Machine Learning for Astrophysics

Supervised Learning strikes back

Outline

Introduction to ML & Supervised ML

- Introduction
- Regression, Regularization
- Classification, Logistic Regression
- Bias/Variance trade-off

Supervised ML strikes back

- Support Vector Machines
- Gaussian Processes
- Nearest Neighbors
- Ensemble Methods: random forests
- Gradient Boosting

You are here

Unsupervised ML

- Clustering: KMeans, DBScan, GMM, Agglomerative Clustering
- Anomaly Detection
- Dimensionality Reduction:
  - linear: PCA, NMF, ICA
  - manifold learning: LLE, IsoMap, t-SNE
- Self-Organizing Maps

Deep Learning

- A few notes on HPC and GPUs (*maybe*)
- Basics of NN: computation graphs
- Training a NN: *forth*- and *back*-propagation
- Optimization Algorithms
- Transfer Learning
- Autoencoders?
- Bayesian NN, Probabilistic BNN

Deep Learning, The Revenge

- (*hints on*) Reinforcement Learning
- Convolutional Neural Networks
- ResNet, Inception Module and MobileNet
- Generative Adversarial Networks
- (*hints on*) Recurrent Neural Networks
- Variational Autoencoders
- Transformers

The Holy Trinity of Machine Learning

Supervised Learning

Data: (x, y)
x is the data, with associated labels y

Goal:
learn a function that maps
$x \rightarrow y$

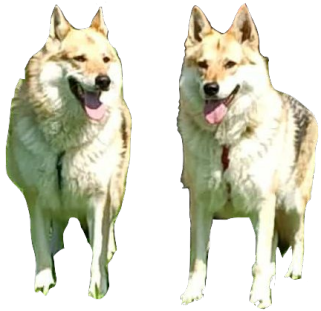
"This thing is a dog"

Unsupervised Learning

Data: x
there are no labels, only data x

Goal:

learn underlying structure of x



"These two things look alike"

Reinforcement Learning

Data: *state-action* pairs

Goal:
learn a policy π
maximizing future rewards

"Cuddling this thing will make you happy"

Supervised Learning

Data: (x, y)
x is the data, with associated labels y

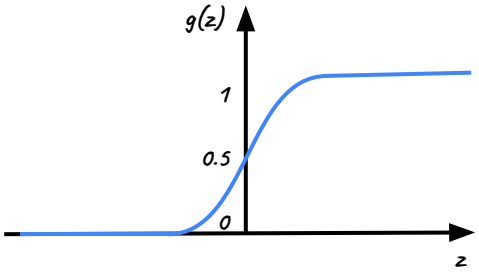
Goal:
learn a function that maps
$x \rightarrow y$

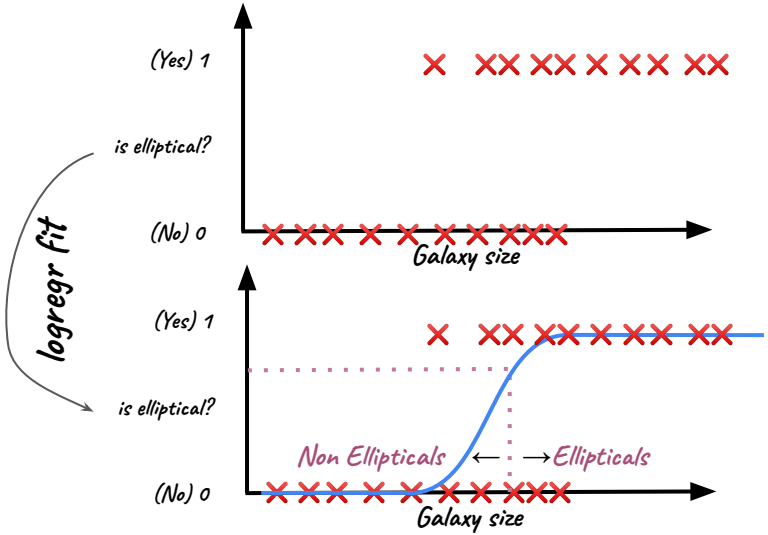
"This thing is a dog"

Build **models** that learn
how to ***combine*** **inputs**
to ***produce*** **predictions**
(even) on **never seen before data**

Logistic regression is the centerpiece of classification problems. It originates from the sigmoid (or logistic) function:

$$\boxed{g(z) = \frac{1}{1+e^{-z}}}$$





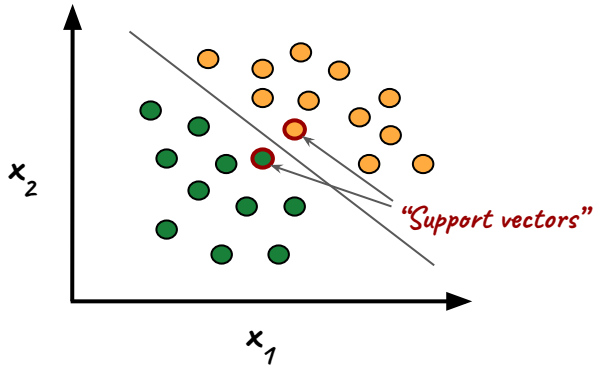
And with this function, define the logistic regression model:

$$f_{\vec{w},b} = g(\vec{w} \cdot \vec{x} + b) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$$

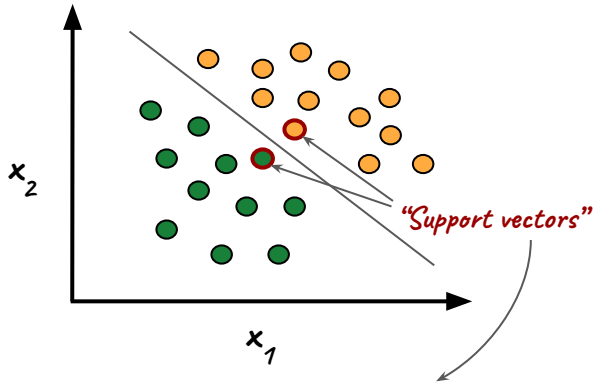
where $f_{w,b}$ gives the probability that x belongs to a certain class (e.g. if the galaxy is elliptical once given a galaxy size in kpc).
With logistic regression you train on the training set, finding the optimal values for the weights w, b.
The logistic loss function (*logloss*) evaluated on a single training example is:

$$\mathcal{L}(f_{\vec{w},b}(x^{(i)}, y^{(i)})) = -y^{(i)} \log(f_{\vec{w},b}(x^{(i)})) - (1 - y^{(i)}) \log(1 - f_{\vec{w},b}(x^{(i)}))$$

A **Support Vector Machine** (SVM) is a supervised learning algorithm, mainly employed for classification tasks. SVMs are based on the idea of finding a hyperplane that best (and maximally) divides a dataset into two classes.

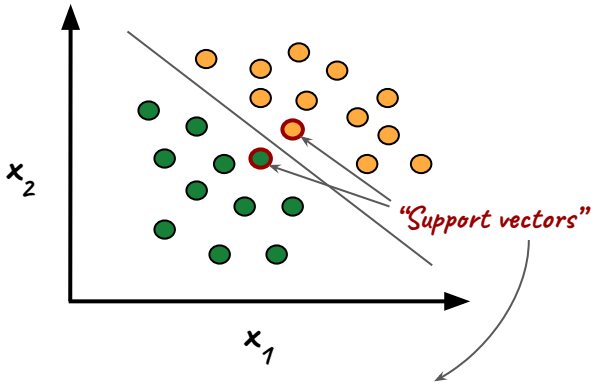


A **Support Vector Machine** (SVM) is a supervised learning algorithm, mainly employed for classification tasks. SVMs are based on the idea of finding a hyperplane that best (and maximally) divides a dataset into two classes.



The data points nearest to the hyperplane. Their removal would alter the position of the dividing hyperplane.

A **Support Vector Machine** (SVM) is a supervised learning algorithm, mainly employed for classification tasks. SVMs are based on the idea of finding a hyperplane that best (and maximally) divides a dataset into two classes.

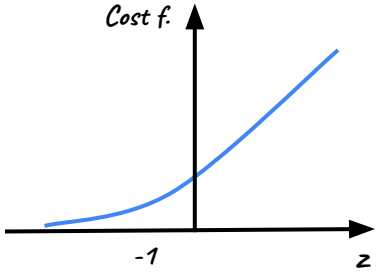


The data points nearest to the hyperplane. Their removal would alter the position of the dividing hyperplane.

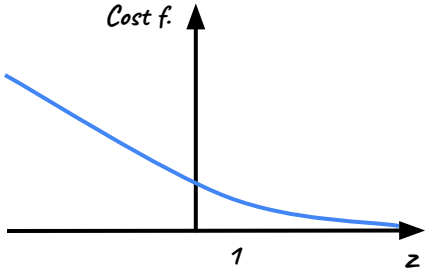
For logistic regression, the loss function for a given example $(x^{(i)}, y^{(i)})$ is:

$$\mathcal{L}(g(z), y^{(i)})) = -y^{(i)} \log(g(z)) - (1 - y^{(i)}) \log(1 - g(z))$$

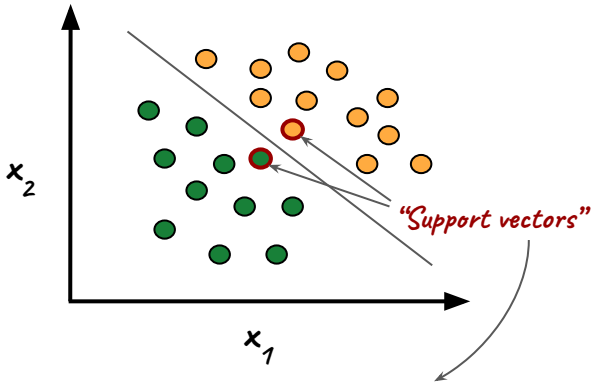
if $y = 0 \longrightarrow -\log(1-g(z))$



if $y = 1 \longrightarrow -\log(g(z))$



A **Support Vector Machine** (SVM) is a supervised learning algorithm, mainly employed for classification tasks. SVMs are based on the idea of finding a hyperplane that best (and maximally) divides a dataset into two classes.

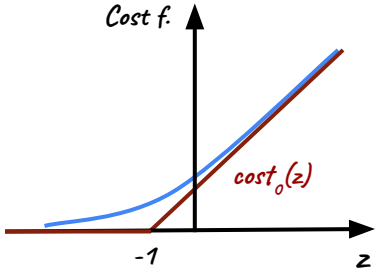


The data points nearest to the hyperplane. Their removal would alter the position of the dividing hyperplane.

For logistic regression, the loss function for a given example $(x^{(i)}, y^{(i)})$ is:

$$\mathcal{L}(g(z), y^{(i)})) = -y^{(i)} \log(g(z)) - (1 - y^{(i)}) \log(1 - g(z))$$

if $y = 0 \longrightarrow -\log(1-g(z))$

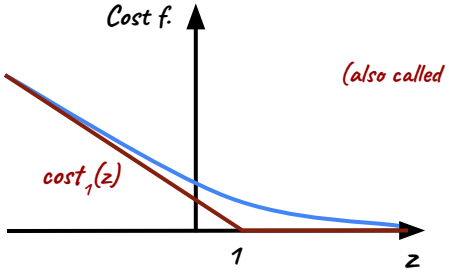


For a SVM, let's modify the cost function:

if $y = 1$, $cost_1(z)$ is $\longrightarrow$
0 for $z > 1$
linear for $z < 1$

$\longleftarrow$ if $y = 0$, $cost_0(z)$ is
0 for $z < -1$
linear for $z > -1$

if $y = 1 \longrightarrow -\log(g(z))$



(also called "hinge loss")

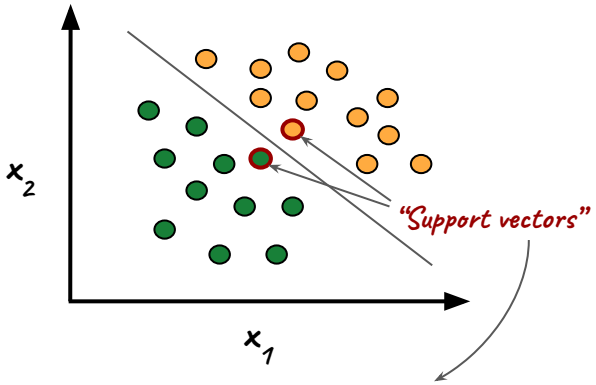
The SVM cost function therefore becomes:

$$\left.\begin{aligned} J(z) &= \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\mathrm{cost}_1(z^{(i)}) + (1-y^{(i)})\mathrm{cost}_0(z^{(i)})\right) + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2 \\ &= C\sum_{i=1}^{m}\left(y^{(i)}\mathrm{cost}_1(z^{(i)}) + (1-y^{(i)})\mathrm{cost}_0(z^{(i)})\right) + \frac{1}{2}\sum_{j=1}^{n} w_j^2 \end{aligned}\right\}$$

for a positive example $(y^{(i)} = 1) \longrightarrow z = w \cdot x \geq 1$

for a negative example $(y^{(i)} = 0) \longrightarrow z = w \cdot x \leq -1$

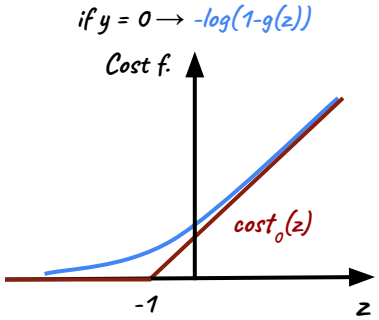
A **Support Vector Machine** (SVM) is a supervised learning algorithm, mainly employed for classification tasks. SVMs are based on the idea of finding a hyperplane that best (and maximally) divides a dataset into two classes.



The data points nearest to the hyperplane. Their removal would alter the position of the dividing hyperplane.

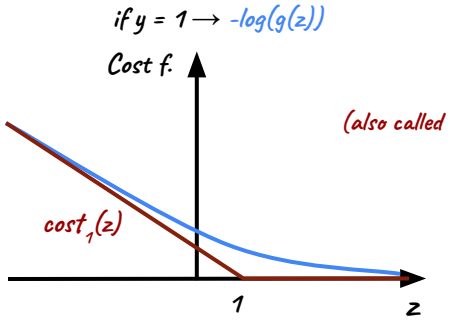
For logistic regression, the loss function for a given example $(x^{(i)}, y^{(i)})$ is:

$$\mathcal{L}(g(z), y^{(i)})) = -y^{(i)} \log(g(z)) - (1 - y^{(i)}) \log(1 - g(z))$$



For a SVM, let's modify the cost function:
if y = 1, cost$_1$(z) is →
0 for z > 1
linear for z < 1

← if y = 0, cost$_0$(z) is
0 for z < -1
linear for z > -1



(also called "hinge loss")

I'll spare you the gory mathematical details, but that forces the decision boundary to be the one with the largest margin from the training example

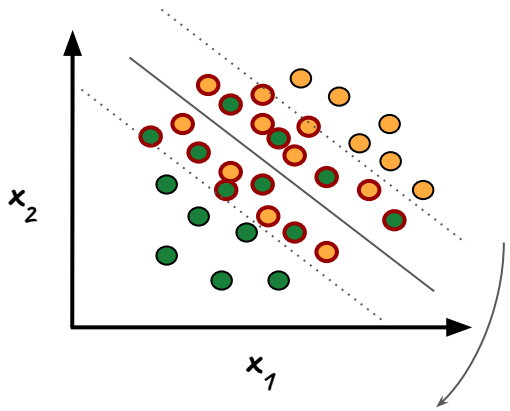
The SVM cost function therefore becomes:

$$\left.\begin{aligned} J(z) &= \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\text{cost}_1(z^{(i)}) + (1-y^{(i)})\text{cost}_0(z^{(i)})\right) + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2 \\ &= C\sum_{i=1}^{m}\left(y^{(i)}\text{cost}_1(z^{(i)}) + (1-y^{(i)})\text{cost}_0(z^{(i)})\right) + \frac{1}{2}\sum_{j=1}^{n} w_j^2 \end{aligned}\right\}$$

for a positive example $(y^{(i)} = 1) \longrightarrow z = w \cdot x \geq 1$

for a negative example $(y^{(i)} = 0) \longrightarrow z = w \cdot x \leq -1$

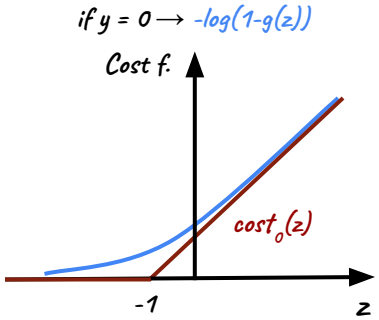
A **Support Vector Machine** (SVM) is a supervised learning algorithm, mainly employed for classification tasks. SVMs are based on the idea of finding a hyperplane that best (and maximally) divides a dataset into two classes.



You can account for the presence of interlopers by adding a tolerance margin (soft margins)

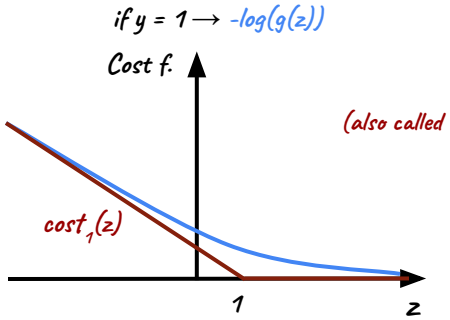
For logistic regression, the loss function for a given example $(x^{(i)}, y^{(i)})$ is:

$$\mathcal{L}(g(z), y^{(i)})) = -y^{(i)} \log(g(z)) - (1 - y^{(i)}) \log(1 - g(z))$$



For a SVM, let's modify the cost function:
if y = 1, cost$_1$(z) is →
0 for z > 1
linear for z < 1

← if y = 0, cost$_0$(z) is
0 for z < -1
linear for z > -1



(also called "hinge loss")

I'll spare you the gory mathematical details, but that forces the decision boundary to be the one with the largest margin from the training example

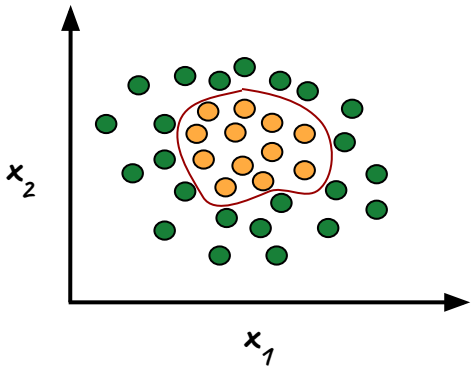
The SVM cost function therefore becomes:

$$\left.\begin{aligned} J(z) &= \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\text{cost}_1(z^{(i)}) + (1-y^{(i)})\text{cost}_0(z^{(i)})\right) + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2 \\ &= C\sum_{i=1}^{m}\left(y^{(i)}\text{cost}_1(z^{(i)}) + (1-y^{(i)})\text{cost}_0(z^{(i)})\right) + \frac{1}{2}\sum_{j=1}^{n} w_j^2 \end{aligned}\right\}$$

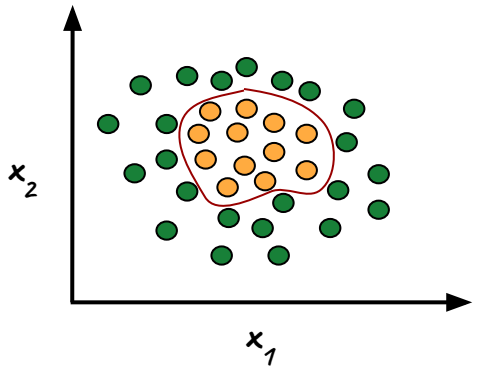
for a positive example $(y^{(i)} = 1) \longrightarrow z = w \cdot x \geq 1$

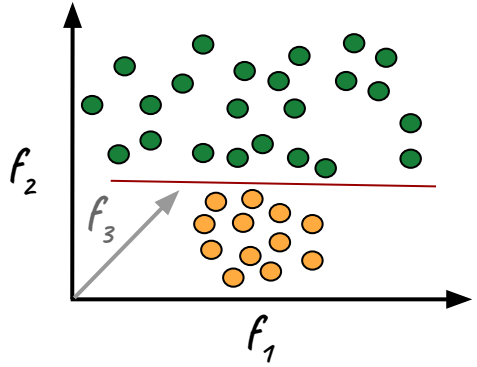
for a negative example $(y^{(i)} = 0) \longrightarrow z = w \cdot x \leq -1$

That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.

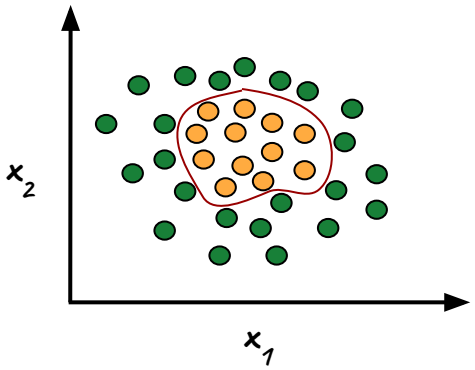


That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.





That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.

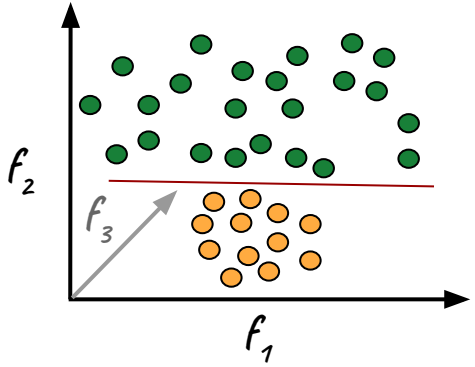


Let's say that our model is:

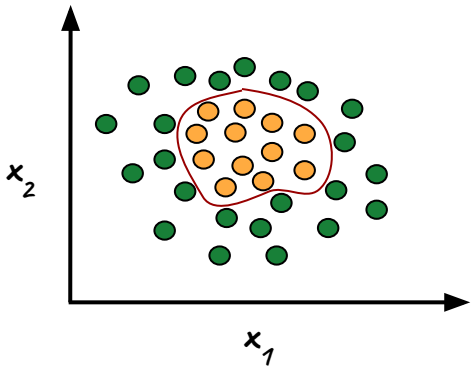
$$w_0 + w_1 \underbrace{x_1^2}_{f_1} + w_2 \underbrace{x_2^2}_{f_2} + w_3 \underbrace{x_1 x_2}_{f_3}$$

$\geq 0 \longrightarrow$ orange dot $\quad y^{(i)} = 1$

$< 0 \longrightarrow$ green dot $\quad y^{(i)} = 0$



That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.



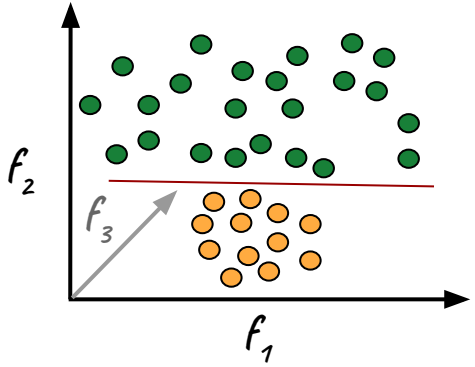
Let's say that our model is:

$$w_0 + w_1 f_1 + w_2 f_2 + w_3 f_3$$

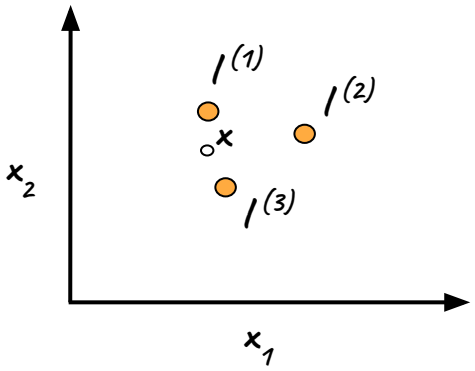
$\geq 0 \longrightarrow$ orange dot $\quad y^{(i)} = 1$

$< 0 \longrightarrow$ green dot $\quad y^{(i)} = 0$

the question is: can we automatically calculate the best choice of features f_i – given certain assumptions – that maximally separates the two classes?



That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.



Let's say that our model is:

$$w_0 + w_1 f_1 + w_2 f_2 + w_3 f_3$$

$\geq 0 \longrightarrow$ orange dot $\quad y^{(i)} = 1$

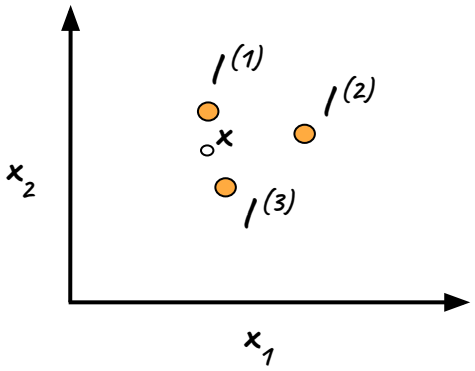
$< 0 \longrightarrow$ green dot $\quad y^{(i)} = 0$

the question is: can we automatically calculate the best choice of features f_i – given certain assumptions – that maximally separates the two classes?

Given x, compute new features depending on the proximity to landmarks points $l^{(1)}, l^{(2)}, l^{(3)}$

$$\left.\begin{aligned} f_1 &= \text{similarity}(x, l^{(1)}) \\ f_2 &= \text{similarity}(x, l^{(2)}) \\ f_3 &= \text{similarity}(x, l^{(3)}) \end{aligned}\right\} \text{similarity} = \text{KERNEL FUNCTION } K(x, l^{(i)})$$

That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.



Let's say that our model is:

$$w_0 + w_1 f_1 + w_2 f_2 + w_3 f_3$$

$\geq 0 \longrightarrow$ orange dot $\quad y^{(i)} = 1$

$< 0 \longrightarrow$ green dot $\quad y^{(i)} = 0$

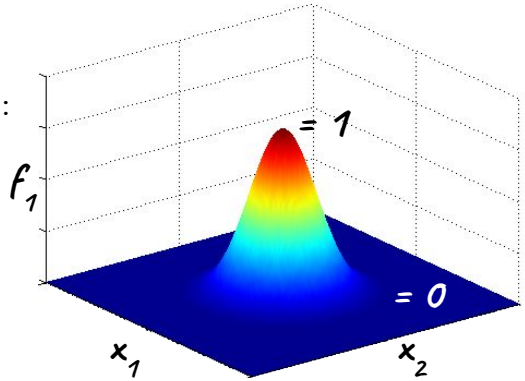
the question is: can we automatically calculate the best choice of features f_i – given certain assumptions – that maximally separates the two classes?

Given x, compute new features depending on the proximity to landmarks points $l^{(1)}, l^{(2)}, l^{(3)}$

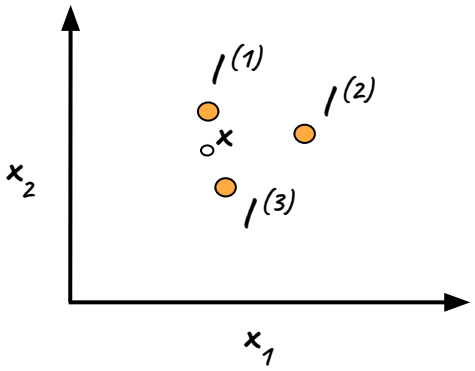
$$\left.\begin{aligned} f_1 &= \text{similarity}(x, l^{(1)}) \\ f_2 &= \text{similarity}(x, l^{(2)}) \\ f_3 &= \text{similarity}(x, l^{(3)}) \end{aligned}\right\} \text{similarity} = \text{KERNEL FUNCTION } K(x, l^{(i)})$$

e.g. Gaussian kernel (or **radial basis function, RBF**):

$$K(x, l^{(i)}) = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$$

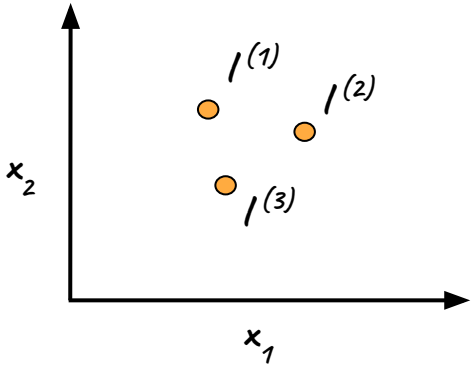


That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.



Let's assume e.g. a model with:

$w_0 = -.5,\ w_1 = +1,\ w_2 = +1\ w_3 = 0$



Let's say that our model is:

$$w_0 + w_1 f_1 + w_2 f_2 + w_3 f_3$$

$\geq 0 \longrightarrow$ orange dot $\quad y^{(i)} = 1$

$< 0 \longrightarrow$ green dot $\quad y^{(i)} = 0$

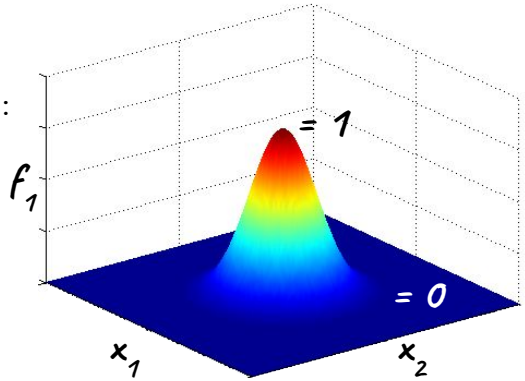
the question is: can we automatically calculate the best choice of features f_i – given certain assumptions – that maximally separates the two classes?

Given x, compute new features depending on the proximity to landmarks points $l^{(1)}, l^{(2)}, l^{(3)}$

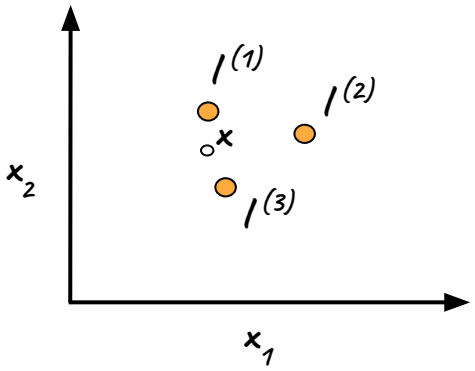
$$\left.\begin{aligned} f_1 &= \text{similarity}(x, l^{(1)}) \\ f_2 &= \text{similarity}(x, l^{(2)}) \\ f_3 &= \text{similarity}(x, l^{(3)}) \end{aligned}\right\} \text{similarity} = \text{KERNEL FUNCTION } K(x, l^{(i)})$$

e.g. Gaussian kernel (or **radial basis function, RBF**):

$$K(x, l^{(i)}) = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$$

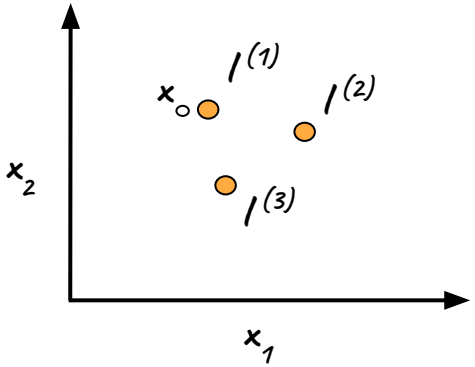


That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.



Let's assume e.g. a model with:

$w_0 = -.5,\ w_1 = +1,\ w_2 = +1\ w_3 = 0$



Let's say that our model is:

$$w_0 + w_1 f_1 + w_2 f_2 + w_3 f_3$$

$\geq 0 \longrightarrow$ orange dot $\quad y^{(i)} = 1$

$< 0 \longrightarrow$ green dot $\quad y^{(i)} = 0$

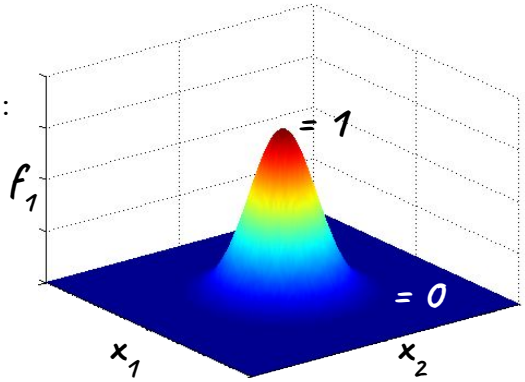
the question is: can we automatically calculate the best choice of features f_i – given certain assumptions – that maximally separates the two classes?

Given x, compute new features depending on the proximity to landmarks points $l^{(1)}, l^{(2)}, l^{(3)}$

$$\left.\begin{aligned} f_1 &= \text{similarity}(x, l^{(1)}) \\ f_2 &= \text{similarity}(x, l^{(2)}) \\ f_3 &= \text{similarity}(x, l^{(3)}) \end{aligned}\right\} \text{similarity} = \text{KERNEL FUNCTION } K(x, l^{(i)})$$

e.g. Gaussian kernel (or **radial basis function, RBF**):

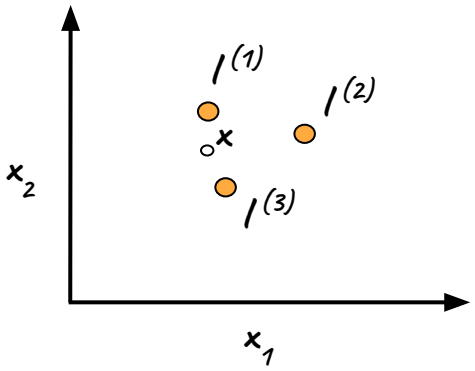
$$K(x, l^{(i)}) = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$$



if x is close to $l^{(1)} \longrightarrow f_1 = 1,\ f_2 = 0,\ f_3 = 0$

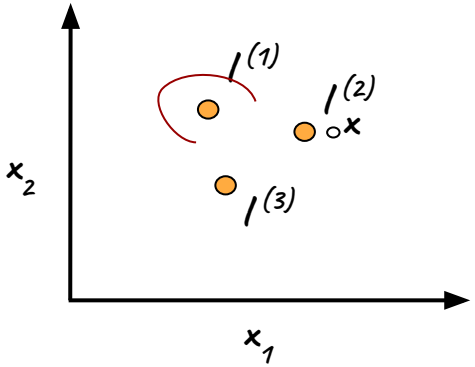
$\longrightarrow -0.5 + 1 = +0.5 \geq 0 \longrightarrow$ we predict $y^{(i)} = 1$

That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.



Let's assume e.g. a model with:

$w_0 = -.5,\ w_1 = +1,\ w_2 = +1\ w_3 = 0$



Let's say that our model is:

$$w_0 + w_1 f_1 + w_2 f_2 + w_3 f_3$$

$\geq 0 \longrightarrow$ orange dot $\quad y^{(i)} = 1$

$< 0 \longrightarrow$ green dot $\quad y^{(i)} = 0$

the question is: can we automatically calculate the best choice of features f_i – given certain assumptions – that maximally separates the two classes?

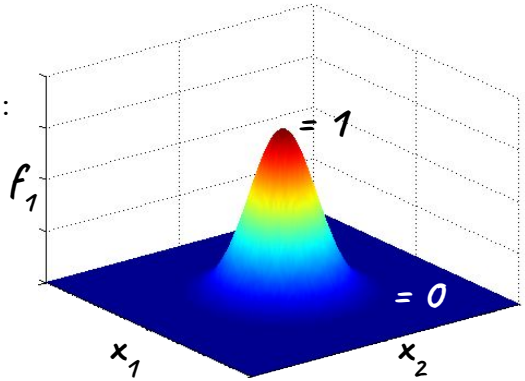
Given x, compute new features depending on the proximity to landmarks points $l^{(1)}, l^{(2)}, l^{(3)}$

$$\left.\begin{aligned} f_1 &= \text{similarity}(x, l^{(1)}) \\ f_2 &= \text{similarity}(x, l^{(2)}) \\ f_3 &= \text{similarity}(x, l^{(3)}) \end{aligned}\right\}$$

similarity = KERNEL FUNCTION $K(x, l^{(i)})$

e.g. Gaussian kernel (or **radial basis function, RBF**):

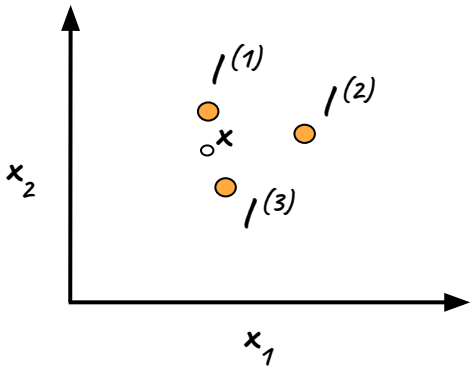
$$K(x, l^{(i)}) = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$$



if x is close to $l^{(2)} \longrightarrow f_1 = 0,\ f_2 = 1,\ f_3 = 0$

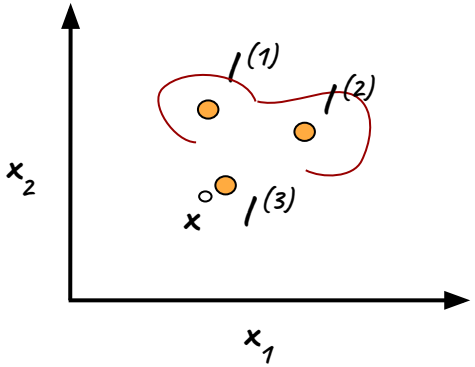
$\longrightarrow -0.5 + 1 = +0.5 \geq 0 \longrightarrow$ we predict $y^{(i)} = 1$

That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.



Let's assume e.g. a model with:

$w_0 = -.5,\ w_1 = +1,\ w_2 = +1\ w_3 = 0$



Let's say that our model is:

$$w_0 + w_1 f_1 + w_2 f_2 + w_3 f_3$$

$\geq 0 \longrightarrow$ orange dot $\quad y^{(i)} = 1$

$< 0 \longrightarrow$ green dot $\quad y^{(i)} = 0$

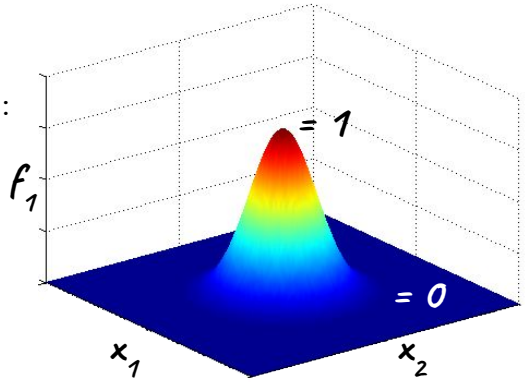
the question is: can we automatically calculate the best choice of features f_i – given certain assumptions – that maximally separates the two classes?

Given x, compute new features depending on the proximity to landmarks points $l^{(1)}, l^{(2)}, l^{(3)}$

$$\left.\begin{aligned} f_1 &= \text{similarity}(x, l^{(1)}) \\ f_2 &= \text{similarity}(x, l^{(2)}) \\ f_3 &= \text{similarity}(x, l^{(3)}) \end{aligned}\right\} \text{similarity} = \text{KERNEL FUNCTION } K(x, l^{(i)})$$

e.g. Gaussian kernel (or **radial basis function, RBF**):

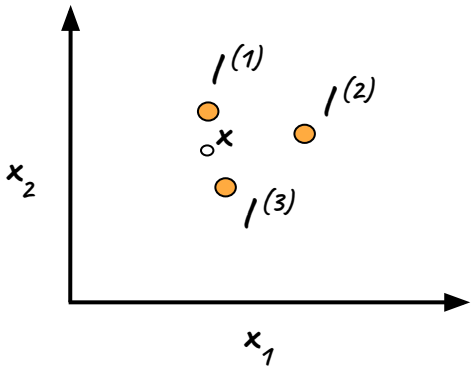
$$K(x, l^{(i)}) = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$$



if x is close to $l^{(3)} \longrightarrow f_1 = 0,\ f_2 = 0,\ f_3 = 1$

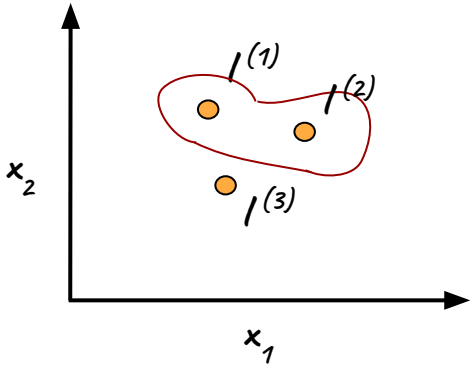
$\longrightarrow -0.5 + 0 = -0.5 < 0 \longrightarrow$ we predict $y^{(i)} = 0$

That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.



Let's assume e.g. a model with:

$w_0 = -.5,\ w_1 = +1,\ w_2 = +1\ w_3 = 0$



Let's say that our model is:

$$w_0 + w_1 f_1 + w_2 f_2 + w_3 f_3$$

$\geq 0 \longrightarrow$ orange dot $\quad y^{(i)} = 1$

$< 0 \longrightarrow$ green dot $\quad y^{(i)} = 0$

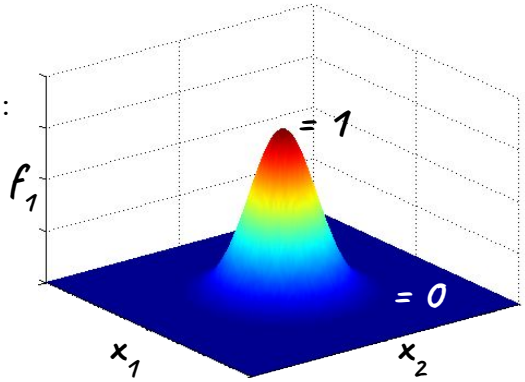
the question is: can we automatically calculate the best choice of features f_i – given certain assumptions – that maximally separates the two classes?

Given x, compute new features depending on the proximity to landmarks points $l^{(1)}, l^{(2)}, l^{(3)}$

$$\left.\begin{aligned} f_1 &= \text{similarity}(x, l^{(1)}) \\ f_2 &= \text{similarity}(x, l^{(2)}) \\ f_3 &= \text{similarity}(x, l^{(3)}) \end{aligned}\right\} \text{similarity} = \text{KERNEL FUNCTION } K(x, l^{(i)})$$

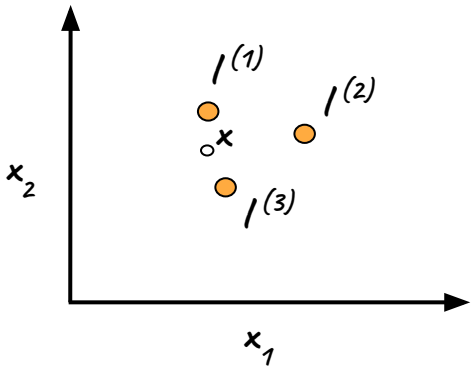
e.g. Gaussian kernel (or **radial basis function, RBF**):

$$K(x, l^{(i)}) = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$$



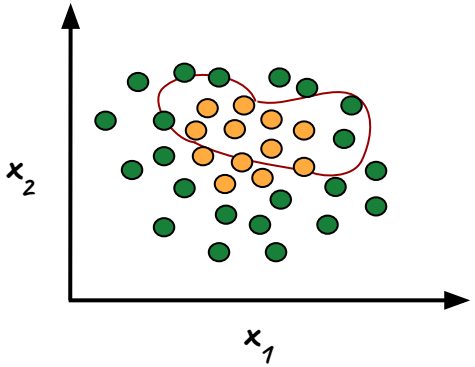
As such, we were able to identify a complex decision boundary for **that** particular model.

That was a linear SVM. How to adapt SVM to learn non-linear structures? Enters the *kernel trick*.



Let's assume e.g. a model with:

$w_0 = -.5,\ w_1 = +1,\ w_2 = +1\ w_3 = 0$



Let's say that our model is:

$$w_0 + w_1 f_1 + w_2 f_2 + w_3 f_3$$

$\geq 0 \longrightarrow$ orange dot $\quad y^{(i)} = 1$

$< 0 \longrightarrow$ green dot $\quad y^{(i)} = 0$

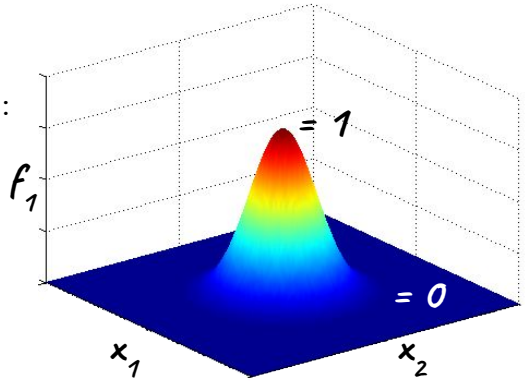
the question is: can we automatically calculate the best choice of features f_i – given certain assumptions – that maximally separates the two classes?

Given x, compute new features depending on the proximity to landmarks points $l^{(1)}, l^{(2)}, l^{(3)}$

$$\left.\begin{aligned} f_1 &= \text{similarity}(x, l^{(1)}) \\ f_2 &= \text{similarity}(x, l^{(2)}) \\ f_3 &= \text{similarity}(x, l^{(3)}) \end{aligned}\right\} \text{similarity} = \text{KERNEL FUNCTION } K(x, l^{(i)})$$

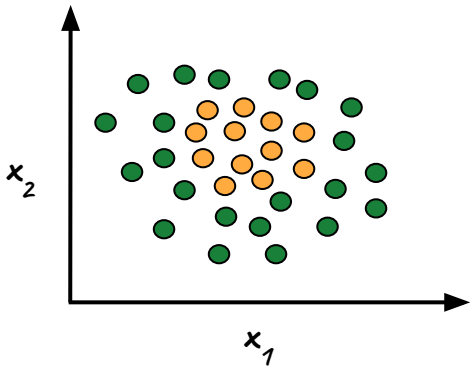
e.g. Gaussian kernel (or **radial basis function, RBF**):

$$K(x, l^{(i)}) = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$$



As such, we were able to identify a complex decision boundary for **that** particular model. The weights w_i should therefore be optimized.

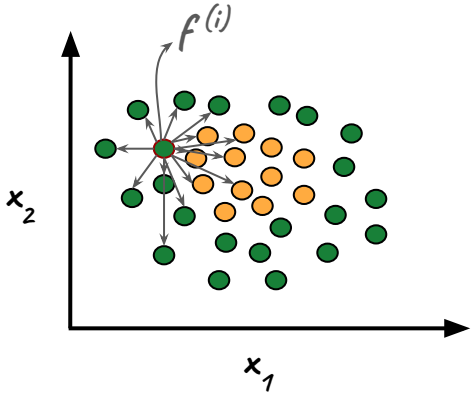
How do we choose the landmarks $l^{(i)}$? And what other kernels are available?



Given a training set of m examples $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$, we choose the landmarks as $l^{(i)} = x^{(i)}$.

$\rightarrow$ *landmarks = the whole training set*

How do we choose the landmarks $l^{(i)}$? And what other kernels are available?



Given a training set of m examples $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$, we choose the landmarks as $l^{(i)} = x^{(i)}$.

$\rightarrow$ landmarks = the whole training set

Basically for every example $x^{(i)}$ we measure the similarity (kernel) with all the other points in the training set, and place the information on a vector called the *feature vector:*

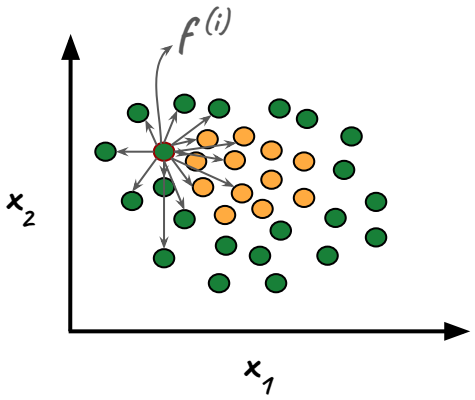
$$f^{(i)} = [f_0^{(i)}, f_1^{(i)} \dots, f_i^{(i)}, \dots, f_m^{(i)}]$$

(1 by convention)

$f_i^{(i)} = K(x^{(i)}, l^{(i)}) = 1$

$f_1^{(i)} = K(x^{(i)}, l^{(1)})$

How do we choose the landmarks $l^{(i)}$? And what other kernels are available?



Given a training set of m examples $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$, we choose the landmarks as $l^{(i)} = x^{(i)}$.

$\rightarrow$ landmarks = the whole training set

Basically for every example $x^{(i)}$ we measure the similarity (kernel) with all the other points in the training set, and place the information on a vector called the *feature vector:*

$$f^{(i)} = [f_0^{(i)}, f_1^{(i)} \dots, f_i^{(i)}, \dots, f_m^{(i)}]$$

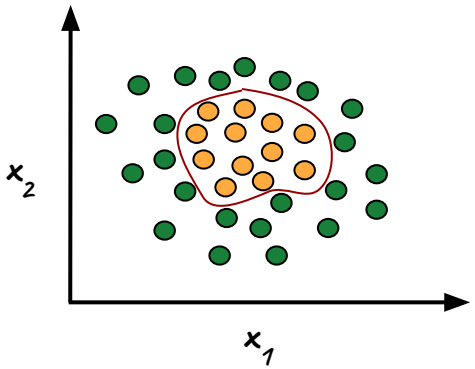
(1 by convention)

$f_i^{(i)} = K(x^{(i)}, l^{(i)}) = 1$

$f_1^{(i)} = K(x^{(i)}, l^{(1)})$

So, we have a feature vector $f^{(i)}$, a weight vector $W = [w_0, w_1, \dots, w_m]$, and a model:

$$w_0 + w_1 f_1^{(i)} + w_2 f_2^{(i)} + \dots + w_m f_m^{(i)} = W^T f^{(i)} \begin{cases} \text{if } W^T f^{(i)} \geq 0 \rightarrow \text{predict } y = 1 \\ \text{if } W^T f^{(i)} < 0 \rightarrow \text{predict } y = 0 \end{cases}$$



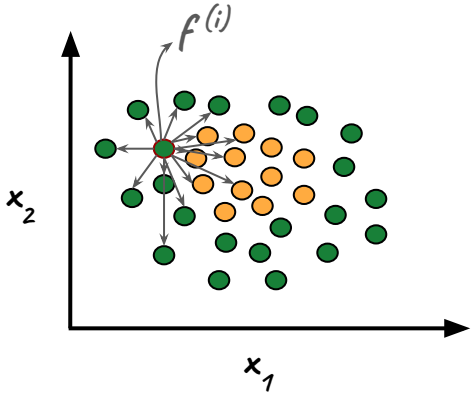
trained by minimizing the SVM cost as a function of $W^T f^{(i)}$

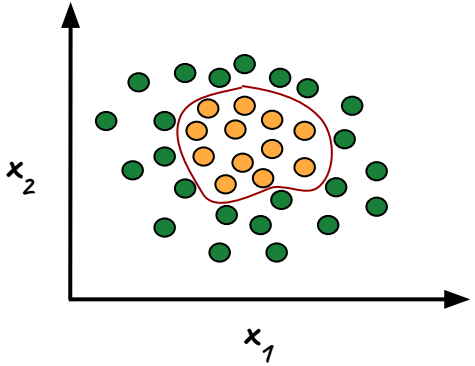
$$J(w_i) = C \sum_{i=1}^{m} \left(y^{(i)} \text{cost}_1(W^T f^{(i)}) + (1 - y^{(i)}) \text{cost}_0(W^T f^{(i)}) \right) + \frac{1}{2} \sum_{j=1}^{m} w_j^2$$

In this case (RBF Kernel) the model hyperparameters are:

- C (the inverse of the regularization term)
- $\gamma = 1/\sigma^2$

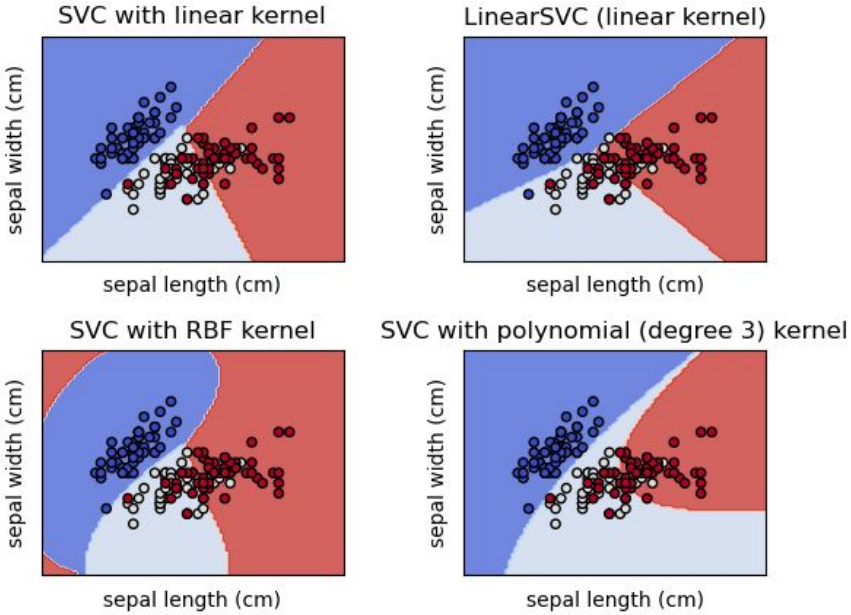
How do we choose the landmarks $l^{(i)}$? And what other kernels are available?





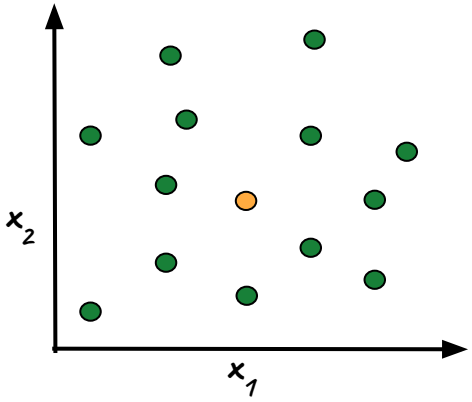
The most commonly used kernels are:

- linear kernel (that is, no kernel altogether)
- radial basis function: $K(x, l^{(i)}) = \exp\{-\gamma \, ||x - l^{(i)}||^2\}$
- polynomial kernel: $K(x, l^{(i)}) = (x^T l^{(i)} + const)^d$
- chi-square kernel: $K(x, l^{(i)}) = \exp\{-\gamma \, \Sigma_i \, [(x - l^{(i)})^2 / (x + l^{(i)})]\}$
- … and more esoteric ones (https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics.pairwise)



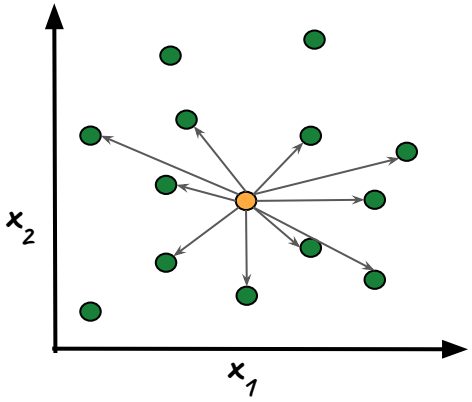
The intuition behind Nearest Neighbors (NN) algorithms is quite straightforward.

The intuition behind Nearest Neighbors (NN) algorithms is quite straightforward.
You have a **training** (or ***reference***) sample, $x^{(i)}$, $y^{(i)}$, and a **target** example. The principle is to find a number of **reference** examples closest to the **target** example in the features space, and predict the **target** label y from these.



The number of samples can be a user-defined constant (***k-nearest neighbors***), or vary based on the local density of points (***radius-based neighbors***).
The distance can, in general, be **any** metric measure: standard Euclidean distance is the most common choice, but also χ^2 distance, Cosine Distance, Manhattan Distance…

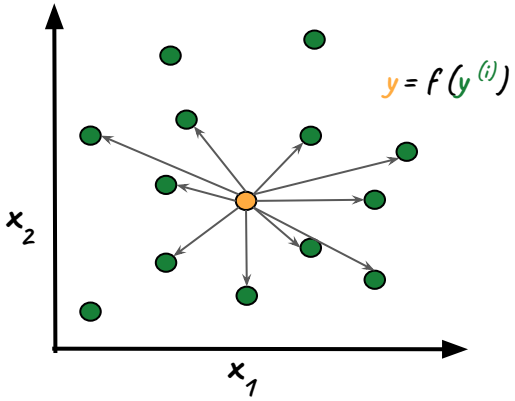
The intuition behind Nearest Neighbors (NN) algorithms is quite straightforward.
You have a **training** (or ***reference***) sample, $x^{(i)}$, $y^{(i)}$, and a **target** example. The principle is to find a number of **reference** examples closest to the **target** example in the features space, and predict the **target** label y from these.



The number of samples can be a user-defined constant (***k-nearest neighbors***), or vary based on the local density of points (***radius-based neighbors***).
The distance can, in general, be **any** metric measure: standard Euclidean distance is the most common choice, but also χ^2 distance, Cosine Distance, Manhattan Distance…

Nearest Neighbors (NN)

The intuition behind Nearest Neighbors (NN) algorithms is quite straightforward.
You have a **training** (or ***reference***) sample, $x^{(i)}$, $y^{(i)}$, and a **target** example. The principle is to find a number of **reference** examples closest to the **target** example in the features space, and predict the **target** label y from these.



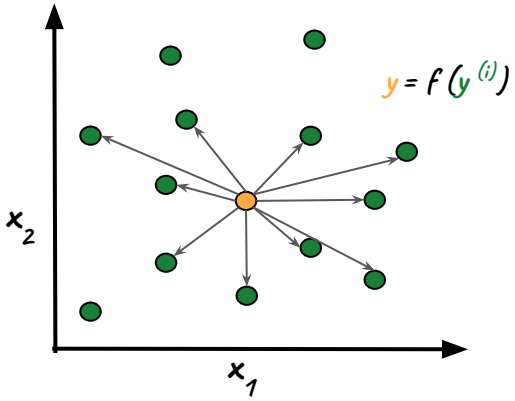
The number of samples can be a user-defined constant (***k-nearest neighbors***), or vary based on the local density of points (***radius-based neighbors***).
The distance can, in general, be **any** metric measure: standard Euclidean distance is the most common choice, but also χ^2 distance, Cosine Distance, Manhattan Distance…

←

For regression tasks, the predicted label y is a function of the NN **reference** labels $y^{(i)}$, i.e. the mean, or the weighted-mean where the weights could be the distance between the **reference** features and the **target** features, or error-weighted distance, or whatever user-defined weights are in your dataset, there is a Universe of possibilities.

Nearest Neighbors (NN)

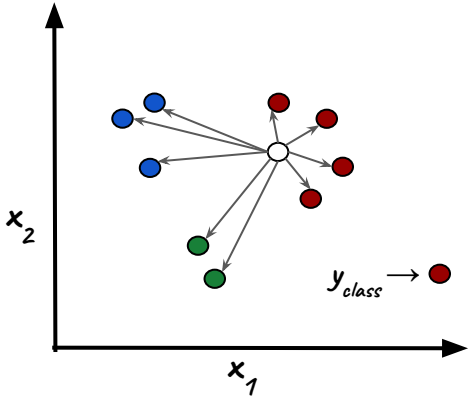
The intuition behind Nearest Neighbors (NN) algorithms is quite straightforward.
You have a **training** (or ***reference***) sample, $x^{(i)}$, $y^{(i)}$, and a **target** example. The principle is to find a number of **reference** examples closest to the **target** example in the features space, and predict the **target** label y from these.



The number of samples can be a user-defined constant (***k-nearest neighbors***), or vary based on the local density of points (***radius-based neighbors***).
The distance can, in general, be **any** metric measure: standard Euclidean distance is the most common choice, but also χ^2 distance, Cosine Distance, Manhattan Distance…

←

For regression tasks, the predicted label y is a function of the NN **reference** labels $y^{(i)}$, i.e. the mean, or the weighted-mean where the weights could be the distance between the **reference** features and the **target** features, or error-weighted distance, or whatever user-defined weights are in your dataset, there is a Universe of possibilities.

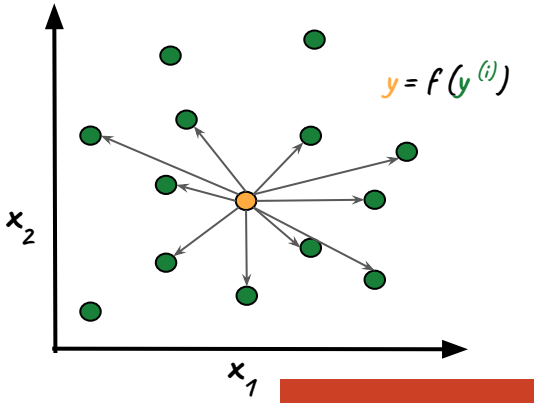


←

For classification tasks, the predicted label y is computed from a (weighted) <u>majority vote</u> of the NN **reference** classes. The assigned label class is the one that has the most representatives within the nearest neighbors of the point.
The votes weighting could be:

- uniform ("*ognuno vale uno*"); a really, really bad idea
- distance-weighted; closer neighbors will have a greater influence than farther neighbors
- whatever user-defined weights you opt to use

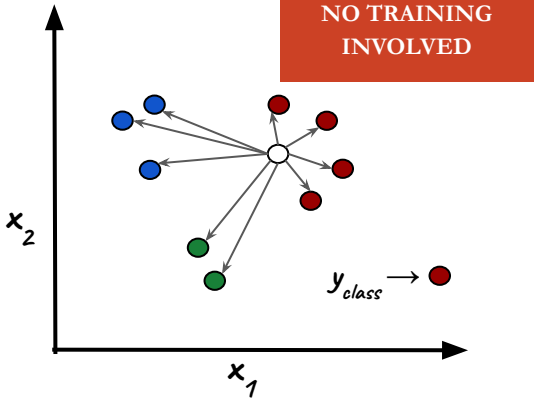
The intuition behind Nearest Neighbors (NN) algorithms is quite straightforward.
You have a **training** (or ***reference***) sample, $x^{(i)}$, $y^{(i)}$, and a **target** example. The principle is to find a number of **reference** examples closest to the **target** example in the features space, and predict the **target** label y from these.



The number of samples can be a user-defined constant (***k-nearest neighbors***), or vary based on the local density of points (***radius-based neighbors***).
The distance can, in general, be **any** metric measure: standard Euclidean distance is the most common choice, but also χ^2 distance, Cosine Distance, Manhattan Distance…

←

For regression tasks, the predicted label y is a function of the NN **reference** labels $y^{(i)}$, i.e. the mean, or the weighted-mean where the weights could be the distance between the **reference** features and the **target** features, or error-weighted distance, or whatever user-defined weights are in your dataset, there is a Universe of possibilities.



←

For classification tasks, the predicted label y is computed from a (weighted) <u>majority vote</u> of the NN **reference** classes. The assigned label class is the one that has the most representatives within the nearest neighbors of the point.
The votes weighting could be:

- uniform ("*ognuno vale uno*"); a really, really bad idea
- distance-weighted; closer neighbors will have a greater influence than farther neighbors
- whatever user-defined weights you opt to use

Nearest Neighbors (NN) - How to find neighbors?

Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

Brute force

With brute force computation, you just measure the distance metric between the target example and **every** reference example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

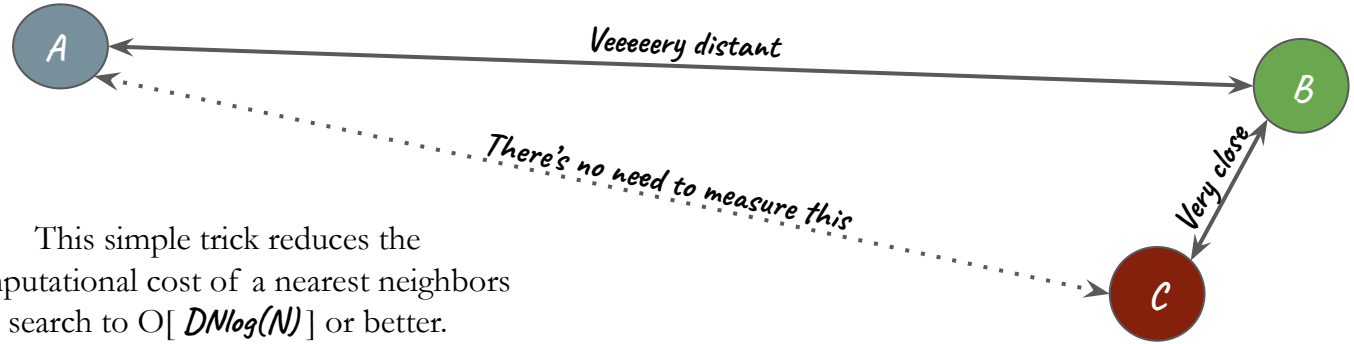
Nearest Neighbors (NN) - How to find neighbors?

Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

Brute force

With brute force computation, you just measure the distance metric between the target example and **every** reference example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

To address the computational inefficiencies of the brute-force approach, a variety of *tree-based data structures* have been invented. These structures attempt to reduce the required number of distance calculations by efficiently encoding aggregate distance information.



This simple trick reduces the computational cost of a nearest neighbors search to O[$DN\log(N)$] or better.

A very distant from B
C very close to B

→A very distant from C without the need to actually compute the distance

Nearest Neighbors (NN) - How to find neighbors?

Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

Brute force

With brute force computation, you just measure the distance metric between the **target** example and **every** **reference** example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

KD tree

KD tree is a binary tree structure which recursively partitions the **reference** parameter space along axes, dividing it into regions into which data points are filed. At that point, for a given **target** example, you just look for the correct partition, and only then switch to brute force calculation of the distances.

Nearest Neighbors (NN) - How to find neighbors?

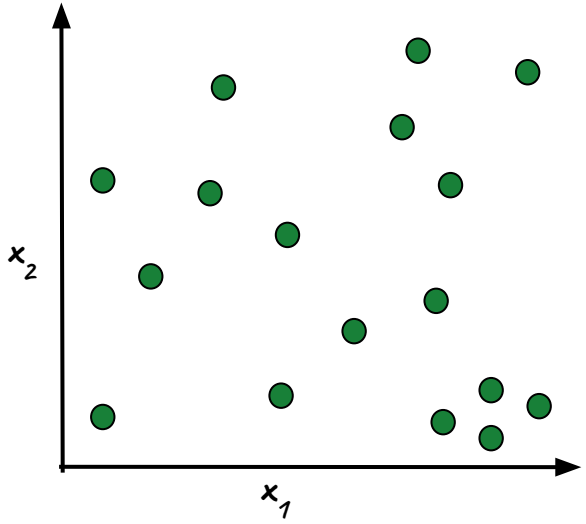
Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

Brute force

With brute force computation, you just measure the distance metric between the target example and **every** reference example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

KD tree

KD tree is a binary tree structure which recursively partitions the reference parameter space along axes, dividing it into regions into which data points are filed. At that point, for a given target example, you just look for the correct partition, and only then switch to brute force calculation of the distances.



Nearest Neighbors (NN) - How to find neighbors?

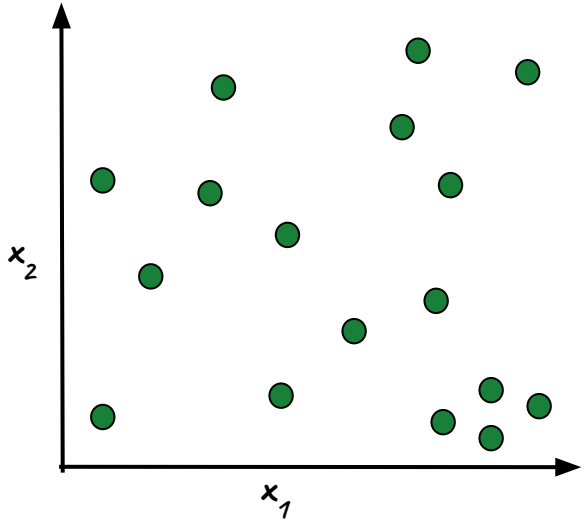
Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

Brute force

With brute force computation, you just measure the distance metric between the target example and **every** reference example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

KD tree

KD tree is a binary tree structure which recursively partitions the reference parameter space along axes, dividing it into regions into which data points are filed. At that point, for a given target example, you just look for the correct partition, and only then switch to brute force calculation of the distances.



Recursively repeat:

1) *pick a random axis*
2) *for all the points in a partition, find i.e. the median*
3) *split in two partitions*

Until a maximum number of points in a partition ("leaf") are reached, e.g. 5 in the example on the left.

Nearest Neighbors (NN) - How to find neighbors?

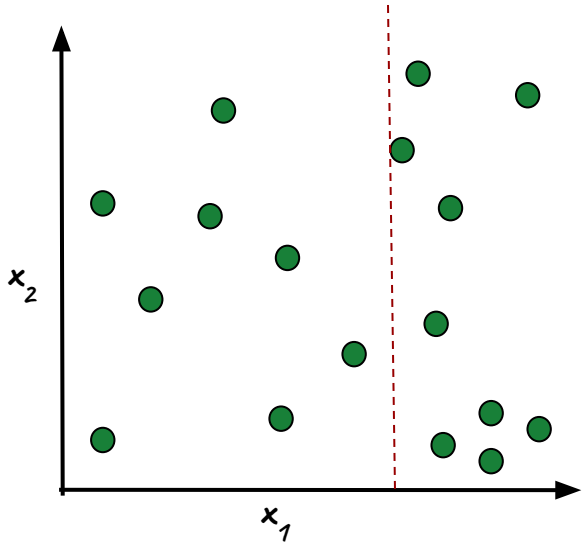
Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

Brute force

With brute force computation, you just measure the distance metric between the target example and **every** reference example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

KD tree

KD tree is a binary tree structure which recursively partitions the reference parameter space along axes, dividing it into regions into which data points are filed. At that point, for a given target example, you just look for the correct partition, and only then switch to brute force calculation of the distances.



Recursively repeat:

1) *pick a random axis*
2) *for all the points in a partition, find i.e. the median*
3) *split in two partitions*

Until a maximum number of points in a partition ("leaf") are reached, e.g. 5 in the example on the left.

Nearest Neighbors (NN) - How to find neighbors?

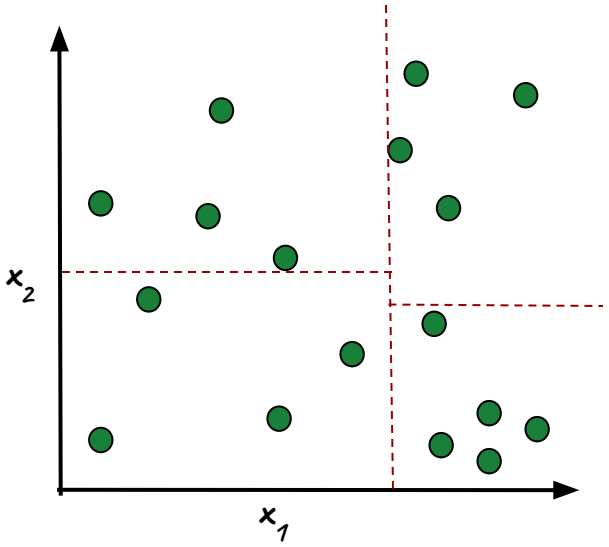
Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

Brute force

With brute force computation, you just measure the distance metric between the target example and **every** reference example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

KD tree

KD tree is a binary tree structure which recursively partitions the reference parameter space along axes, dividing it into regions into which data points are filed. At that point, for a given target example, you just look for the correct partition, and only then switch to brute force calculation of the distances.



Recursively repeat:

1) *pick a random axis*
2) *for all the points in a partition, find i.e. the median*
3) *split in two partitions*

Until a maximum number of points in a partition ("leaf") are reached, e.g. 5 in the example on the left.

Nearest Neighbors (NN) - How to find neighbors?

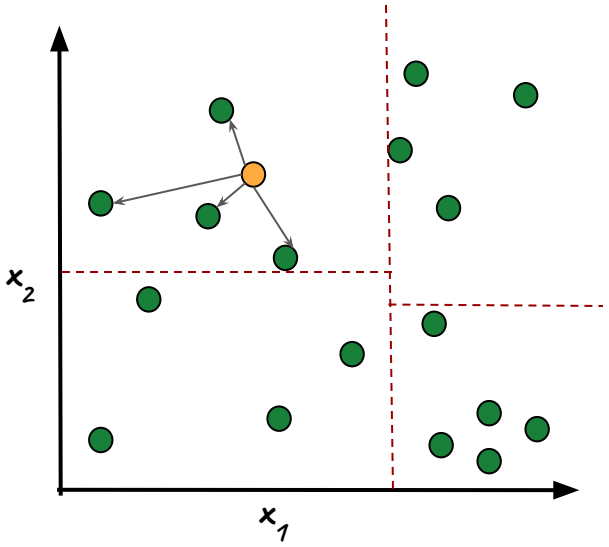
Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

Brute force

With brute force computation, you just measure the distance metric between the target example and **every** reference example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

KD tree

KD tree is a binary tree structure which recursively partitions the reference parameter space along axes, dividing it into regions into which data points are filed. At that point, for a given target example, you just look for the correct partition, and only then switch to brute force calculation of the distances.



Recursively repeat:

1) *pick a random axis*
2) *for all the points in a partition, find i.e. the median*
3) *split in two partitions*

Until a maximum number of points in a partition ("leaf") are reached, e.g. 5 in the example on the left.

Nearest Neighbors (NN) - How to find neighbors?

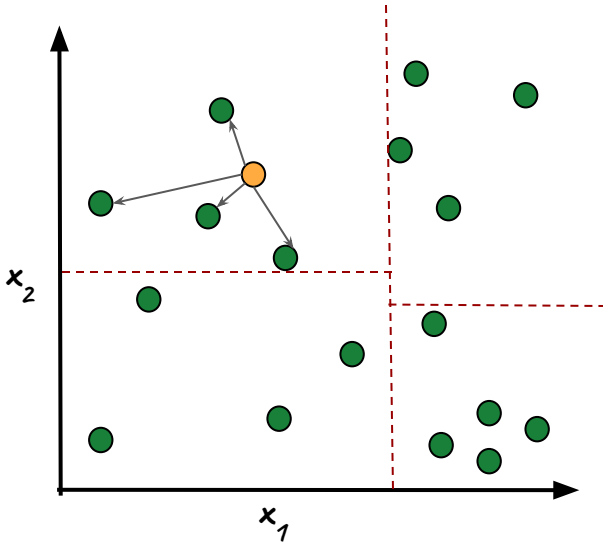
Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

Brute force

With brute force computation, you just measure the distance metric between the target example and **every** reference example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

KD tree

KD tree is a binary tree structure which recursively partitions the reference parameter space along axes, dividing it into regions into which data points are filed. At that point, for a given target example, you just look for the correct partition, and only then switch to brute force calculation of the distances.



Recursively repeat:

1) pick a random axis
2) for all the points in a partition, find i.e. the median
3) split in two partitions

Until a maximum number of points in a partition ("leaf") are reached, e.g. 5 in the example on the left.

Striking resemblance with decision trees and random forests (more on that later)

Nearest Neighbors (NN) - How to find neighbors?

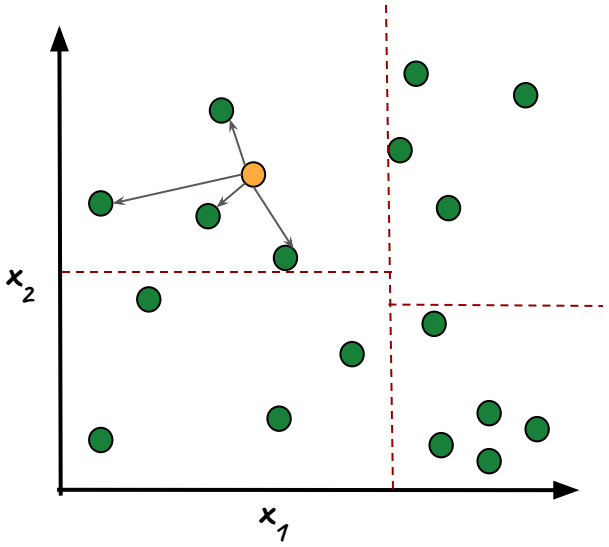
Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

Brute force

With brute force computation, you just measure the distance metric between the target example and **every** reference example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

KD tree

KD tree is a binary tree structure which recursively partitions the reference parameter space along axes, dividing it into regions into which data points are filed. At that point, for a given target example, you just look for the correct partition, and only then switch to brute force calculation of the distances.



Recursively repeat:

1) *pick a random axis*
2) *for all the points in a partition, find i.e. the median*
3) *split in two partitions*

Until a maximum number of points in a partition ("leaf") are reached, e.g. 5 in the example on the left.

Striking resemblance with decision trees and random forests (more on that later)

Building a a KD tree is very fast: partitioning is performed only along the data axes, so no dimensional distances need to be computed.
Once built, the nearest neighbor of a query point can be determined with only O[$\log(N)$] distance computations.
Though the KD tree approach is very fast for low-dimensional ($D<20$) neighbors searches, it becomes inefficient as D grows very large: this is one manifestation of the so-called "*curse of dimensionality*" (more on that next week in Unsupervised Learning)

Nearest Neighbors (NN) - How to find neighbors?

Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

Brute force

With brute force computation, you just measure the distance metric between the **target** example and **every** **reference** example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

KD tree

KD tree is a binary tree structure which recursively partitions the **reference** parameter space along axes, dividing it into regions into which data points are filed. At that point, for a given **target** example, you just look for the correct partition, and only then switch to brute force calculation of the distances.

Ball tree

KD trees partition **reference** data along Cartesian axes. Ball trees partition data in a series of nesting hyperspheres. This makes tree construction more costly than that of the KD tree, but results in a data structure which can be very efficient on highly structured data, even in very high dimensions.

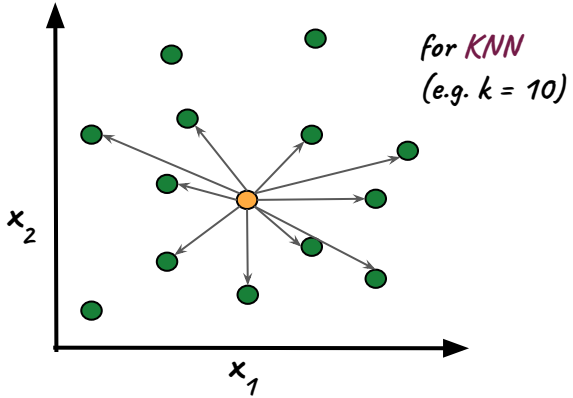
Finding the nearest neighbors can be achieved in three different ways, depending on the size of your samples.

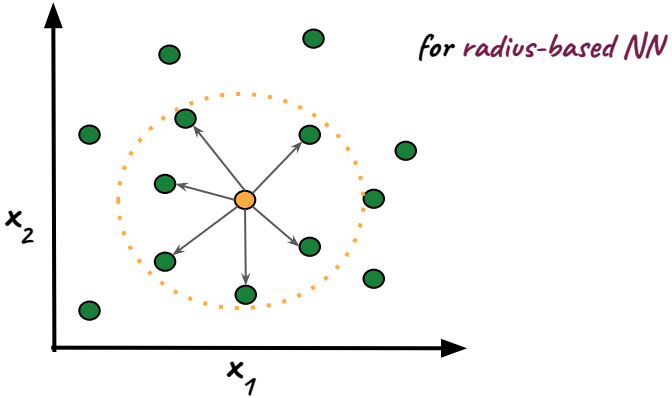
Brute force With brute force computation, you just measure the distance metric between the target example and **every** reference example. For N samples in D dimensions, this approach scales as O[DN^2]. Works fine for small dataset, quickly explodes as the number of samples N grows.

KD tree KD tree is a binary tree structure which recursively partitions the reference parameter space along axes, dividing it into regions into which data points are filed. At that point, for a given target example, you just look for the correct partition, and only then switch to brute force calculation of the distances.

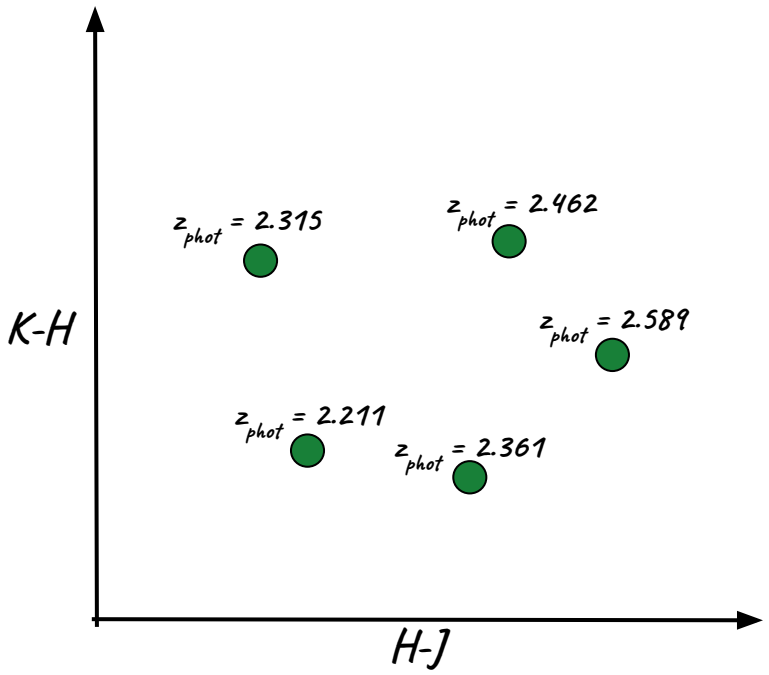
Ball tree KD trees partition reference data along Cartesian axes. Ball trees partition data in a series of nesting hyperspheres. This makes tree construction more costly than that of the KD tree, but results in a data structure which can be very efficient on highly structured data, even in very high dimensions.

Once you have candidates NN:

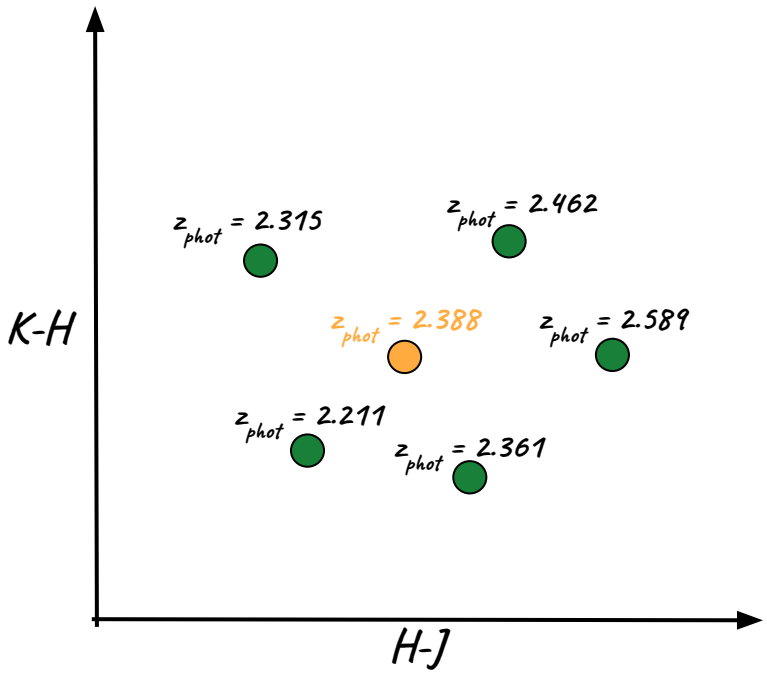




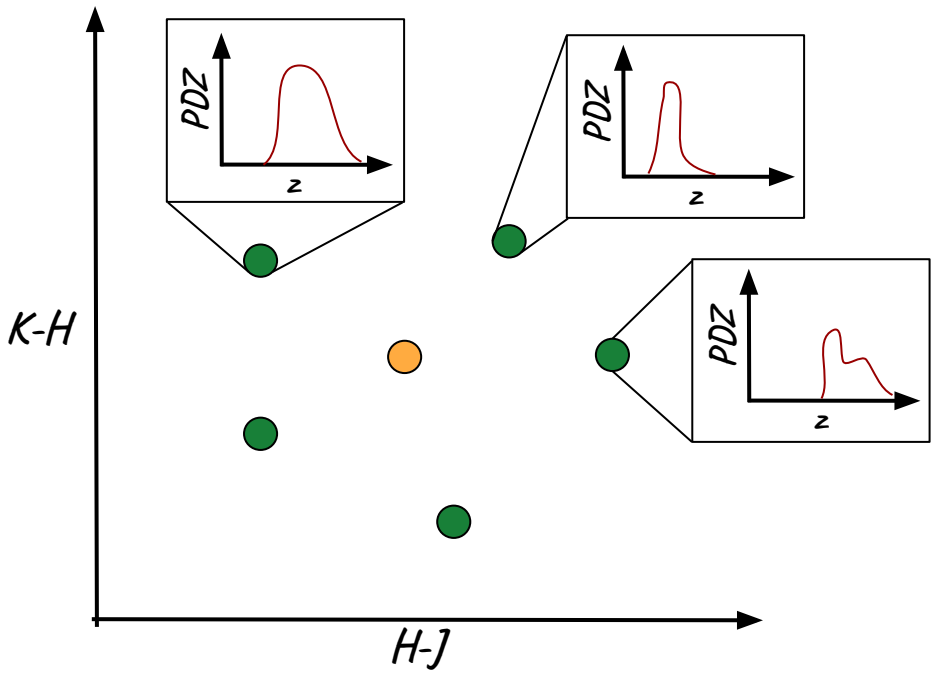
Consider the following example: measuring galaxies' distances based on their observed photometry (photometric redshift).
The features are colors and magnitudes (e.g. *H-J* and *K-H*), the labels the ***photo-z***. You have a **reference sample**, built from applying classical SED fitting methods (i.e. MAGPHYS, Cigale, Lephare, whatever) to the observed photometry.



Consider the following example: measuring galaxies' distances based on their observed photometry (photometric redshift).
The features are colors and magnitudes (e.g. *H-J* and *K-H*), the labels the ***photo-z***. You have a **reference sample**, built from applying classical SED fitting methods (i.e. MAGPHYS, Cigale, Lephare, whatever) to the observed photometry.
Whenever a **target** galaxy arrives, you can look for the 5 nearest-neighbors, and predict the ***photo-z*** by taking the distance-weighted mean of the reference galaxies ***photo-z***.

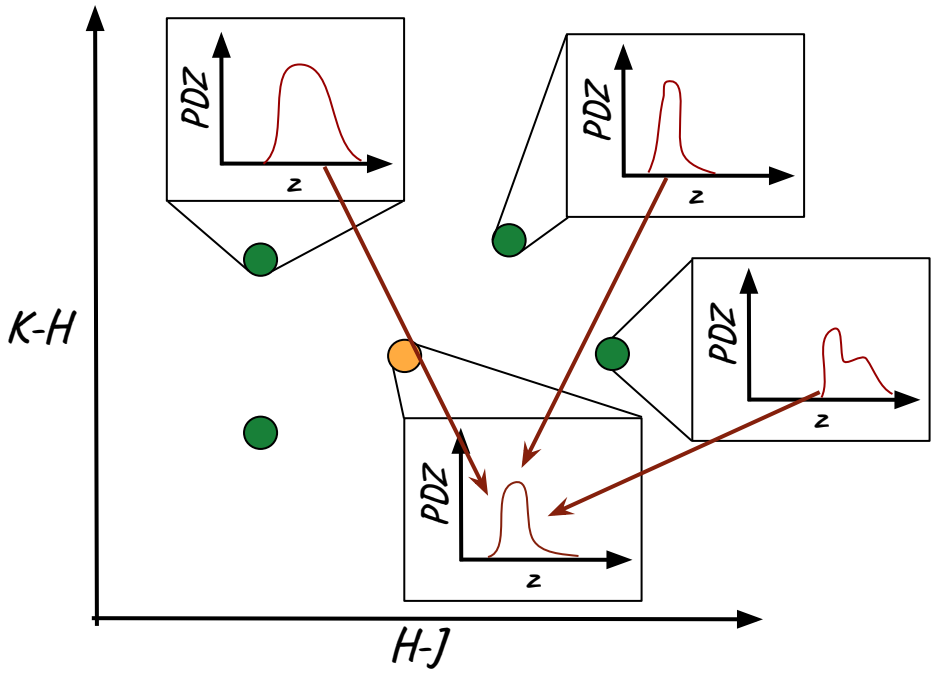


Consider the following example: measuring galaxies' distances based on their observed photometry (photometric redshift).
The features are colors and magnitudes (e.g. *H-J* and *K-H*), the labels the ***photo-z***. You have a **reference sample**, built from applying classical SED fitting methods (i.e. MAGPHYS, Cigale, Lephare, whatever) to the observed photometry.
Whenever a **target** galaxy arrives, you can look for the 5 nearest-neighbors, and predict the ***photo-z*** by taking the distance-weighted mean of the reference galaxies ***photo-z***.



But there's more, if you consider that the reference sample is not just a collection of numbers, but "placeholders", carrying with them the whole set of information that traditional SED fitting algorithms carry. Those (usually Bayesian) methods output the full posterior distribution function for the parameters, not just the point predictions (PDZ for ***photo-z***).

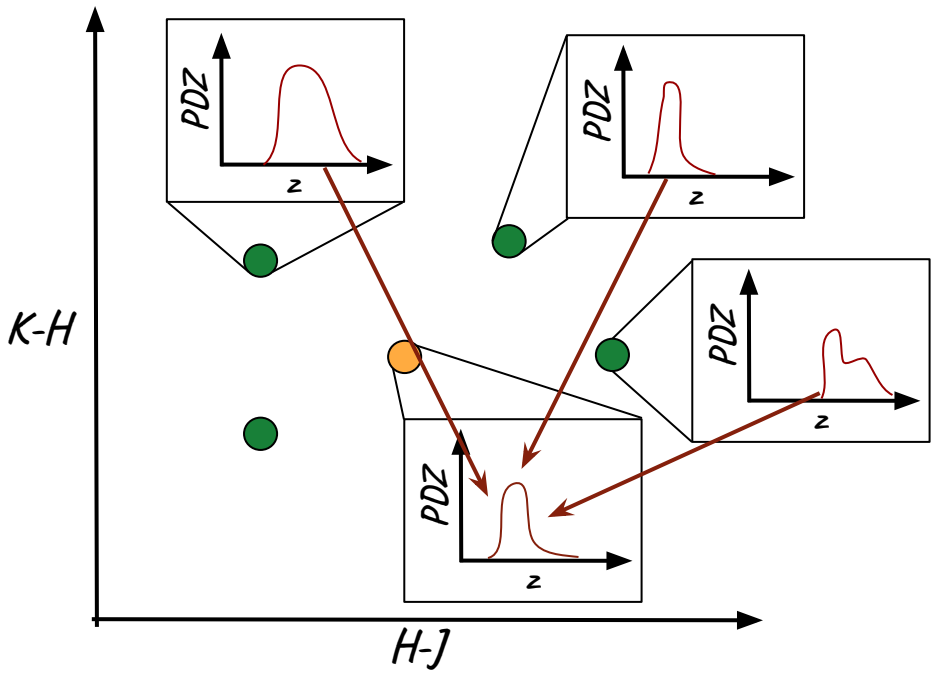
Consider the following example: measuring galaxies' distances based on their observed photometry (photometric redshift).
The features are colors and magnitudes (e.g. *H-J* and *K-H*), the labels the ***photo-z***. You have a **reference sample**, built from applying classical SED fitting methods (i.e. MAGPHYS, Cigale, Lephare, whatever) to the observed photometry.
Whenever a **target** galaxy arrives, you can look for the 5 nearest-neighbors, and predict the ***photo-z*** by taking the distance-weighted mean of the reference galaxies ***photo-z***.



But there's more, if you consider that the reference sample is not just a collection of numbers, but "placeholders", carrying with them the whole set of information that traditional SED fitting algorithms carry. Those (usually Bayesian) methods output the full posterior distribution function for the parameters, not just the point predictions (PDZ for ***photo-z***).

Under the assumption that similar photometries with similar error would naturally lead to similar PDZs, you can combine the PDZs in the **reference sample**'s nearest neighbors to "measure" the **target** galaxy's PDZ.

Consider the following example: measuring galaxies' distances based on their observed photometry (photometric redshift).
The features are colors and magnitudes (e.g. *H-J* and *K-H*), the labels the ***photo-z***. You have a **reference sample**, built from applying classical SED fitting methods (i.e. MAGPHYS, Cigale, Lephare, whatever) to the observed photometry.
Whenever a **target** galaxy arrives, you can look for the 5 nearest-neighbors, and predict the ***photo-z*** by taking the distance-weighted mean of the reference galaxies ***photo-z***.



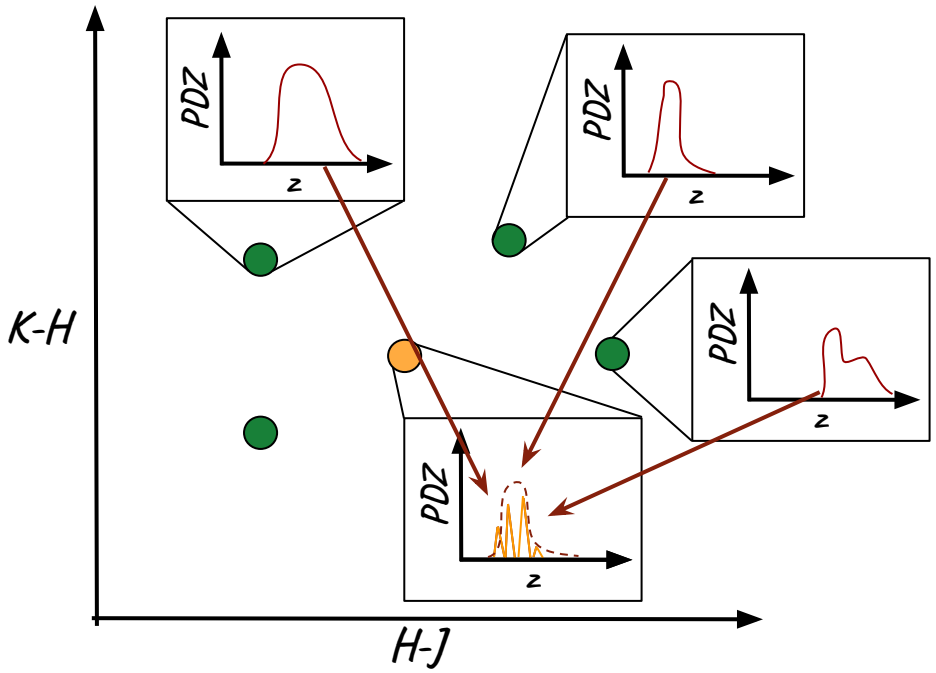
But there's more, if you consider that the reference sample is not just a collection of numbers, but "placeholders", carrying with them the whole set of information that traditional SED fitting algorithms carry. Those (usually Bayesian) methods output the full posterior distribution function for the parameters, not just the point predictions (PDZ for ***photo-z***).

Under the assumption that similar photometries with similar error would naturally lead to similar PDZs, you can combine the PDZs in the **reference sample**'s nearest neighbors to "measure" the **target** galaxy's PDZ.

What I've just described here is the **NNPZ**, the <u>official</u> Euclid pipeline (along with a Self-Organizing Map calibration phase, more on Monday) to measure the PDZ for the > 10^9 galaxies that the survey will observe, necessary to measure the cosmic shear.

https://doi.org/10.1051/0004-6361/202039403

Consider the following example: measuring galaxies' distances based on their observed photometry (photometric redshift).
The features are colors and magnitudes (e.g. *H-J* and *K-H*), the labels the ***photo-z***. You have a **reference sample**, built from applying classical SED fitting methods (i.e. MAGPHYS, Cigale, Lephare, whatever) to the observed photometry.
Whenever a **target** galaxy arrives, you can look for the 5 nearest-neighbors, and predict the ***photo-z*** by taking the distance-weighted mean of the reference galaxies ***photo-z***.



But there's more, if you consider that the reference sample is not just a collection of numbers, but "placeholders", carrying with them the whole set of information that traditional SED fitting algorithms carry. Those (usually Bayesian) methods output the full posterior distribution function for the parameters, not just the point predictions (PDZ for ***photo-z***).

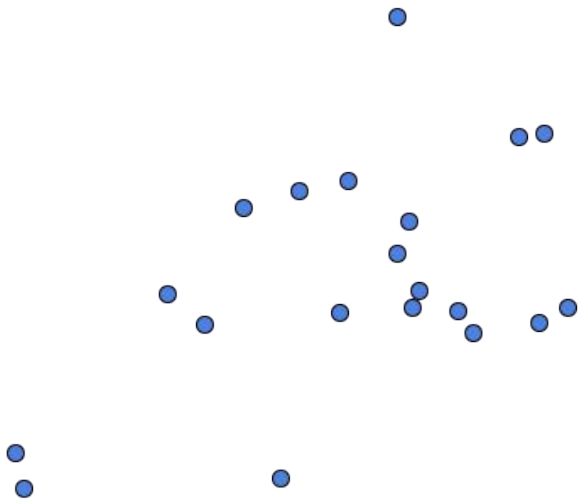
Under the assumption that similar photometries with similar error would naturally lead to similar PDZs, you can combine the PDZs in the **reference sample**'s nearest neighbors to "measure" the **target** galaxy's PDZ.

What I've just described here is the **NNPZ**, the <u>official</u> Euclid pipeline (along with a Self-Organizing Map calibration phase, more on Monday) to measure the PDZ for the > 10^9 galaxies that the survey will observe, necessary to measure the cosmic shear.

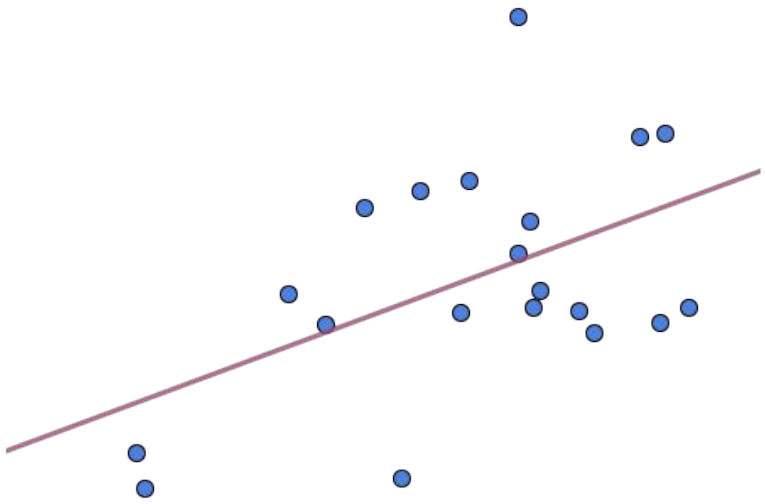
https://doi.org/10.1051/0004-6361/202039403

Gaussian Processes are perhaps less known (at least as a Supervised ML technique), but are extremely powerful as they can be solved analytically while still being able to model relatively complex systems.

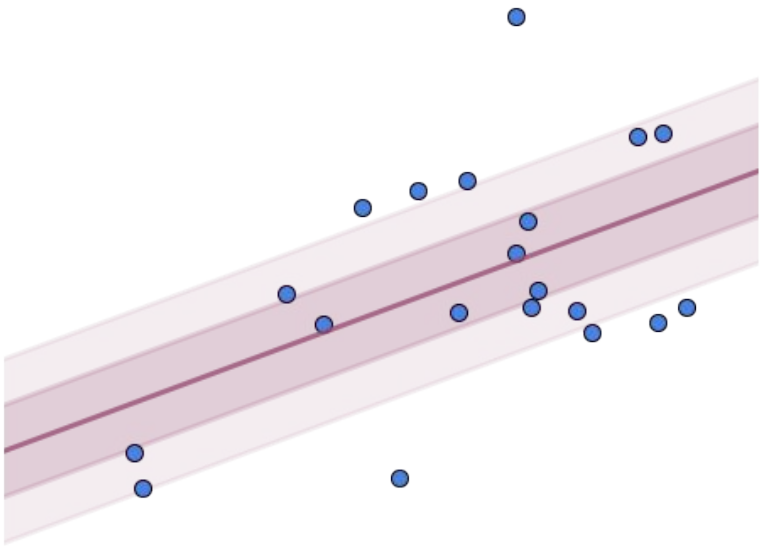
Gaussian Processes are perhaps less known (at least as a Supervised ML technique), but are extremely powerful as they can be solved analytically while still being able to model relatively complex systems.
Let's say we want to fit this particular dataset with a linear model, and estimate the remaining noise variance (a predictive distribution).



Gaussian Processes are perhaps less known (at least as a Supervised ML technique), but are extremely powerful as they can be solved analytically while still being able to model relatively complex systems.
Let's say we want to fit this particular dataset with a linear model, and estimate the remaining noise variance (a predictive distribution).



Gaussian Processes are perhaps less known (at least as a Supervised ML technique), but are extremely powerful as they can be solved analytically while still being able to model relatively complex systems.
Let's say we want to fit this particular dataset with a linear model, and estimate the remaining noise variance (a predictive distribution).



This is the typical linear regression algorithm showed yesterday, which gives back informative results once coupled with the variance of the observed (training) data, but misses completely the uncertainty on the model itself.
How to capture it?

Gaussian Processes are perhaps less known (at least as a Supervised ML technique), but are extremely powerful as they can be solved analytically while still being able to model relatively complex systems.
Let's say we want to fit this particular dataset with a linear model, and estimate the remaining noise variance (a predictive distribution).

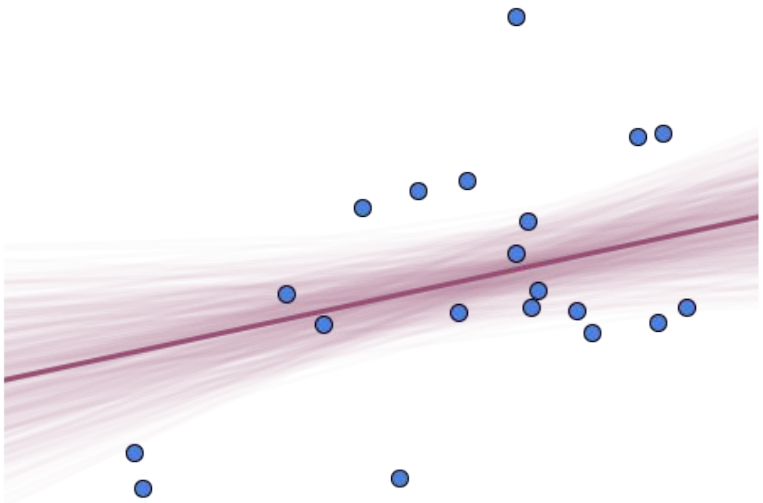
Enters Bayesian linear regression.
Let's forget for the data for a second, and consider the model we're trying to fit. Without data, the model is a set of all the possible linear solutions that are in the parameter space. We'll call it the ***prior samples*** (on the left, 70 possible linear solutions with some priors on the weight $\boldsymbol{w}$ and bias $\boldsymbol{b}$) *.

Gaussian Processes are perhaps less known (at least as a Supervised ML technique), but are extremely powerful as they can be solved analytically while still being able to model relatively complex systems.
Let's say we want to fit this particular dataset with a linear model, and estimate the remaining noise variance (a predictive distribution).

Enters Bayesian linear regression.
Let's forget for the data for a second, and consider the model we're trying to fit. Without data, the model is a set of all the possible linear solutions that are in the parameter space. We'll call it the ***prior samples*** (on the left, 70 possible linear solutions with some priors on the weight $\boldsymbol{w}$ and bias $\boldsymbol{b}$) *.
Now, we use Bayes' theorem to update the model such that it produces samples that agree with our data.

Gaussian Processes are perhaps less known (at least as a Supervised ML technique), but are extremely powerful as they can be solved analytically while still being able to model relatively complex systems.
Let's say we want to fit this particular dataset with a linear model, and estimate the remaining noise variance (a predictive distribution).

"Posterior samples"

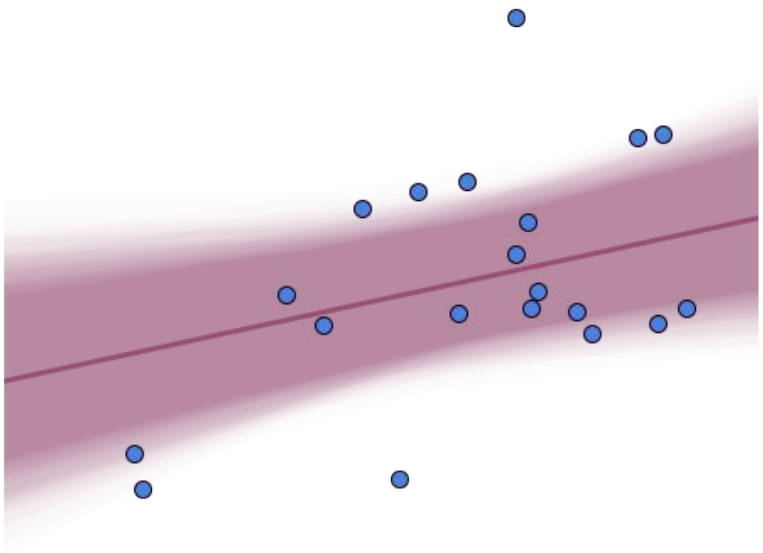


Enters Bayesian linear regression.
Let's forget for the data for a second, and consider the model we're trying to fit. Without data, the model is a set of all the possible linear solutions that are in the parameter space. We'll call it the *prior samples* (on the left, 70 possible linear solutions with some priors on the weight w and bias b).
Now, we use Bayes' theorem to update the model such that it produces samples that agree with our data.
In this way we get the *posterior samples*:

$$\text{Prior samples (Model)} + \text{Data} \xrightarrow[\text{theorem}]{\text{Bayes'}} \text{Posterior samples}$$

Gaussian Processes are perhaps less known (at least as a Supervised ML technique), but are extremely powerful as they can be solved analytically while still being able to model relatively complex systems.
Let's say we want to fit this particular dataset with a linear model, and estimate the remaining noise variance (a predictive distribution).

"Posterior samples"

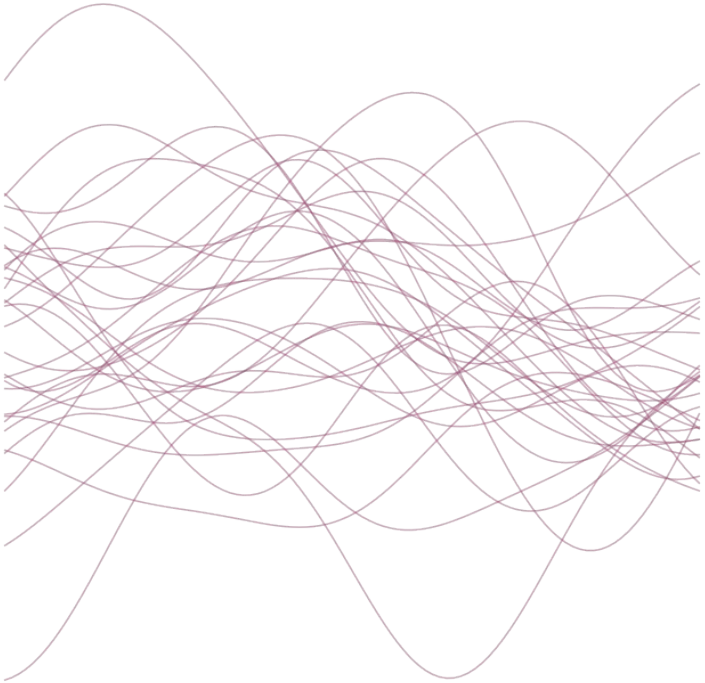


Enters Bayesian linear regression.
Let's forget for the data for a second, and consider the model we're trying to fit. Without data, the model is a set of all the possible linear solutions that are in the parameter space. We'll call it the *prior samples* (on the left, 70 possible linear solutions with some priors on the weight w and bias b).
Now, we use Bayes' theorem to update the model such that it produces samples that agree with our data.
In this way we get the *posterior samples*:

$$\text{Prior samples (Model)} + \text{Data} \xrightarrow[\text{theorem}]{\text{Bayes'}} \text{Posterior samples}$$

Applying a noise variance to each sample will lead to a predictive distribution, that we can average over many (in principle infinite) samples.
As you can see, the intervals become a bit wider far away from the bulk of data, which is a big improvement with respect to the fixed intervals we had with simple linear regression.

Gaussian Processes are perhaps less known (at least as a Supervised ML technique), but are extremely powerful as they can be solved analytically while still being able to model relatively complex systems.
Let's say we want to fit this particular dataset with a linear model, and estimate the remaining noise variance (a predictive distribution).

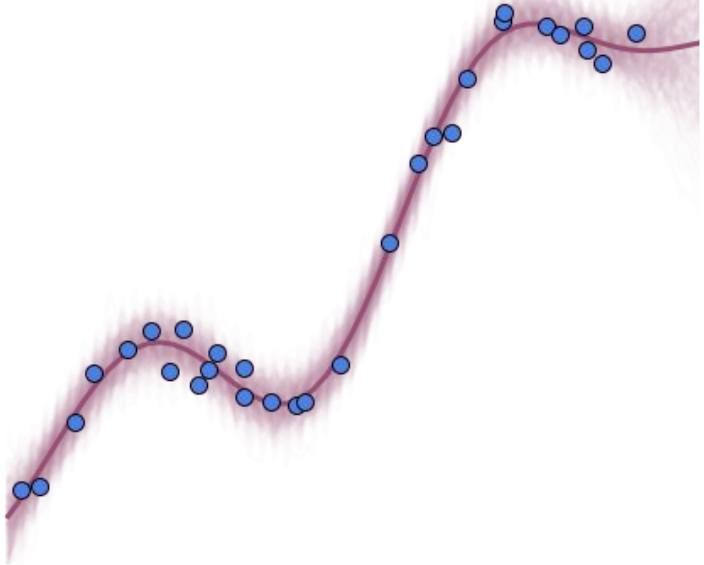


Enters Bayesian linear regression.
Let's forget for the data for a second, and consider the model we're trying to fit. Without data, the model is a set of all the possible linear solutions that are in the parameter space. We'll call it the *prior samples* (on the left, 70 possible linear solutions with some priors on the weight w and bias b).
Now, we use Bayes' theorem to update the model such that it produces samples that agree with our data.
In this way we get the *posterior samples*:

$$\text{Prior samples (Model)} + \text{Data} \xrightarrow[\text{theorem}]{\text{Bayes'}} \text{Posterior samples}$$

Applying a noise variance to each sample will lead to a predictive distribution, that we can average over many (in principle infinite) samples.
As you can see, the intervals become a bit wider far away from the bulk of data, which is a big improvement with respect to the fixed intervals we had with simple linear regression.
The mechanics of GPs is exactly the same, but instead of dealing with lines, we deal with samples over general functions.

Gaussian Processes are perhaps less known (at least as a Supervised ML technique), but are extremely powerful as they can be solved analytically while still being able to model relatively complex systems.
Let's say we want to fit this particular dataset with a linear model, and estimate the remaining noise variance (a predictive distribution).



Enters Bayesian linear regression.
Let's forget for the data for a second, and consider the model we're trying to fit. Without data, the model is a set of all the possible linear solutions that are in the parameter space. We'll call it the *prior samples* (on the left, 70 possible linear solutions with some priors on the weight w and bias b).
Now, we use Bayes' theorem to update the model such that it produces samples that agree with our data.
In this way we get the *posterior samples*:

$$\text{Prior samples (Model)} + \text{Data} \xrightarrow[\text{theorem}]{\text{Bayes'}} \text{Posterior samples}$$

Applying a noise variance to each sample will lead to a predictive distribution, that we can average over many (in principle infinite) samples.
As you can see, the intervals become a bit wider far away from the bulk of data, which is a big improvement with respect to the fixed intervals we had with simple linear regression.
The mechanics of GPs is exactly the same, but instead of dealing with lines, we deal with samples over general functions.

How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?
We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

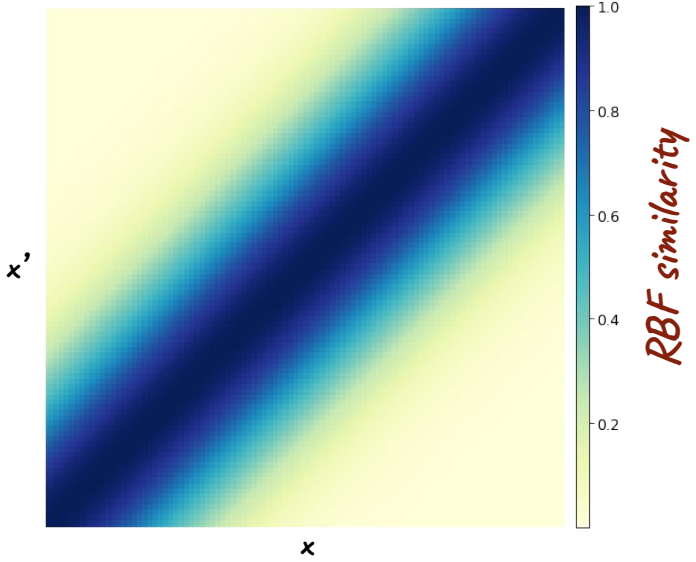
How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, x and x', the kernel is a function measuring their similarity: $K(x, x' | \tau)$ with τ the vector of hyperparameters used to tune it. K close to 1 means similar inputs, K close to 0 means dissimilar inputs.

The most common used ***kernel*** function is, as you can guess, the RBF:

$$K(x, x'|\tau) = \sigma^2 \exp\left(-\frac{1}{2}\left(\frac{x-x'}{l}\right)^2\right)$$



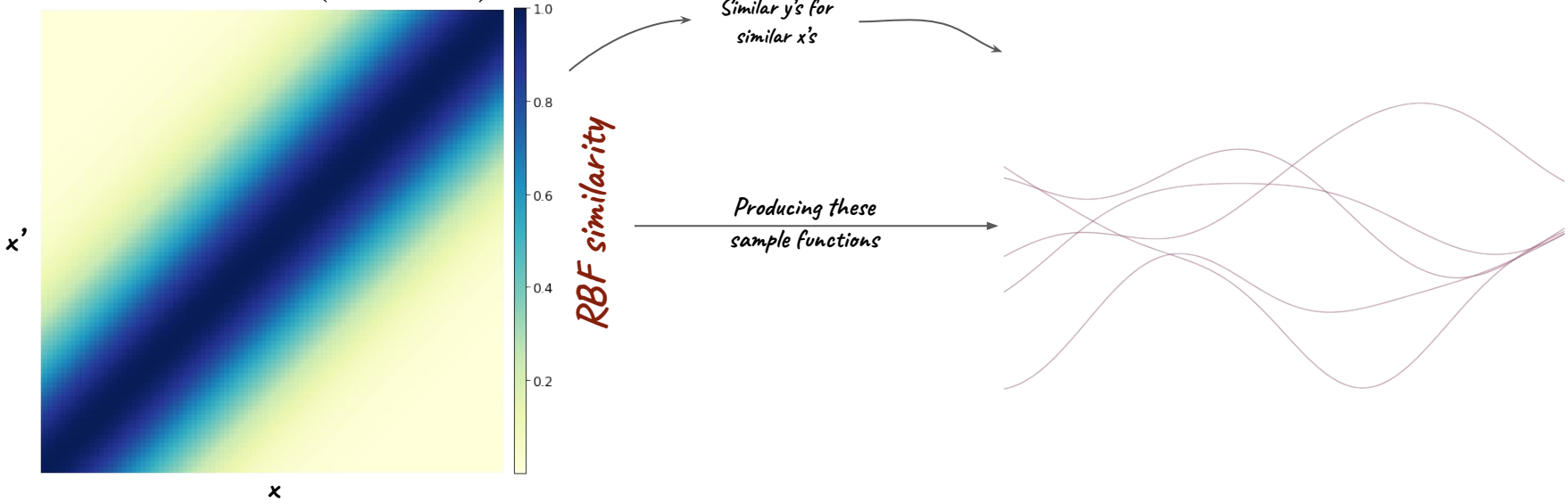
How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, **x** and **x'**, the kernel is a function measuring their similarity: ***K(x, x' |τ)*** with ***τ*** the vector of hyperparameters used to tune it. ***K*** close to 1 means similar inputs, ***K*** close to 0 means dissimilar inputs.

The most common used ***kernel*** function is, as you can guess, the RBF:

$$K(x, x'|\tau) = \sigma^2 \exp\left(-\frac{1}{2}\left(\frac{x-x'}{l}\right)^2\right)$$



How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

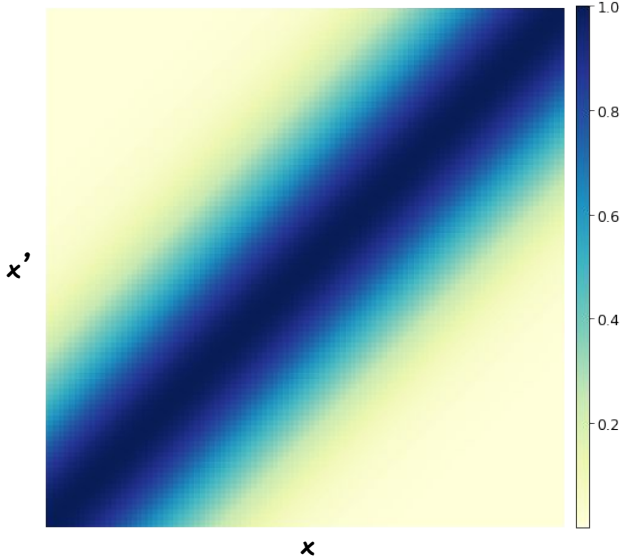
We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, x and x', the kernel is a function measuring their similarity: $K(x, x' | \tau)$ with τ the vector of hyperparameters used to tune it. K close to 1 means similar inputs, K close to 0 means dissimilar inputs.

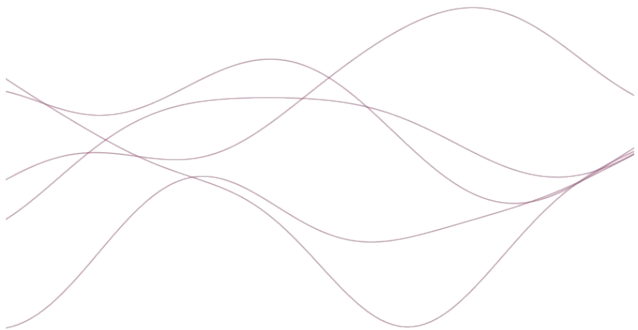
The most common used ***kernel*** function is, as you can guess, the RBF:

$$K(x, x'|\tau) = \sigma^2 \exp\left(-\frac{1}{2}\left(\frac{x-x'}{l}\right)^2\right)$$

"length scale".



Producing these sample functions



How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

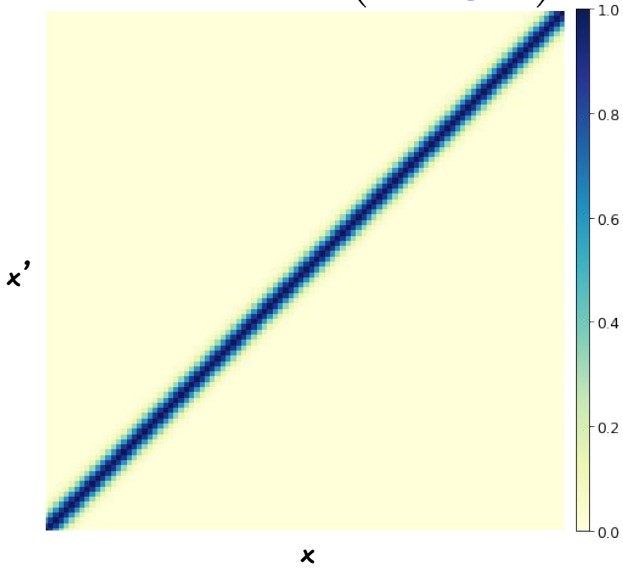
We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, **x** and **x'**, the kernel is a function measuring their similarity: ***K(x, x' |τ)*** with ***τ*** the vector of hyperparameters used to tune it. ***K*** close to 1 means similar inputs, ***K*** close to 0 means dissimilar inputs.

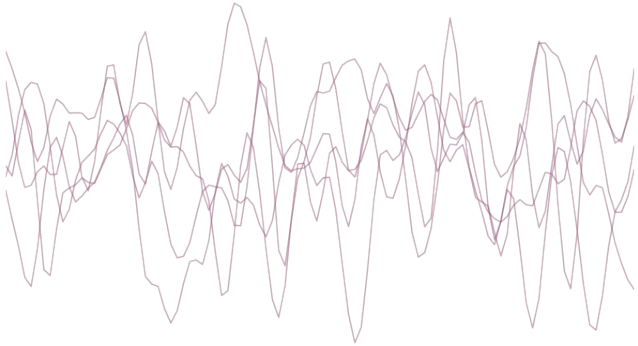
The most common used ***kernel*** function is, as you can guess, the RBF:

$$K(x, x'|\tau) = \sigma^2 \exp\left(-\frac{1}{2}\left(\frac{x-x'}{l}\right)^2\right)$$

"length scale": *for smaller values, most input pairs are considered different, therefore the samples wiggle more rapidly*



Producing these sample functions



How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

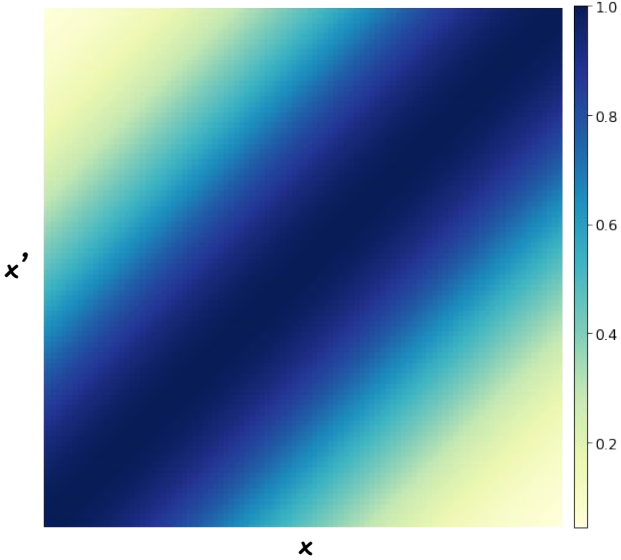
We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, **x** and **x'**, the kernel is a function measuring their similarity: ***K(x, x' |τ)*** with ***τ*** the vector of hyperparameters used to tune it. ***K*** close to 1 means similar inputs, ***K*** close to 0 means dissimilar inputs.

The most common used ***kernel*** function is, as you can guess, the RBF:

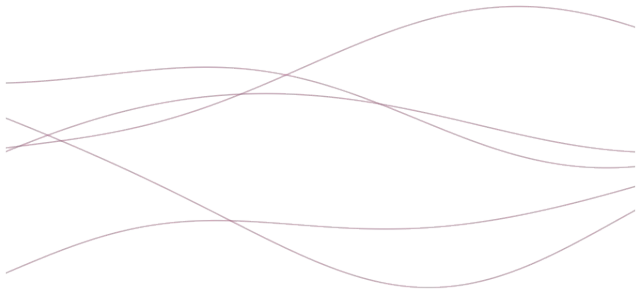
$$K(x, x'|\tau) = \sigma^2 \exp\left(-\frac{1}{2}\left(\frac{x-x'}{l}\right)^2\right)$$

"length scale": *for bigger values, most input pairs are considered similar, therefore the samples are smoother*



RBF similarity

Producing these sample functions



How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

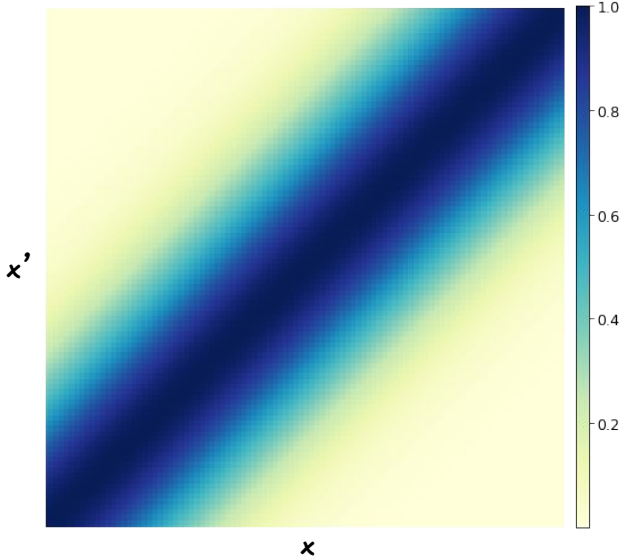
We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, x and x', the kernel is a function measuring their similarity: $K(x, x' | \tau)$ with τ the vector of hyperparameters used to tune it. K close to 1 means similar inputs, K close to 0 means dissimilar inputs.

The most common used ***kernel*** function is, as you can guess, the RBF:

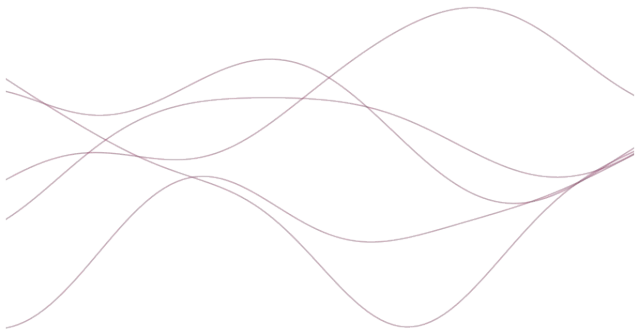
$$K(x, x'|\tau) = \sigma^2 \exp\left(-\frac{1}{2}\left(\frac{x-x'}{l}\right)^2\right)$$

"output scale": determines the scale of the y axis.



RBF similarity

Producing these sample functions



How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

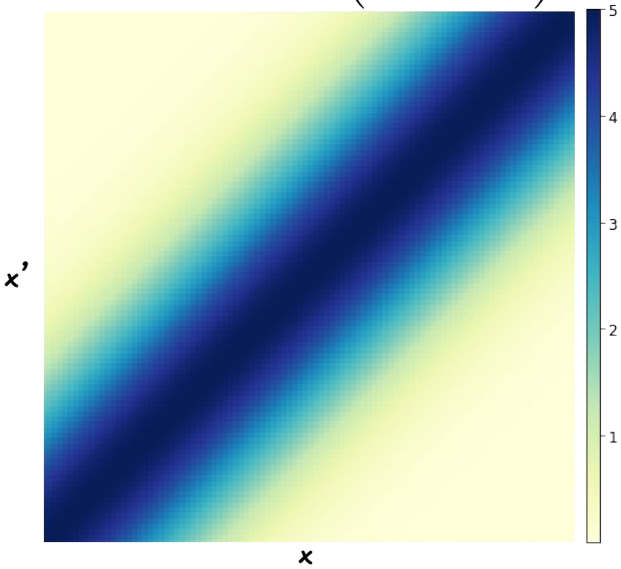
We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, **x** and **x'**, the kernel is a function measuring their similarity: ***K(x, x' |τ)*** with ***τ*** the vector of hyperparameters used to tune it. ***K*** close to 1 means similar inputs, ***K*** close to 0 means dissimilar inputs.

The most common used ***kernel*** function is, as you can guess, the RBF:

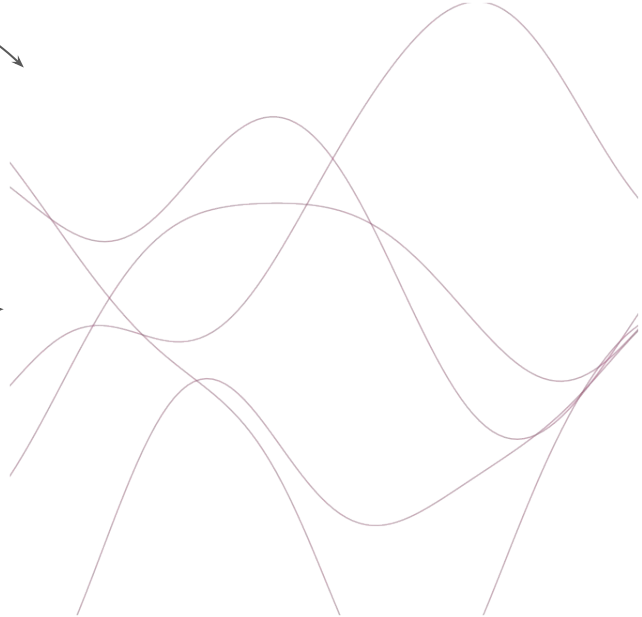
$$K(x, x'|\tau) = \sigma^2 \exp\left(-\frac{1}{2}\left(\frac{x-x'}{l}\right)^2\right)$$

"output scale": *determines the scale of the y axis. Increase it and the functions will span more of the vertical axes (notice the color limits on the left)*



RBF similarity

Producing these sample functions



How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

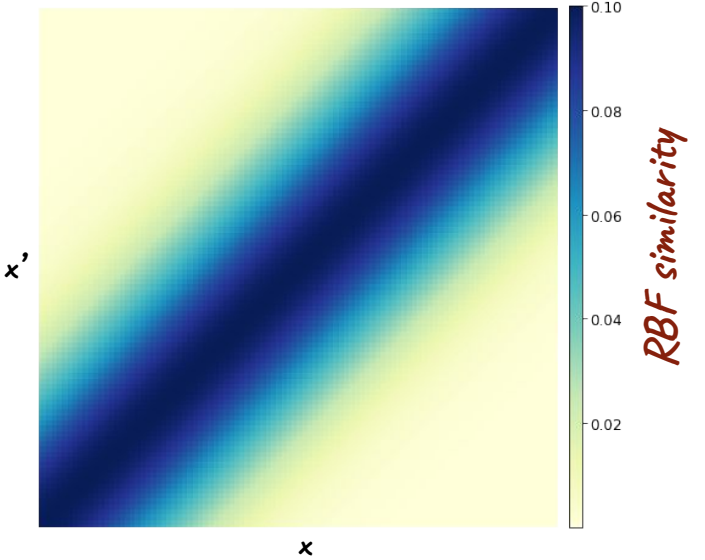
We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, **x** and **x'**, the kernel is a function measuring their similarity: ***K(x, x' |τ)*** with **τ** the vector of hyperparameters used to tune it. ***K*** close to 1 means similar inputs, ***K*** close to 0 means dissimilar inputs.

The most common used ***kernel*** function is, as you can guess, the RBF:

$$K(x, x'|\tau) = \sigma^2 \exp\left(-\frac{1}{2}\left(\frac{x-x'}{l}\right)^2\right)$$

"output scale": *determines the scale of the y axis. Decrease it and the functions will span less of the vertical axes (notice the color limits on the left).*



Producing these sample functions

How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?
We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.
A quick reminder: given two inputs, $\mathbf{x}$ and $\mathbf{x}'$, the kernel is a function measuring their similarity: $K(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau})$ with $\boldsymbol{\tau}$ the vector of hyperparameters used to tune it. K close to 1 means similar inputs, K close to 0 means dissimilar inputs.
There are lots of available ***kernels***, and the nice thing about them is that you can combine them to produce new ones.

$$K_c(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau}_c) = K_a(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau}_a) + K_b(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau}_b)$$

$$K_c(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau}_c) = K_a(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau}_a)\, K_b(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau}_b)$$

$$K(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau}) = cK(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau})$$

$$K(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau}) = g(\mathbf{x})K(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau})g(\mathbf{x}') \text{ for any function } g$$

$$K(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau}) = q(K(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau})) \text{ for positive coefficient polynomial } q$$

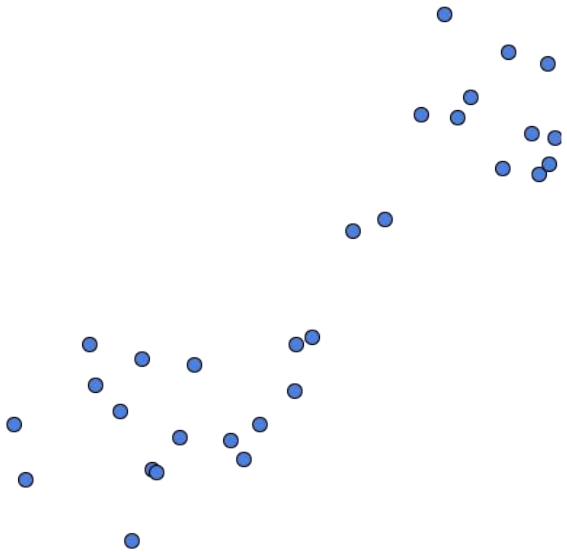
$$K(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau}) = \exp(K(\mathbf{x}, \mathbf{x}' / \boldsymbol{\tau}))$$

How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, $\mathbf{x}$ and $\mathbf{x}'$, the kernel is a function measuring their similarity: $K(\mathbf{x}, \mathbf{x}' \mid \boldsymbol{\tau})$ with $\boldsymbol{\tau}$ the vector of hyperparameters used to tune it. K close to 1 means similar inputs, K close to 0 means dissimilar inputs.

There are lots of available ***kernels***, and the nice thing about them is that you can combine them to produce new ones.

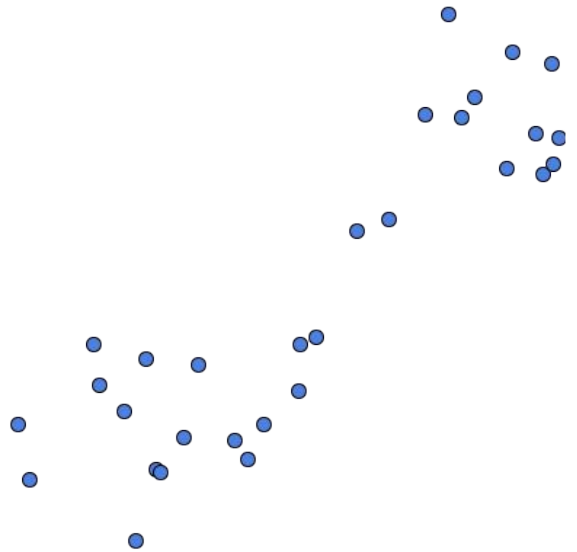


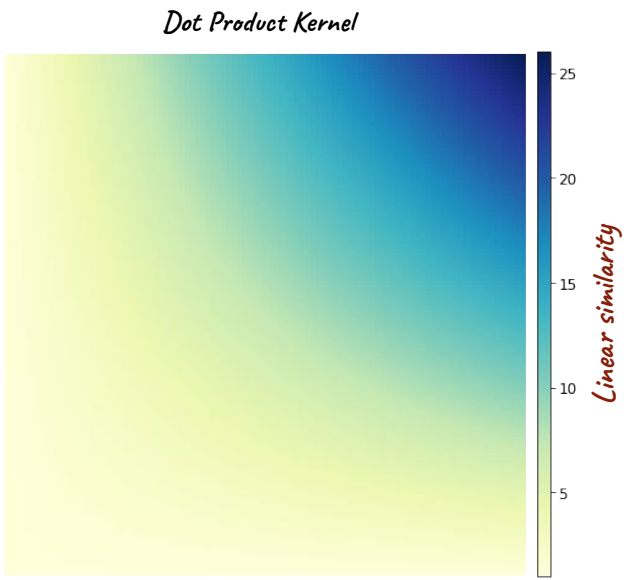
How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

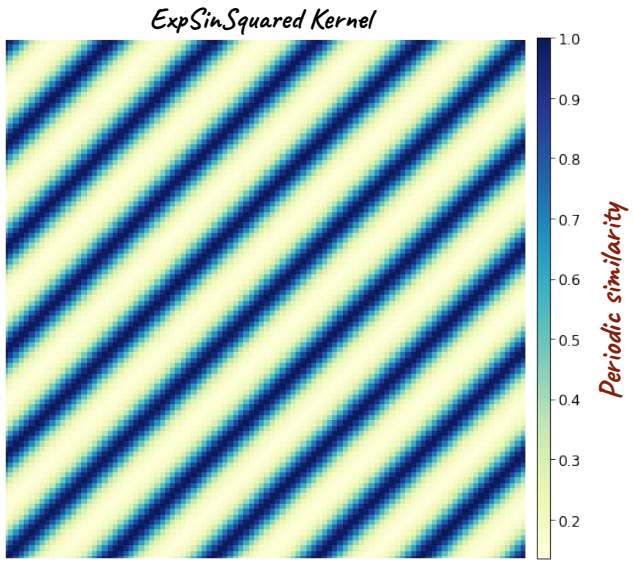
We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, x and x', the kernel is a function measuring their similarity: $K(x, x' | \tau)$ with τ the vector of hyperparameters used to tune it. K close to 1 means similar inputs, K close to 0 means dissimilar inputs.

There are lots of available ***kernels***, and the nice thing about them is that you can combine them to produce new ones.





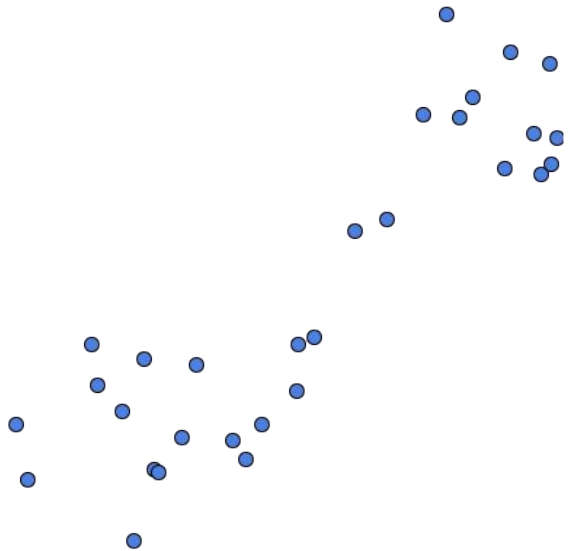


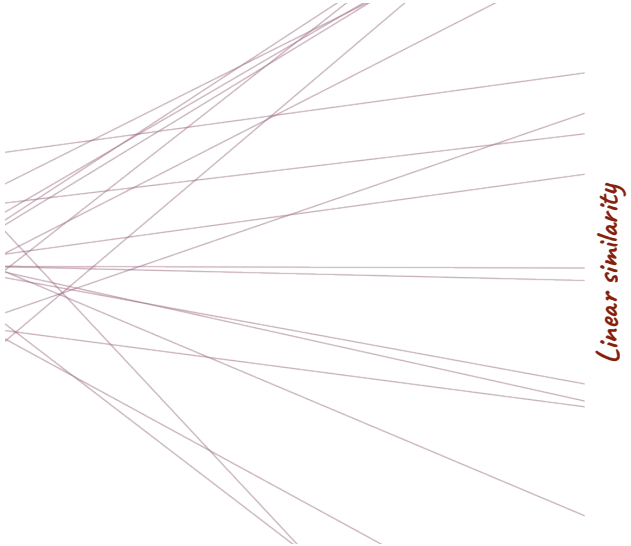
How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

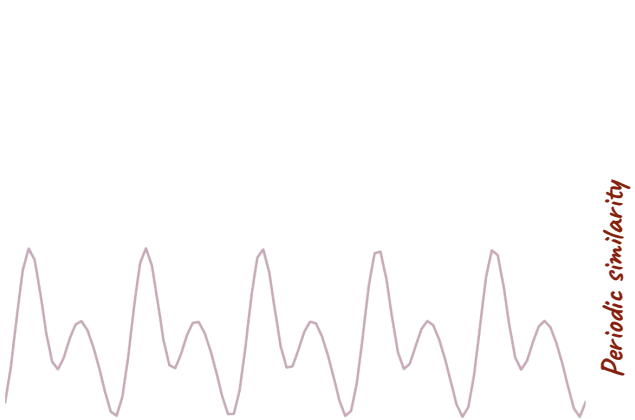
We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, x and x', the kernel is a function measuring their similarity: $K(x, x' | \tau)$ with τ the vector of hyperparameters used to tune it. K close to 1 means similar inputs, K close to 0 means dissimilar inputs.

There are lots of available ***kernels***, and the nice thing about them is that you can combine them to produce new ones.





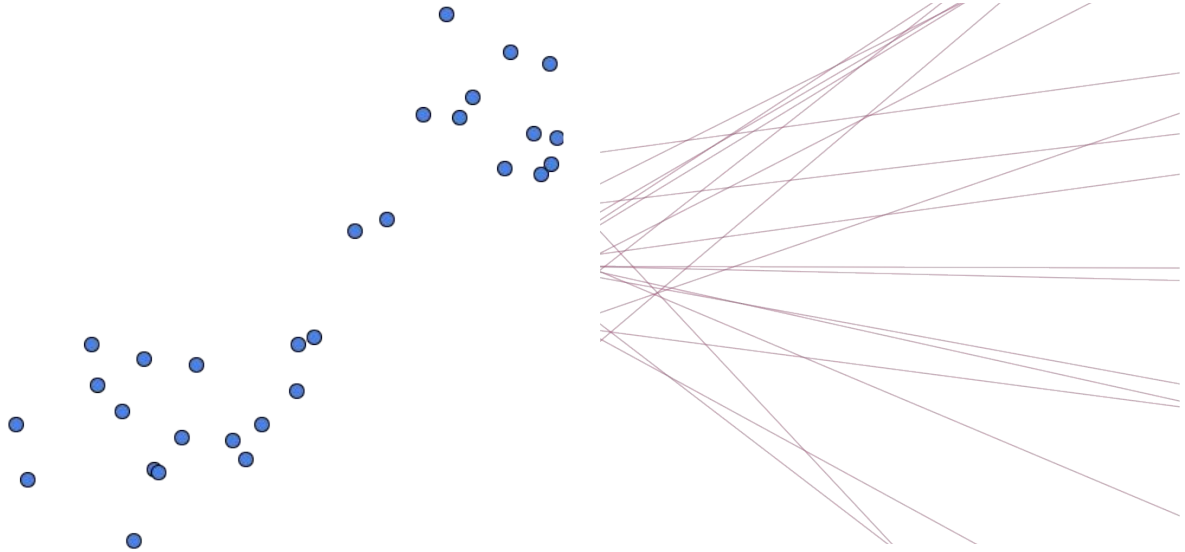


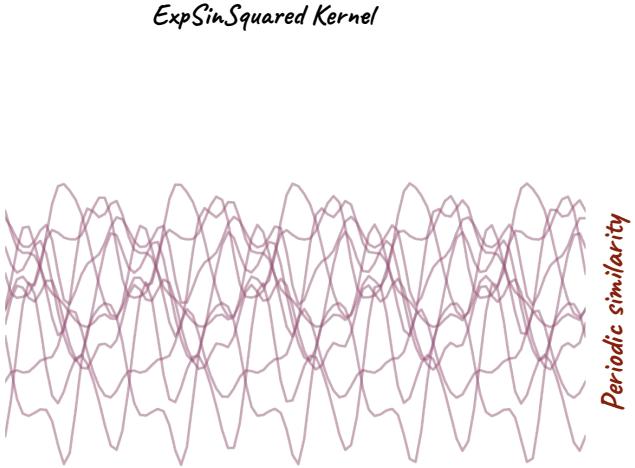
How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, x and x', the kernel is a function measuring their similarity: $K(x, x' | \tau)$ with τ the vector of hyperparameters used to tune it. K close to 1 means similar inputs, K close to 0 means dissimilar inputs.

There are lots of available ***kernels***, and the nice thing about them is that you can combine them to produce new ones.



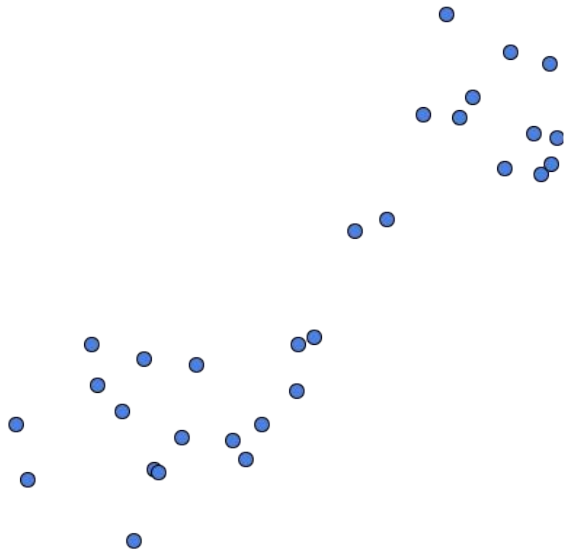


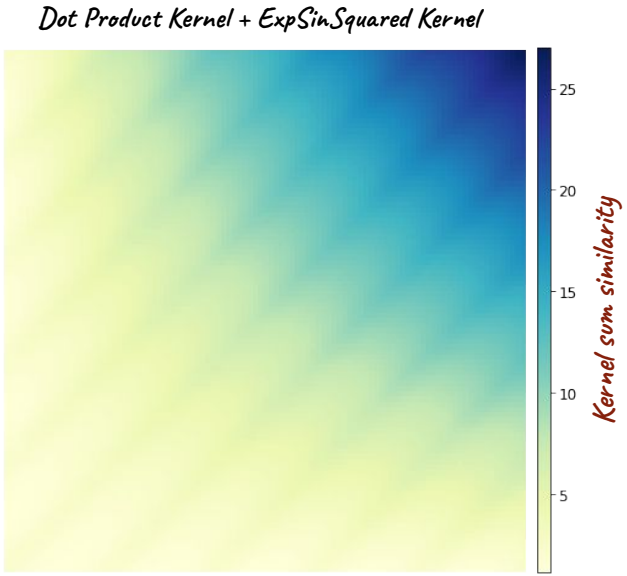
How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

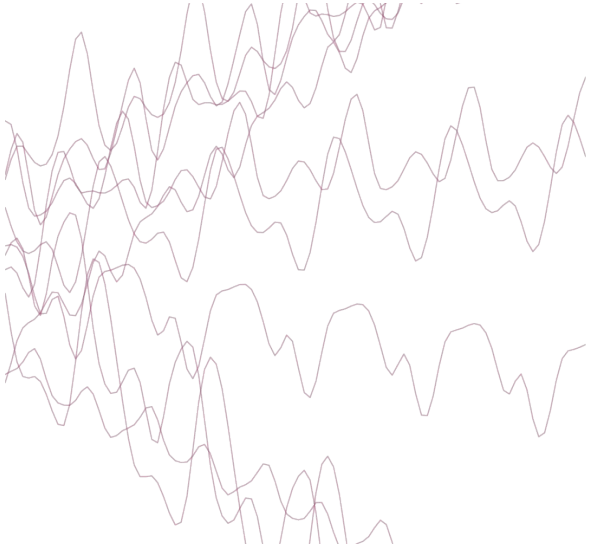
We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

A quick reminder: given two inputs, x and x', the kernel is a function measuring their similarity: $K(x, x' | \tau)$ with τ the vector of hyperparameters used to tune it. K close to 1 means similar inputs, K close to 0 means dissimilar inputs.

There are lots of available ***kernels***, and the nice thing about them is that you can combine them to produce new ones.





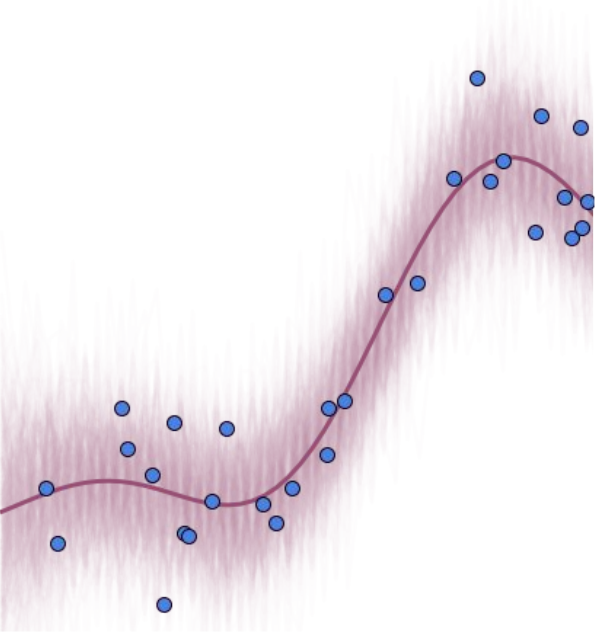


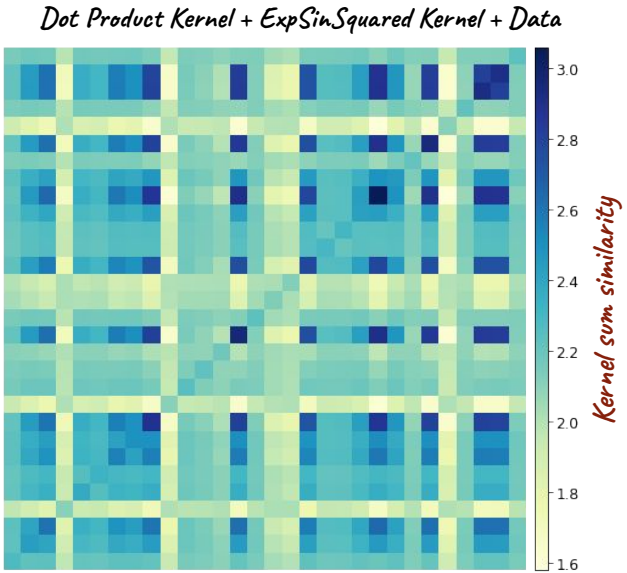
How to control the model? In other words, how can we determine which particular set of functions are more or less likely to be sampled by the particular dataset I'm trying to model?

We need a function that somehow measures the similarity between the input points in our data… ah, right, I've just showed it to you talking about SVMs: we need ***kernels***.

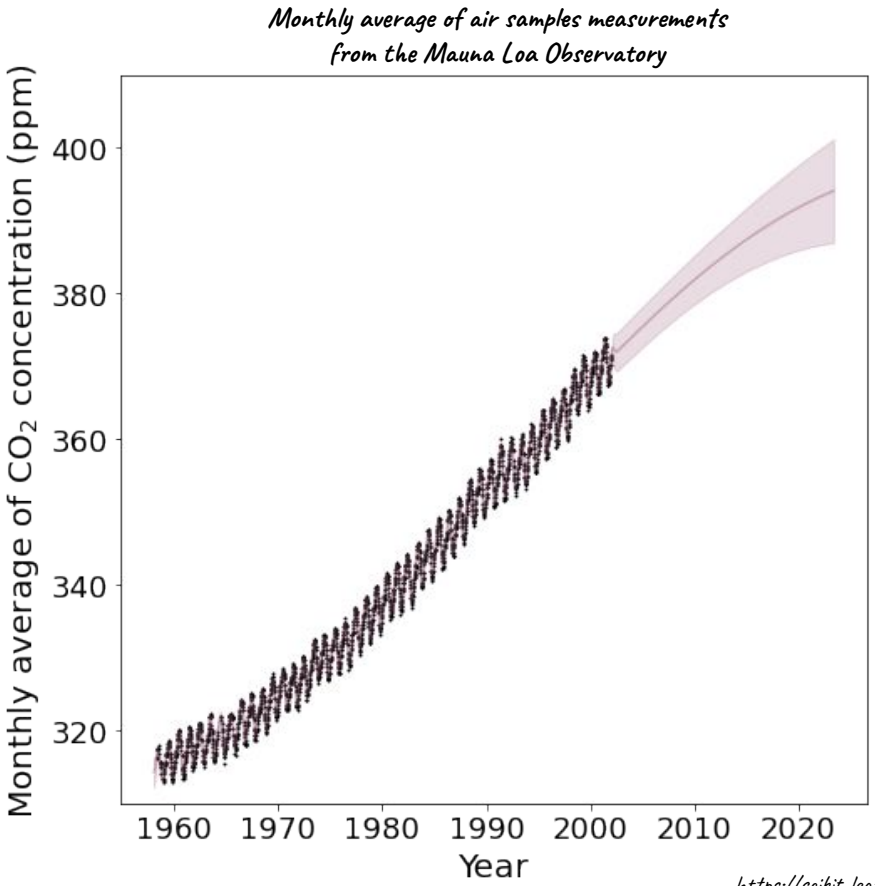
A quick reminder: given two inputs, x and x', the kernel is a function measuring their similarity: $K(x, x' | \tau)$ with τ the vector of hyperparameters used to tune it. K close to 1 means similar inputs, K close to 0 means dissimilar inputs.

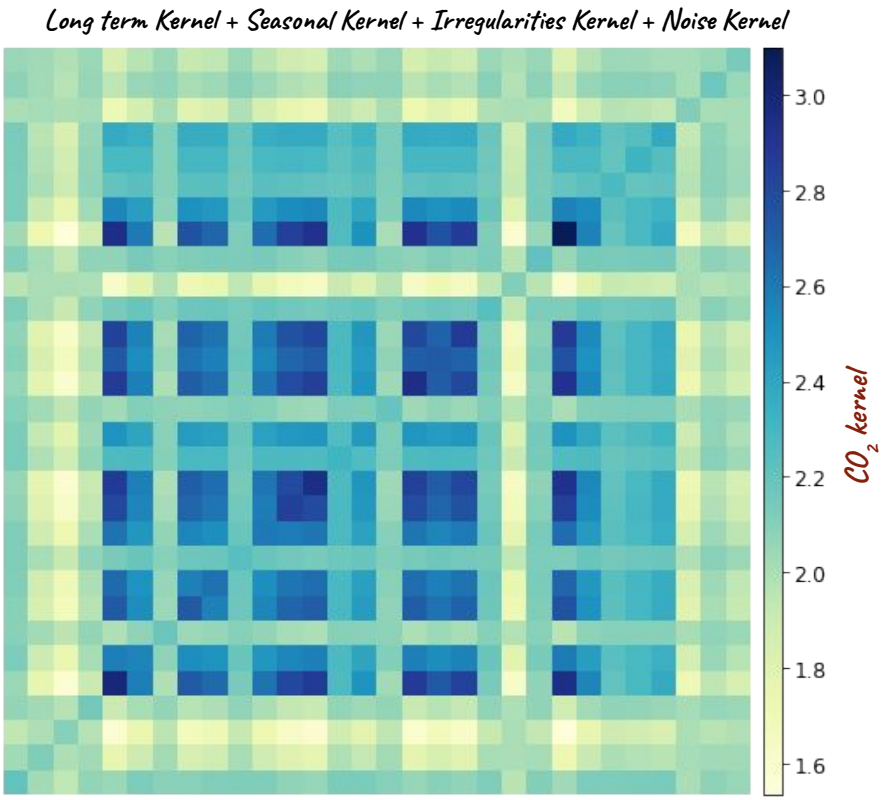
There are lots of available ***kernels***, and the nice thing about them is that you can combine them to produce new ones.





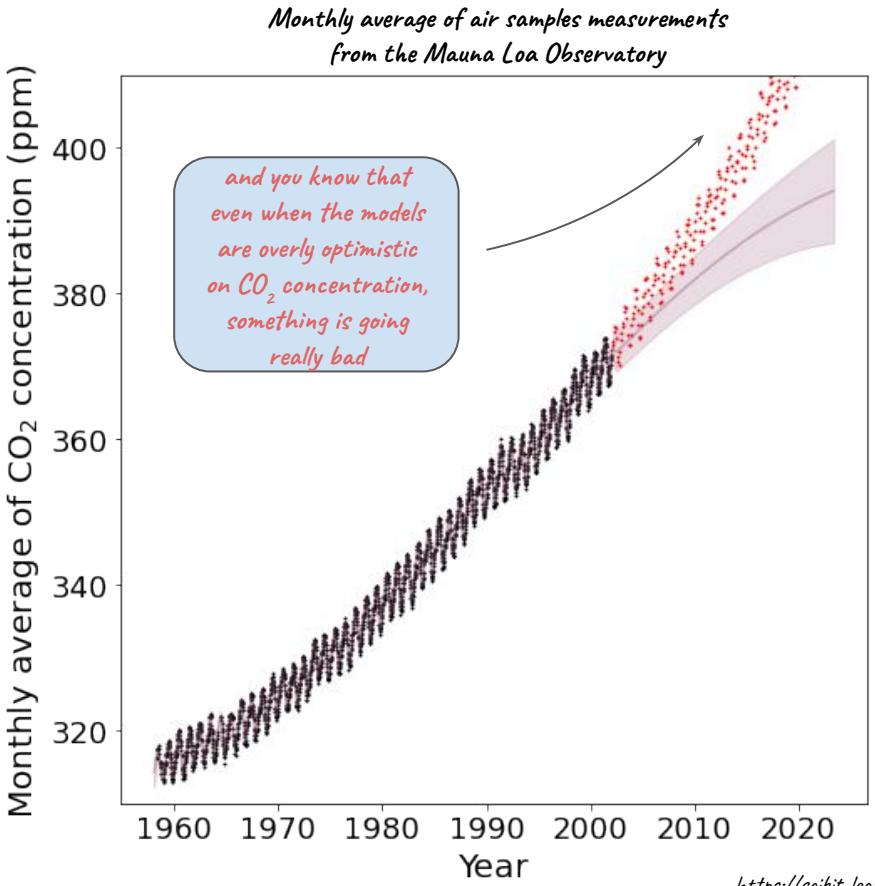
That was the quick and dirty example I've made for the slides. Here's a more sophisticated application you'll find in the notebooks.

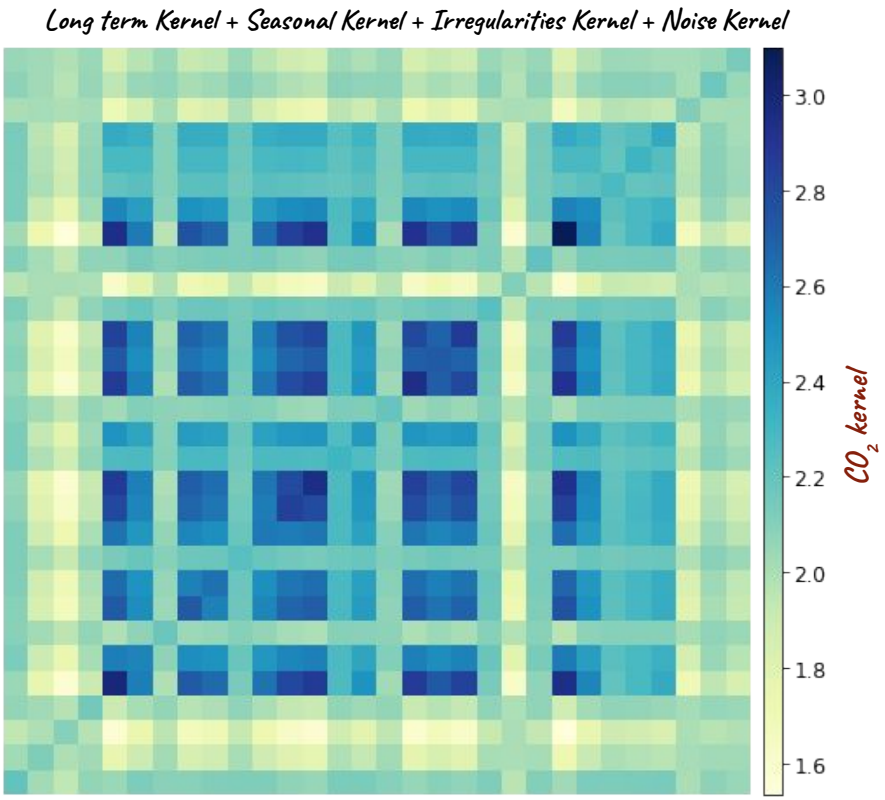




https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_co2.html#sphx-glr-auto-examples-gaussian-process-plot-gpr-co2-py

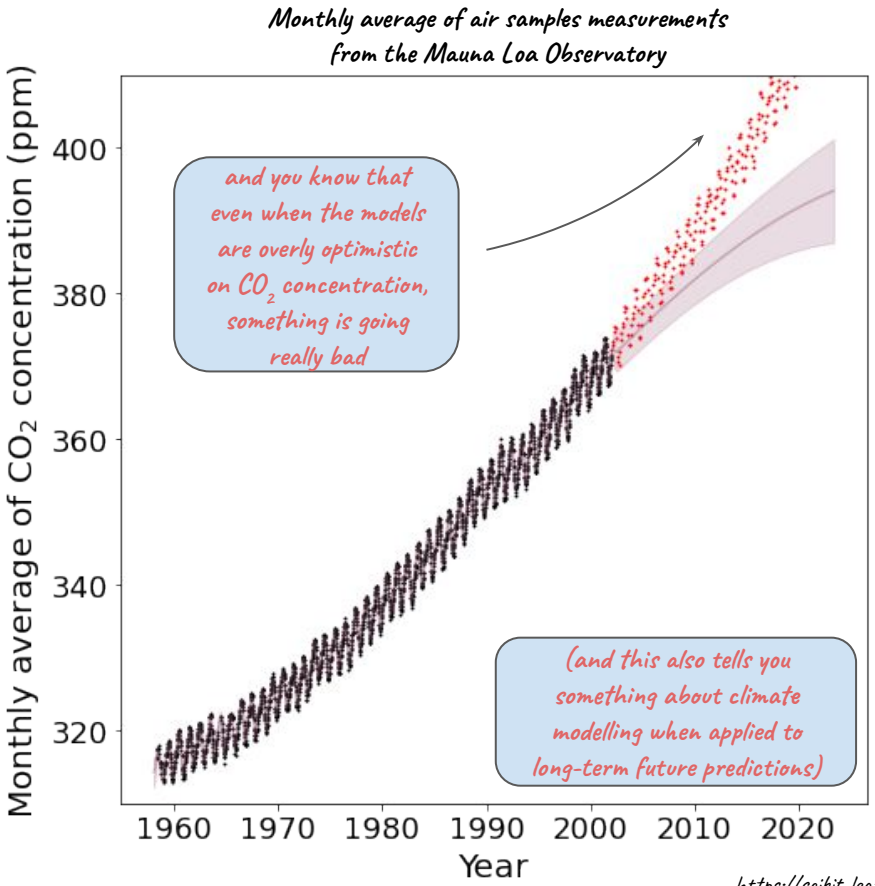
That was the quick and dirty example I've made for the slides. Here's a more sophisticated application you'll find in the notebooks.

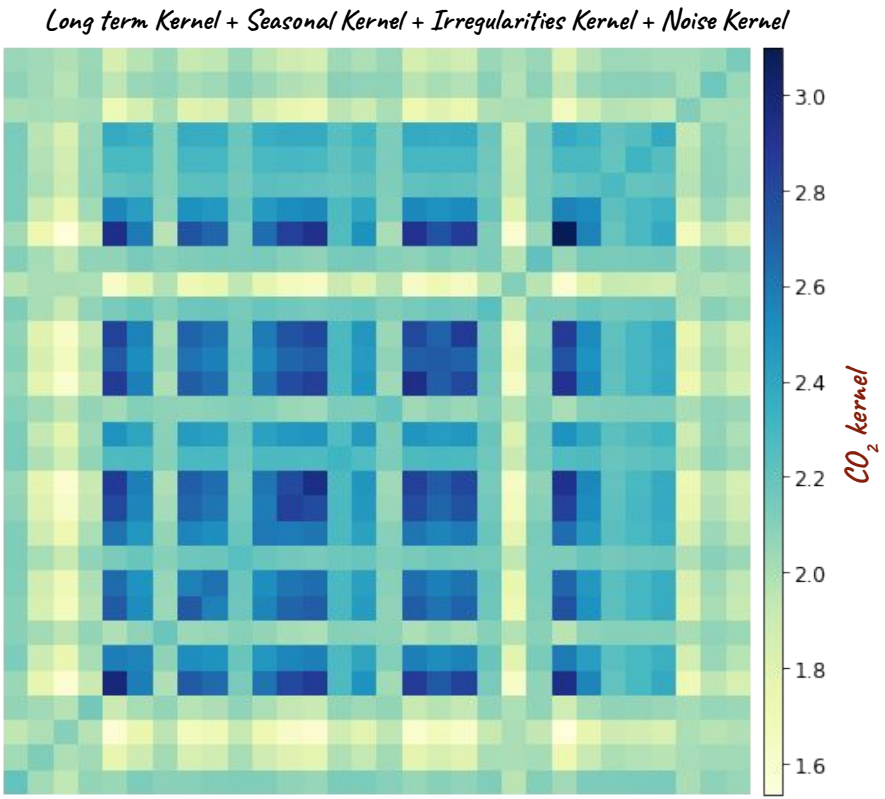




https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_co2.html#sphx-glr-auto-examples-gaussian-process-plot-gpr-co2-py

That was the quick and dirty example I've made for the slides. Here's a more sophisticated application you'll find in the notebooks.





https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_co2.html#sphx-glr-auto-examples-gaussian-process-plot-gpr-co2-py

What's the math behind GPs? (just a little bit, I swear)

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i$$

$$\varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i$$

$$\varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

Let's put everything into matrices and vectors:

X, X^* Matrices where each row is a vector input, for data (X) and test (X^*)

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i$$

$$\varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

Let's put everything into matrices and vectors:

X, X^*	Matrices where each row is a vector input, for data (X) and test (X^*)
y	Vector of all the observed outputs

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i \qquad \varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

Let's put everything into matrices and vectors:

X, X^*	Matrices where each row is a vector input, for data (X) and test (X^*)
y	Vector of all the observed outputs
f, f^*	Unobserved true functions outputs for all the inputs, data (f) and test (f^*)

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i$$

$$\varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

Let's put everything into matrices and vectors:

X, X^*	Matrices where each row is a vector input, for data (X) and test (X^*)
y	Vector of all the observed outputs
f, f^*	Unobserved true functions outputs for all the inputs, data (f) and test (f^*)
$K_{X,X}$	N by N matrix of all kernel similarities $K(x_i, x_j \mid \tau)$ for any two inputs

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i \qquad \varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

Let's put everything into matrices and vectors:

X, X^*	Matrices where each row is a vector input, for data (X) and test (X^*)
y	Vector of all the observed outputs
f, f^*	Unobserved true functions outputs for all the inputs, data (f) and test (f^*)
$K_{X^*,X^*}, K_{X^*,X}, K_{X,X^*}, K_{X,X}$	N by N matrix of all kernel similarities $K(x_i, x_j \mid \tau)$ for any two inputs

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i \qquad \varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

Let's put everything into matrices and vectors:

X, X^*	Matrices where each row is a vector input, for data (X) and test (X^*)
y	Vector of all the observed outputs
f, f^*	Unobserved true functions outputs for all the inputs, data (f) and test (f^*)
$K_{X^*,X^*}, K_{X^*,X}, K_{X,X^*}, K_{X,X}$	N by N matrix of all kernel similarities $K(x_i, x_j / \tau)$ for any two inputs

Gaussian Process Assumption

we assume that the observations y and the true function outputs at our test points f^* are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right) \qquad \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i \qquad \varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

Let's put everything into matrices and vectors:

X, X^*	Matrices where each row is a vector input, for data (X) and test (X^*)
y	Vector of all the observed outputs
f, f^*	Unobserved true functions outputs for all the inputs, data (f) and test (f^*)
$K_{X^*,X^*}, K_{X^*,X}, K_{X,X^*}, K_{X,X}$	N by N matrix of all kernel similarities $K(x_i, x_j / \tau)$ for any two inputs

Gaussian Process Assumption

we assume that the observations y and the true function outputs at our test points f^* are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\boxed{\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix}} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right)$$

$$\hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

Paste together the observed y and the unobserved true function output at out test points into a long vector

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i \qquad \varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

Let's put everything into matrices and vectors:

X, X^*	Matrices where each row is a vector input, for data (X) and test (X^*)
y	Vector of all the observed outputs
f, f^*	Unobserved true functions outputs for all the inputs, data (f) and test (f^*)
$K_{X^*,X^*}, K_{X^*,X}, K_{X,X^*}, K_{X,X}$	N by N matrix of all kernel similarities $K(x_i, x_j / \tau)$ for any two inputs

Gaussian Process Assumption

we assume that the observations y and the true function outputs at our test points f^* are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right)$$

$$\hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

Paste together the observed y and the unobserved true function output at out test points into a long vector

This thing is distributed as a multivariate normal with mean zero and that particular covariance matrix

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i \qquad \varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

Let's put everything into matrices and vectors:

X, X^*	Matrices where each row is a vector input, for data (X) and test (X^*)
y	Vector of all the observed outputs
f, f^*	Unobserved true functions outputs for all the inputs, data (f) and test (f^*)
$K_{X^*,X^*}, K_{X^*,X}, K_{X,X^*}, K_{X,X}$	N by N matrix of all kernel similarities $K(x_i, x_j / \tau)$ for any two inputs

Gaussian Process Assumption

we assume that the observations y and the true function outputs at our test points f^* are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right)$$

$$\hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

Paste together the observed y and the unobserved true function output at out test points into a long vector

This thing is distributed as a multivariate normal with mean zero and that particular covariance matrix

That covariance matrix is made up of our kernel similarities between all inputs

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i \qquad \varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

Let's put everything into matrices and vectors:

X, X^*	Matrices where each row is a vector input, for data (X) and test (X^*)
y	Vector of all the observed outputs
f, f^*	Unobserved true functions outputs for all the inputs, data (f) and test (f^*)
$K_{X^*,X^*}, K_{X^*,X}, K_{X,X^*}, K_{X,X}$	N by N matrix of all kernel similarities $K(x_i, x_j / \tau)$ for any two inputs

Gaussian Process Assumption

we assume that the observations y and the true function outputs at our test points f^* are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right)$$

Paste together the observed y and the unobserved true function output at out test points into a long vector

This thing is distributed as a multivariate normal with mean zero and that particular covariance matrix

$$\hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

For the submatrix governing y, we add the noise variance to the diagonal

What's the math behind GPs? (just a little bit, I swear)

Data: $D = \{(x_i, y_i)\}$ with $i = 1 \dots N$

Task: provide predictive distributions for M test inputs $\{x_j^*\}$ with $j = 1 \dots M$

Let's assume our y's are noisy observations of some true function:

$$f(x): y_i = f(x_i) + \varepsilon_i \qquad \varepsilon_i \sim N(0, \sigma_\varepsilon^2)$$

means normal distribution centred on 0 and with variance σ_ε, which is another model hyperparameter

Let's put everything into matrices and vectors:

X, X^*	Matrices where each row is a vector input, for data (X) and test (X^*)
y	Vector of all the observed outputs
f, f^*	Unobserved true functions outputs for all the inputs, data (f) and test (f^*)
$K_{X^*,X^*}, K_{X^*,X}, K_{X,X^*}, K_{X,X}$	N by N matrix of all kernel similarities $K(x_i, x_j / \tau)$ for any two inputs

Gaussian Process Assumption

we assume that the observations y and the true function outputs at our test points f^* are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right)$$

$$\hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

Assuming that the covariance matrix is just the similarities of x's has the effect of smoothing the whole models (which is common in ML)

Gaussian Process Assumption

we assume that the observations *y* and the true function outputs at our test points *f are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:**

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right)$$

$$\hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

How is the data factored in? If a vector is normally distributed, and we observe it partially, Bayesian inference will tell us the distribution over the unobserved elements given the observed ones.

Gaussian Process Assumption

we assume that the observations y and the true function outputs at our test points f^* are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right) \qquad \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

How is the data factored in? If a vector is normally distributed, and we observe it partially, Bayesian inference will tell us the distribution over the unobserved elements given the observed ones.

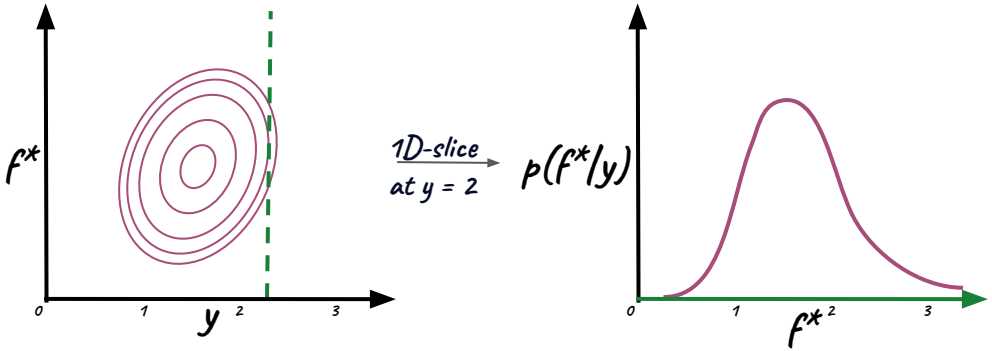
Specifically, given our data $\mathcal{D}$ and our test inputs X^*, the distribution over the true function vector f^* is another multivariate Normal:

$$\mathbf{f}^* | \mathbf{X}^*, \mathcal{D} \sim \mathcal{N}(\mu_{f^*}, \Sigma_{f^*})$$

$$\mu_{f^*} = \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} \mathbf{K}_{\mathbf{X},\mathbf{X}}^{-1} \mathbf{y}$$

$$\Sigma_{f^*} = \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} - \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} \mathbf{K}_{\mathbf{X},\mathbf{X}}^{-1} \mathbf{K}_{\mathbf{X},\mathbf{X}^*}$$

This is called *conditioning*. Let's see a simple example 1D example:



Gaussian Process Assumption

we assume that the observations *y* and the true function outputs at our test points *f are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:**

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f*} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right) \qquad \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

How is the data factored in? If a vector is normally distributed, and we observe it partially, Bayesian inference will tell us the distribution over the unobserved elements given the observed ones.

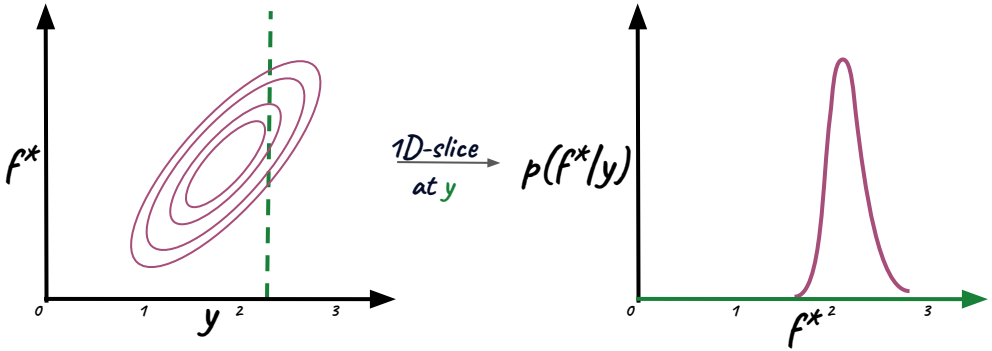
Specifically, given our data $\mathcal{D}$ and our test inputs X^*, the distribution over the true function vector f^* is another multivariate Normal:

$$\mathbf{f*}|\mathbf{X*}, \mathcal{D} \sim \mathcal{N}(\mu_{f^*}, \Sigma_{f^*})$$

$$\mu_{f^*} = \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} \mathbf{K}_{\mathbf{X},\mathbf{X}}^{-1} \mathbf{y}$$

$$\Sigma_{f^*} = \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} - \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} \mathbf{K}_{\mathbf{X},\mathbf{X}}^{-1} \mathbf{K}_{\mathbf{X},\mathbf{X}^*}$$

This is called *conditioning*. Let's see a simple example 1D example:



Gaussian Process Assumption

we assume that the observations *y* and the true function outputs at our test points *f are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:**

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f*} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right) \qquad \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

How is the data factored in? If a vector is normally distributed, and we observe it partially, Bayesian inference will tell us the distribution over the unobserved elements given the observed ones.

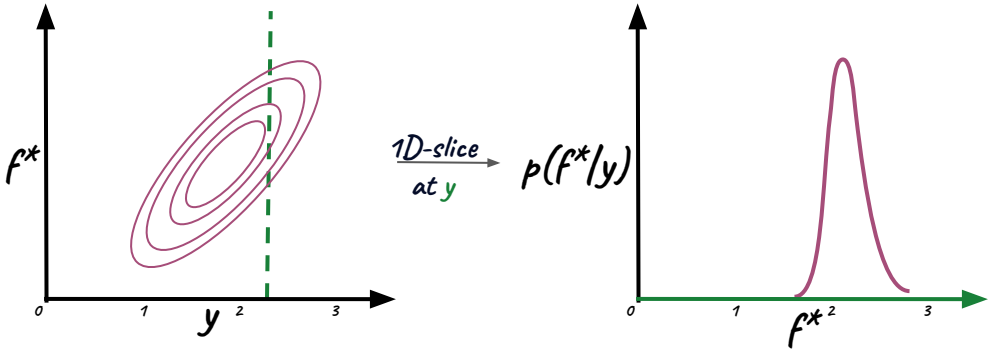
Specifically, given our data $\mathcal{D}$ and our test inputs X^*, the distribution over the true function vector f^* is another multivariate Normal:

$$\mathbf{f*}|\mathbf{X*}, \mathcal{D} \sim \mathcal{N}(\mu_{f^*}, \Sigma_{f^*})$$

$$\mu_{f^*} = \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} \mathbf{K}_{\mathbf{X},\mathbf{X}}^{-1} \mathbf{y}$$

$$\Sigma_{f^*} = \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} - \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} \mathbf{K}_{\mathbf{X},\mathbf{X}}^{-1} \mathbf{K}_{\mathbf{X},\mathbf{X}^*}$$

This is called *conditioning*. Let's see a simple example 1D example:



So, all this expression is doing is slicing a multivariate normal on multiple dimension, something that is computationally feasible on a Normal, but harder or impossible on a generic multivariate.

Gaussian Process Assumption

we assume that the observations *y* and the true function outputs at our test points *f
are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right)$$

$$\hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

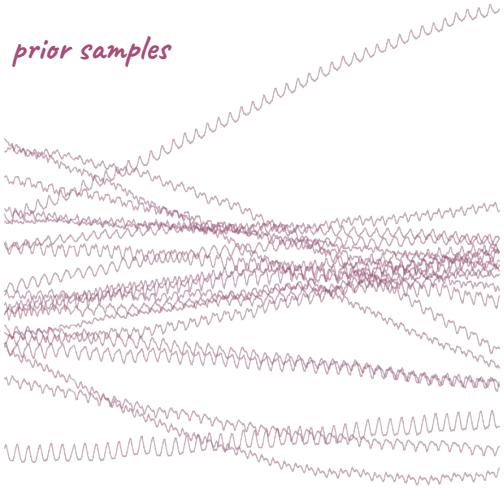
Let's wrap it up.

Gaussian Process Assumption

we assume that the observations *y* and the true function outputs at our test points *f
are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right) \qquad \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

Let's wrap it up.
The GP assumption gives us our *prior samples*. Each function comes from drawing a sample from the Normal (here with the CO_2 kernel).

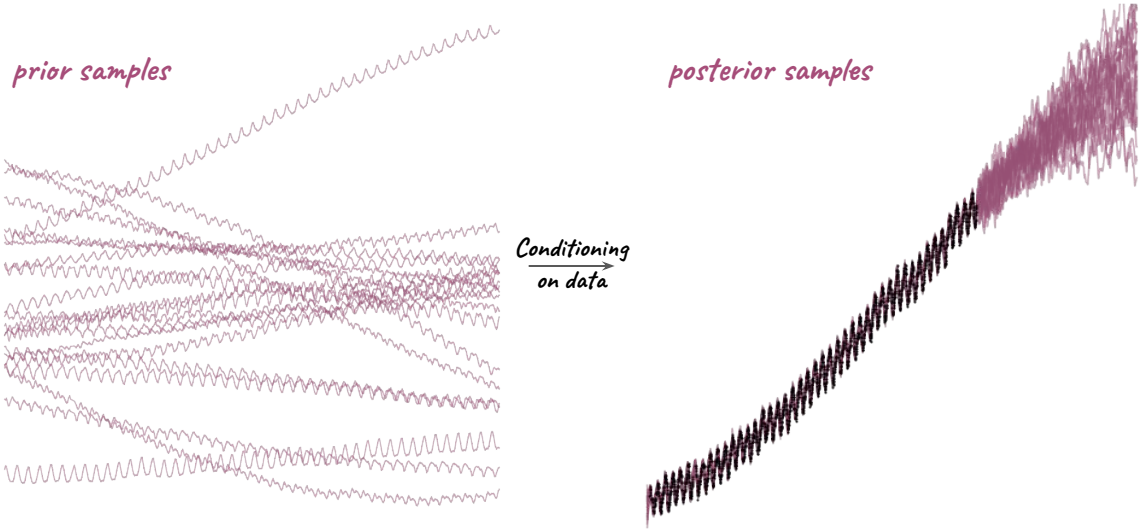


Gaussian Process Assumption

we assume that the observations *y* and the true function outputs at our test points *f
are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right) \qquad \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

Let's wrap it up.
The GP assumption gives us our *prior samples*. Each function comes from drawing a sample from the Normal (here with the CO_2 kernel).
To get the *posterior samples*, we just sample according to the Normal, after conditioning on the data.



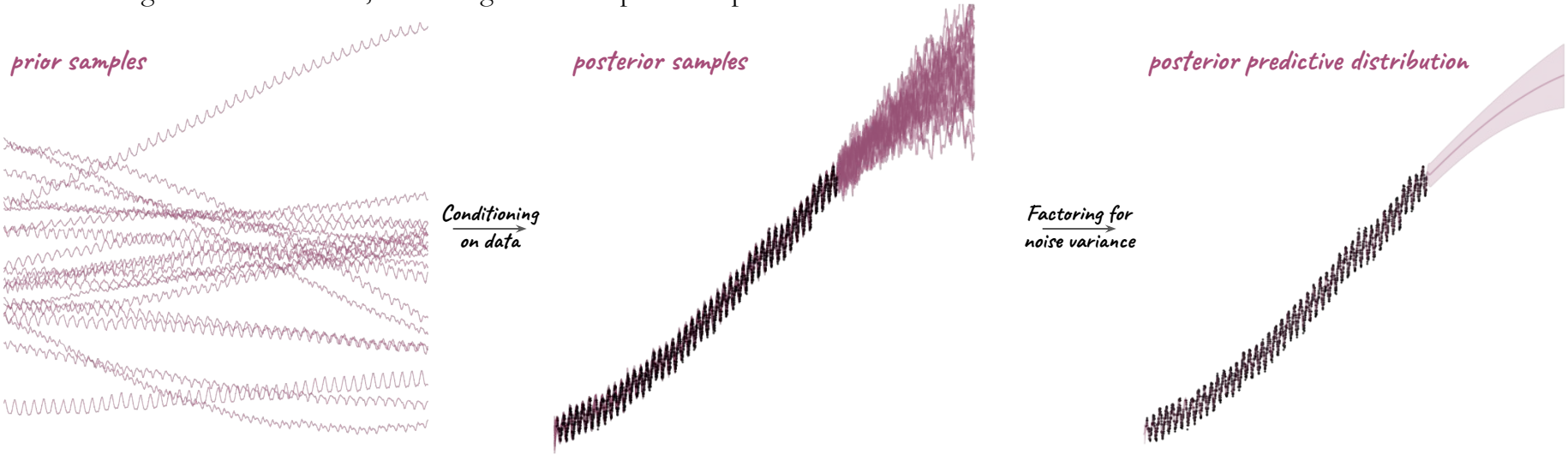
That's a wrap!

Gaussian Process Assumption

we assume that the observations *y* and the true function outputs at our test points *f
are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right) \qquad \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

Let's wrap it up.
The GP assumption gives us our *prior samples*. Each function comes from drawing a sample from the Normal (here with the CO_2 kernel).
To get the *posterior samples*, we just sample according to the Normal, after conditioning on the data.
Once factoring for noise variance, this also gives us the posterior predictive distribution.



Gaussian Process Assumption

we assume that the observations *y* and the true function outputs at our test points *f
are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right) \qquad \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

Let's wrap it up.
The GP assumption gives us our *prior samples*. Each function comes from drawing a sample from the Normal (here with the CO_2 kernel).
To get the *posterior samples*, we just sample according to the Normal, after conditioning on the data.
Once factoring for noise variance, this also gives us the posterior predictive distribution.
How about the kernel hyperparameters ***τ*** and the noise variance σ_ϵ^2?
Those are picked by maximizing the log-likelihood of ***y*** after integrating out the possible functions:

$$\log p(\mathbf{y}|\mathbf{X}, \tau, \sigma_{\epsilon^2}) = \log \int p(\mathbf{y}|\hat{\mathbf{f}}, \sigma_{\epsilon^2}) p(\hat{\mathbf{f}}|\mathbf{X}, \tau) d\hat{f} = \log \mathcal{N}(\mathbf{y}|0, \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}})$$

scikit-learn does that for you, don't worry

Gaussian Process Assumption

we assume that the observations *y* and the true function outputs at our test points *f are jointly distributed as an (N+M)-dimensional multivariate Normal distribution:**

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} & \mathbf{K}_{\mathbf{X},\mathbf{X}^*} \\ \mathbf{K}_{\mathbf{X}^*,\mathbf{X}} & \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} \end{bmatrix} \right) \qquad \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}} = \mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma_\epsilon^2 \mathbb{I}$$

Let's wrap it up.
The GP assumption gives us our *prior samples*. Each function comes from drawing a sample from the Normal (here with the CO_2 kernel).
To get the *posterior samples*, we just sample according to the Normal, after conditioning on the data.
Once factoring for noise variance, this also gives us the posterior predictive distribution.
How about the kernel hyperparameters τ and the noise variance σ_ϵ^2?
Those are picked by maximizing the log-likelihood of *y* after integrating out the possible functions:

$$\log p(\mathbf{y}|\mathbf{X}, \tau, \sigma_{\epsilon^2}) = \log \int p(\mathbf{y}|\hat{\mathbf{f}}, \sigma_{\epsilon^2}) p(\hat{\mathbf{f}}|\mathbf{X}, \tau) d\hat{f} = \log \mathcal{N}(\mathbf{y}|0, \hat{\mathbf{K}}_{\mathbf{X},\mathbf{X}})$$

scikit-learn does that for you, don't worry

Drawbacks of GPs:

- inverting $K_{X,X}$ takes $O(N^2)$ time, which makes GPs slow for large values of N
- designing the kernel require **considerable** work, it's not out-of-the box
- for certain kernels (e.g. RBFs), prediction might break down in high dimensional feature spaces

Decision trees are the building block of random forest, ensemble methods and gradient boosted algorithms, three terms that "*squeeze squeeze*" ontologically refer to the same thing. As all the best things in life, they are extremely simple and yet powerful: gradient boosted algorithms are the lone ML technique that is able to compete with (deep) Neural Networks in terms of performances.
Let's say that we have this dataset of crucial astrophysical importance:

Decision trees are the building block of random forest, ensemble methods and gradient boosted algorithms, three terms that "*squeeze squeeze*" ontologically refer to the same thing. As all the best things in life, they are extremely simple and yet powerful: gradient boosted algorithms are the lone ML technique that is able to compete with (deep) Neural Networks in terms of performances.
Let's say that we have this dataset of crucial astrophysical importance:

Brutally taken from Andrew Ng's Machine Learning Course, please Andrew don't sue me, big fan btw.

Decision trees are the building block of random forest, ensemble methods and gradient boosted algorithms, three terms that "*squeeze squeeze*" ontologically refer to the same thing. As all the best things in life, they are extremely simple and yet powerful: gradient boosted algorithms are the lone ML technique that is able to compete with (deep) Neural Networks in terms of performances.
Let's say that we have this dataset of crucial astrophysical importance:

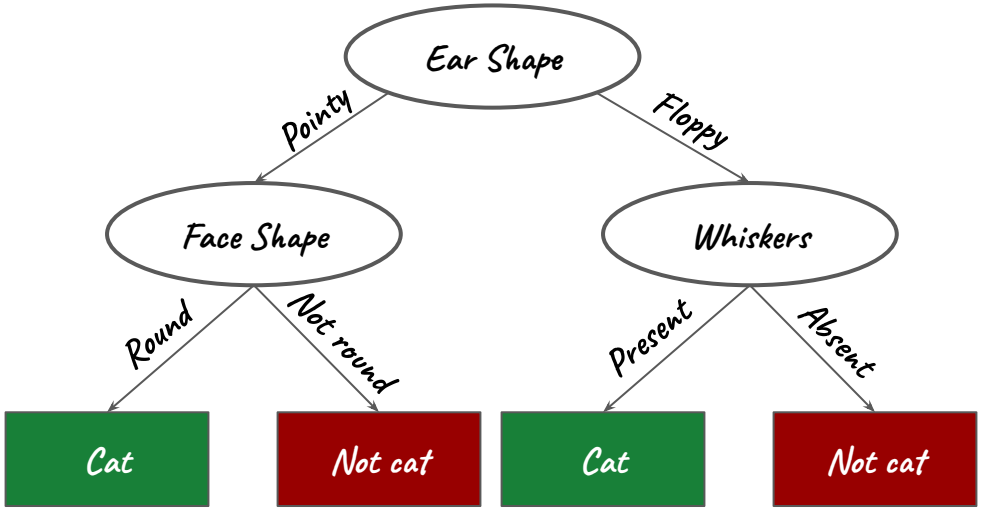
	Ear Shape	Face Shape	Whiskers	Cat?
	Pointy	Round	Present	Yes
	Floppy	Not round	Present	Yes
	Floppy	Round	Absent	No
	Pointy	Not round	Present	No
	Pointy	Round	Present	Yes
	Pointy	Round	Absent	Yes
	Floppy	Not round	Absent	No
	Pointy	Round	Absent	Yes
	Floppy	Round	Absent	No
	Floppy	Round	Absent	No

The Sloan Digital Cats Survey – III, with five cats and five dogs.
To keep it simple, we have only categorical features and label.
The question is: can we **train** a model to automatically classify a new example as cat or dog based on its observed features?

Decision trees are the building block of random forest, ensemble methods and gradient boosted algorithms, three terms that "*squeeze squeeze*" ontologically refer to the same thing. As all the best things in life, they are extremely simple and yet powerful: gradient boosted algorithms are the lone ML technique that is able to compete with (deep) Neural Networks in terms of performances.
Let's say that we have this dataset of crucial astrophysical importance:

	Ear Shape	Face Shape	Whiskers	Cat?
	Pointy	Round	Present	Yes
	Floppy	Not round	Present	Yes
	Floppy	Round	Absent	No
	Pointy	Not round	Present	No
	Pointy	Round	Present	Yes
	Pointy	Round	Absent	Yes
	Floppy	Not round	Absent	No
	Pointy	Round	Absent	Yes
	Floppy	Round	Absent	No
	Floppy	Round	Absent	No

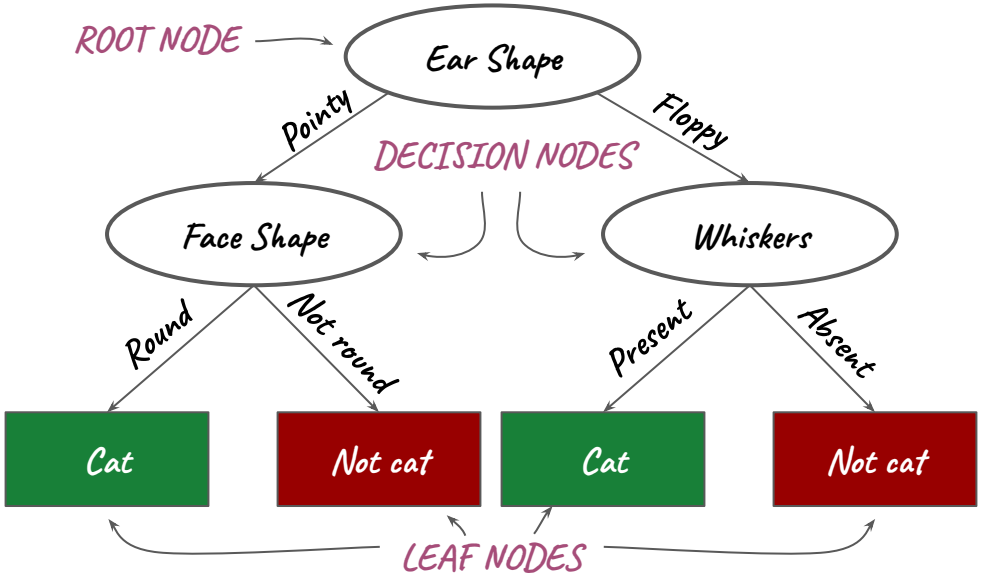
The Sloan Digital Cats Survey – III, with five cats and five dogs.
To keep it simple, we have only categorical features and label.
The question is: can we **train** a model to automatically classify a new example as cat or dog based on its observed features?



Decision trees are the building block of random forest, ensemble methods and gradient boosted algorithms, three terms that "*squeeze squeeze*" ontologically refer to the same thing. As all the best things in life, they are extremely simple and yet powerful: gradient boosted algorithms are the lone ML technique that is able to compete with (deep) Neural Networks in terms of performances.
Let's say that we have this dataset of crucial astrophysical importance:

	Ear Shape	Face Shape	Whiskers	Cat?
	Pointy	Round	Present	Yes
	Floppy	Not round	Present	Yes
	Floppy	Round	Absent	No
	Pointy	Not round	Present	No
	Pointy	Round	Present	Yes
	Pointy	Round	Absent	Yes
	Floppy	Not round	Absent	No
	Pointy	Round	Absent	Yes
	Floppy	Round	Absent	No
	Floppy	Round	Absent	No

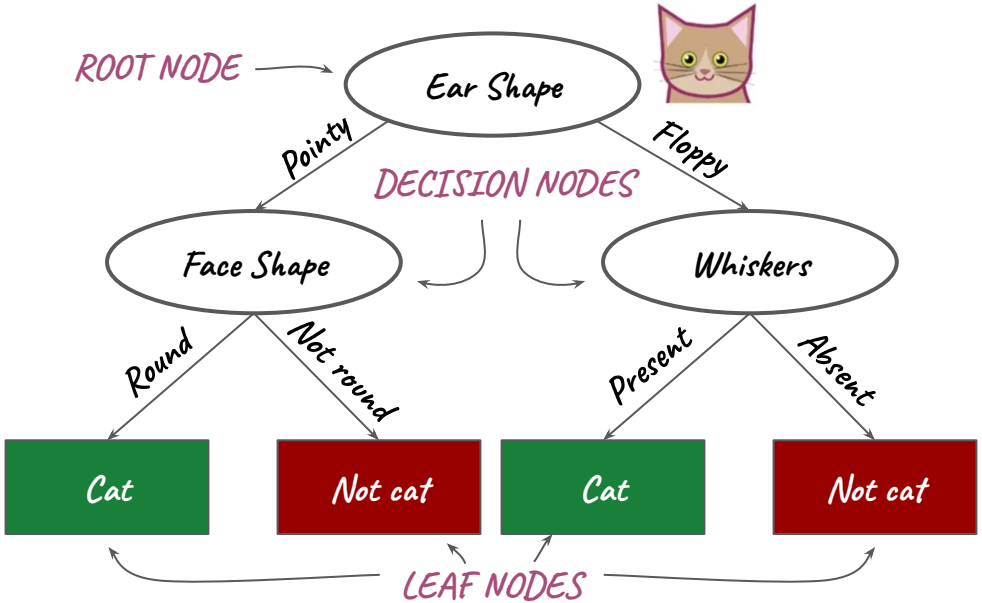
The Sloan Digital Cats Survey – III, with five cats and five dogs.
To keep it simple, we have only categorical features and label.
The question is: can we **train** a model to automatically classify a new example as cat or dog based on its observed features?



Decision trees are the building block of random forest, ensemble methods and gradient boosted algorithms, three terms that "*squeeze squeeze*" ontologically refer to the same thing. As all the best things in life, they are extremely simple and yet powerful: gradient boosted algorithms are the lone ML technique that is able to compete with (deep) Neural Networks in terms of performances.
Let's say that we have this dataset of crucial astrophysical importance:

	Ear Shape	Face Shape	Whiskers	Cat?
	Pointy	Round	Present	Yes
	Floppy	Not round	Present	Yes
	Floppy	Round	Absent	No
	Pointy	Not round	Present	No
	Pointy	Round	Present	Yes
	Pointy	Round	Absent	Yes
	Floppy	Not round	Absent	No
	Pointy	Round	Absent	Yes
	Floppy	Round	Absent	No
	Floppy	Round	Absent	No

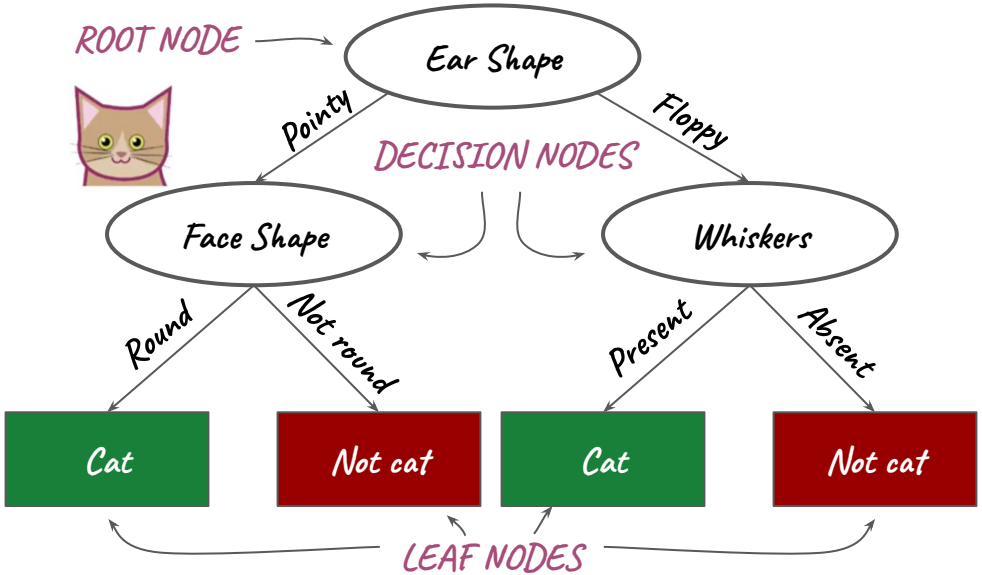
The Sloan Digital Cats Survey – III, with five cats and five dogs.
To keep it simple, we have only categorical features and label.
If a new example arrives, you just roll it down the tree, and see in which leaf it lands: that's the predicted label.



Decision trees are the building block of random forest, ensemble methods and gradient boosted algorithms, three terms that "*squeeze squeeze*" ontologically refer to the same thing. As all the best things in life, they are extremely simple and yet powerful: gradient boosted algorithms are the lone ML technique that is able to compete with (deep) Neural Networks in terms of performances.
Let's say that we have this dataset of crucial astrophysical importance:

	Ear Shape	Face Shape	Whiskers	Cat?
	Pointy	Round	Present	Yes
	Floppy	Not round	Present	Yes
	Floppy	Round	Absent	No
	Pointy	Not round	Present	No
	Pointy	Round	Present	Yes
	Pointy	Round	Absent	Yes
	Floppy	Not round	Absent	No
	Pointy	Round	Absent	Yes
	Floppy	Round	Absent	No
	Floppy	Round	Absent	No

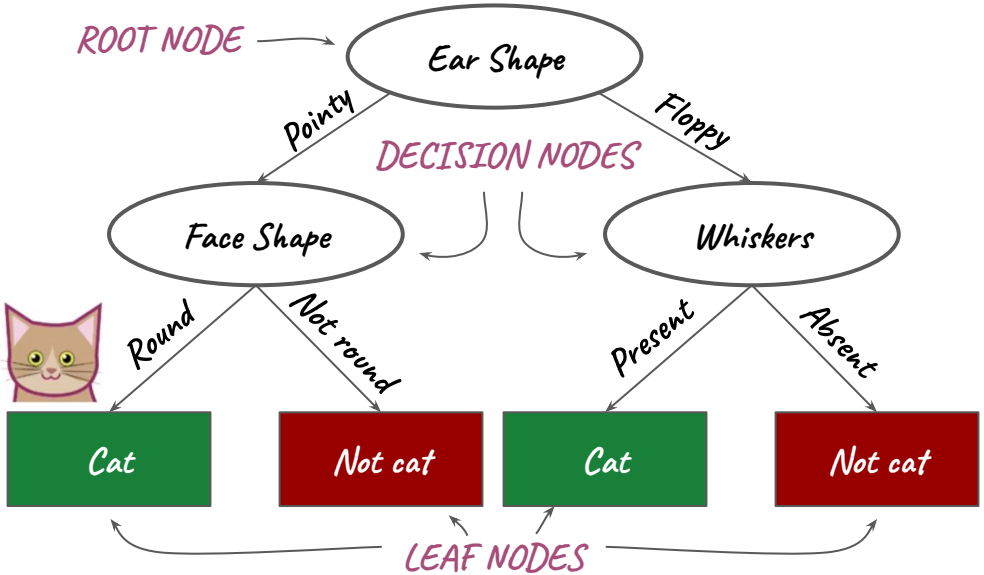
The Sloan Digital Cats Survey – III, with five cats and five dogs.
To keep it simple, we have only categorical features and label.
If a new example arrives, you just roll it down the tree, and see in which leaf it lands: that's the predicted label.



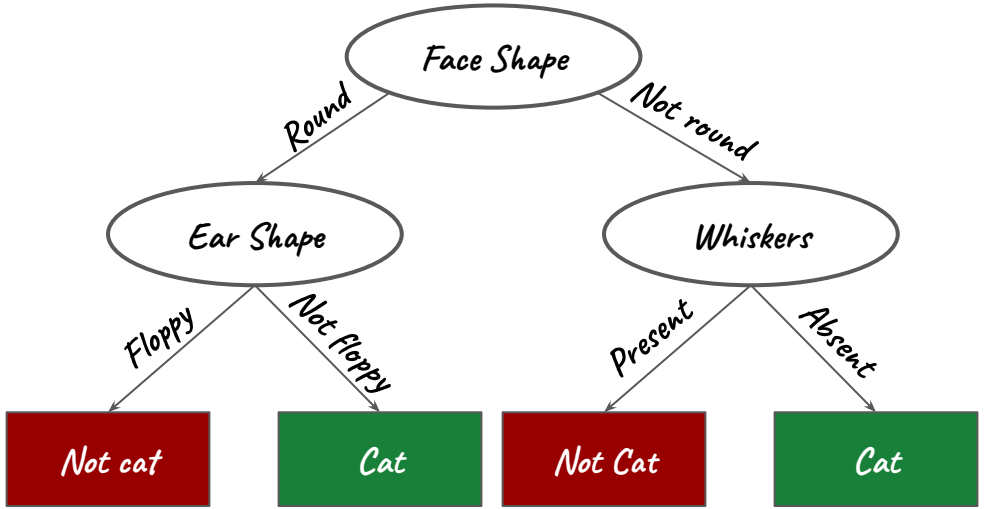
Decision trees are the building block of random forest, ensemble methods and gradient boosted algorithms, three terms that "*squeeze squeeze*" ontologically refer to the same thing. As all the best things in life, they are extremely simple and yet powerful: gradient boosted algorithms are the lone ML technique that is able to compete with (deep) Neural Networks in terms of performances.
Let's say that we have this dataset of crucial astrophysical importance:

	Ear Shape	Face Shape	Whiskers	Cat?
	Pointy	Round	Present	Yes
	Floppy	Not round	Present	Yes
	Floppy	Round	Absent	No
	Pointy	Not round	Present	No
	Pointy	Round	Present	Yes
	Pointy	Round	Absent	Yes
	Floppy	Not round	Absent	No
	Pointy	Round	Absent	Yes
	Floppy	Round	Absent	No
	Floppy	Round	Absent	No

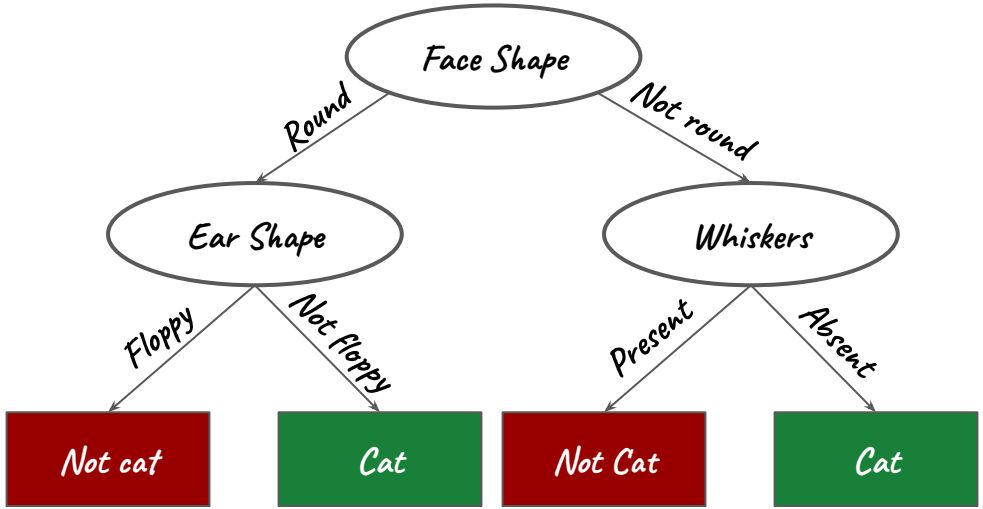
The Sloan Digital Cats Survey – III, with five cats and five dogs.
To keep it simple, we have only categorical features and label.
If a new example arrives, you just roll it down the tree, and see in which leaf it lands: that's the predicted label.

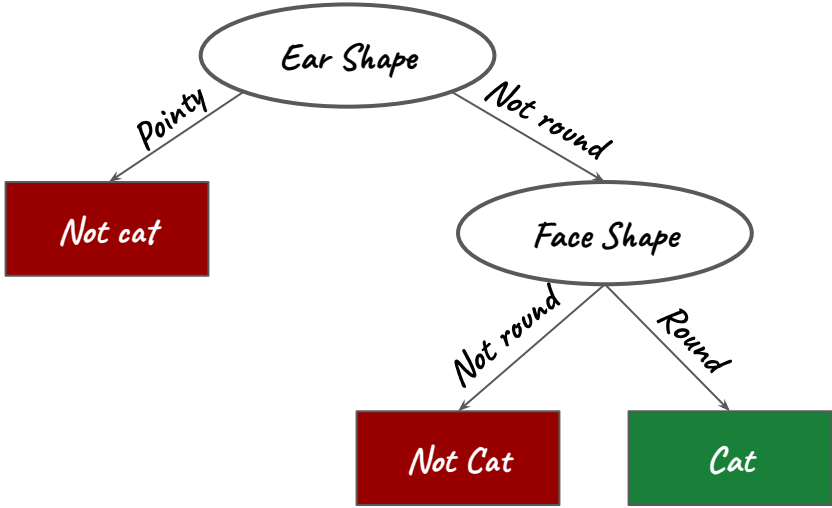


Of course, that's not the only possible decision tree that can be built from that **training** set.

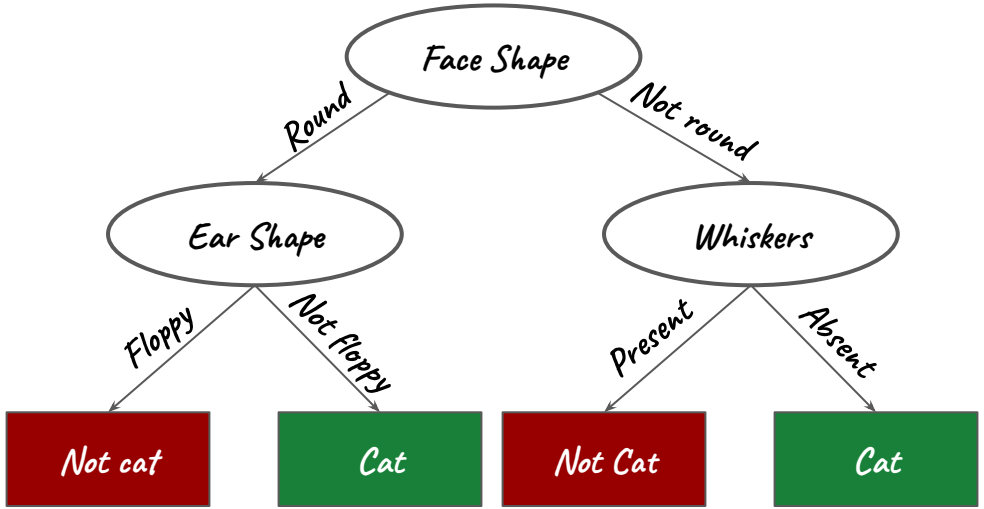


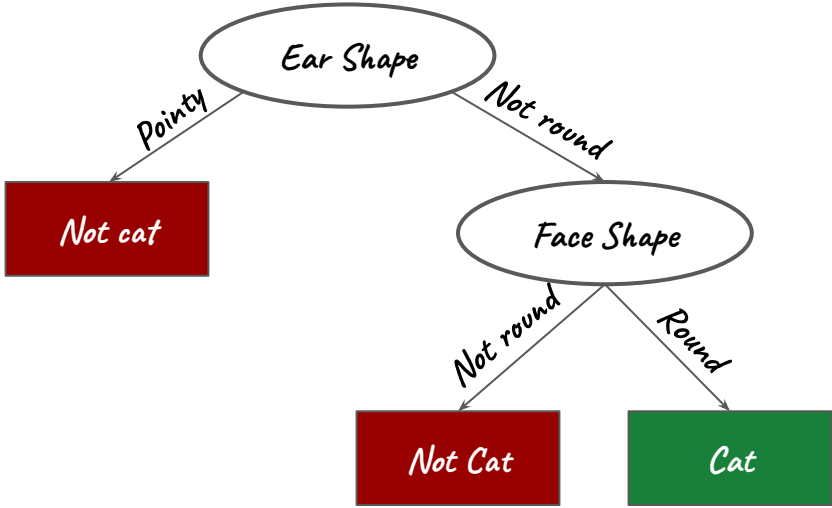
Of course, that's not the only possible decision tree that can be built from that **training** set.



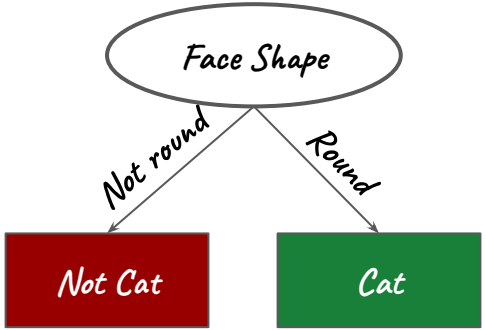


Of course, that's not the only possible decision tree that can be built from that **training** set.





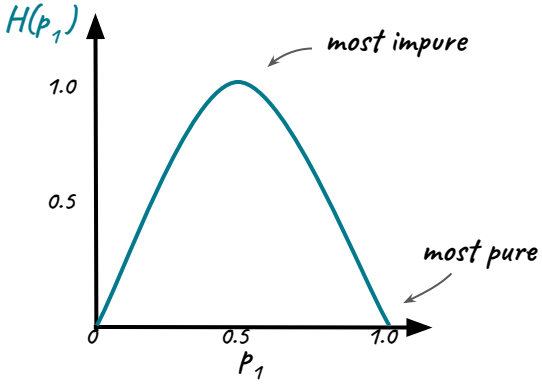
Some will do worse on the **training** / **dev** set, some will do better, etc …
The question is: out of all the possible decision trees, how to choose the one that generalizes to the data the most? How do you get an algorithm to automatically pick the best one?



Decision Trees - Information Gain

The feature upon which split the data at each node is chosen as the one that maximizes the ***purity*** *(or minimizes impurity)*

The feature upon which split the data at each node is chosen as the one that maximizes the ***purity*** *(or minimizes impurity)*
The sample ***purity/impurity*** is measured from the ***entropy*** H, function of the fraction p of examples in the sample that belong to a category.

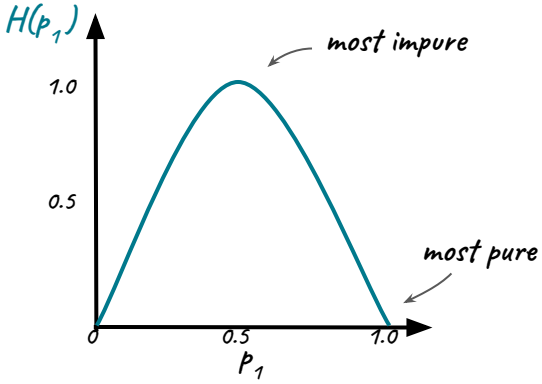


$$H(p_j) = -\sum_j p_j \log_j(p_j)$$

$$H(p_1) = -p_1 \log_2(p_1) - p_0 \log_2(p_0) = -p_1 \log_2(p_1) - (1 - p_1) \log_2(1 - p_1)$$

The feature upon which split the data at each node is chosen as the one that maximizes the ***purity*** *(or minimizes impurity)*

The sample ***purity/impurity*** is measured from the ***entropy*** H, function of the fraction p of examples in the sample that belong to a category.



$$H(p_j) = -\sum_j p_j \log_j(p_j)$$

$$H(p_1) = -p_1 \log_2(p_1) - p_0 \log_2(p_0) = -p_1 \log_2(p_1) - (1-p_1)\log_2(1-p_1)$$

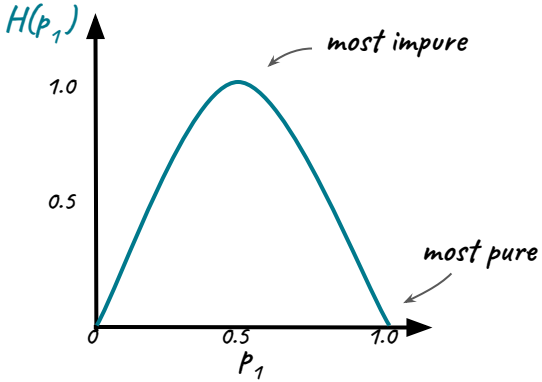
e.g. $p_1 \rightarrow$ fraction of examples that are cats; $H(p_1)$ is the ***entropy*** of p_1; $p_0 = 1 - p_1$

if $p_1 = 0.50 \rightarrow H(p_1) = 1.00 \leftarrow$ ***most impure***

if $p_1 = 0.83 \rightarrow H(p_1) = 0.65$

if $p_1 = 1.00 \rightarrow H(p_1) = 0.00 \leftarrow$ ***most pure***

The feature upon which split the data at each node is chosen as the one that maximizes the ***purity*** *(or minimizes impurity)*
The sample ***purity/impurity*** is measured from the ***entropy*** H, function of the fraction p of examples in the sample that belong to a category.



$$H(p_j) = -\sum_j p_j \log_j(p_j)$$

$$H(p_1) = -p_1 \log_2(p_1) - p_0 \log_2(p_0) = -p_1 \log_2(p_1) - (1-p_1)\log_2(1-p_1)$$

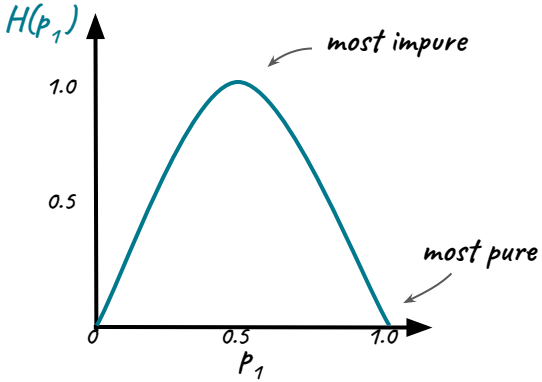
e.g. $p_1 \rightarrow$ fraction of examples that are cats; $H(p_1)$ is the ***entropy*** of p_1; $p_0 = 1 - p_1$
if $p_1 = 0.50 \rightarrow H(p_1) = 1.00 \leftarrow$ ***most impure***
if $p_1 = 0.83 \rightarrow H(p_1) = 0.65$
if $p_1 = 1.00 \rightarrow H(p_1) = 0.00 \leftarrow$ ***most pure***

In decision trees, the reduction of entropy is called the ***INFORMATION GAIN***, upon which the split choice is based.

e.g.
ear shape split $\rightarrow$ ***left*** $H = 0.72$; ***right*** $H = 0.72$
face shape split $\rightarrow$ ***left*** $H = 0.99$; ***right*** $H = 0.92$
whiskers split $\rightarrow$ ***left*** $H = 0.81$; ***right*** $H = 0.91$

Take the mean of the left/right split, weighted by the number of examples going in the sub-branch, subtracted to the entropy of the root node

The feature upon which split the data at each node is chosen as the one that maximizes the *purity* (or minimizes impurity)
The sample *purity/impurity* is measured from the *entropy H*, function of the fraction *p* of examples in the sample that belong to a category.



$$H(p_j) = -\sum_j p_j \log_j(p_j)$$

$$H(p_1) = -p_1 \log_2(p_1) - p_0 \log_2(p_0) = -p_1 \log_2(p_1) - (1-p_1)\log_2(1-p_1)$$

e.g. $p_1 \rightarrow$ fraction of examples that are cats; $H(p_1)$ is the *entropy* of p_1; $p_0 = 1 - p_1$
if $p_1 = 0.50 \rightarrow H(p_1) = 1.00 \leftarrow$ most impure
if $p_1 = 0.83 \rightarrow H(p_1) = 0.65$
if $p_1 = 1.00 \rightarrow H(p_1) = 0.00 \leftarrow$ most pure

In decision trees, the reduction of entropy is called the *INFORMATION GAIN*, upon which the split choice is based.

e.g.
ear shape split → *left H = 0.72; right H = 0.72*
face shape split → *left H = 0.99; right H = 0.92*
whiskers split → *left H = 0.81; right H = 0.91*

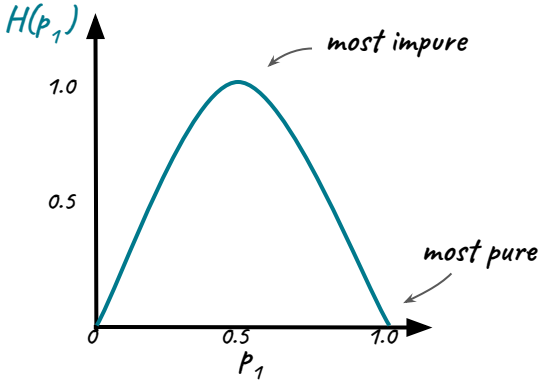
Take the mean of the left/right split, weighted by the number of examples going in the sub-branch, subtracted to the entropy of the root node

$$\text{Information Gain} = H(p_1^{\text{root}}) - (w^{\text{left}} H(p_1^{\text{left}}) + w^{\text{right}} H(p_1^{\text{right}}))$$

the fraction of examples going to the left/right branch

The chosen feature split is the one with the **highest** *INFORMATION GAIN*.

The feature upon which split the data at each node is chosen as the one that maximizes the *purity* (or minimizes impurity)
The sample *purity/impurity* is measured from the *entropy H*, function of the fraction *p* of examples in the sample that belong to a category.



$$H(p_j) = -\sum_j p_j \log_j(p_j)$$

$$H(p_1) = -p_1 \log_2(p_1) - p_0 \log_2(p_0) = -p_1 \log_2(p_1) - (1-p_1)\log_2(1-p_1)$$

e.g. $p_1 \rightarrow$ fraction of examples that are cats; $H(p_1)$ is the *entropy* of p_1; $p_0 = 1 - p_1$
if $p_1 = 0.50 \rightarrow H(p_1) = 1.00 \leftarrow$ most impure
if $p_1 = 0.83 \rightarrow H(p_1) = 0.65$
if $p_1 = 1.00 \rightarrow H(p_1) = 0.00 \leftarrow$ most pure

In decision trees, the reduction of entropy is called the *INFORMATION GAIN*, upon which the split choice is based.

e.g.
ear shape split → *left H = 0.72; right H = 0.72*
face shape split → *left H = 0.99; right H = 0.92*
whiskers split → *left H = 0.81; right H = 0.91*

Take the mean of the left/right split, weighted by the number of examples going in the sub-branch, subtracted to the entropy of the root node

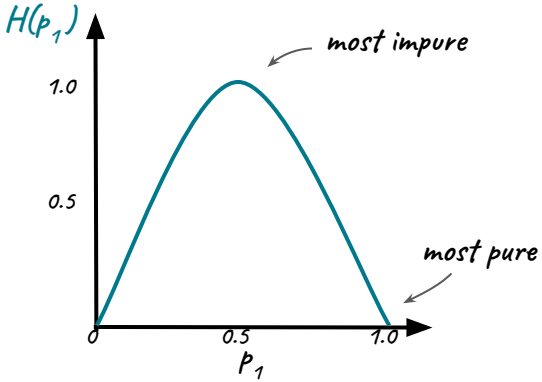
$$\text{Information Gain} = H(p_1^{\text{root}}) - (w^{\text{left}} H(p_1^{\text{left}}) + w^{\text{right}} H(p_1^{\text{right}}))$$

the fraction of examples going to the left/right branch

The chosen feature split is the one with the **highest** *INFORMATION GAIN*.

IG (ear shape split) = 0.28
IG (face shape split) = 0.03
IG (whiskers split) = 0.12

The feature upon which split the data at each node is chosen as the one that maximizes the *purity* (or minimizes impurity)
The sample *purity/impurity* is measured from the *entropy H*, function of the fraction *p* of examples in the sample that belong to a category.



$$H(p_j) = -\sum_j p_j \log_j(p_j)$$

$$H(p_1) = -p_1 \log_2(p_1) - p_0 \log_2(p_0) = -p_1 \log_2(p_1) - (1-p_1)\log_2(1-p_1)$$

e.g. $p_1 \rightarrow$ fraction of examples that are cats; $H(p_1)$ is the *entropy* of p_1; $p_0 = 1 - p_1$
if $p_1 = 0.50 \rightarrow H(p_1) = 1.00 \leftarrow$ *most impure*
if $p_1 = 0.83 \rightarrow H(p_1) = 0.65$
if $p_1 = 1.00 \rightarrow H(p_1) = 0.00 \leftarrow$ *most pure*

In decision trees, the reduction of entropy is called the *INFORMATION GAIN*, upon which the split choice is based.

e.g.
ear shape split → *left H = 0.72; right H = 0.72*
face shape split → *left H = 0.99; right H = 0.92*
whiskers split → *left H = 0.81; right H = 0.91*

Take the mean of the left/right split, weighted by the number of examples going in the sub-branch, subtracted to the entropy of the root node

$$\text{Information Gain} = H(p_1^{\text{root}}) - (w^{\text{left}} H(p_1^{\text{left}}) + w^{\text{right}} H(p_1^{\text{right}}))$$

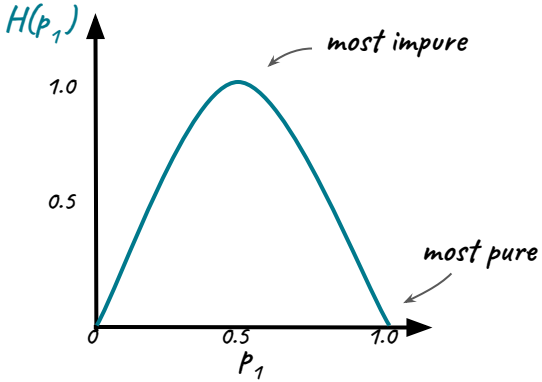
the fraction of examples going to the left/right branch

The chosen feature split is the one with the **highest** *INFORMATION GAIN*.

IG (ear shape split) = 0.28
IG (face shape split) = 0.03
IG (whiskers split) = 0.12

and that's why the first split was on the ear shape feature

The feature upon which split the data at each node is chosen as the one that maximizes the *purity* *(or minimizes impurity)*
The sample *purity/impurity* is measured from the *entropy H*, function of the fraction *p* of examples in the sample that belong to a category.



$$H(p_j) = -\sum_j p_j \log_j(p_j)$$

$$H(p_1) = -p_1 \log_2(p_1) - p_0 \log_2(p_0) = -p_1 \log_2(p_1) - (1-p_1)\log_2(1-p_1)$$

e.g. $p_1 \rightarrow$ fraction of examples that are cats; $H(p_1)$ is the *entropy* of p_1; $p_0 = 1 - p_1$
if $p_1 = 0.50 \rightarrow H(p_1) = 1.00 \leftarrow$ *most impure*
if $p_1 = 0.83 \rightarrow H(p_1) = 0.65$
if $p_1 = 1.00 \rightarrow H(p_1) = 0.00 \leftarrow$ *most pure*

Another commonly used metric for purity/impurity is the Gini coefficient

In decision trees, the reduction of entropy is called the *INFORMATION GAIN*, upon which the split choice is based.

e.g.
ear shape split → *left H = 0.72; right H = 0.72*
face shape split → *left H = 0.99; right H = 0.92*
whiskers split → *left H = 0.81; right H = 0.91*

Take the mean of the left/right split, weighted by the number of examples going in the sub-branch, subtracted to the entropy of the root node

$$\text{Information Gain} = H(p_1^{\text{root}}) - (w^{\text{left}} H(p_1^{\text{left}}) + w^{\text{right}} H(p_1^{\text{right}}))$$

the fraction of examples going to the left/right branch

The chosen feature split is the one with the **highest** *INFORMATION GAIN*.

IG (ear shape split) = 0.28
IG (face shape split) = 0.03
IG (whiskers split) = 0.12

and that's why the first split was on the ear shape feature

So, at each step a decision tree measures the *IG* for all possible features, and picks the one with the highest *IG*.

Decision Trees Recursive Algorithm

So, at each step a decision tree measures the *IG* for all possible features, and picks the one with the highest *IG*.
Therefore it splits the dataset according to the selected feature, and create left/right branches of the tree, and it keeps repeating this until a stopping criterion is met:

- when a node is 100% one class

Decision Trees Recursive Algorithm

So, at each step a decision tree measures the *IG* for all possible features, and picks the one with the highest *IG*.
Therefore it splits the dataset according to the selected feature, and create left/right branches of the tree, and it keeps repeating this until a stopping criterion is met:

- when a node is 100% one class
- when a splitting node will result in the tree exceeding a **max_depth**

Decision Trees Recursive Algorithm

So, at each step a decision tree measures the *IG* for all possible features, and picks the one with the highest *IG*.
Therefore it splits the dataset according to the selected feature, and create left/right branches of the tree, and it keeps repeating this until a stopping criterion is met:

- when a node is 100% one class
- when a splitting node will result in the tree exceeding a **max_depth**
- when the *IG* from additional splits is less than a chosen threshold **min_impurity_decrease**

Decision Trees Recursive Algorithm

So, at each step a decision tree measures the *IG* for all possible features, and picks the one with the highest *IG*.
Therefore it splits the dataset according to the selected feature, and create left/right branches of the tree, and it keeps repeating this until a stopping criterion is met:

- when a node is 100% one class
- when a splitting node will result in the tree exceeding a **max_depth**
- when the *IG* from additional splits is less than a chosen threshold **min_impurity_decrease**
- when the number of examples in a node is below a certain threshold **min_samples_leaf**

At that point the tree has arrived to a **leaf node**, and stops the splitting process for that particular branch.

Decision Trees Recursive Algorithm

So, at each step a decision tree measures the *IG* for all possible features, and picks the one with the highest *IG*.
Therefore it splits the dataset according to the selected feature, and create left/right branches of the tree, and it keeps repeating this until a stopping criterion is met:

- when a node is 100% one class
- when a splitting node will result in the tree exceeding a **max_depth**
- when the *IG* from additional splits is less than a chosen threshold **min_impurity_decrease**
- when the number of examples in a node is below a certain threshold **min_samples_leaf**

At that point the tree has arrived to a **leaf node**, and stops the splitting process for that particular branch.
When a categorical feature can take more than two possible values (e.g. **Pointy/Floppy/Oval shapes** for ears) use one-hot encoding.

Decision Trees Recursive Algorithm

So, at each step a decision tree measures the *IG* for all possible features, and picks the one with the highest *IG*.
Therefore it splits the dataset according to the selected feature, and create left/right branches of the tree, and it keeps repeating this until a stopping criterion is met:

- when a node is 100% one class
- when a splitting node will result in the tree exceeding a **max_depth**
- when the *IG* from additional splits is less than a chosen threshold **min_impurity_decrease**
- when the number of examples in a node is below a certain threshold **min_samples_leaf**

At that point the tree has arrived to a **leaf node**, and stops the splitting process for that particular branch.
When a categorical feature can take more than two possible values (e.g. **Pointy/Floppy/Oval shapes** for ears) use one-hot encoding.
What if I have numerical features?

So, at each step a decision tree measures the *IG* for all possible features, and picks the one with the highest *IG*.
Therefore it splits the dataset according to the selected feature, and create left/right branches of the tree, and it keeps repeating this until a stopping criterion is met:

- when a node is 100% one class
- when a splitting node will result in the tree exceeding a **max_depth**
- when the *IG* from additional splits is less than a chosen threshold **min_impurity_decrease**
- when the number of examples in a node is below a certain threshold **min_samples_leaf**

At that point the tree has arrived to a **leaf node**, and stops the splitting process for that particular branch.
When a categorical feature can take more than two possible values (e.g. **Pointy/Floppy/Oval shapes** for ears) use one-hot encoding.
What if I have numerical features?

	Ear Shape	Face Shape	Whiskers	Weight (kg)	Cat?
	Pointy	Round	Present	5.8	Yes
	Floppy	Not round	Present	7.3	Yes
	Floppy	Round	Absent	15	No
	Pointy	Not round	Present	42.1	No
	Pointy	Round	Present	4.6	Yes
	Pointy	Round	Absent	5.7	Yes
	Floppy	Not round	Absent	15.2	No
	Pointy	Round	Absent	10.3	Yes
	Floppy	Round	Absent	13.4	No
	Floppy	Round	Absent	19.4	No

So, at each step a decision tree measures the *IG* for all possible features, and picks the one with the highest *IG*.
Therefore it splits the dataset according to the selected feature, and create left/right branches of the tree, and it keeps repeating this until a stopping criterion is met:

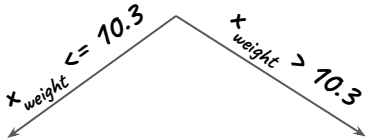
- when a node is 100% one class
- when a splitting node will result in the tree exceeding a ***max_depth***
- when the *IG* from additional splits is less than a chosen threshold ***min_impurity_decrease***
- when the number of examples in a node is below a certain threshold ***min_samples_leaf***

At that point the tree has arrived to a ***leaf node***, and stops the splitting process for that particular branch.
When a categorical feature can take more than two possible values (e.g. ***Pointy/Floppy/Oval shapes*** for ears) use one-hot encoding.
What if I have numerical features?

	Ear Shape	Face Shape	Whiskers	Weight (kg)	Cat?
	Pointy	Round	Present	5.8	Yes
	Floppy	Not round	Present	7.3	Yes
	Floppy	Round	Absent	15	No
	Pointy	Not round	Present	42.1	No
	Pointy	Round	Present	4.6	Yes
	Pointy	Round	Absent	5.7	Yes
	Floppy	Not round	Absent	15.2	No
	Pointy	Round	Absent	10.3	Yes
	Floppy	Round	Absent	13.4	No
	Floppy	Round	Absent	19.4	No

The concept is still the same: the feature is chosen by maximizing the *IG*, so the algorithm will measure the one obtained by splitting on the ***weights*** feature.
In this case, there are various possible x_{weight} values upon which to split: the decision tree will try all the different thresholds, measure the *IG* for each one, and pick the one carrying the highest, to compare with the *IG* arriving from splitting at the other features.

In this case, is 10.3 kg, so the eventual split would be on:

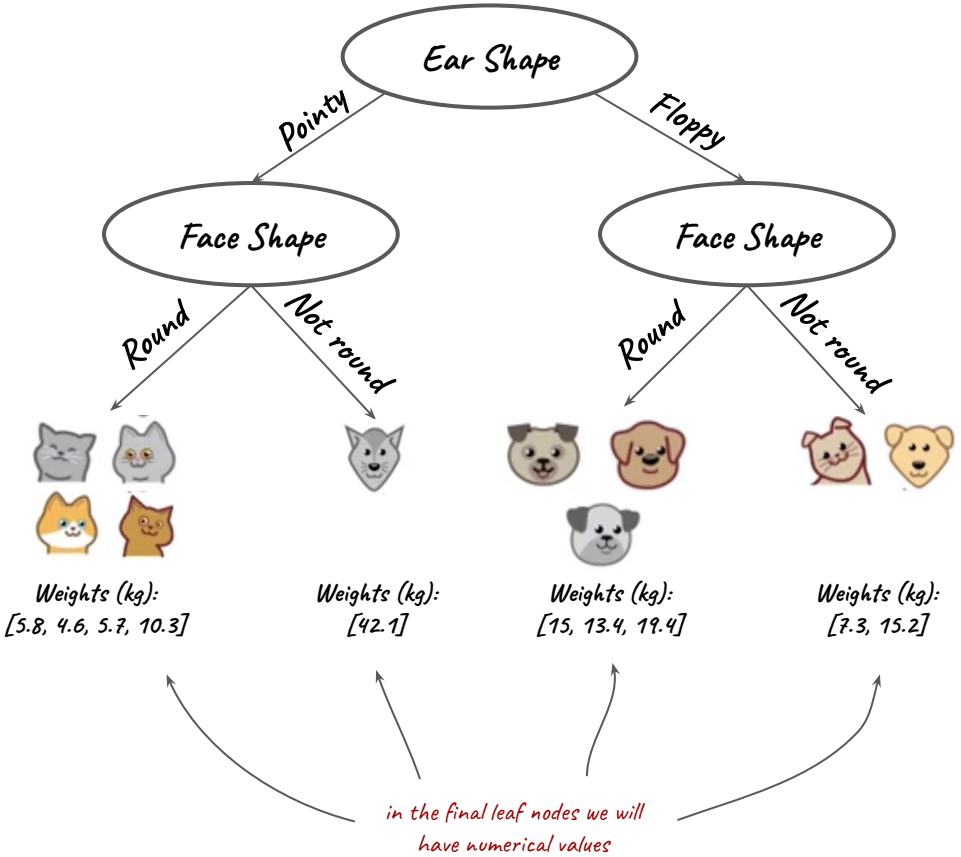


Those were decision trees for classification tasks. What if I want to predict a number, and not a class?

	Ear Shape	Face Shape	Whiskers	Weight (kg)
	Pointy	Round	Present	5.8
	Floppy	Not round	Present	7.3
	Floppy	Round	Absent	15
	Pointy	Not round	Present	42.1
	Pointy	Round	Present	4.6
	Pointy	Round	Absent	5.7
	Floppy	Not round	Absent	15.2
	Pointy	Round	Absent	10.3
	Floppy	Round	Absent	13.4
	Floppy	Round	Absent	19.4

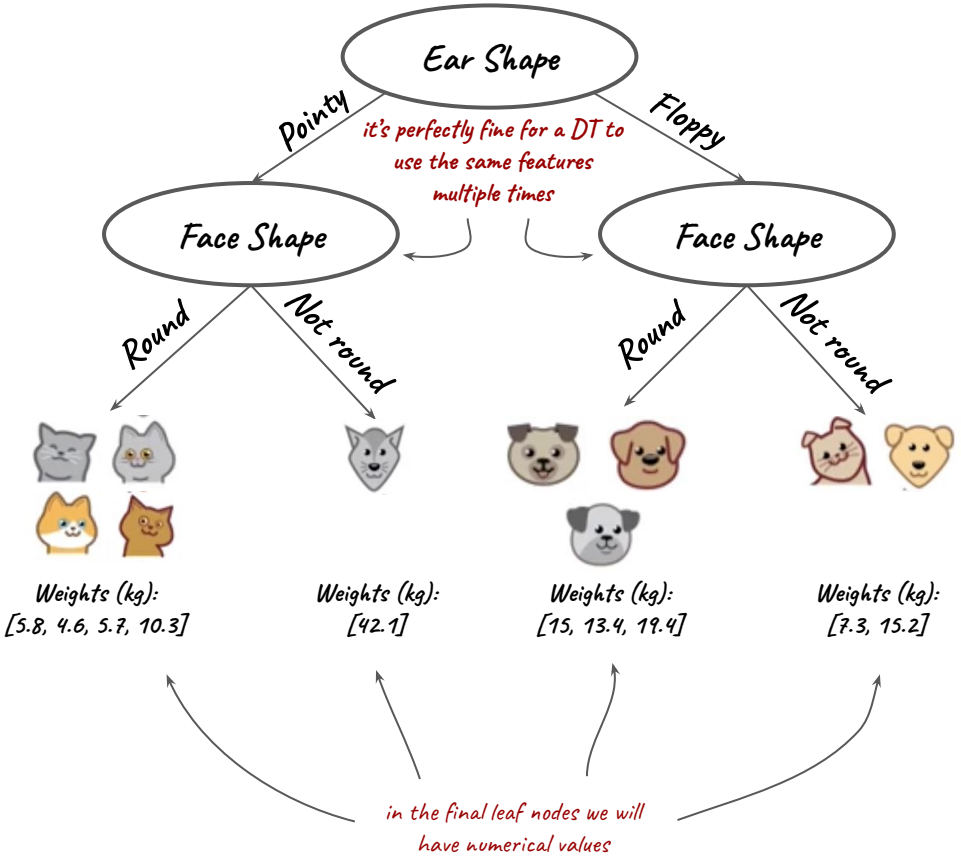
Those were decision trees for classification tasks. What if I want to predict a number, and not a class?

	Ear Shape	Face Shape	Whiskers	Weight (kg)
	Pointy	Round	Present	5.8
	Floppy	Not round	Present	7.3
	Floppy	Round	Absent	15
	Pointy	Not round	Present	42.1
	Pointy	Round	Present	4.6
	Pointy	Round	Absent	5.7
	Floppy	Not round	Absent	15.2
	Pointy	Round	Absent	10.3
	Floppy	Round	Absent	13.4
	Floppy	Round	Absent	19.4



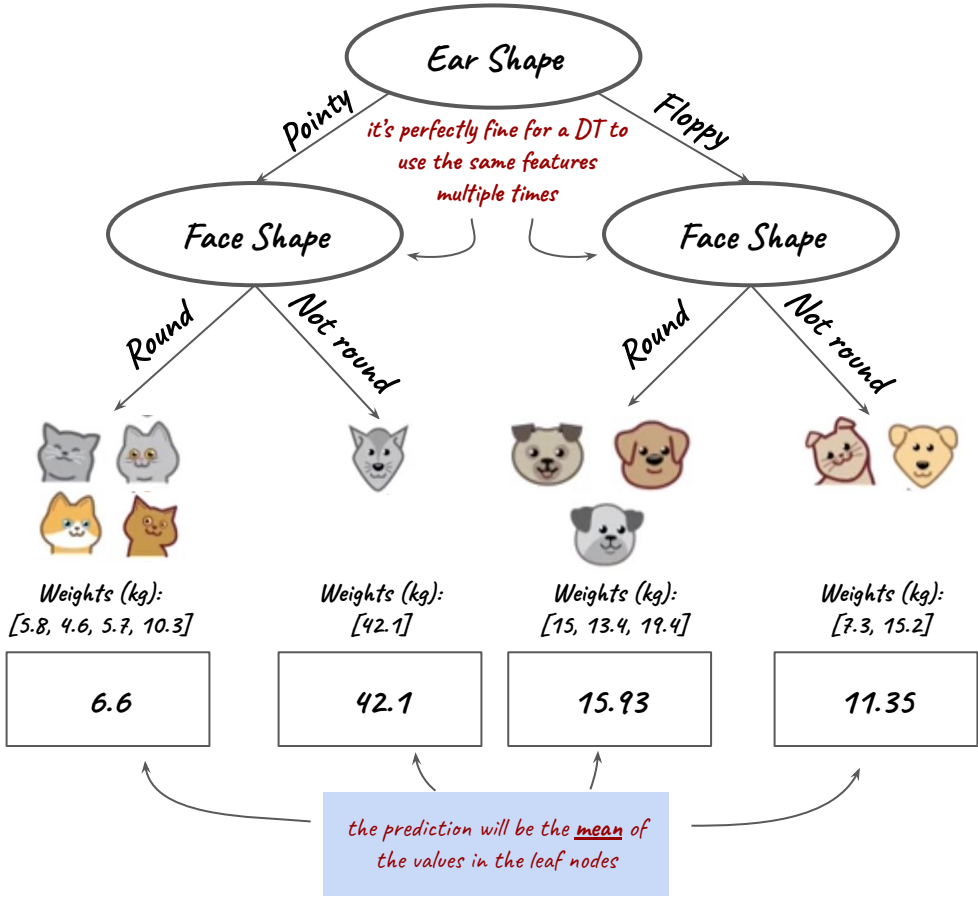
Those were decision trees for classification tasks. What if I want to predict a number, and not a class?

	Ear Shape	Face Shape	Whiskers	Weight (kg)
	Pointy	Round	Present	5.8
	Floppy	Not round	Present	7.3
	Floppy	Round	Absent	15
	Pointy	Not round	Present	42.1
	Pointy	Round	Present	4.6
	Pointy	Round	Absent	5.7
	Floppy	Not round	Absent	15.2
	Pointy	Round	Absent	10.3
	Floppy	Round	Absent	13.4
	Floppy	Round	Absent	19.4



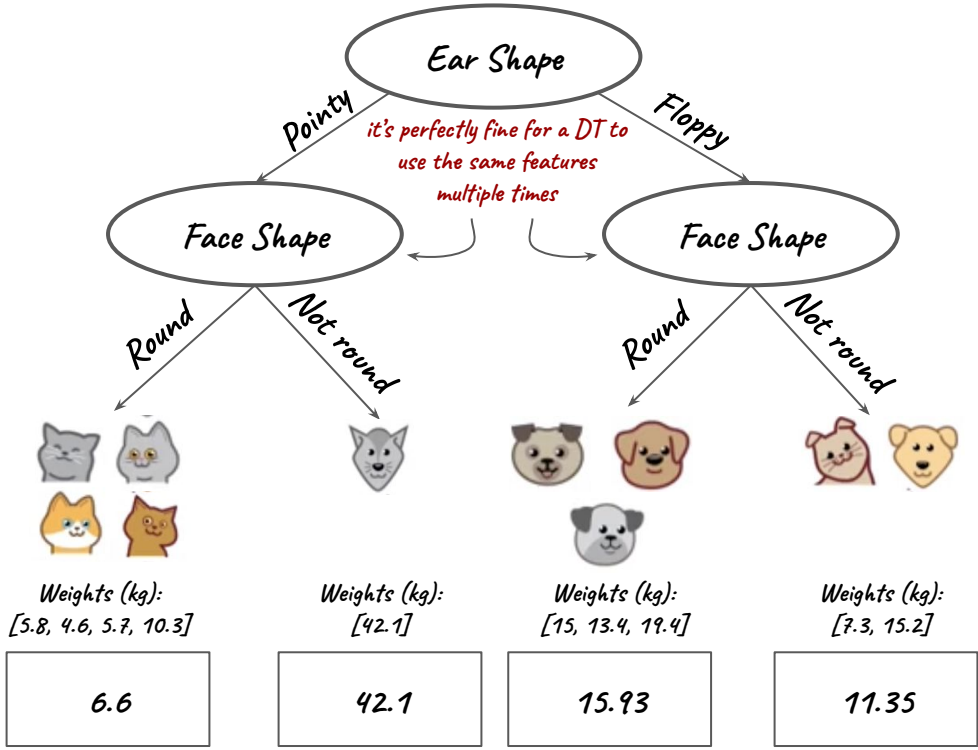
Those were decision trees for classification tasks. What if I want to predict a number, and not a class?

	Ear Shape	Face Shape	Whiskers	Weight (kg)
	Pointy	Round	Present	5.8
	Floppy	Not round	Present	7.3
	Floppy	Round	Absent	15
	Pointy	Not round	Present	42.1
	Pointy	Round	Present	4.6
	Pointy	Round	Absent	5.7
	Floppy	Not round	Absent	15.2
	Pointy	Round	Absent	10.3
	Floppy	Round	Absent	13.4
	Floppy	Round	Absent	19.4



Those were decision trees for classification tasks. What if I want to predict a number, and not a class?

	Ear Shape	Face Shape	Whiskers	Weight (kg)
	Pointy	Round	Present	5.8
	Floppy	Not round	Present	7.3
	Floppy	Round	Absent	15
	Pointy	Not round	Present	42.1
	Pointy	Round	Present	4.6
	Pointy	Round	Absent	5.7
	Floppy	Not round	Absent	15.2
	Pointy	Round	Absent	10.3
	Floppy	Round	Absent	13.4
	Floppy	Round	Absent	19.4

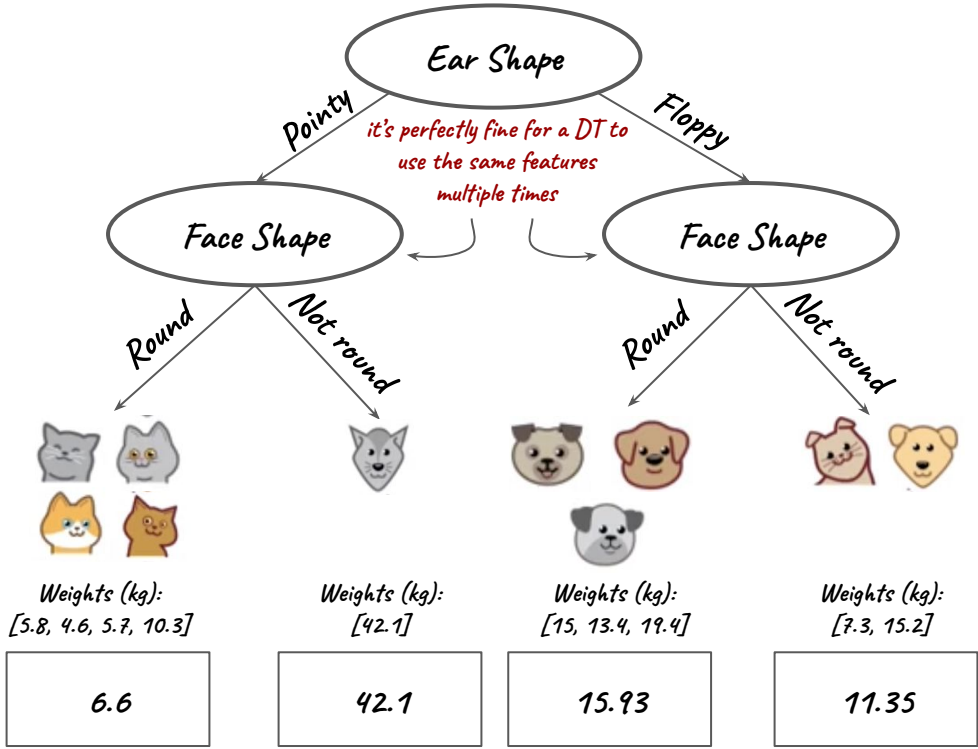


$$\Delta\text{Var} = \text{Var}(\text{root}) - (w^{\text{left}}\,\text{Var}(\text{left}) + w^{\text{right}}\,\text{Var}(\text{right}))$$

The feature to split on is chosen as the one that reduces the most the VARIANCE between the root node and the weighted left/right branches.

Those were decision trees for classification tasks. What if I want to predict a number, and not a class?

	Ear Shape	Face Shape	Whiskers	Weight (kg)
	Pointy	Round	Present	5.8
	Floppy	Not round	Present	7.3
	Floppy	Round	Absent	15
	Pointy	Not round	Present	42.1
	Pointy	Round	Present	4.6
	Pointy	Round	Absent	5.7
	Floppy	Not round	Absent	15.2
	Pointy	Round	Absent	10.3
	Floppy	Round	Absent	13.4
	Floppy	Round	Absent	19.4



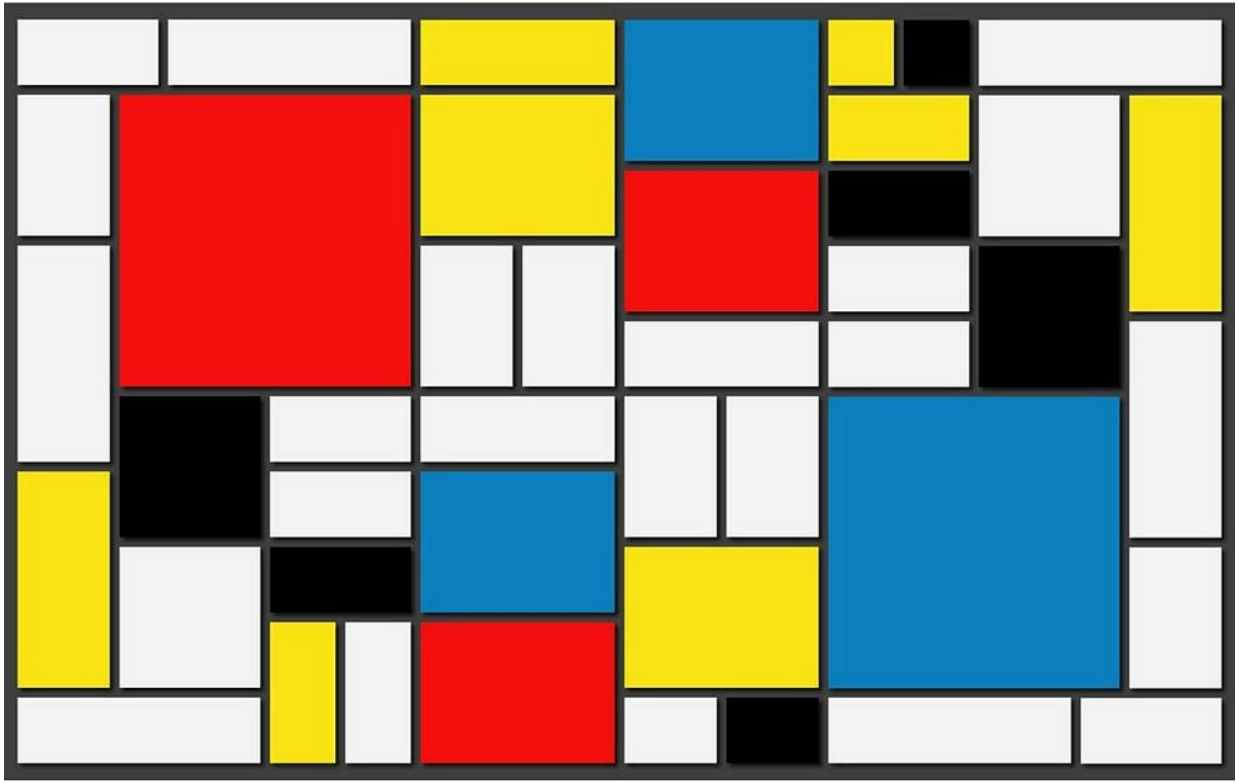
$$\Delta\mathrm{Var} = \mathrm{Var}(\mathrm{root}) - (w^{\mathrm{left}}\,\mathrm{Var}(\mathrm{left}) + w^{\mathrm{right}}\,\mathrm{Var}(\mathrm{right}))$$

The whole thing works also for **multilabel (multi-output)** predictions (e.g. weight and height). The algorithms will chose the feature that reduces the **average** variance.

The feature to split on is chosen as the one that reduces the most the VARIANCE between the root node and the weighted left/right branches.

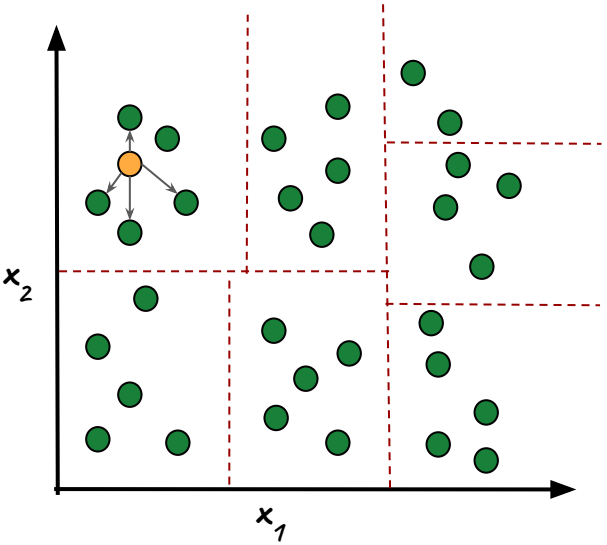
What is a decision tree actually doing on the feature space?

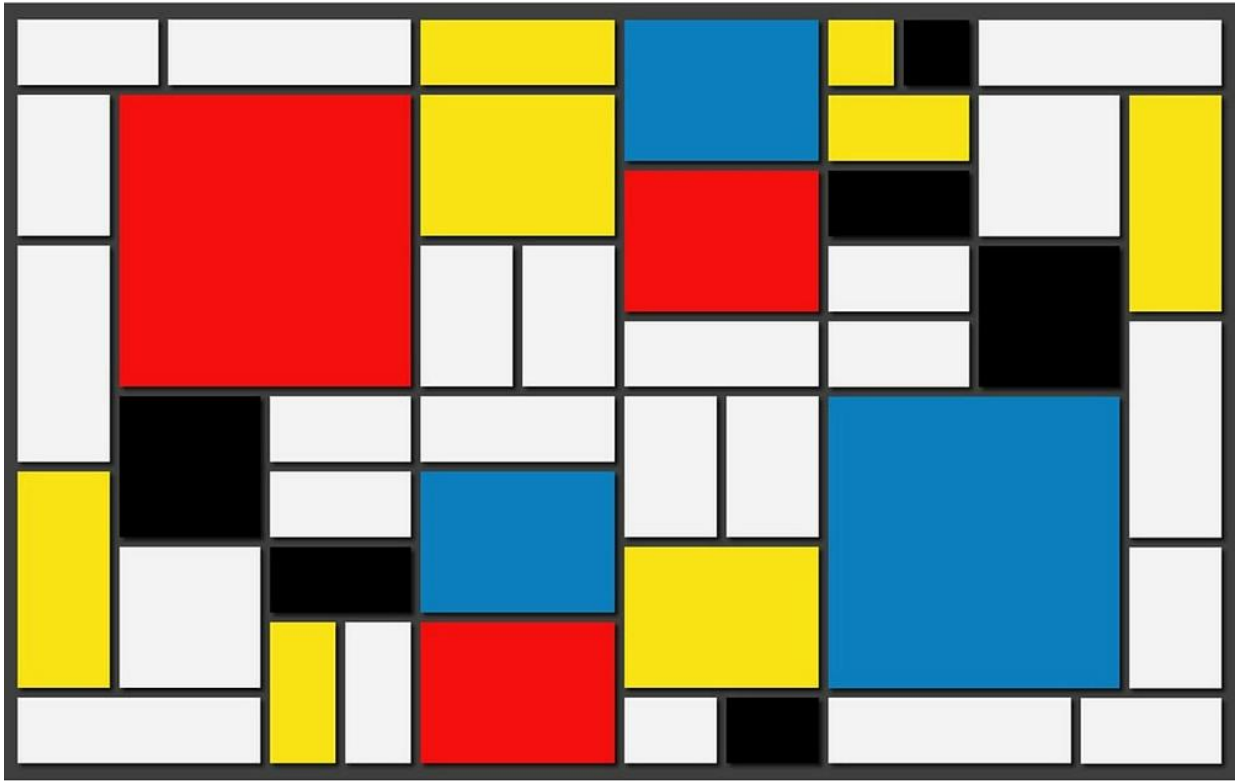
A decision tree is partitioning the feature space, grouping together close training examples in the a bag (***leaf***) with the least possible variance. When a new target examples rolls down the tree, it arrives in the final leaf, and the prediction comes from averaging the training values. That's similar to what NN is doing, but with a smarter/ad hoc feature space partitioning.



What is a decision tree actually doing on the feature space?

A decision tree is partitioning the feature space, grouping together close training examples in the a bag (***leaf***) with the least possible variance. When a new target examples rolls down the tree, it arrives in the final leaf, and the prediction comes from averaging the training values. That's similar to what NN is doing, but with a smarter/ad hoc feature space partitioning.

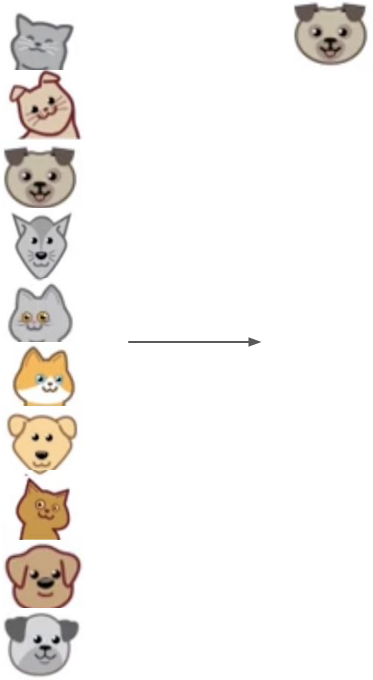




The greatest weakness of a single **decision tree** is that it clearly overfits to that particular **training set**.
The solution is to use multiple decision trees, a **tree ensemble**.
Every tree in the ensemble is built from a *sampling-with-replacement* version of the **training set**.

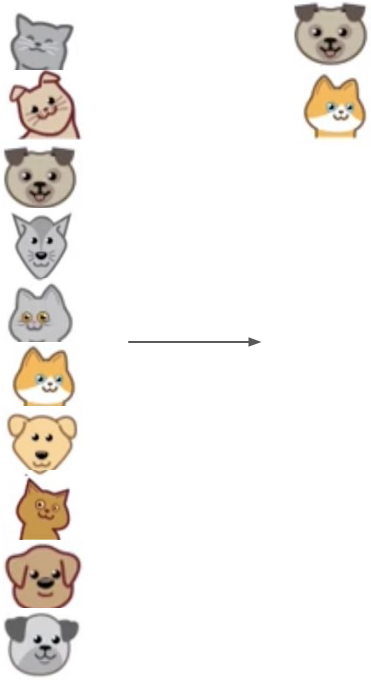
training set

The greatest weakness of a single **decision tree** is that it clearly overfits to that particular **training set**.
The solution is to use multiple decision trees, a **tree ensemble**.
Every tree in the ensemble is built from a *sampling-with-replacement* version of the **training set**.



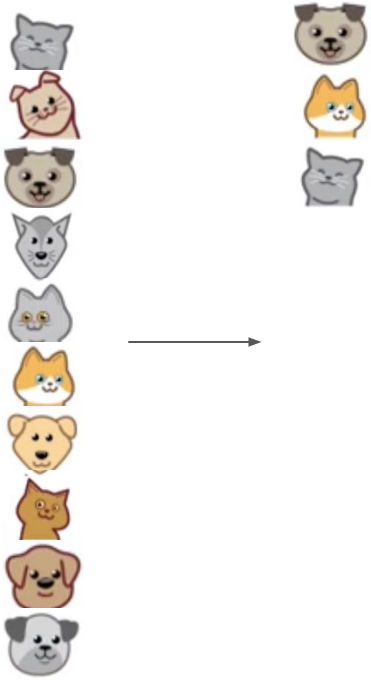
training set

The greatest weakness of a single **decision tree** is that it clearly overfits to that particular **training set**.
The solution is to use multiple decision trees, a **tree ensemble**.
Every tree in the ensemble is built from a *sampling-with-replacement* version of the **training set**.



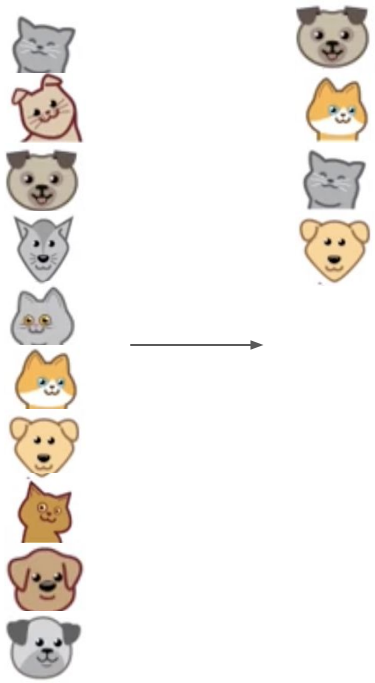
training set

The greatest weakness of a single **decision tree** is that it clearly overfits to that particular **training set**.
The solution is to use multiple decision trees, a **tree ensemble**.
Every tree in the ensemble is built from a *sampling-with-replacement* version of the **training set**.



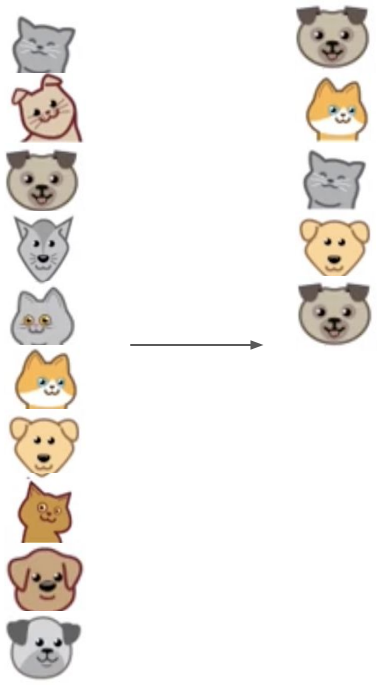
training set

The greatest weakness of a single **decision tree** is that it clearly overfits to that particular **training set**.
The solution is to use multiple decision trees, a **tree ensemble**.
Every tree in the ensemble is built from a *sampling-with-replacement* version of the **training set**.



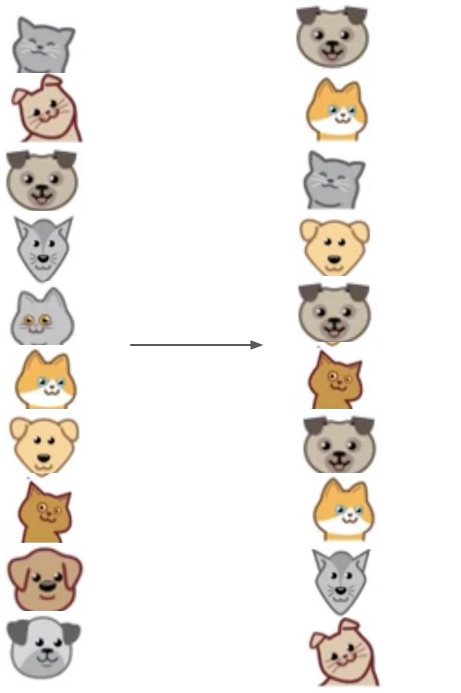
training set

The greatest weakness of a single **decision tree** is that it clearly overfits to that particular **training set**.
The solution is to use multiple decision trees, a **tree ensemble**.
Every tree in the ensemble is built from a *sampling-with-replacement* version of the **training set**.



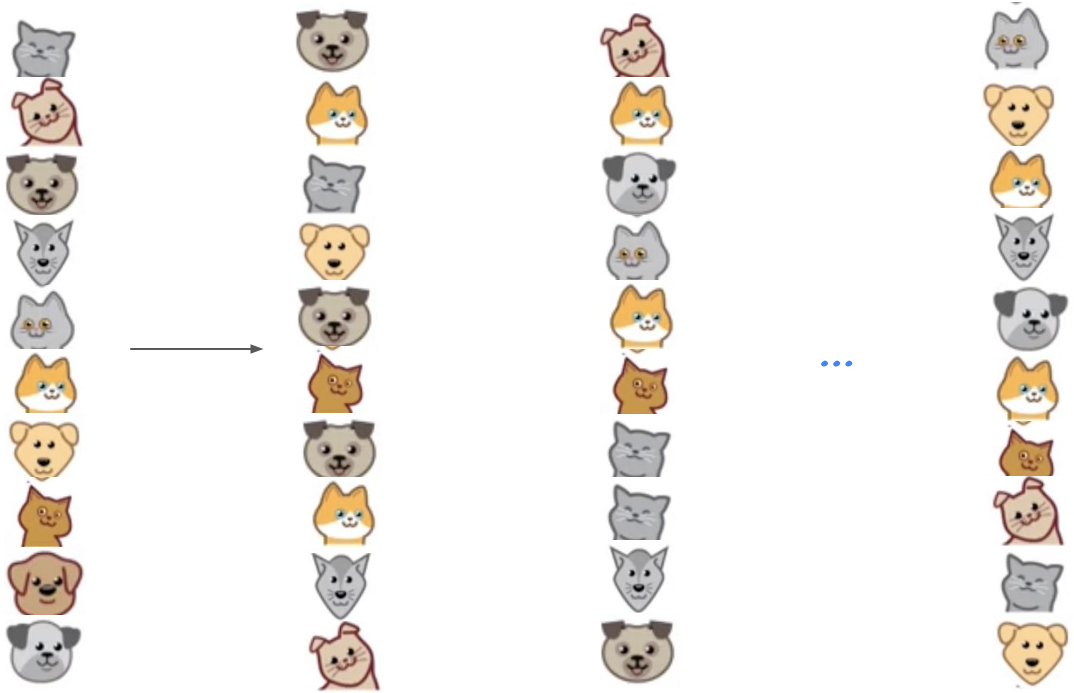
training set

The greatest weakness of a single **decision tree** is that it clearly overfits to that particular **training set**.
The solution is to use multiple decision trees, a **tree ensemble**.
Every tree in the ensemble is built from a *sampling-with-replacement* version of the **training set**.



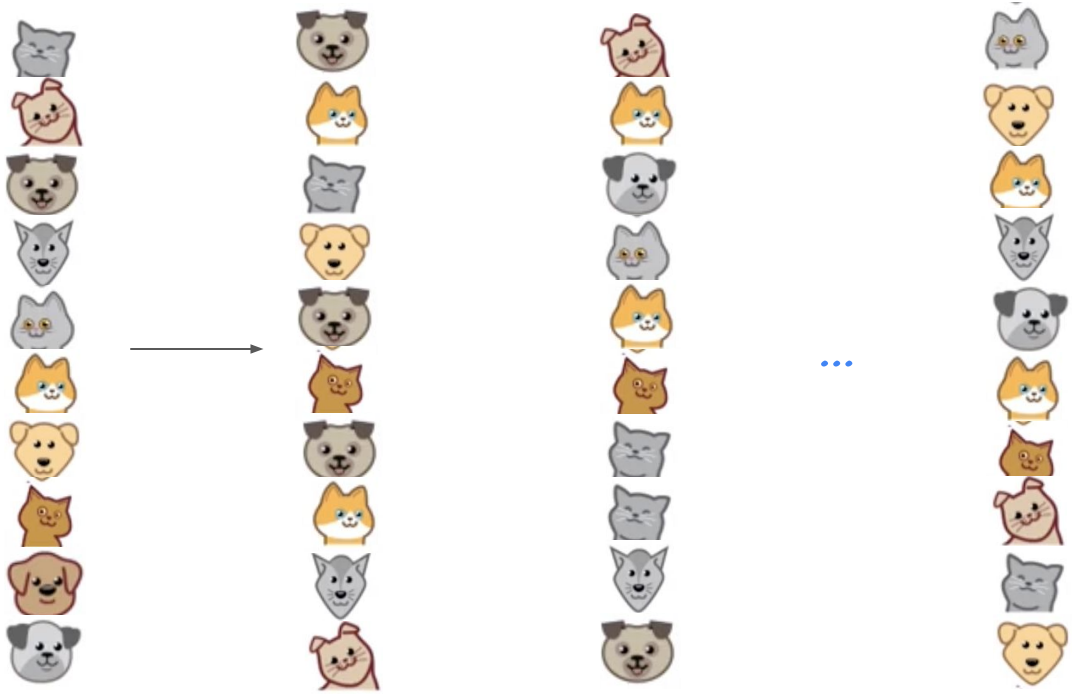
training set *Set #1 → DT#1*

The greatest weakness of a single **decision tree** is that it clearly overfits to that particular **training set**.
The solution is to use multiple decision trees, a **tree ensemble**.
Every tree in the ensemble is built from a *sampling-with-replacement* version of the **training set**.



training set Set #1 → DT#1 Set #2 → DT#2 Set #B → DT#B

The greatest weakness of a single **decision tree** is that it clearly overfits to that particular **training set**.
The solution is to use multiple decision trees, a **tree ensemble**.
Every tree in the ensemble is built from a *sampling-with-replacement* version of the **training set**.

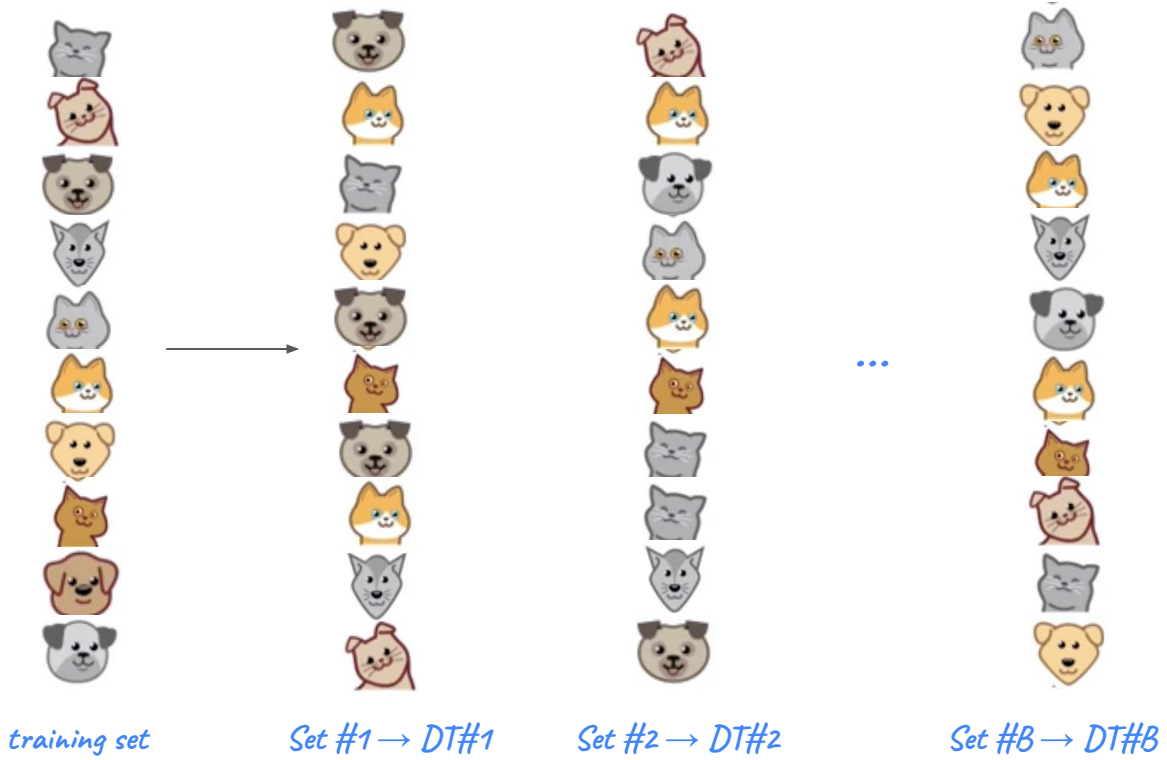


training set Set #1 → DT#1 Set #2 → DT#2 Set #B → DT#B

This technique is called *BAGGING*, short version for bootstrap aggregating.

For $i = 1 \ldots B$ use sampling with replacement to create a new training set of size m and train a **decision tree** on the new dataset.

The greatest weakness of a single **decision tree** is that it clearly overfits to that particular **training set**.
The solution is to use multiple decision trees, a **tree ensemble**.
Every tree in the ensemble is built from a *sampling-with-replacement* version of the **training set**.



This technique is called *BAGGING*, short version for bootstrap aggregating.

For $i = 1 \dots B$ use sampling with replacement to create a new training set of size m and train a **decision tree** on the new dataset.

To maximize generalization, randomize the feature choice in this way: at each node, when choosing the feature to use to split, pick a random subset of $k < n$ (usually $k = \sqrt{n}$) features and allow the algorithm to only choose from that subset of features.

This is the *random forest* algorithm.

Tree Ensemble and Random Forest

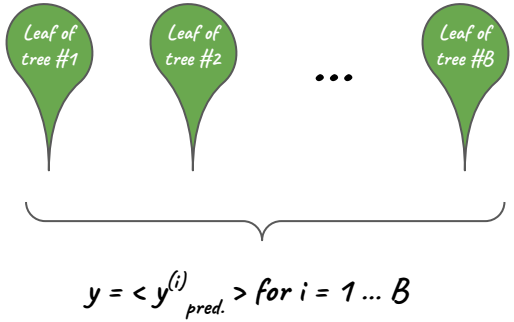
The greatest weakness of a single **decision tree** is that it clearly overfits to that particular **training set**.
The solution is to use multiple decision trees, a **tree ensemble**.
Every tree in the ensemble is built from a *sampling-with-replacement* version of the **training set**.

Classification

For classification tasks, the predicted label y is taken by choosing the class of the single leaf of the whole forest with the highest probability, or by either a majority vote coming from the final leaves of each classifier (the DTs), or (as scikit-learn does) by averaging the classifiers probabilistic predictions.

Regression

For regression tasks, the predicted label y is computed as the mean of the predictions made by the trees in the forest.



This technique is called *BAGGING*, short version for bootstrap aggregating.

For $i = 1 \dots B$ use sampling with replacement to create a new training set of size m and train a **decision tree** on the new dataset.

To maximize generalization, randomize the feature choice in this way: at each node, when choosing the feature to use to split, pick a random subset of $k < n$ (usually $k = \sqrt{n}$) features and allow the algorithm to only choose from that subset of features.

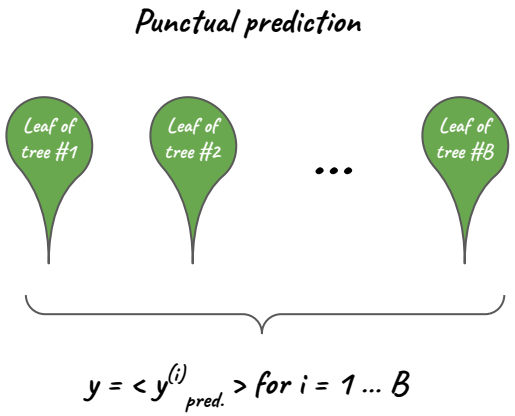
This is the *random forest* algorithm.

In this way, a random forest regressor is able to give a punctual prediction given a certain set of features.

Random Forests and the posterior distribution function

In this way, a random forest regressor is able to give a punctual prediction given a certain set of features.
As in NN algorithms, there is a trick to extract a sort of posterior distribution function for that predicted label ***y***, under the same assumption that similar observations will lead to similar PDFs:

- the punctual prediction ***y*** is the mean of all the predictions made by the trees in the forest, e.g. between the values in the final leaves

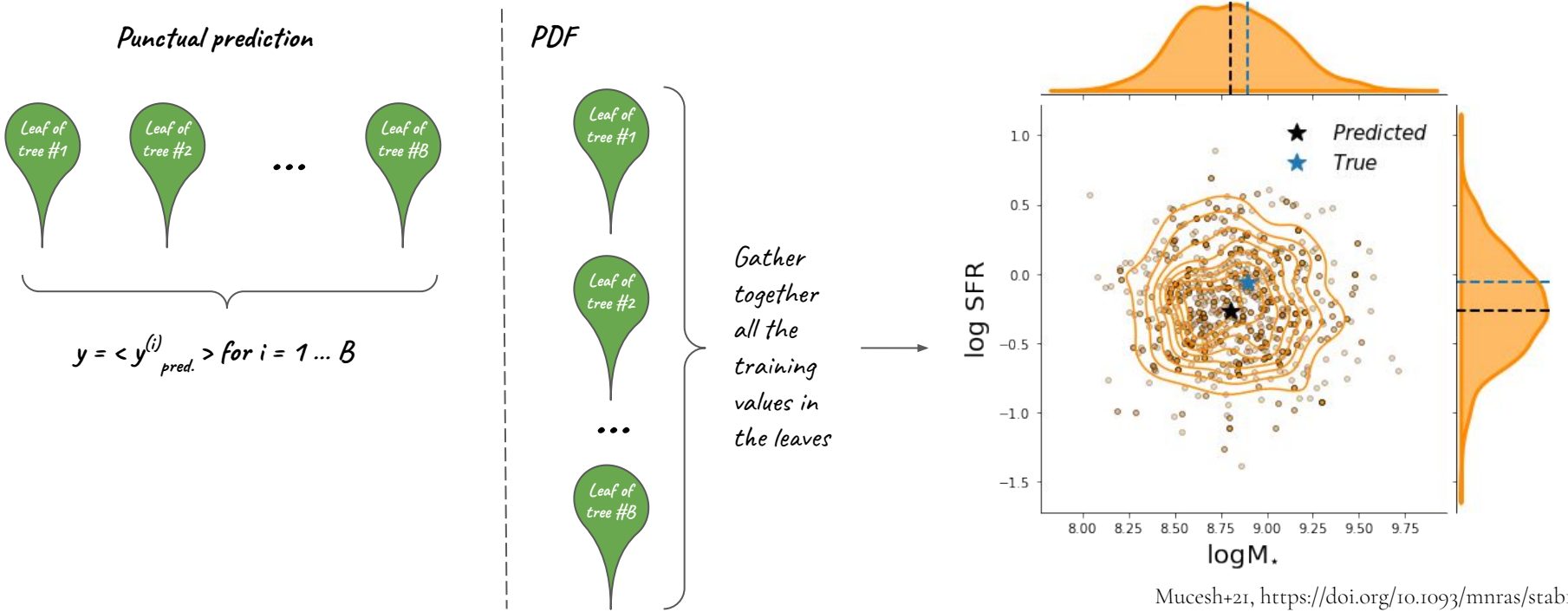


Mucesh+21, https://doi.org/10.1093/mnras/stab164

Random Forests and the posterior distribution function

In this way, a random forest regressor is able to give a punctual prediction given a certain set of features.
As in NN algorithms, there is a trick to extract a sort of posterior distribution function for that predicted label ***y***, under the same assumption that similar observations will lead to similar PDFs:

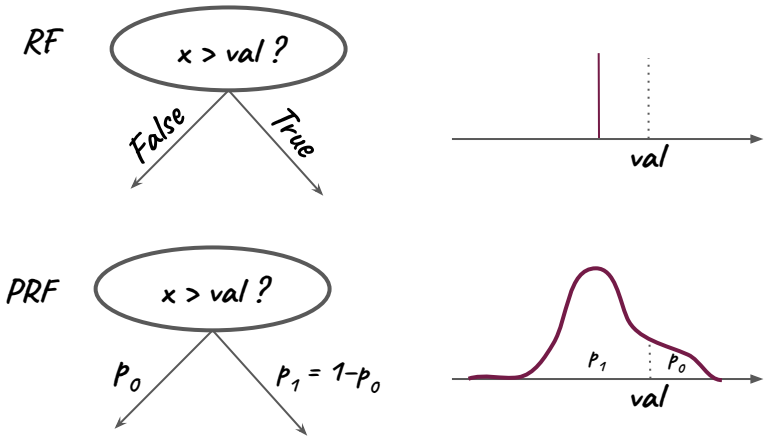
- the punctual prediction ***y*** is the mean of all the predictions made by the trees in the forest, e.g. between the values in the final leaves
- the posterior distribution function ***PDF*** is obtained by putting together the training values in the final leaves, and estimate their density (e.g. with KDE)



Mucesh+21, https://doi.org/10.1093/mnras/stab164

Turns out there is also a way for factoring in the noise in the features/labels.
Probabilistic Random Forest (PRF) algorithm treats the features and labels as probability distribution functions, rather than deterministic quantities.

Labels and features as PDFs



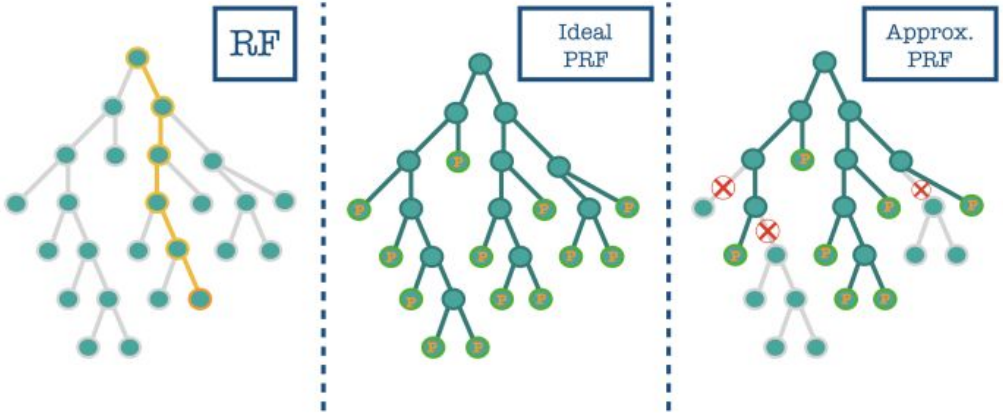
"In the presence of feature uncertainty the split is not deterministic. At every node each object may propagate into both branches with some probability. Therefore, while in an RF tree an object goes through a single trajectory, in a PRF tree the object may propagate through all the tree nodes, with some probability."

The final probability of each object at the leaf nodes is the combined probability for it to take all the turn that led to it from the root node.

Reis+18, https://doi.org/10.3847/1538-3881/aaf101

Turns out there is also a way for factoring in the noise in the features/labels.
Probabilistic Random Forest (PRF) algorithm treats the features and labels as probability distribution functions, rather than deterministic quantities.

Labels and features as PDFs



"In the presence of feature uncertainty the split is not deterministic. At every node each object may propagate into both branches with some probability. Therefore, while in an RF tree an object goes through a single trajectory, in a PRF tree the object may propagate through all the tree nodes, with some probability."
The final probability of each object at the leaf nodes is the combined probability for it to take all the turn that led to it from the root node.

Training of a PRF

Training is done by choosing the most homogeneous possibile splits, as in RF, but using a modified version of the *Gini impurity* as the cost function. Stopping criterion and other details are the same as in a normal RF.

Prediction by a PRF

The prediction is determined via majority vote (as in the scikit-learn version of RF), where the vote of each tree comes from the leaf that achieved the *highest* probability.

Selective propagation scheme

This works as smooth as silk from a theoretical point of view, but it is computationally ***EXTREMELY*** expensive. Therefore not all the possible propagation schemes down the trees are maintained, but only the ones over a certain threshold (e.g. 5%).

Reis+18, https://doi.org/10.3847/1538-3881/aaf101

Random forests are fine, but there is a way to increase their ability to generalize and make accurate predictions, up to the levels where only NN now are used to play: **gradient boosting**.
The idea is quite simple: focus more on the subsets that are underperforming, and learn how to make them perform better. Combine multiple badly performing trees (weak learners) to make one performing strong learner.
This is achieved with a tweak to the bagging technique:

For $i = 1 \dots B$ use sampling with replacement to create a new training set of size m and train a decision tree on the new dataset.

Random forests are fine, but there is a way to increase their ability to generalize and make accurate predictions, up to the levels where only NN now are used to play: **gradient boosting**.
The idea is quite simple: focus more on the subsets that are underperforming, and learn how to make them perform better. Combine multiple badly performing trees (weak learners) to make one performing strong learner.
This is achieved with a tweak to the bagging technique:

> For $i = 1 \dots B$ use sampling with replacement to create a new training set of size m, but instead of picking from all the possible examples with equal $(1/m)$ probability, make it more likely to pick misclassified examples from previously trained tree; or assign different weights to the training examples, which will be higher for misclassified examples, and train a decision tree on the new dataset.

Random forests are fine, but there is a way to increase their ability to generalize and make accurate predictions, up to the levels where only NN now are used to play: **gradient boosting**.
The idea is quite simple: focus more on the subsets that are underperforming, and learn how to make them perform better. Combine multiple badly performing trees (weak learners) to make one performing strong learner.
This is achieved with a tweak to the bagging technique:

> For $i = 1 \dots B$ use sampling with replacement to create a new training set of size m, but instead of picking from all the possible examples with equal $(1/m)$ probability, make it more likely to pick misclassified examples from previously trained tree; or assign different weights to the training examples, which will be higher for misclassified examples, and train a decision tree on the new dataset.

AdaBoost:

a split made on a single feature

> In the first iteration of AdaBoost, each example is assigned an equal weight. After building and evaluating the first *stump*, increase the weights of those examples that are difficult to classify, lower the weights for those that are easily classified, and build a second stump. Repeat for a certain number of iterations. Predictions of the final ensemble model is therefore the weighted sum of the predictions made by all the previous stumps.

Random forests are fine, but there is a way to increase their ability to generalize and make accurate predictions, up to the levels where only NN now are used to play: **gradient boosting**.
The idea is quite simple: focus more on the subsets that are underperforming, and learn how to make them perform better. Combine multiple badly performing trees (weak learners) to make one performing strong learner.
This is achieved with a tweak to the bagging technique:

> For $i = 1 \dots B$ use sampling with replacement to create a new training set of size m, but instead of picking from all the possible examples with equal $(1/m)$ probability, make it more likely to pick misclassified examples from previously trained tree; or assign different weights to the training examples, which will be higher for misclassified examples, and train a decision tree on the new dataset.

AdaBoost:

a split made on a single feature

In the first iteration of AdaBoost, each example is assigned an equal weight. After building and evaluating the first *stump*, increase the weights of those examples that are difficult to classify, lower the weights for those that are easily classified, and build a second stump. Repeat for a certain number of iterations. Predictions of the final ensemble model is therefore the weighted sum of the predictions made by all the previous stumps.

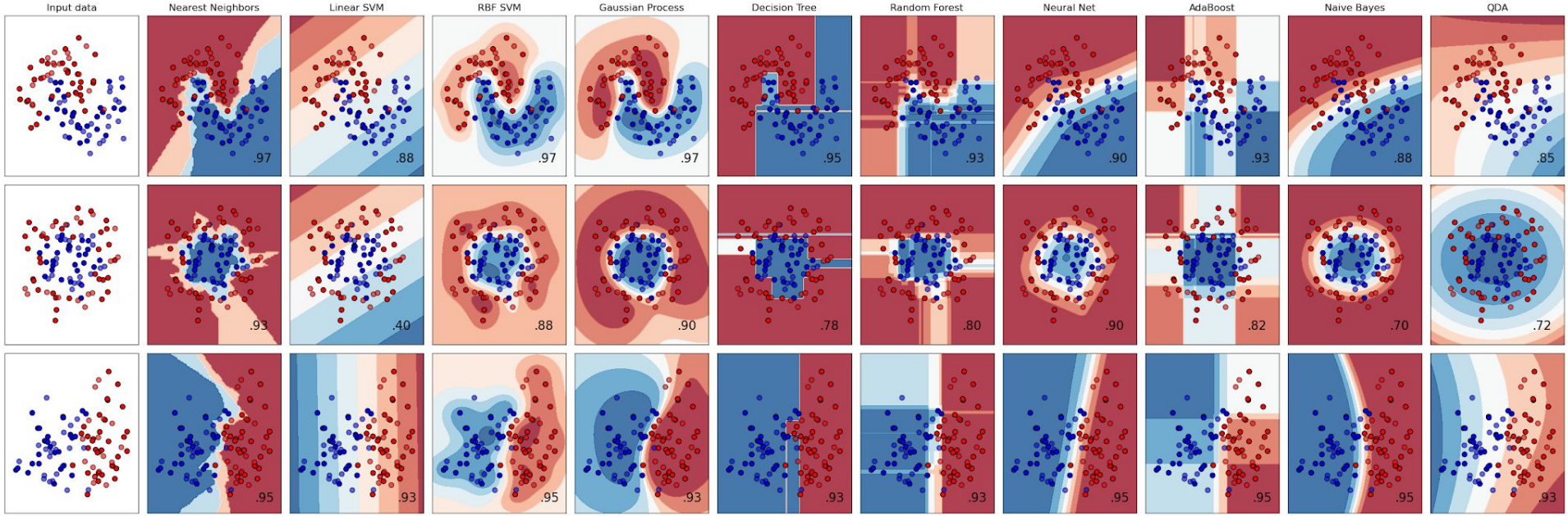
Gradient Boosted Decision Trees (e.g. scikit-learn):

Similar as above, but building full trees, and instead of trying to directly predict the labels y, the model will try to predict the ***residuals*** between the labels y and the predictions y' (MSE loss), trying to minimize them at each iteration. The trees predictions are scaled with a learning rate and the whole process is regularized in order to increase generalization and avoid overfitting.

XGBoost (eXtreme Gradient Boosting)

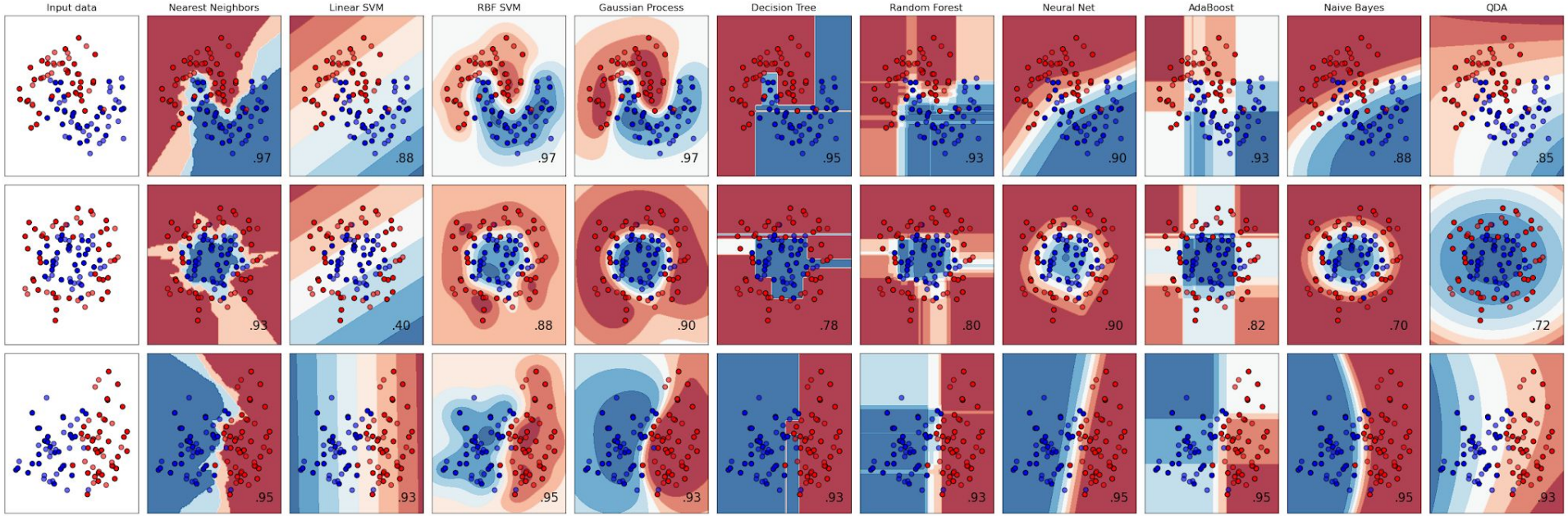
Similar as above, with small differences and details on tree growth policy, it is the most widely used ensemble methods, the one that is able to compete with Deep Learning algorithms in terms of performances.

Classifiers, classified



Classifiers, classified

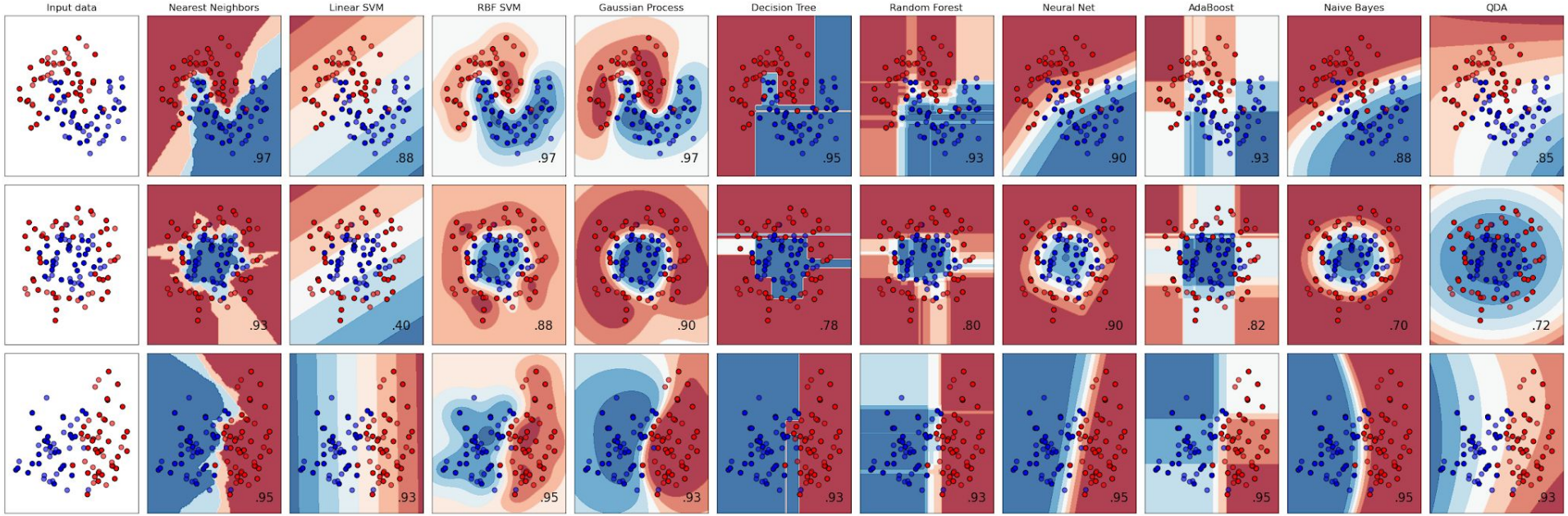
AH-HA STUPID
NEURAL NETWORK



Classifiers, classified

powerful, it's just that I cannot ignore that flebile overfitting aftertaste

AH-HA STUPID NEURAL NETWORK

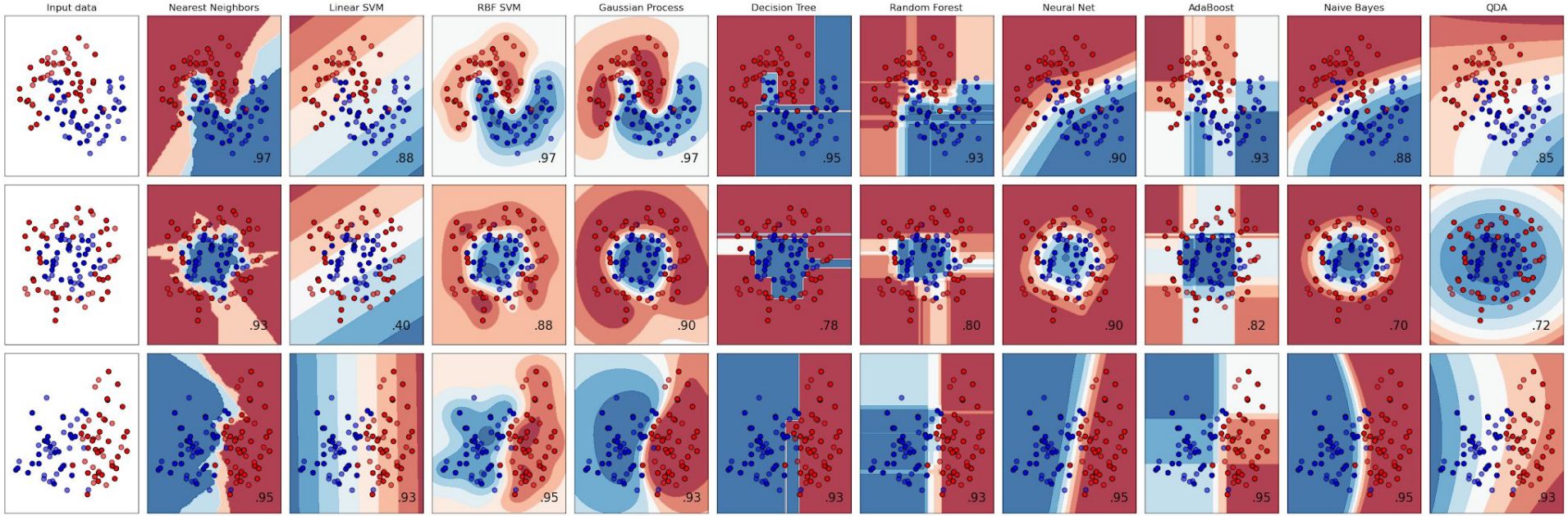


Classifiers, classified

powerful, it's just that I cannot ignore that flebile overfitting aftertaste

Linear SVM is unable to identify non-linear decision bounds...

AH-HA STUPID NEURAL NETWORK



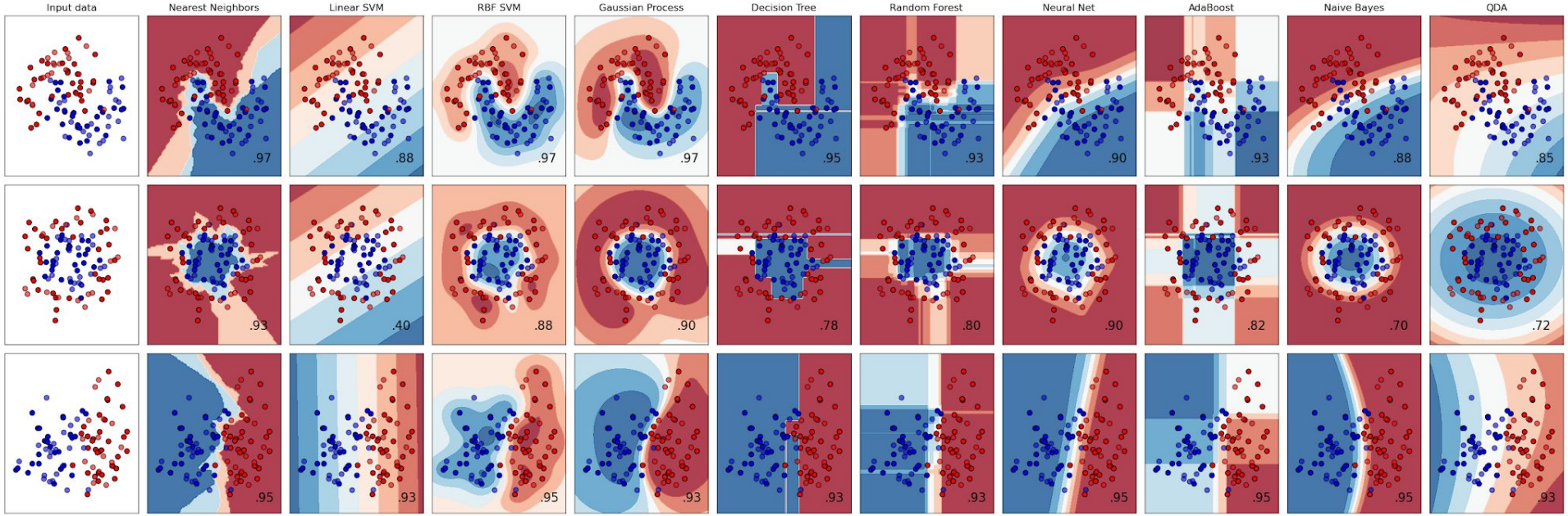
Classifiers, classified

powerful, it's just that I cannot ignore that flebile overfitting aftertaste

Linear SVM is unable to identify non-linear decision bounds...

...and that's why we use the kernel trick (e.g. RBFs)

AH-HA STUPID NEURAL NETWORK



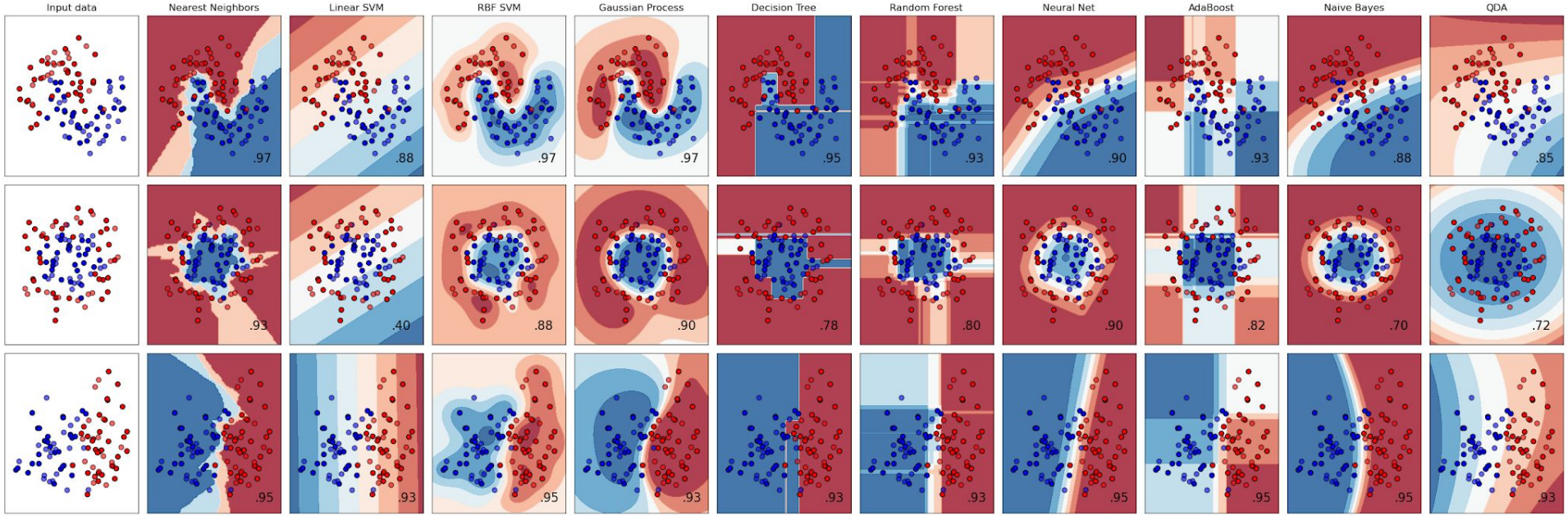
Classifiers, classified

powerful, it's just that I cannot ignore that flebile overfitting aftertaste

Linear SVM is unable to identify non-linear decision bounds...

...and that's why we use the kernel trick (e.g. RBFs)

AH-HA STUPID NEURAL NETWORK



a very complicated mathematical tractation, but look at those nice results

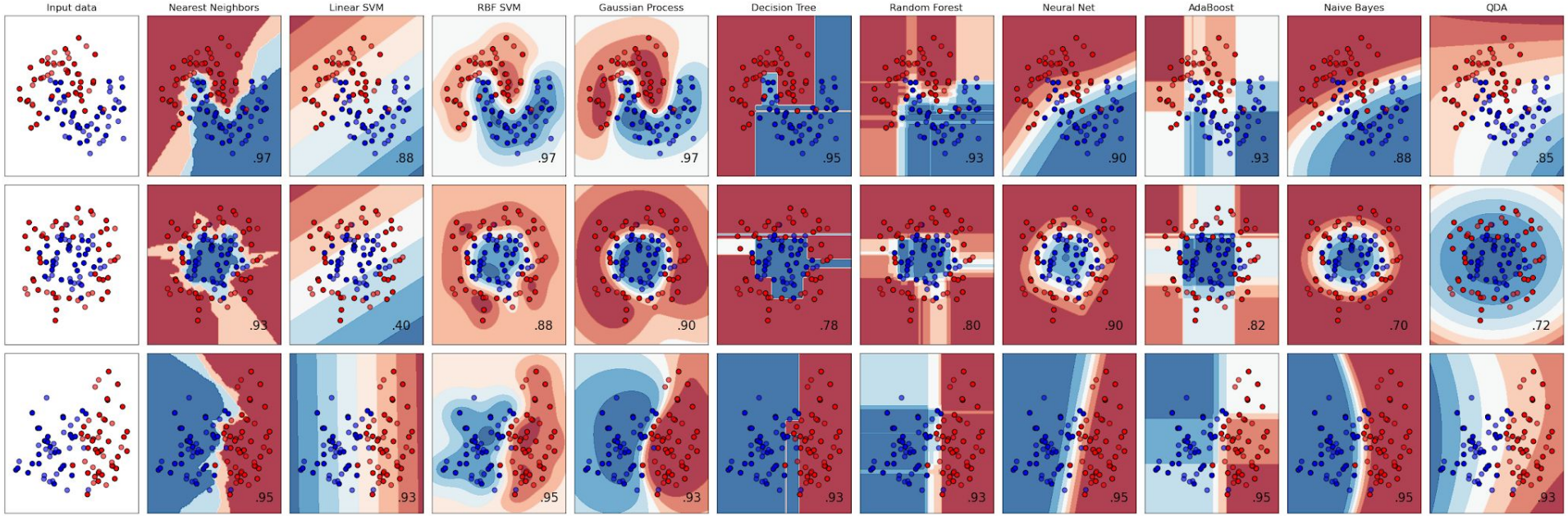
Classifiers, classified

powerful, it's just that I cannot ignore that flebile overfitting aftertaste

Linear SVM is unable to identify non-linear decision bounds...

...and that's why we use the kernel trick (e.g. RBFs)

AH-HA STUPID NEURAL NETWORK



a very complicated mathematical tractation, but look at those nice results

a single DT is very prone to overfitting...

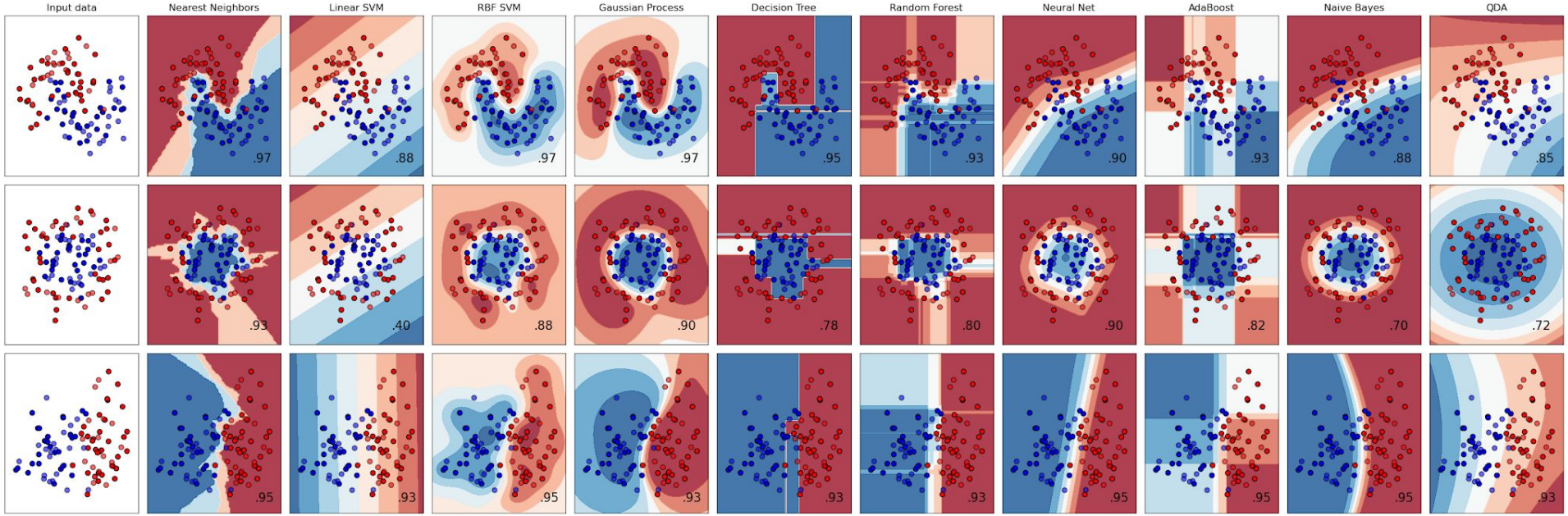
Classifiers, classified

powerful, it's just that I cannot ignore that flebile overfitting aftertaste

Linear SVM is unable to identify non-linear decision bounds...

...and that's why we use the kernel trick (e.g. RBFs)

AH-HA STUPID NEURAL NETWORK



a very complicated mathematical tractation, but look at those nice results

a single DT is very prone to overfitting...

... and that's why RFs and GBMs exists

Classifiers, classified

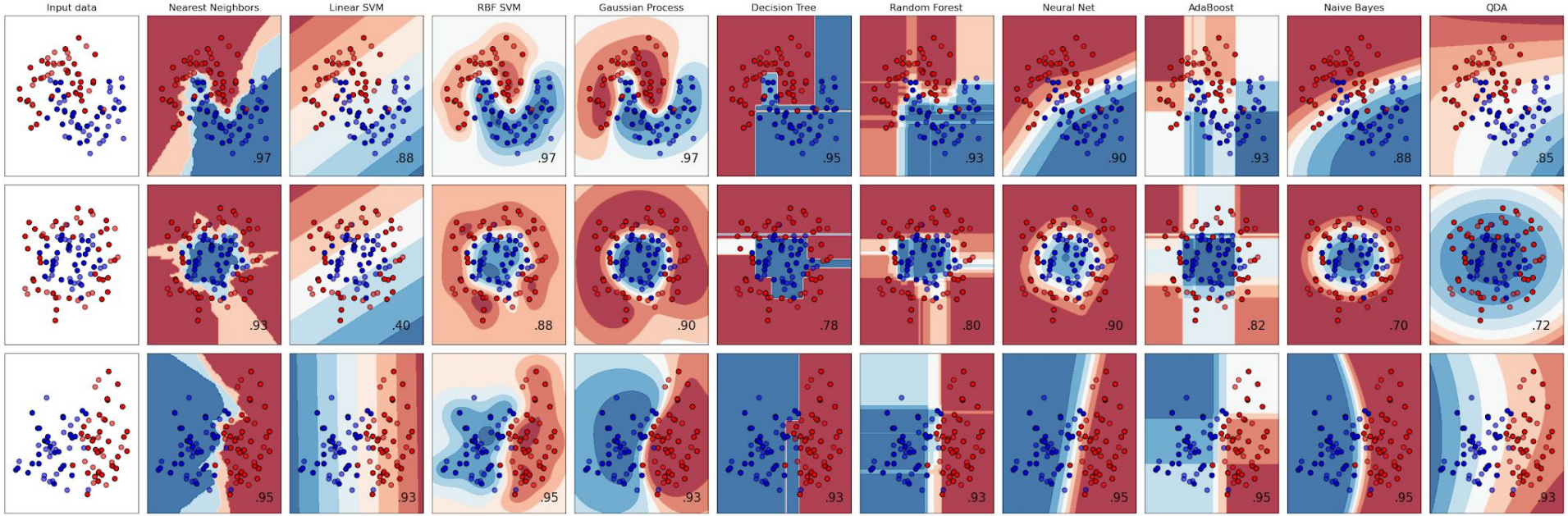
powerful, it's just that I cannot ignore that flebile overfitting aftertaste

Linear SVM is unable to identify non-linear decision bounds...

...and that's why we use the kernel trick (e.g. RBFs)

DTs, RFs and GBMs Mondrianize the feature space

AH-HA STUPID NEURAL NETWORK



a very complicated mathematical tractation, but look at those nice results

a single DT is very prone to overfitting...

... and that's why RFs and GBMs exists