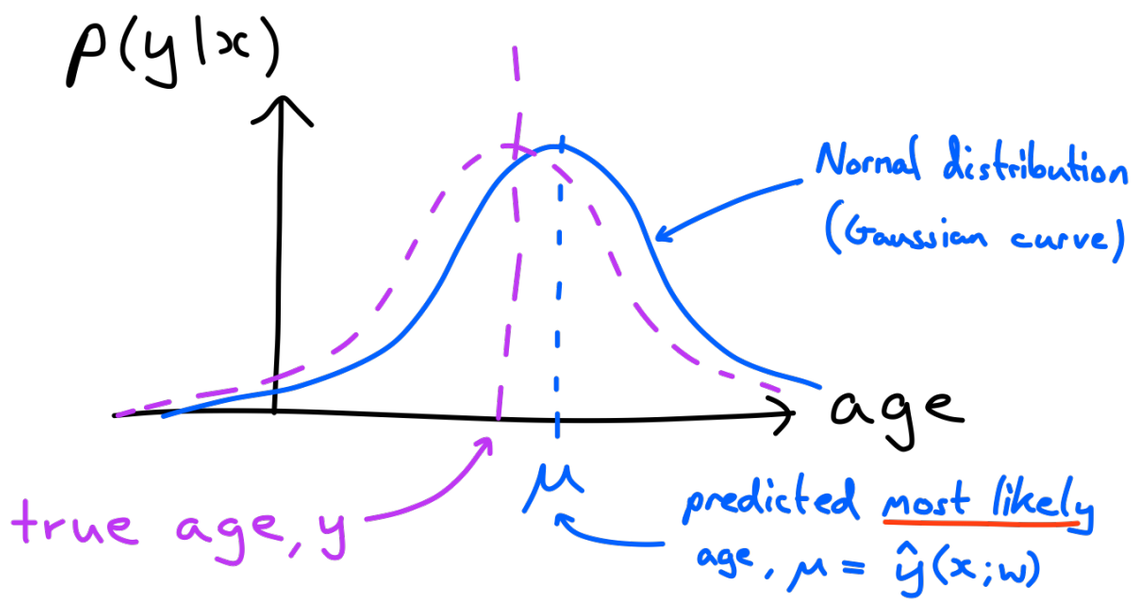
We can usually tell roughly, but we can never be sure, because these two things are not perfectly correlated. What we actually predict, are confidences that a person is a certain age.

This indicates that our predictions might be well modelled by a Gaussian (normal distribution).

but this is an assumption ← assuming a "Gaussian prior"

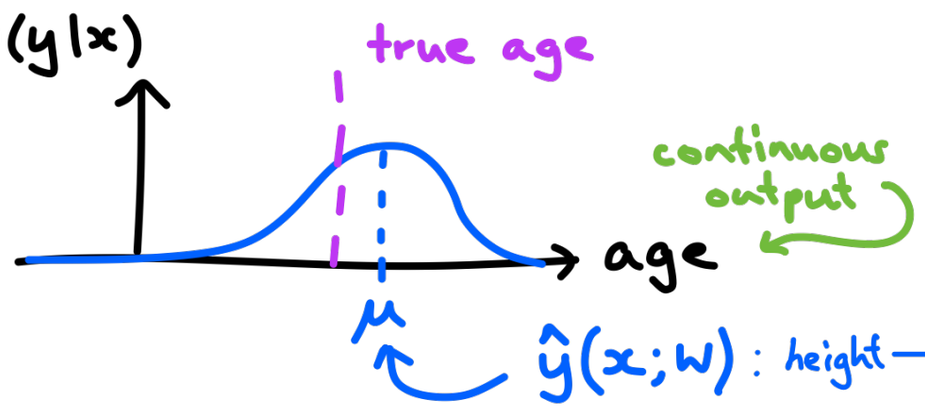
$$p(y|x) = \mathcal{N}(y;\ \hat{y}(x;w),\ \sigma^2)$$

assume that we are predicting the mean



Derivation of mean squared error

continuous output transformed into probability → $p(y|x)$



$$\mathcal{N}(\hat{y};\, y(x;w), \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2}}$$

Normal distribution equation

$$\theta_{MLE} = \underset{\theta}{\text{argmax}} \left[\sum_{i=1}^{m} \log p(y^{(i)} | x^{(i)}; \theta) \right]$$

$$= \underset{\theta}{\text{argmax}} \left[\sum_{i=1}^{m} \log \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2}} \right]$$

$\log ab = \log a + \log b$

$$= \underset{\theta}{\text{argmax}} \left[\sum_{i=1}^{m} \log \frac{1}{\sqrt{2\pi}\sigma} - \sum_{i=1}^{m} \frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2} \right]$$

$$= \underset{\theta}{\text{argmax}} \sum_{i=1}^{m} \left[-\|\hat{y}^{(i)} - y^{(i)}\|^2 \right]$$ (squared L2 norm)

$$= \underset{\theta}{\text{argmin}} \sum_{i} \left[\|\hat{y}^{(i)} - y^{(i)}\|^2 \right]$$

$$= \underset{\theta}{\text{argmin}} \frac{1}{m} \sum_{i}^{m} \left[\|\hat{y}^{(i)} - y^{(i)}\|^2 \right]$$

NICE!

i.e. maximum likelihood = minimise MSE

$$= \underset{\theta}{\text{argmin}} \mathbb{E}_{x \sim \hat{p}_{data}} \|\hat{y}^{(i)} - y^{(i)}\|^2$$

Summary of foundational concepts:

- We are trying to learn functions that map or inputs to our outputs
- Specifically, or goal is to estimate suitable parameters that control these functions
- Our estimates of the ideal parameters should be unbiased (learn parameters that perform our function mapping better) and have low variance
- We are going to train most of our models using maximum likelihood
- The optimal model will trade-off bias and variance
- Sometimes we may **assume** that our labels follow some certain distribution, given our **prior** understanding of the problem.
- Then we can try to predict the parameters of that probability distribution, rather than a point value.
- Learning the best parameters for regression means minimising the square error between our most confident predictions and true labels

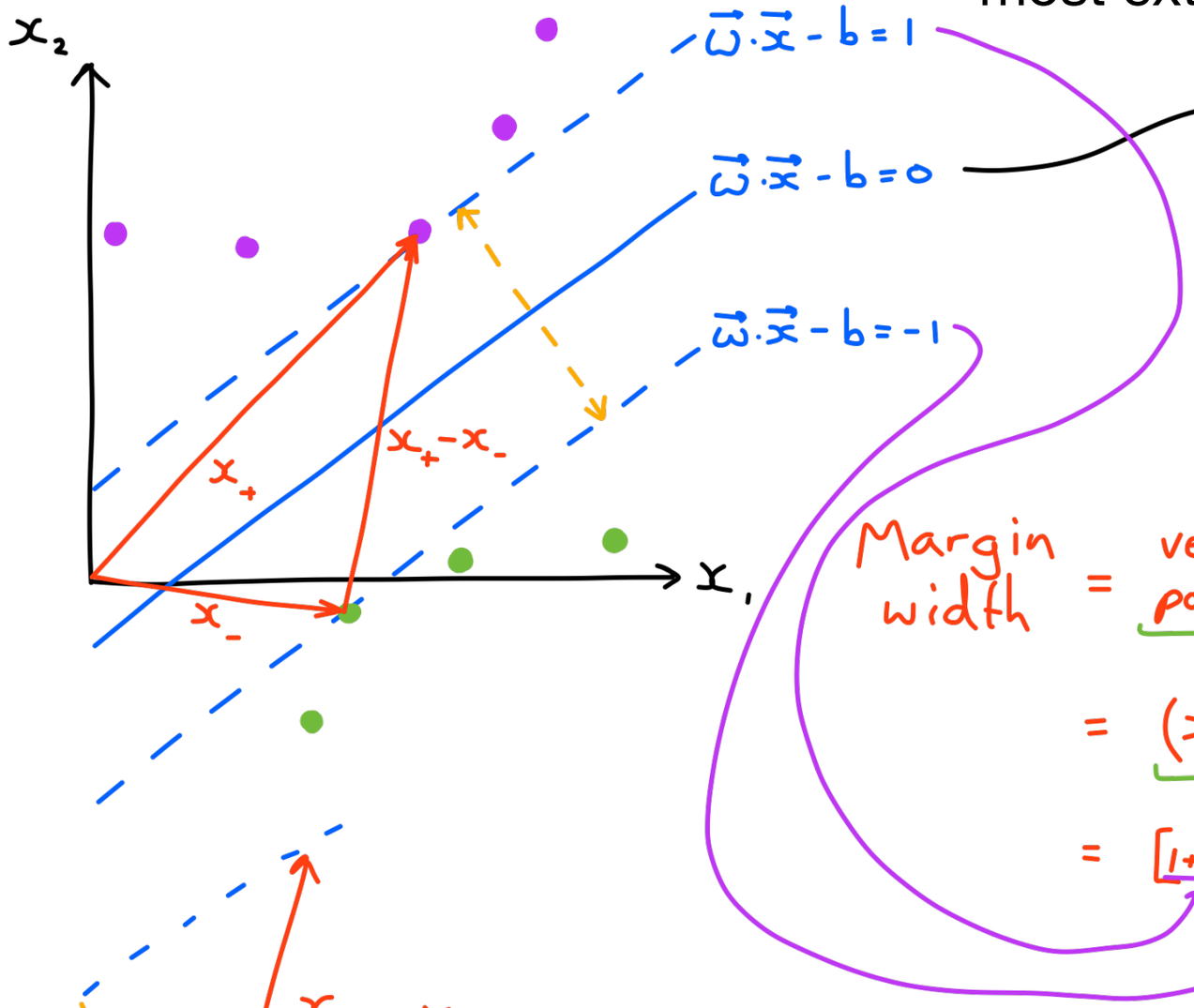
Traditional machine learning approaches

Support Vector Machines (SVMs)
Classification and regression trees (CARTs)
Gaussian Processes (GPs)

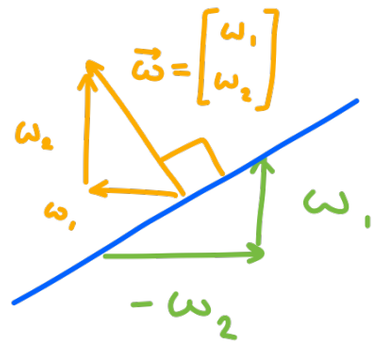
Support Vector Machines

What makes a good split?

Perhaps maximising the gap between the most extreme examples from each class



$$\omega_1 x_1 + \omega_2 x_2 - b = 0 \longrightarrow x_2 = \underbrace{-\frac{\omega_1}{\omega_2}}_{\text{slope}} x_1 + \underbrace{\frac{b}{\omega_2}}_{\text{intercept}}$$



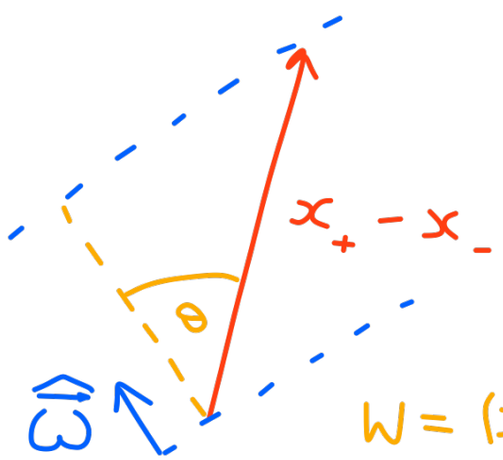
$$g=\begin{bmatrix}-\omega_2\\ \omega_1\end{bmatrix}$$

$$g'=\begin{bmatrix}\omega_1\\ \omega_2\end{bmatrix}=\vec{\omega}$$

Margin width = vector between points on margins projected across width of margin

$$= (x_+ - x_-)\cdot\frac{\vec{\omega}}{\|\vec{\omega}\|} = (\vec{x}_+\cdot\vec{\omega} - \vec{x}_-\cdot\vec{\omega})\cdot\frac{1}{\|\vec{\omega}\|}$$

$$= [1+b-(-1+b)]\cdot\frac{1}{\|\vec{\omega}\|} = \frac{2}{\|\vec{\omega}\|}$$

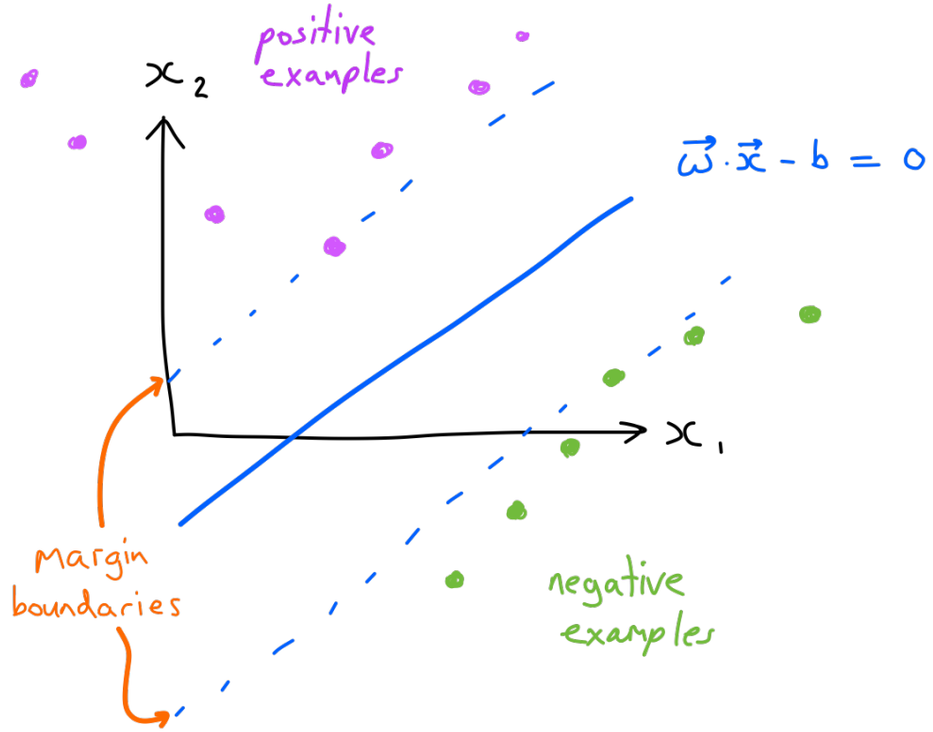


$$W = (x_+ - x_-)\cdot\hat{\omega} = |x_+ - x_-|\cos\theta$$

Maximise width of margin : maximise $\frac{2}{\|\vec{\omega}\|}$ i.e. minimise $\|\vec{\omega}\|$

subject to all datapoints being classified correctly

Hard constraint - what if this can't be done?



Hypothesis:

If $\vec{w}\cdot\vec{x}-b<0$ (below line) predict $\hat{y}=-1$ (negative class)

If $\vec{w}\cdot\vec{x}-b>0$ (above line) predict $\hat{y}=+1$ (positive class)

Why $\vec{w}\cdot\vec{x}-b=-1$ & $\vec{w}\cdot\vec{x}-b=1$?

Binary labels $y^{(i)}\in\{-1,+1\}$ Not $\{0,1\}$!

If $y^{(i)}=-1$, then $\hat{y}^{(i)}=\vec{w}\cdot\vec{x}^{(i)}-b$ should be negative

If $y^{(i)}=+1$, then $\hat{y}^{(i)}=\vec{w}\cdot\vec{x}^{(i)}-b$ should be positive

So if correctly classified $y^{(i)}$ & $\vec{w}\cdot\vec{x}^{(i)}-b$ will have the same sign

Hence if correctly classified $y^{(i)}(\vec{w}\cdot\vec{x}^{(i)}-b)\geq 1$

$\hookrightarrow 1-y^{(i)}(\vec{w}\cdot\vec{x}^{(i)}-b)<0$

control trade-off between correct classification & support vector margin width

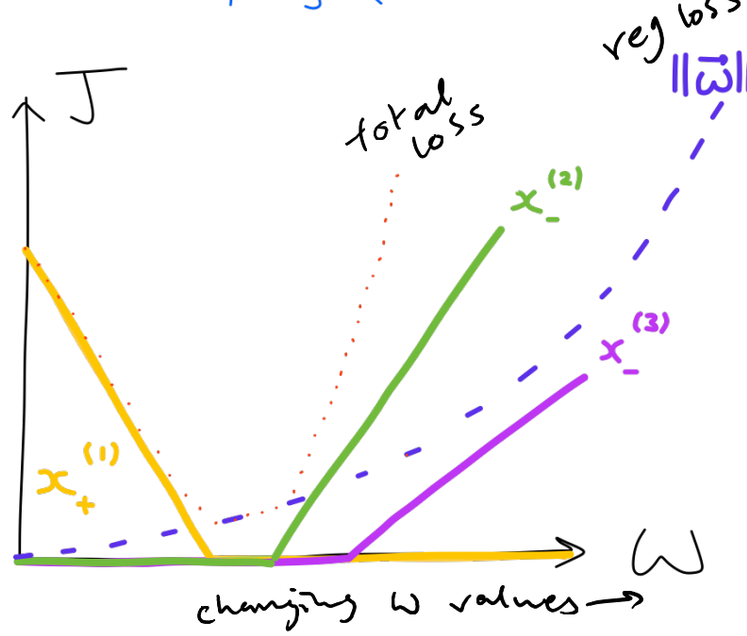
$$J(w)=\frac{1}{n}\sum_i^n\left(\max\left[0,\ 1-y^{(i)}(\vec{w}\cdot\vec{x}^{(i)}-b)\right]\right)+\lambda\|\vec{w}\|^2$$

convex

still minimising $\|\vec{w}\|$

$$\vec{w}^*=\underset{w}{\operatorname{argmin}}\ J \qquad b^*=\underset{b}{\operatorname{argmin}}\ J$$

Descend using optimiser to train SVM



~~Before the kernel trick...~~ what if data has a non-linear relationship?

It is possible to rewrite the linear decision boundary in terms of similarity (dot products) between examples.

$$\text{Hypothesis: } \vec{\omega}\cdot\vec{x} + b = b + \sum_{i=1}^{m} \alpha_i \vec{x}\cdot\vec{x}^{(i)}$$

test sample; ith training sample

$$= b + \alpha_0 \vec{x}\cdot\vec{x}^{(0)} + \alpha_1 \vec{x}\cdot\vec{x}^{(1)} + \ldots + \alpha_m \vec{x}\cdot\vec{x}^{(m)}$$

How similar is $\vec{x}$ to this datapoint $x^{(0)}$

How important is this similarity

1D proj of $\vec{x}$ onto $\vec{x}^{(0)}$

Now the parameter vector we are learning is $\vec{\alpha}$ rather than $\vec{\omega}$

The relationship between our hypothesis and alpha is linear.

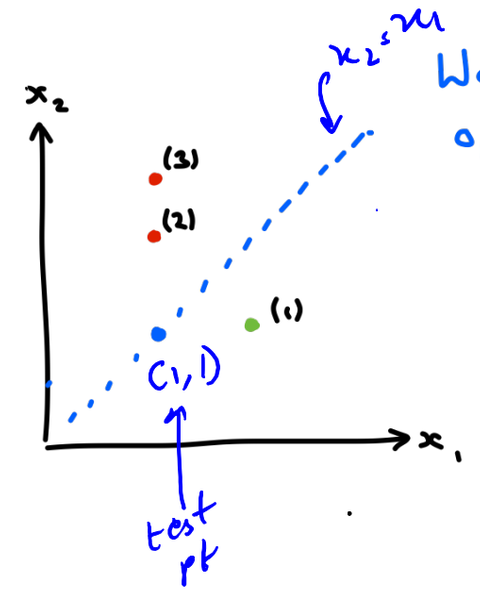
But now, because of the sum, the cost of evaluating the decision function is linear in the number of training examples. $O(n)$

But only the support vectors influence the maximum main hyperplane.

Only support vectors will end up with non-zero alpha coefficients.

Demonstration that $\vec{\omega} \cdot \vec{x} + b$ can be written as $b + \sum_{i=1}^{m} \alpha_i \vec{x} \cdot \vec{x}^{(i)}$

Index	$x_1^{(i)}$	$x_2^{(i)}$
$x^{(1)}$	2	1
$x^{(2)}$	1	2
$x^{(3)}$	1	3



We can see by eye that the optimal parameters are

$b = 0 \quad \vec{\omega} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

e.g. $\vec{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

$\vec{x} \cdot \vec{\omega} + b = 0$ ← for $\vec{x}$ on decision boundary

$1 \times \omega_1 + 1 \times \omega_2 + b = 0 \quad 1 - 1 + 0 = 0$ ✓

$$b + \sum_{i=1}^{m} \alpha_i \vec{x} \cdot \vec{x}^{(i)} = 0$$

$$0 + \alpha_1 \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \alpha_2 \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \alpha_3 \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 3 \end{bmatrix} = 0$$

Solved by $\alpha_1 = 1 \quad \alpha_2 = -1 \quad \alpha_3 = 0$

On margin $\alpha_1, \alpha_2 \neq 0$

example 1 has positive coefficient $\alpha_1 > 0$

example 2 has negative coefficient $\alpha_2 < 0$

example 3 has a zero coefficient $\alpha_3 = 0$

Not on margin → not relevant to hyperplane

The kernel trick

A SVM is only going to be useful if my data can be split well **linearly**, right?

$$b + \sum_{i=1}^{m} \alpha_i \, \phi(\vec{x}) \cdot \phi(\vec{x}^{(i)}) = b + \sum_{i=1}^{m} \alpha_i \, k(\vec{x}, \vec{x}^{(i)})$$

e.g. $\phi(x) = x^2 + 1$

$\vec{x}$ in transformed space

1D dot product in transformed space

kernel function = $\phi(\vec{x}) \cdot \phi(\vec{x}^{(i)})$

similarity between transformed feature vectors

for all examples

is this example important?

How similar is the datapoint that we want to classify in transformed space?

Instead of splitting the original, raw data linearly, we can apply a non-linear transformation to each datapoint and hope that it is linearly separable in this transformed space.

The '*trick*' is that we don't actually transform each example and then take a dot product, but directly evaluate a '*kernel*' that corresponds to doing exactly that.

A valid kernel needs to still correspond to testing the similarity between two vectors, and so must at least correspond to a dot product in *some* transformed space (check out the homework!).

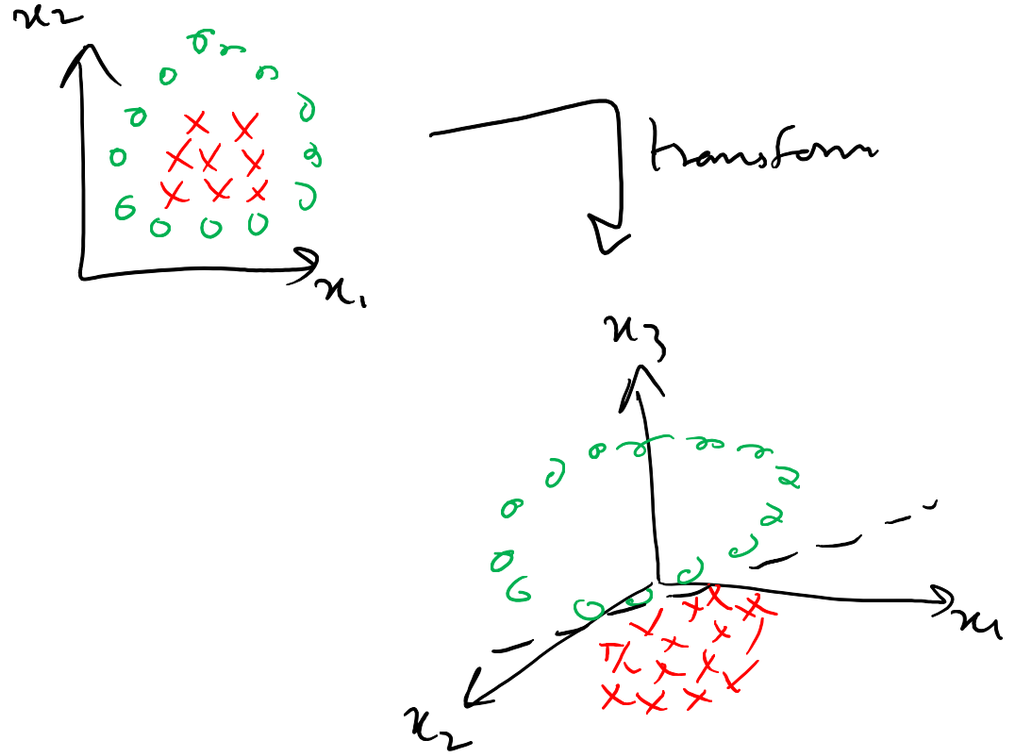
The value of using a kernel is that we can essentially evaluate the similarity between datapoints in a space where they are linearly separable, without having to perform this complex transformation on any of the datapoints - we don't event have to know what it is!

New eqn → $\vec{\alpha} \cdot \vec{k} + b = 0$ where $\vec{k} = \begin{bmatrix} k(\vec{x}, \vec{x}^{(1)}) \\ \vdots \\ k(\vec{x}, \vec{x}^{(N)}) \end{bmatrix}$

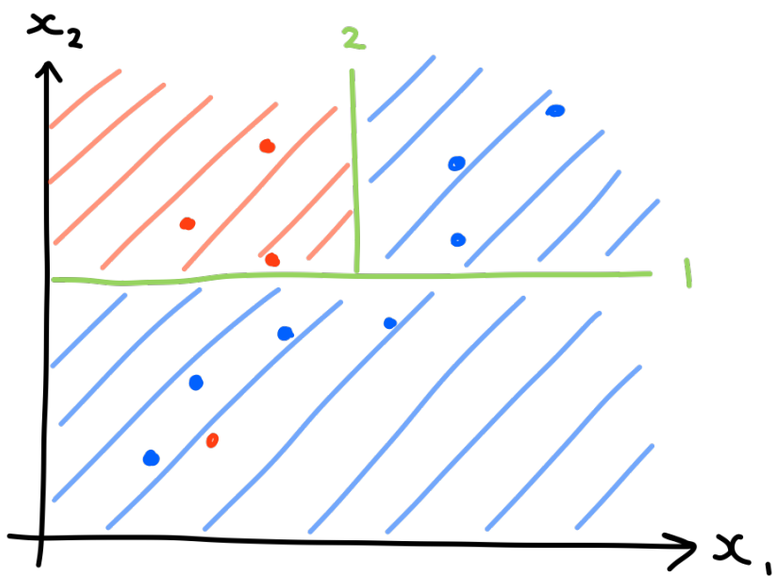
$$\text{Hypothesis: } \hat{y} = \text{sign}\left(b + \sum_{i=1}^{m} \alpha_i \, k(\vec{x}, \vec{x}^{(i)})\right)$$

$$\text{Loss: } \mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} \max\left(0, \, y^{(i)}\left[b + \sum_{i=1}^{m} \alpha_i \, k(\vec{x}, \vec{x}^{(i)})\right]\right) + \lambda \|\vec{\omega}\|^2$$

	$k(\vec{x}, \vec{x}^{(i)})$
Polynomial kernel	$(1 + x^T x^{(i)})^p$
Radial Basis Function (RBF) kernel	$e^{-\frac{(x - x^{(i)})^2}{2\sigma^2}}$



Classification trees



- Classification trees split up the feature space into regions.
- Each region corresponds to a certain class - that of the majority of the datapoints within it.
- When we want to classify a new example, we see what class the region which it falls into is.

How do we know where to split the data?

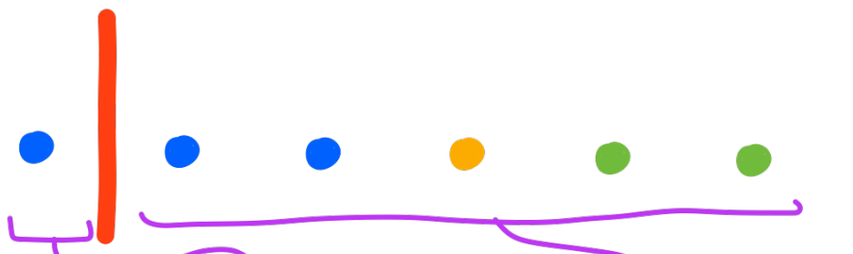
What makes a good split?

When we make an ideal split, the each region will only contain a single class - it will be ***pure***.

$$\text{Gini Impurity, } I_G = \sum_{i=1}^{k} p_i(1-p_i) = \sum_{i=1}^{k} (p_i - p_i^2) = \sum_{i=1}^{k} p_i - \sum_{i=1}^{k} p_i^2 = 1 - \sum_{i=1}^{k} p_i^2$$

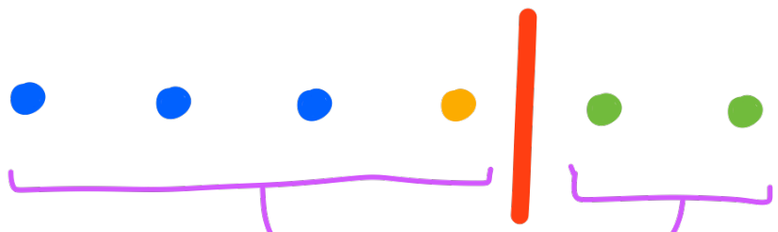
p_i = fraction of examples in each region with class i

Bad split $p_1 = \frac{1}{2}$ $p_0 = \frac{1}{6}$ $p_3 = \frac{1}{3}$



$$\sum I_G = \left[1 - \left(\frac{1}{1}\right)^2\right] + \left[1 - \left(\frac{2}{5}\right)^2 - \left(\frac{1}{5}\right)^2 - \left(\frac{2}{5}\right)^2\right] = 0.64$$

Good split



$$\sum I_G = \left[1 - \left(\frac{3}{4}\right)^2 - \left(\frac{1}{4}\right)^2\right] + \left[1 - \left(\frac{2}{2}\right)^2\right] = 0.375$$

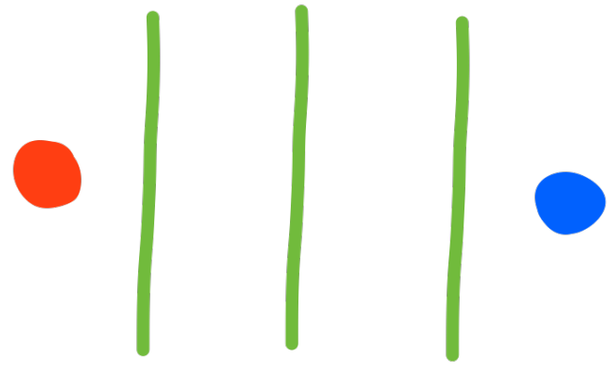
How do we know where to split the data?

Where should we make this split?

Along any continuous axis, there are an infinite many number of values at which we could split the data.

We want the one value that results in the minimum impurity.

But if we split the data at any value anywhere between two datapoints, the created regions will contain the same examples and hence will have the same impurity. So having no knowledge about what position between two subsequent points would be best for the split, we just take the midpoint between them.



All of these splits result in the same impurity, so we just choose the midpoint between points

We need to search for the next best place to split the data across all midpoints between examples, across all axes over all regions.

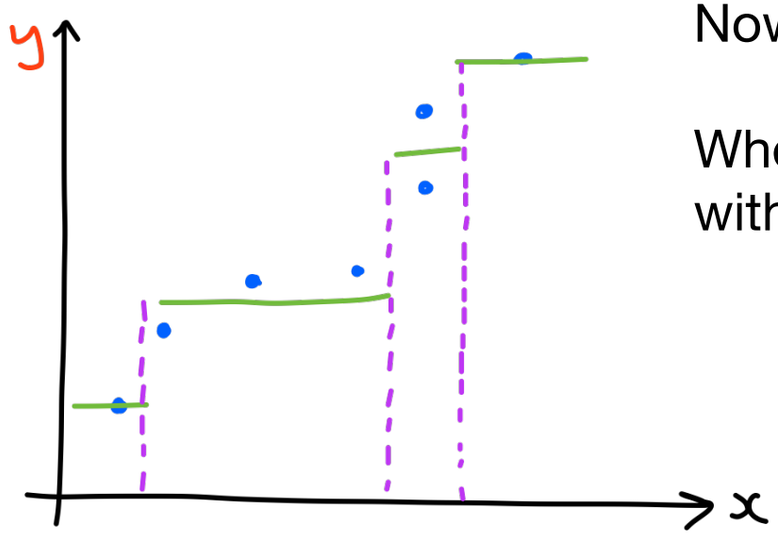
$$x_{split} = \min_{j} \operatorname{argmin}_{i} I_G'(\text{split at midpoint of } x_i + x_{i+1})$$

min over each axis

min over each possible split at example midpoints

```
Whilst stopping criteria not met:
    Initialise empty list to store best splits in each region
    For each region:
        Initialise empty list to store best place to split each axis
        For each feature axis:
            Initialise empty list to store impurities of all possible splits
            Order examples in order of value of feature on this axis
            Compute midpoints between each subsequent examples
            For each midpoint:
                Compute total Geni impurity of splitting here
                Store (axis, midpoint, impurity) tuple in list
```

Regression trees



different constant linear models for each region

Now we are trying to predict a continuous value, rather than a classification.

When we make a split, we model our predictions for each region created with a constant value that minimises the square error in that region.

$$\hat{y}_R^* = \underbrace{\underset{\hat{y}}{\operatorname{argmin}} \sum_{y^{(i)} \in R} (\hat{y} - y^{(i)})^2}_{\text{minimise square error}} = \underbrace{\bar{y} \text{ for } y \in R}_{\text{minimised by mean of labels in this region}}$$

- $\hat{y}_R^*$: optimal (*) prediction ($\hat{}$) in region (R)
- $y^{(i)} \in R$: only for each example in this region R

```
Whilst stopping criteria not met:
    Initialise empty list to store best splits in each region
    For each region:
        Initialise empty list to store best place to split each axis
        For each feature axis:
            Initialise empty list to store impurities of all possible splits
            Order examples in order of value of feature on this axis
            Compute midpoints between each subsequent examples
            For each midpoint:
                Compute total squared error after splitting here
                Store (axis, midpoint, impurity) tuple in list
```

Overview of traditional approaches covered

	SVMs	CARTs	Linear regression
Transform the data			

But what is it these techniques all lack?

All of the techniques above try to split the data directly, or after only a single transformation.

What if we could **repeatedly** stretch and shift the data, many times, to perform complex transformations on the data before trying to linearly split or regress it?

What if we could learn to make these complex transformations **meaningful**, rather than guessing and specifying them beforehand?