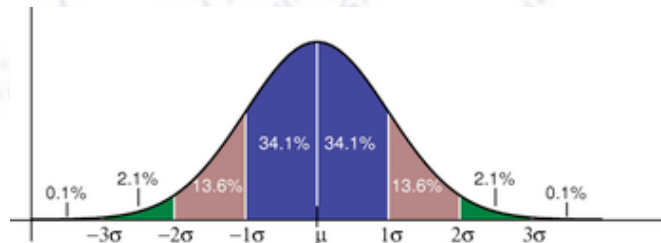


Machine Learning

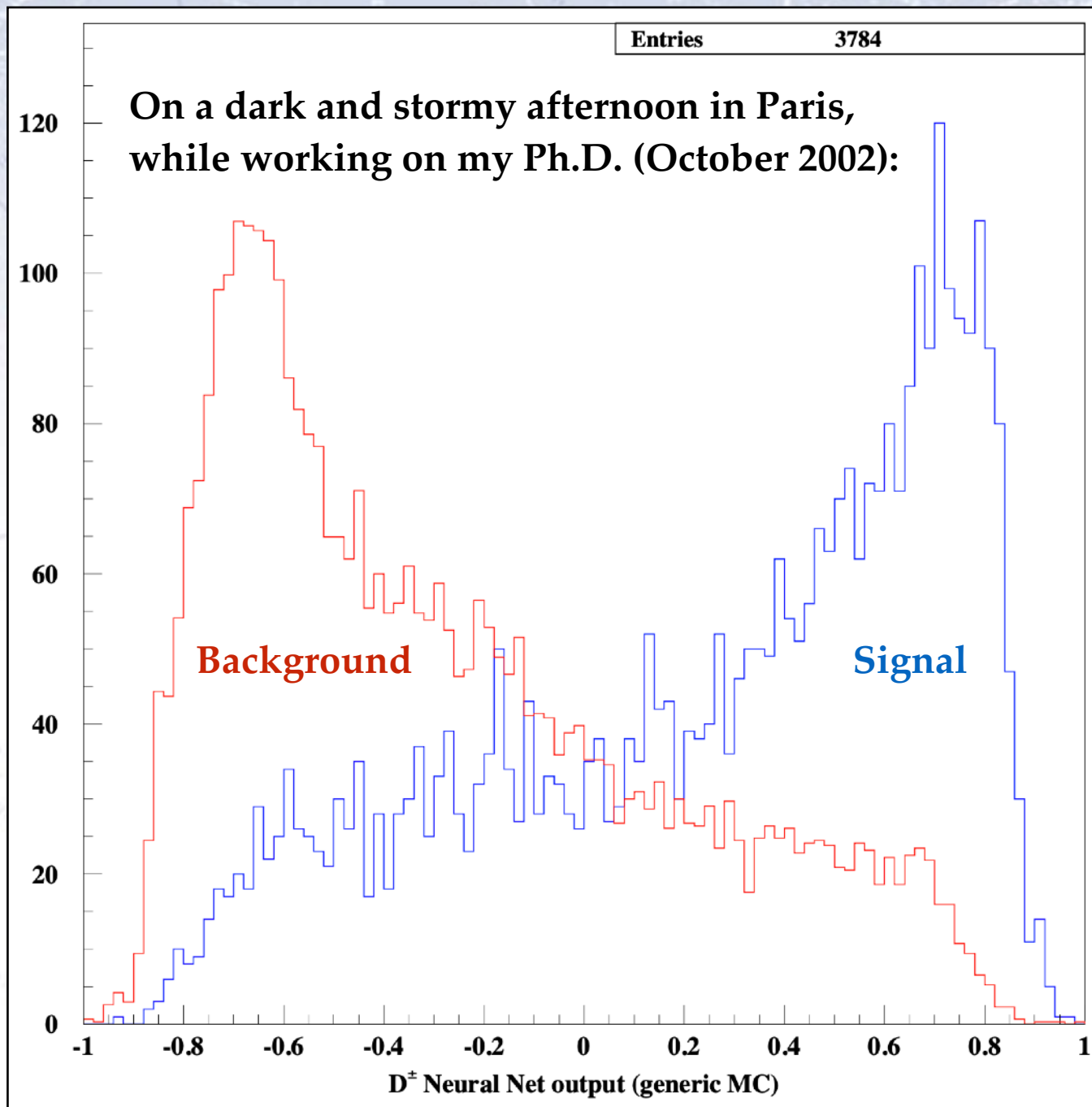
Basics of ML



Troels C. Petersen (NBI)



"Statistics is merely a quantisation of common sense - Machine Learning is a sharpening of it!"



Comment on “The AI Hype”

Machine Learning is a tool like all others (logic, math, computers, statistics, etc.)

Despite the connotations of machine learning and artificial intelligence as a mysterious and radical departure from traditional approaches, we stress that machine learning has a mathematical formulation that is closely tied to statistics, the calculus of variations, approximation theory, and optimal control theory.

[PDG 2024, Review of Machine Learning]

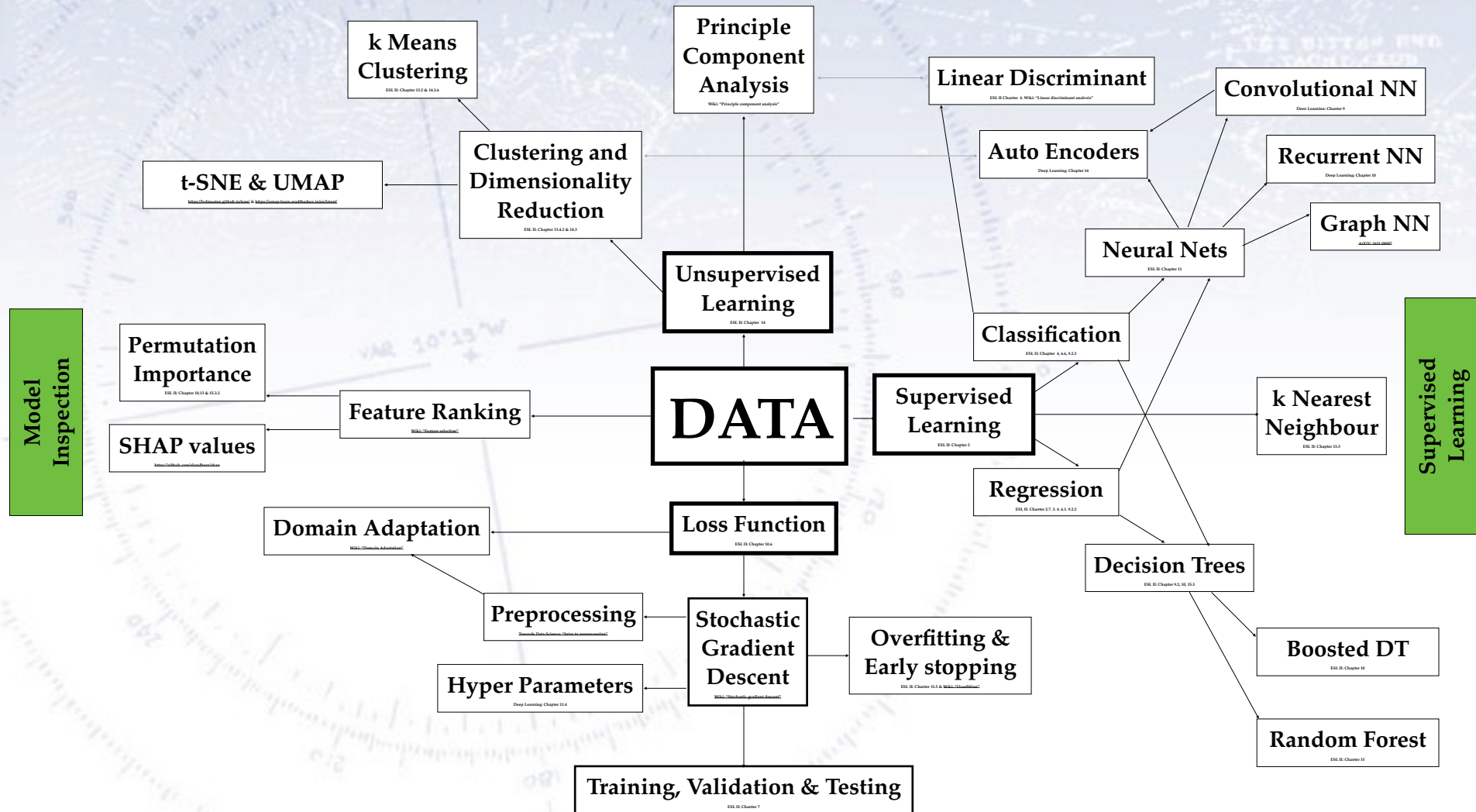
So this is just a sharpening of our tools... albeit a cool sharpening!

Applied Machine Learning

Unsupervised Learning

Overview of subjects

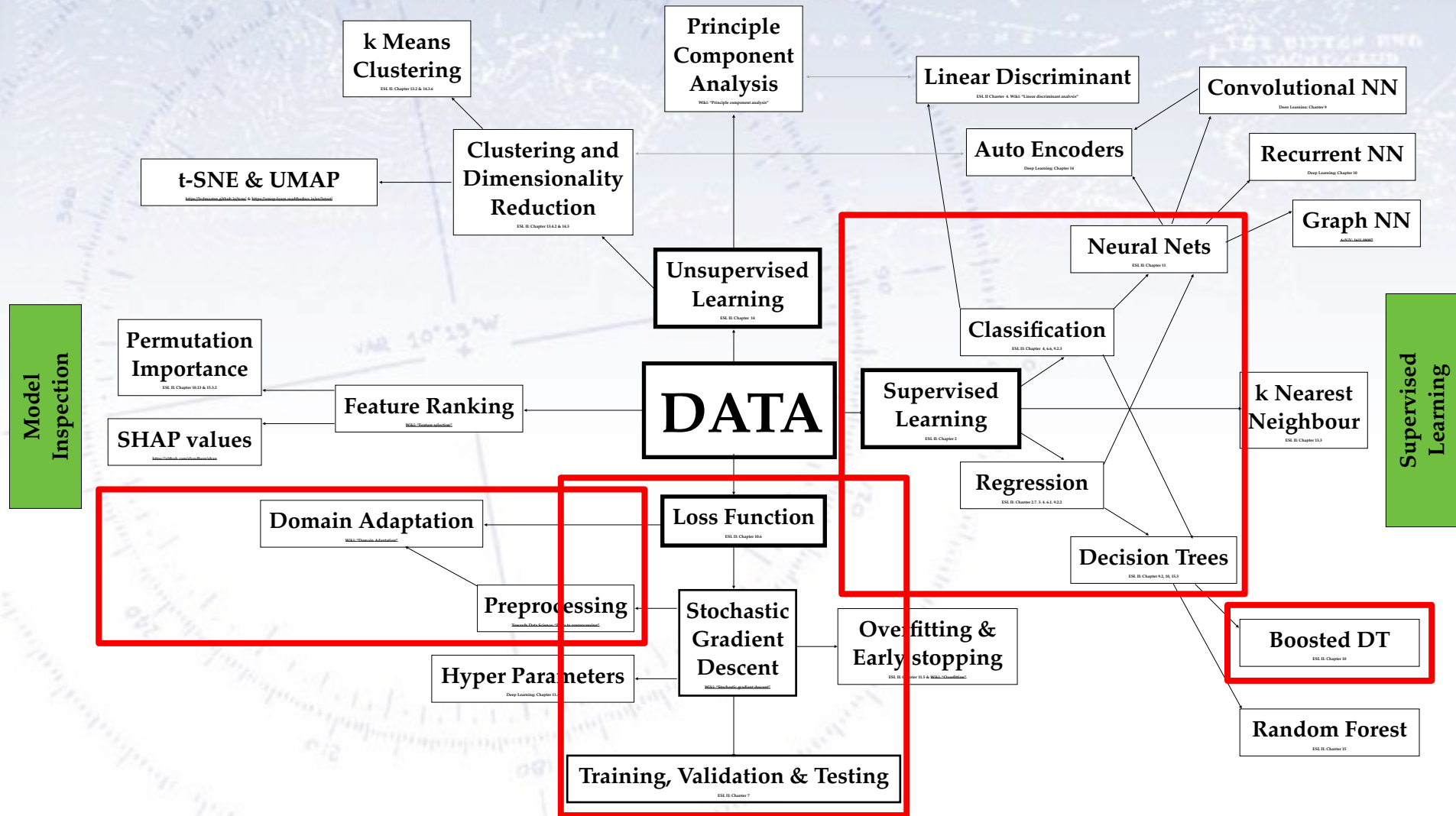
Version 1.3, 14. March 2024



References:

Trevor Hastie et al.: "Elements of Statistics Learning II" (ELS II)
Ian Goodfellow et al.: "Deep Learning"
Wiki: Good reference for most subjects (only specified when essential)
Various blogs / githubs / papers for specific subjects.

Model Optimisation



Outline

What is ML & Humans vs. ML

Two main ingredients:

- Universal Approximation Theorems
- Stochastic Gradient Descent

The linear vs. non-linear case

Tree based models

Neural Network models

Loss functions

Train, Validation & Test

Preprocessing

Domain Adaptation:

- What are the dangers?
- MC signal, data background
- How to find data-MC differences
- How to mend data-MC differences
- Training in/ with data

Coding example

The background is a faded nautical chart. It features a large compass rose with concentric circles and radial lines indicating degrees. Depth soundings are visible as numbers along the coastlines. The word 'MAGNETIC' is printed on the chart. In the upper right, there is a label '1ST BITTER END YACHT CLUB'.

What is ML?

What is Machine Learning?

While there is no formal definition, an early attempt is the following intuition:

“Machine learning programs can perform tasks without being explicitly programmed to do so.”

[Arthur Samuel, US computer pioneer 1901-1990]

“Little Peter is capable of finding his way home without being explicitly taught to do so.”

What is Machine Learning?

While there is no formal definition, an early attempt is the following intuition:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T , as measured by P , improves with experience E ."

[T. Mitchell, "Machine Learning" 1997]

"Little Peter is said to learn from traveling around with respect to finding his way home and the time it takes, if his ability to find his way home, as measured by the time it takes, improves as he travels around."



Humans vs. ML

Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!

However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: Computers:	Humans: Computers:
Non-linear	Humans: Computers:	Humans: Computers:

Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

However, through smart algorithms, computers have learned to deal with it all!

That is essentially what Machine Learning has enabled!

Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!

However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: Computers:	Humans: Computers:
Non-linear	Humans: Computers:	Humans: Computers:

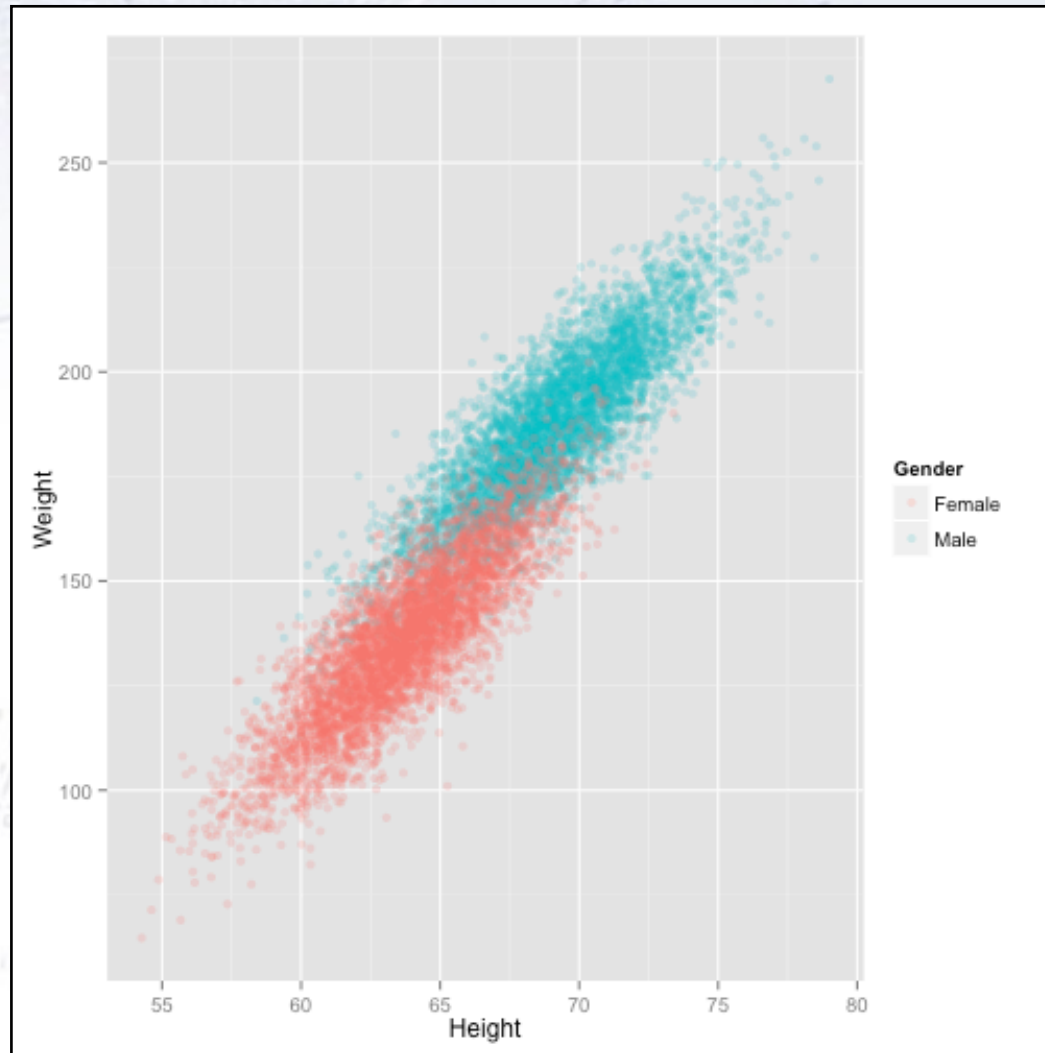
Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

However, through smart algorithms, computers have learned to deal with it all!

That is essentially what Machine Learning has enabled!

Dimensionality and Complexity

Humans & Computers are good at seeing/understanding linear data in few dimensions:



Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!

However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: ✓ Computers: ✓	Humans: Computers:
Non-linear	Humans: Computers:	Humans: Computers:

Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

However, through smart algorithms, computers have learned to deal with it all!

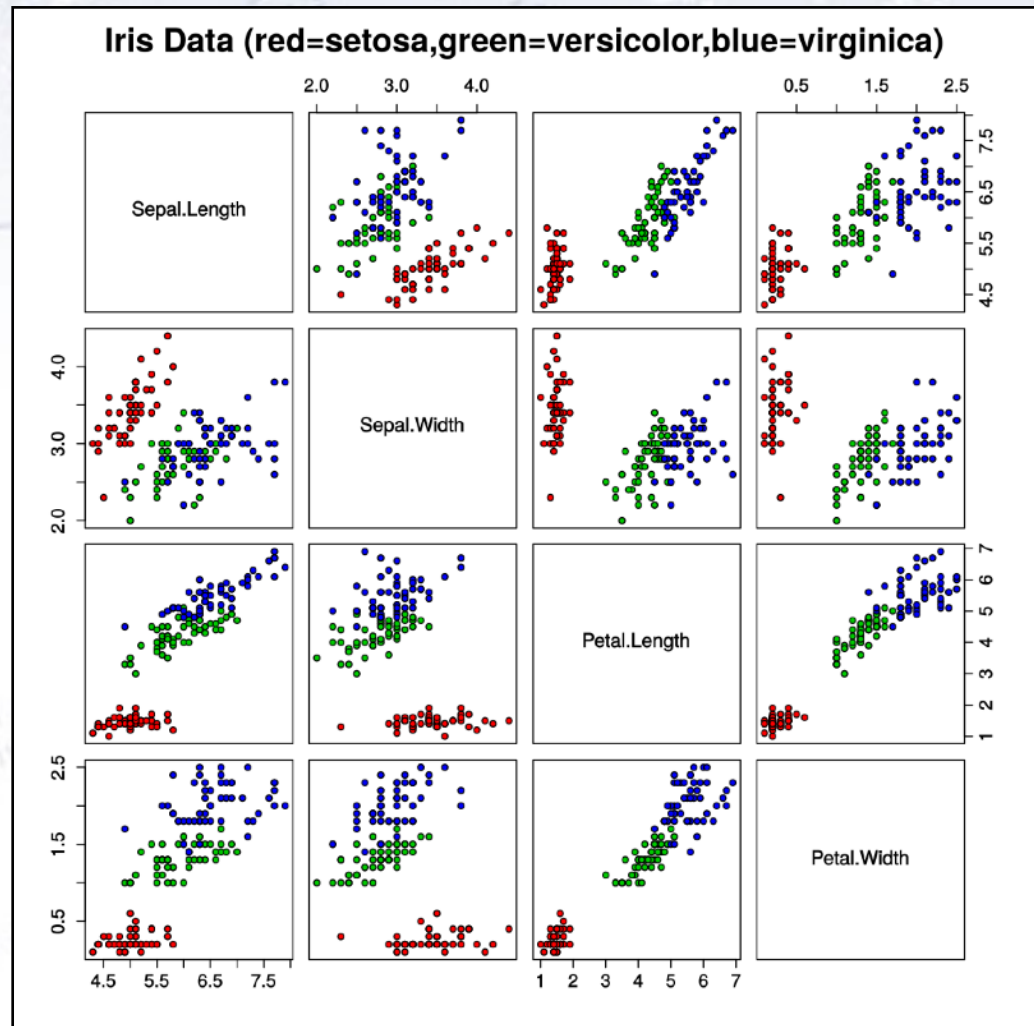
That is essentially what Machine Learning has enabled!

Dimensionality and Complexity

However, when the dimensionality goes beyond 3D, we are lost, even for simple linear data. Computers are not...

Shown is the famous
Fisher Iris dataset:
150 irises (3 kinds) with
4 measurements for each.

4 dimensional data!



Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!

However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: ✓ Computers: ✓	Humans: ÷ Computers: ✓
Non-linear	Humans: Computers:	Humans: Computers:

Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

However, through smart algorithms, computers have learned to deal with it all!

That is essentially what Machine Learning has enabled!



Jackson Pollock

Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!

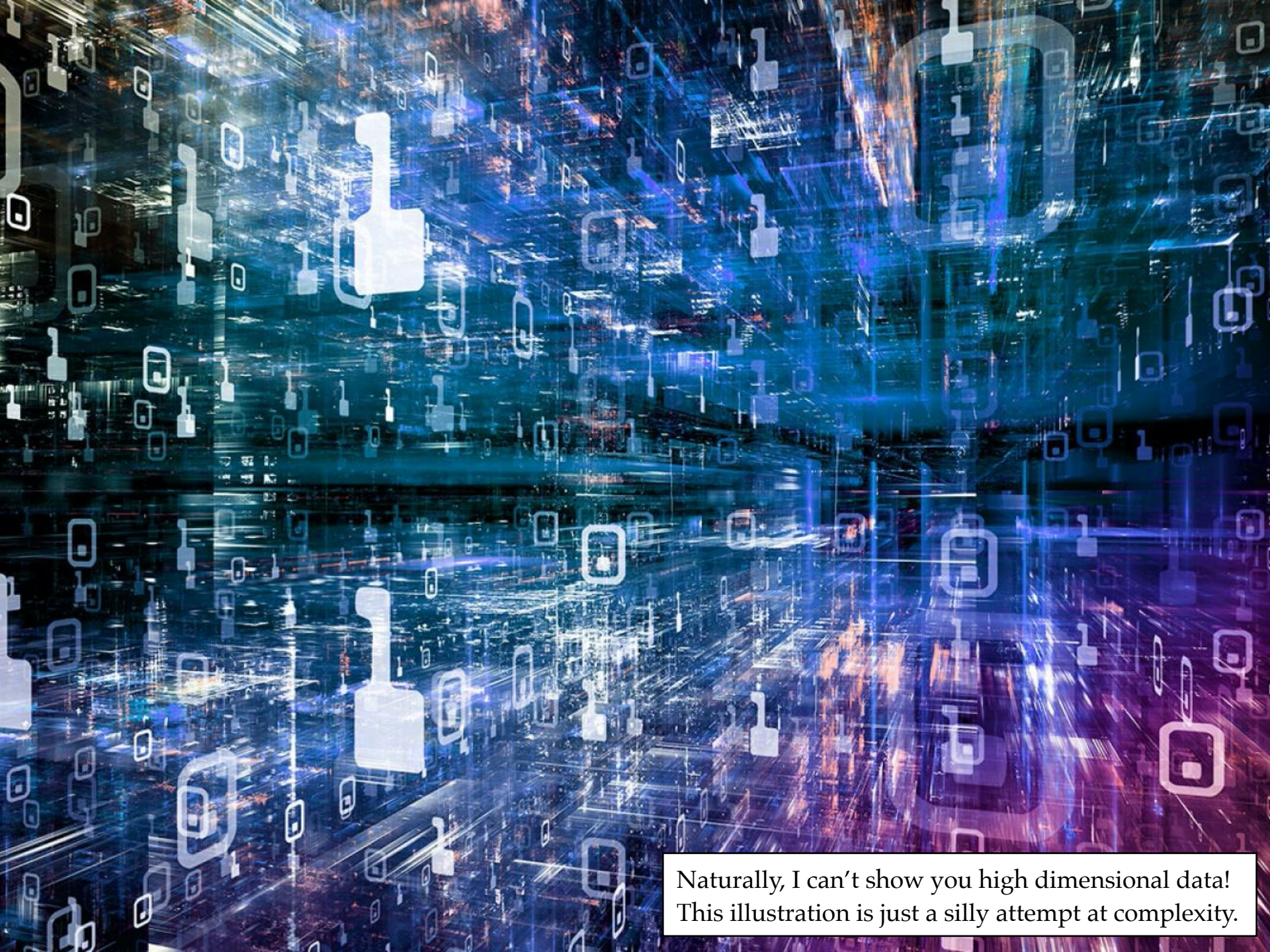
However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: ✓ Computers: ✓	Humans: ÷ Computers: ✓
Non-linear	Humans: ✓ Computers: (✓)	Humans: Computers:

Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

However, through smart algorithms, computers have learned to deal with it all!

That is essentially what Machine Learning has enabled!



Naturally, I can't show you high dimensional data!
This illustration is just a silly attempt at complexity.

Dimensionality and Complexity

Humans are good at seeing/understanding data in few dimensions!

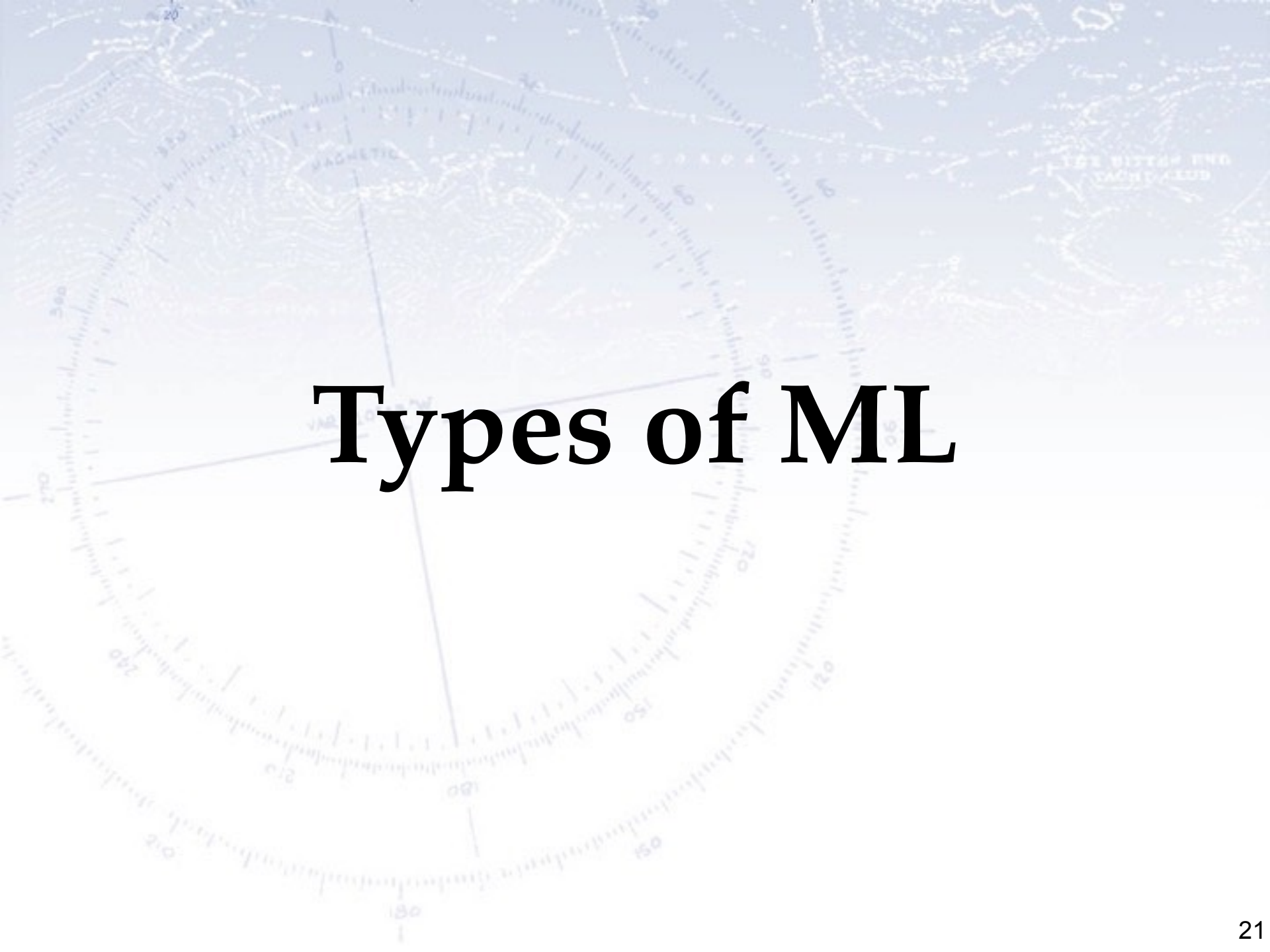
However, as dimensionality grows, complexity grows exponentially (“curse of dimensionality”), and humans are generally not geared for such challenges.

	Low dim.	High dim.
Linear	Humans: ✓ Computers: ✓	Humans: ÷ Computers: ✓
Non-linear	Humans: ✓ Computers: (✓)	Humans: ÷ Computers: (✓)

Computers, on the other hand, are OK with high dimensionality, albeit the growth of the challenge, but have a harder time facing non-linear issues.

However, through smart algorithms, computers have learned to deal with it all!

That is essentially what Machine Learning has enabled!



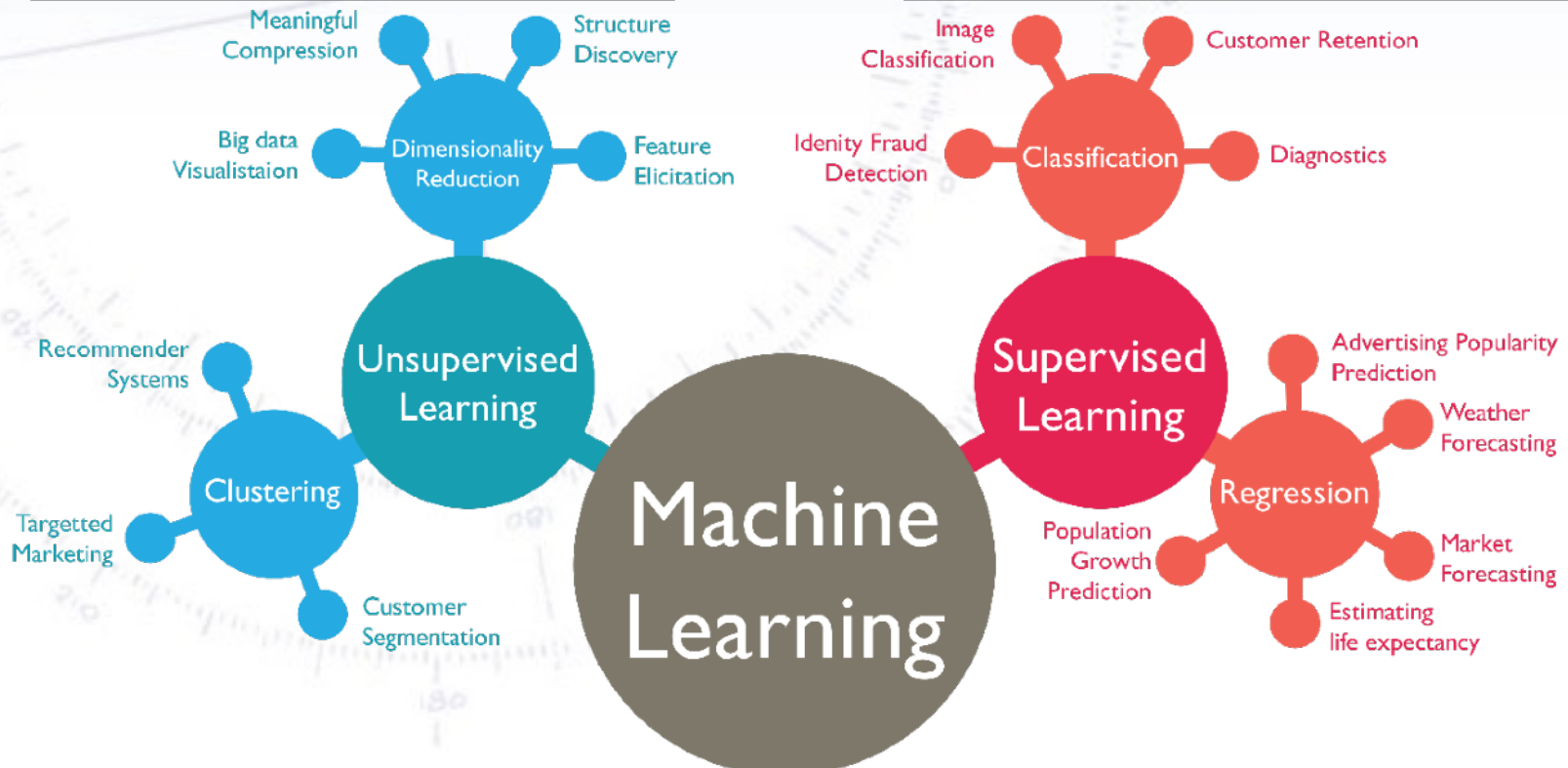
Types of ML

Unsupervised vs. Supervised Classification vs. Regression

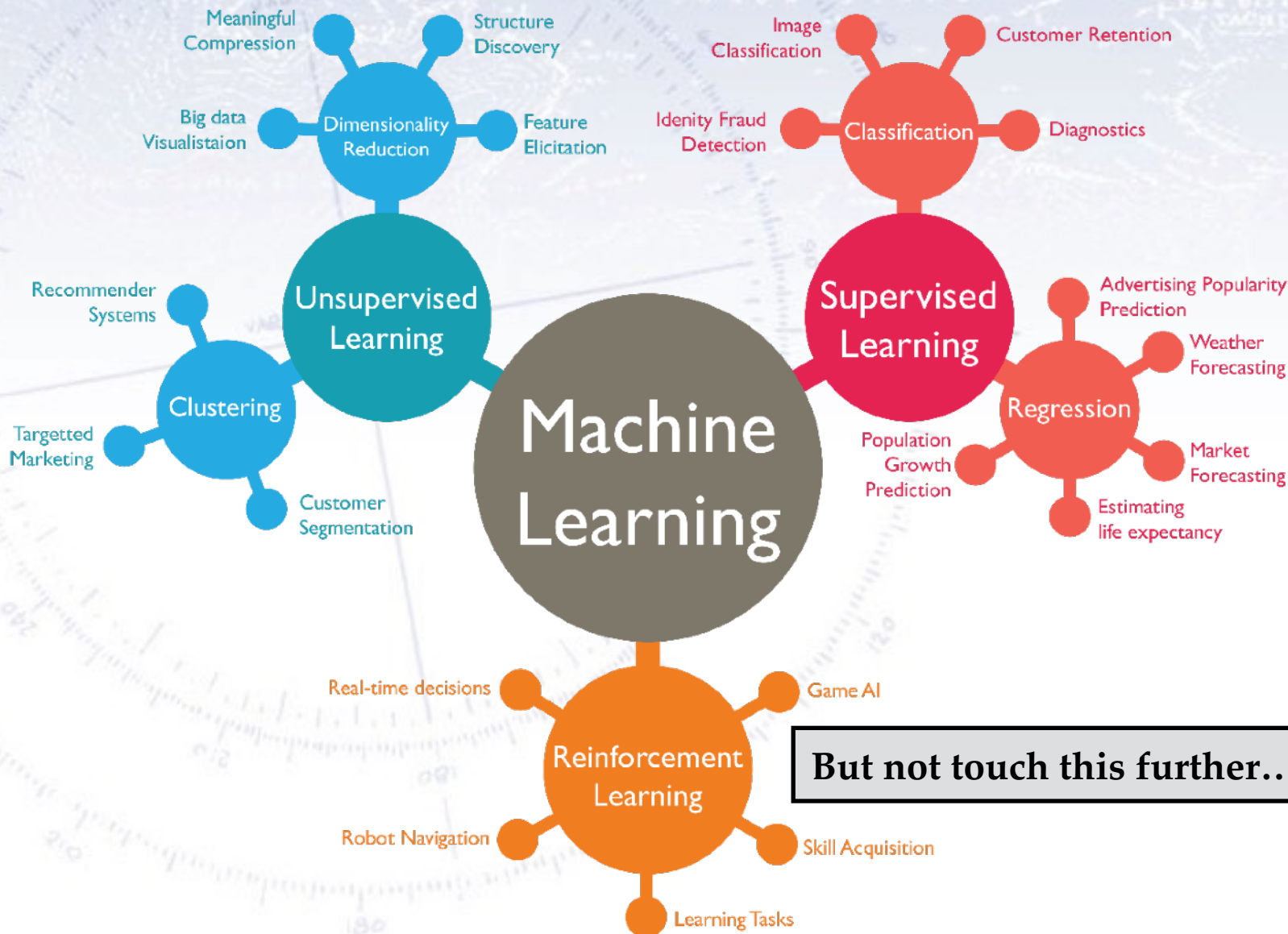
Machine Learning can be supervised (you have correctly labelled examples) or unsupervised (you don't)... [or reinforced]. Following this, one can be using ML to either classify (is it A or B?) or for regression (estimate of X).

But of course also over here!

We will be mostly on this side!



Reinforcement Learning





Two main ingredients:

- 1. Solutions exists**
- 2. How to find them**

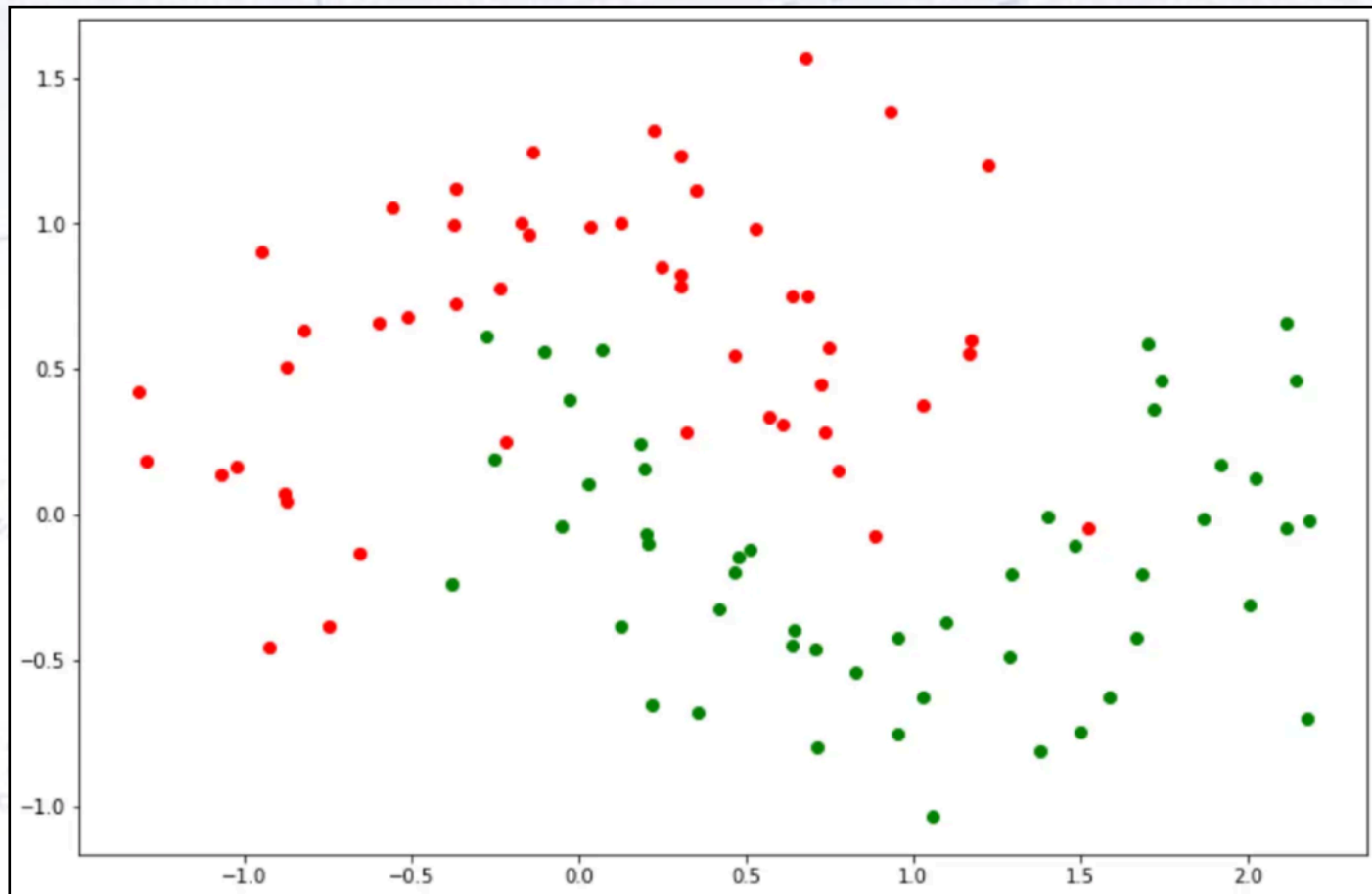


Solutions exists

(Technically called Universal Approximation Theorems)

Where to separate?

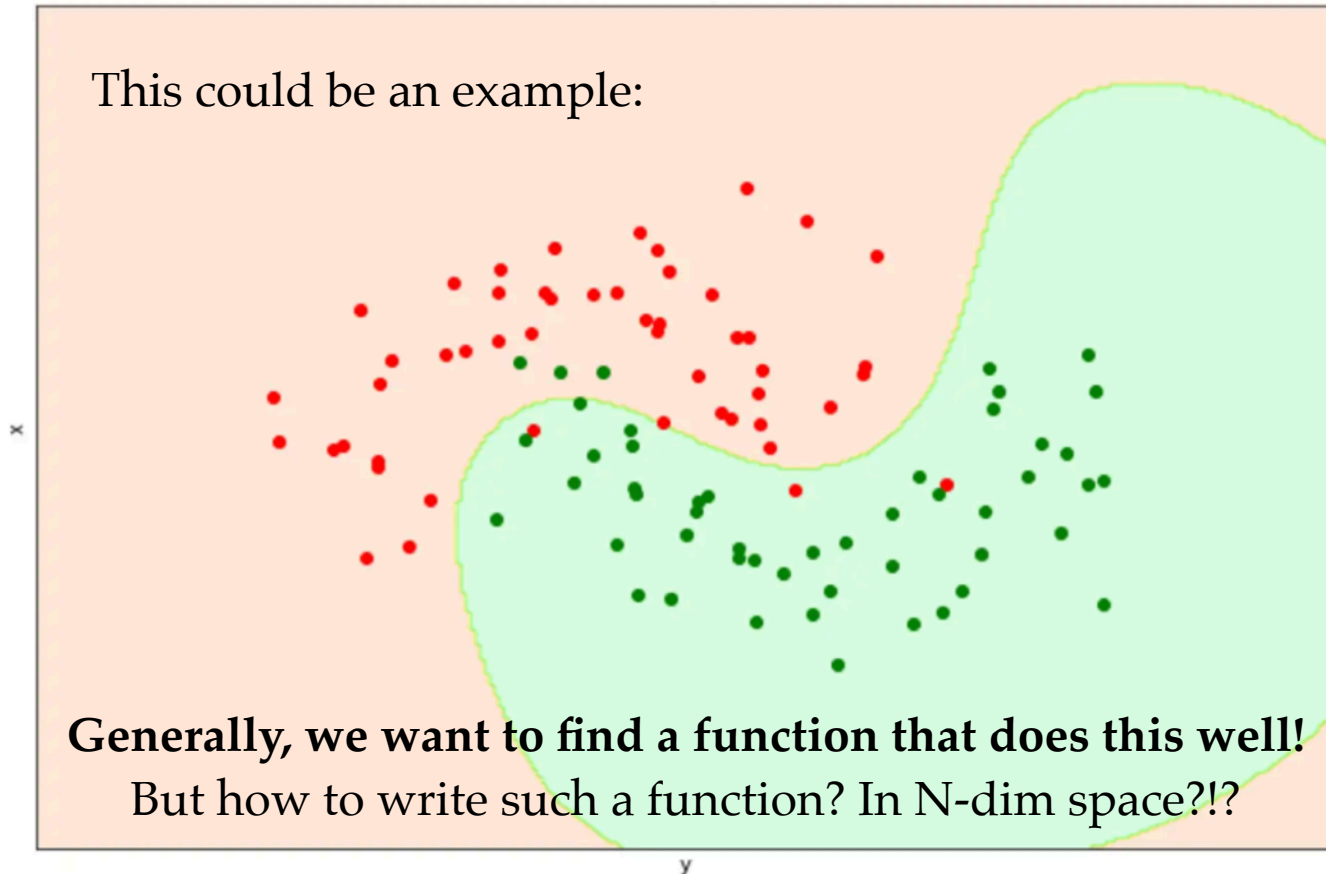
Look at the red and green points, and imagine that you wanted to draw a curve that separates these.



Where to separate?

Look at the red and green points, and imagine that you wanted to draw a curve that separates these.

This could be an example:



Generally, we want to find a function that does this well!
But how to write such a function? In N-dim space?!?

Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

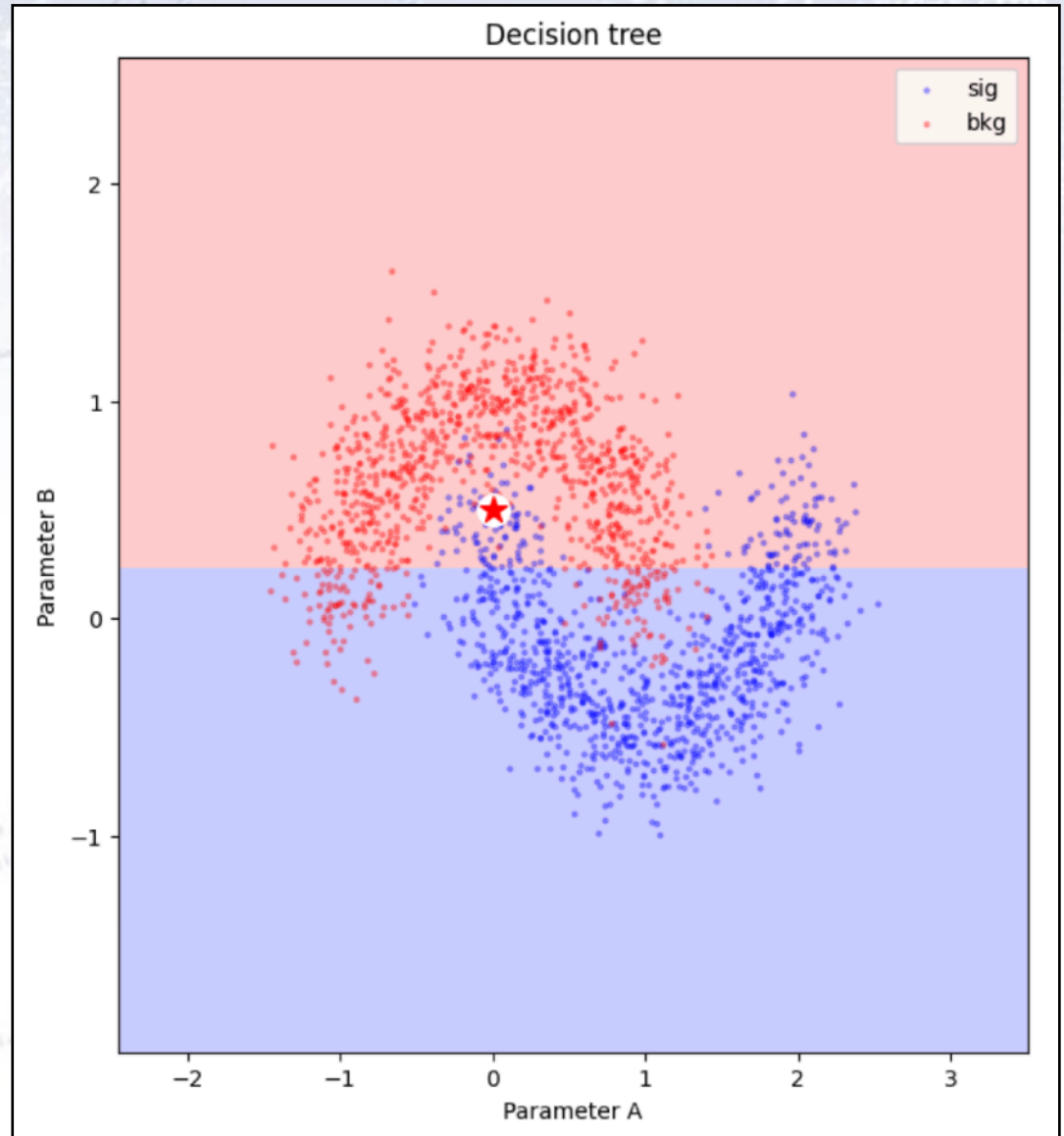
Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

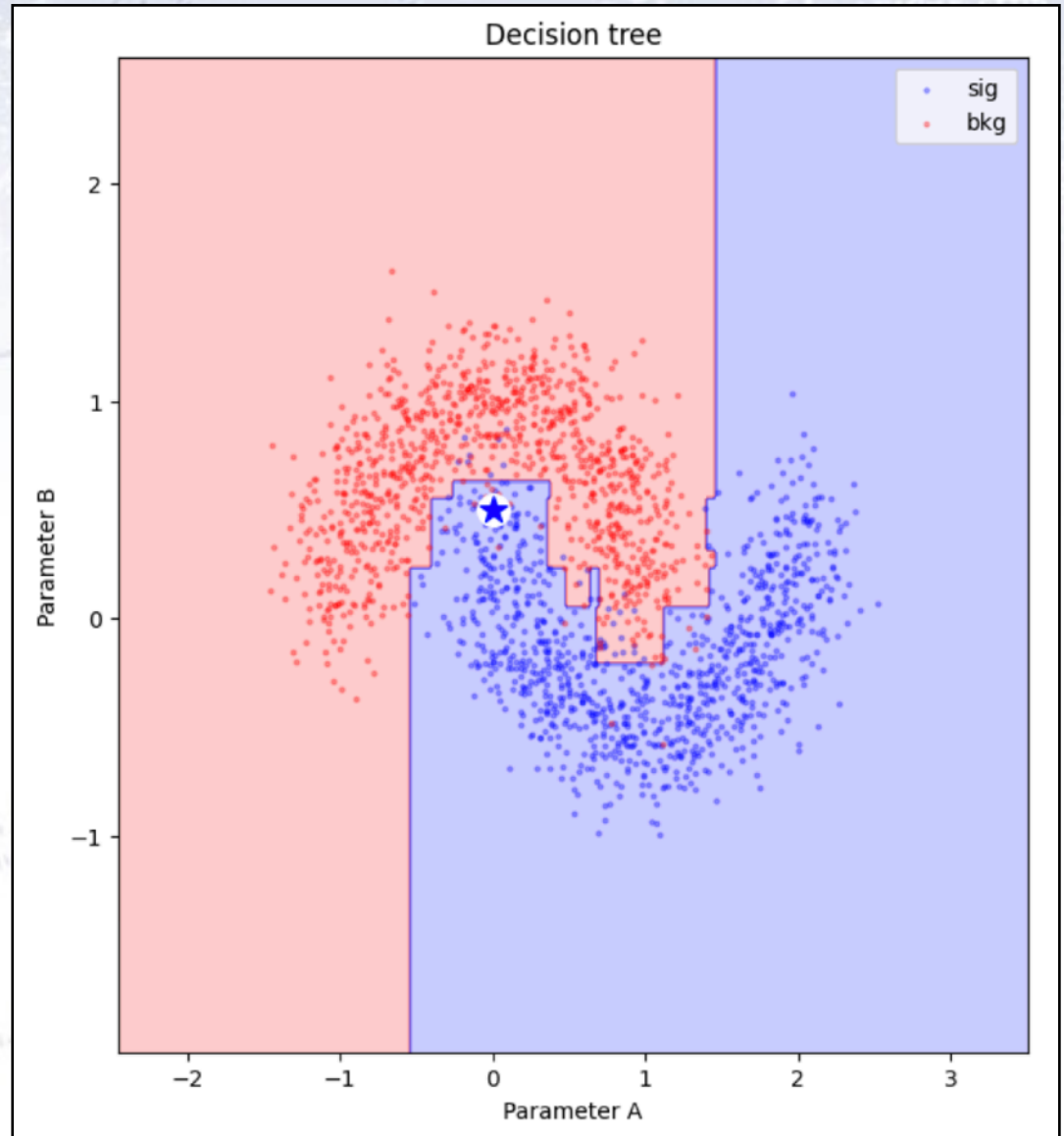
Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approx. Theorems

A simple function can be obtained simply by asking a lot of questions:

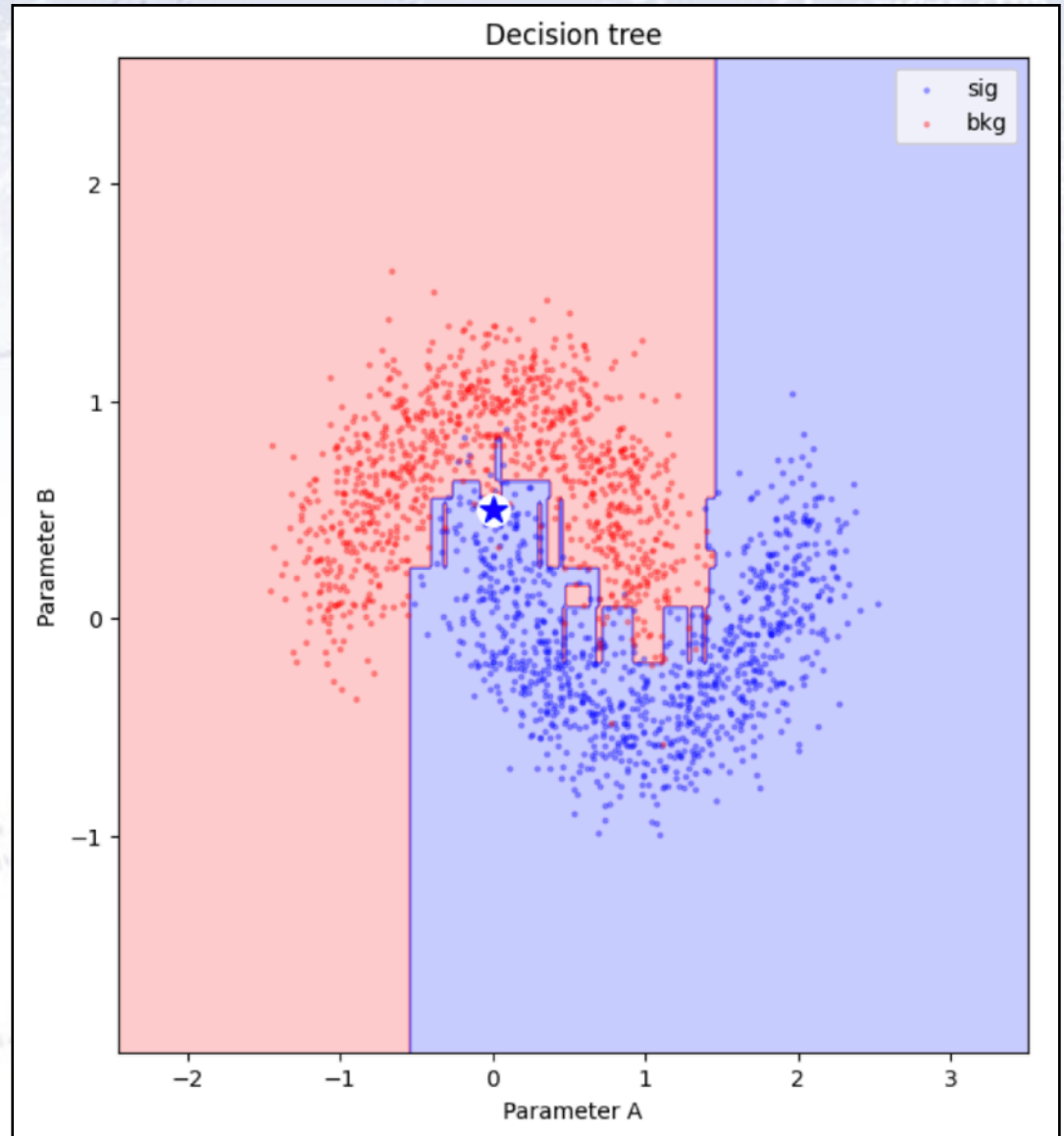
Question: Is $B > 0.23$?

Answer: Yes \rightarrow Red

Answer: No \rightarrow Blue

This question is illustrated in the drawing by the horizontal line with red and blue on the sides.

A Decision Tree consists of asking many such questions, corresponding to setting a lot of lines.



Universal Approximation Theorems

Theorem 5.1.1 (Universal Approximation Theorem) ¹⁰ Let σ be a non-constant, bounded, and monotone-increasing continuous function. Let I_{m_0} denote the m_0 -dimensional unit hypercube $[0, 1]^{m_0}$. The space of continuous functions on I_{m_0} is denoted as $C(I_{m_0})$. Then given any function $f \in C(I_{m_0})$ and $\epsilon > 0$ there exists a set of real constants a_i, b_i and w_{ij} , where $i = 1, \dots, m_1$ and $j = 1, \dots, m_0$ such that we may define

$$F(x_1, \dots, x_{m_0}) = \sum_{i=1}^{m_1} a_i \sigma \left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i \right) \quad (5.6)$$

as an approximate realization of the function f ; that is,

$$|F(x_1, \dots, x_{m_0}) - f(x_1, \dots, x_{m_0})| < \epsilon \quad (5.7)$$

for all x_1, x_2, \dots, x_{m_0} that lie in the input space.

Universal Approximation Theorems

Theorem 5.1.1 (Universal Approximation Theorem) ¹⁰ Let σ be a non-

Summary:

Neural Networks etc. can approximate functions in any dimension very well!

$$F(x_1, \dots, x_{m_0}) = \sum_{i=1} a_i \sigma \left(\sum_{j=1} w_{ij} x_j + b_i \right) \quad (5.6)$$

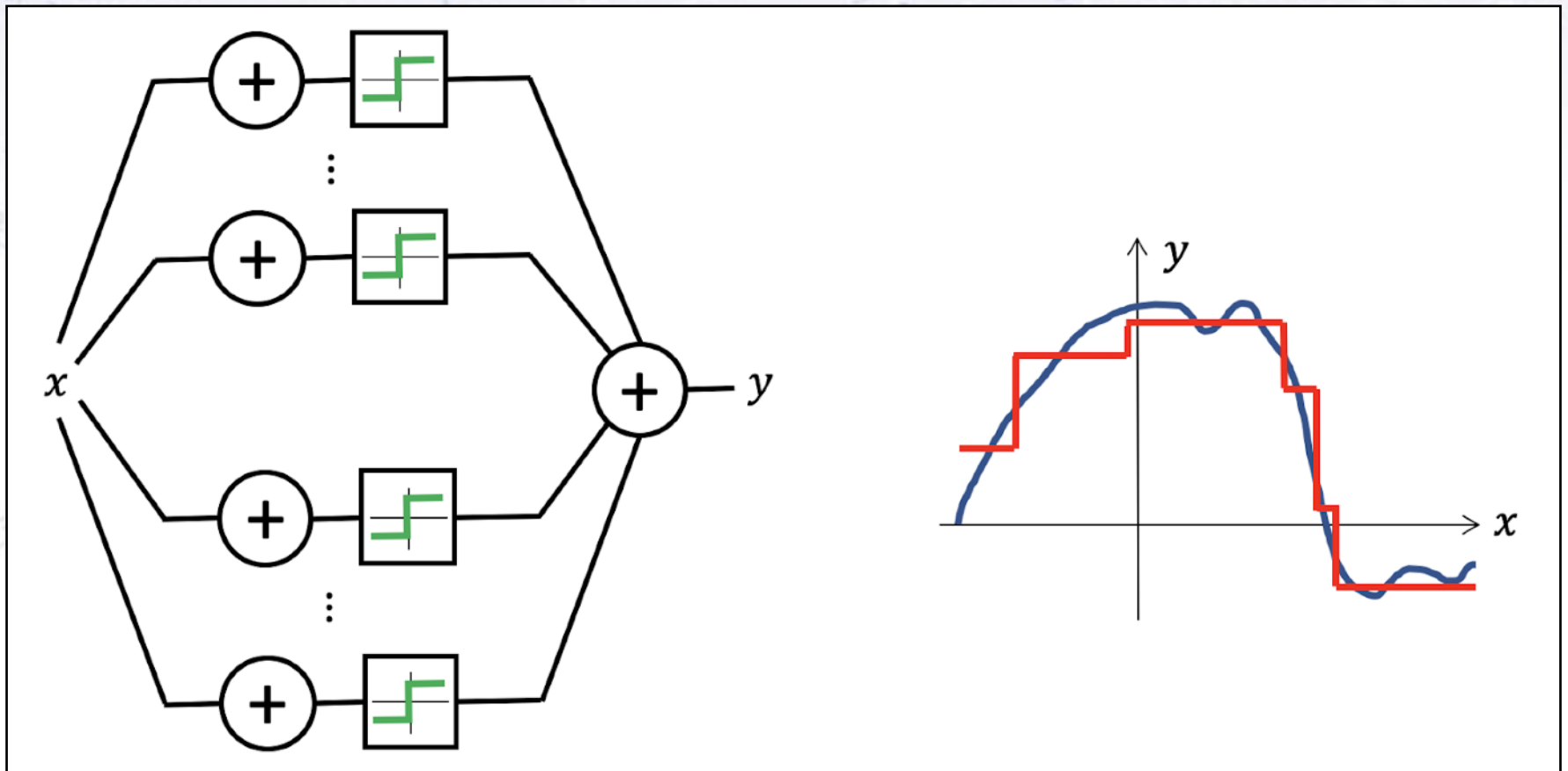
as an approximate realization of the function f ; that is,

$$|F(x_1, \dots, x_{m_0}) - f(x_1, \dots, x_{m_0})| < \epsilon \quad (5.7)$$

for all x_1, x_2, \dots, x_{m_0} that lie in the input space.

Universal Approx. Theorems

Such approximations typically entails a **large amount of parameters**, for which the UATs give no recipe on how to find - only that such a construction is possible.



Universal Approx. Theorems

One main ingredient behind ML are **Universal Approximation Theorems (UAT)**.

These imply that Neural Networks can approximate a very wide variety of functions given simple function constraints and enough degrees of freedom.

This typically entails a **large amount of weights**, for which the UATs give **no recipe on how to find** - only that such a construction is possible.

Even if one assumes that there is no noise in the training set, then there will still be **infinitely many functions that passes through all training points** and not all of them will have the same error on an unseen point (i.e. the test set).

So how to find actual solutions that are behaving nicely?

The background of the slide is a faded, light blue map of a coastal area, likely the Bitter End Yacht Club. Overlaid on the map is a large, semi-transparent circular magnetic compass. The compass has degree markings around its perimeter and a central needle pointing towards the top. The word "MAGNETIC" is visible on the compass face. The map shows various geographical features, including a coastline and some text labels like "BITTER END YACHT CLUB".

How to find these

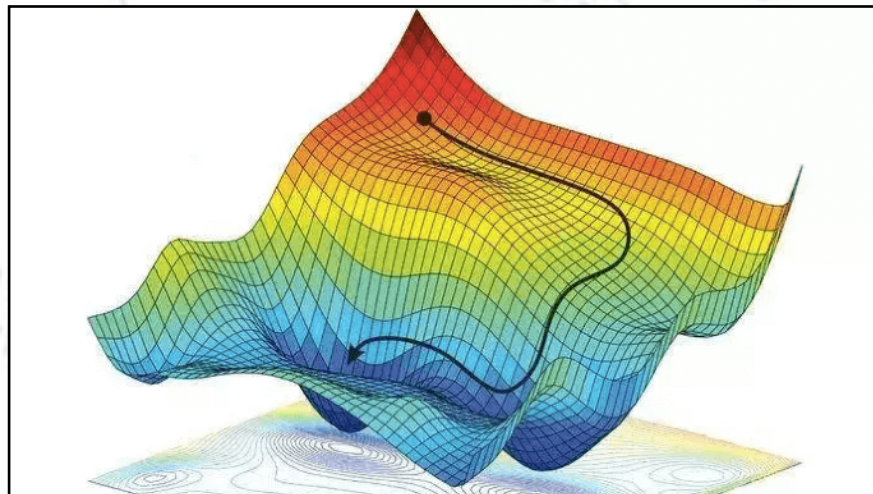
(Technically called Stochastic Gradient Descent)

Stochastic Gradient Descent

The way to obtain the parameters / weights of ML algorithms, is generally by **Stochastic Gradient Descent**.

This “back propagation” algorithm works by computing the gradient of the loss function (to be optimised) with respect to each weight using the chain rule.

One thus computes the gradient one layer at a time, iterating backwards from the last layer (avoiding redundancies). See Goodfellow et al. for details.



(Normal) Gradient Descent

The choice of loss function, L , depends on the problem at hand, and in particular what you find important! You want to minimise this with respect to the model parameters θ :

$$L(\theta) = \frac{1}{N} \sum_i^N L_i(\theta)$$

In order to find the optimal solution, one can use Gradient Descent, typically **based on the whole dataset**:

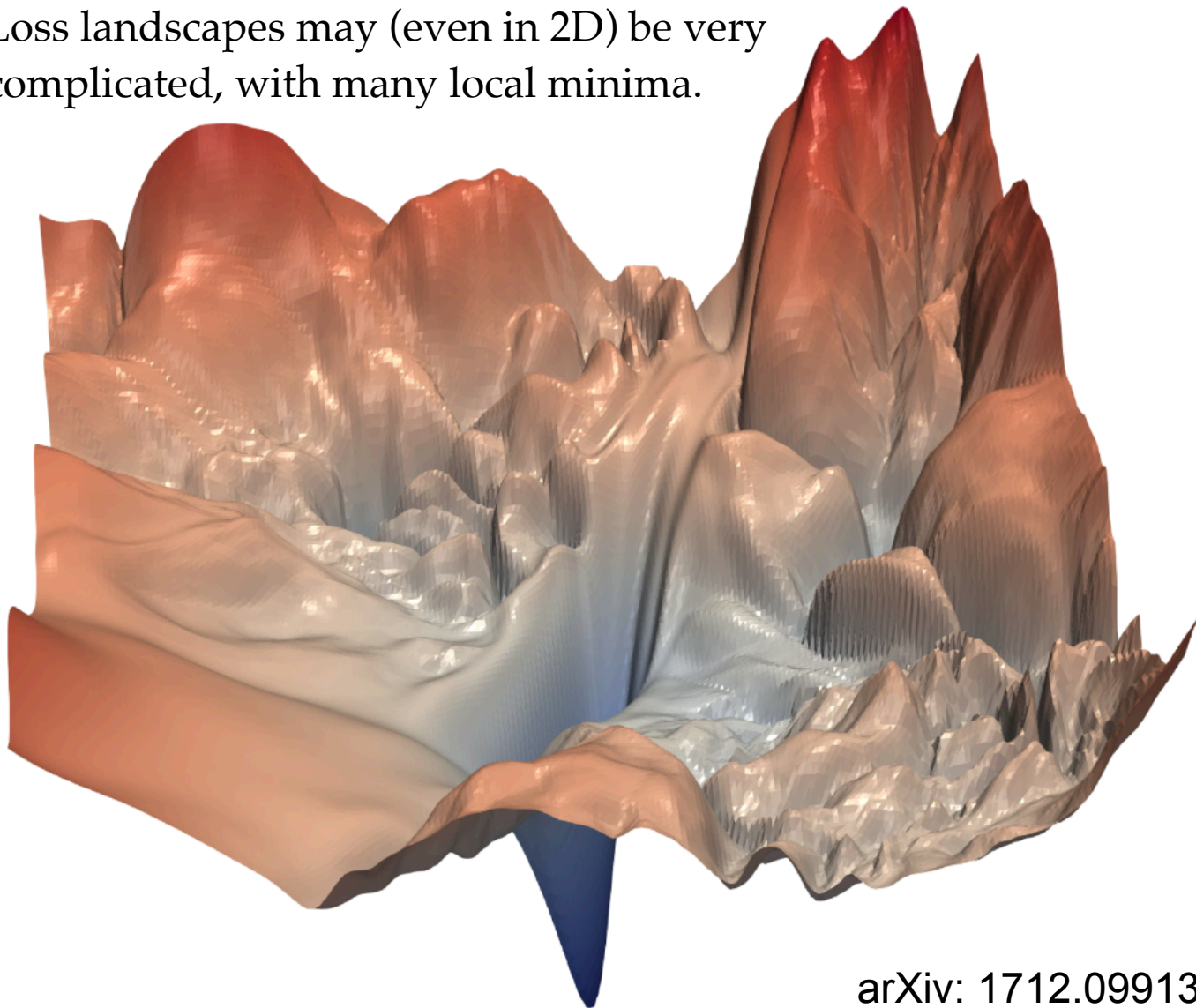
$$\theta_{j+1} = \theta_j - \eta \nabla L(\theta) = \theta_j - \frac{\eta}{N} \sum_i^N \nabla L_i(\theta)$$

This is the procedure used by e.g. Minuit and other minimisation routines.

Note the very important parameter: **Learning rate η** .

(Nasty) Loss Landscapes

Loss landscapes may (even in 2D) be very complicated, with many local minima.



arXiv: 1712.09913

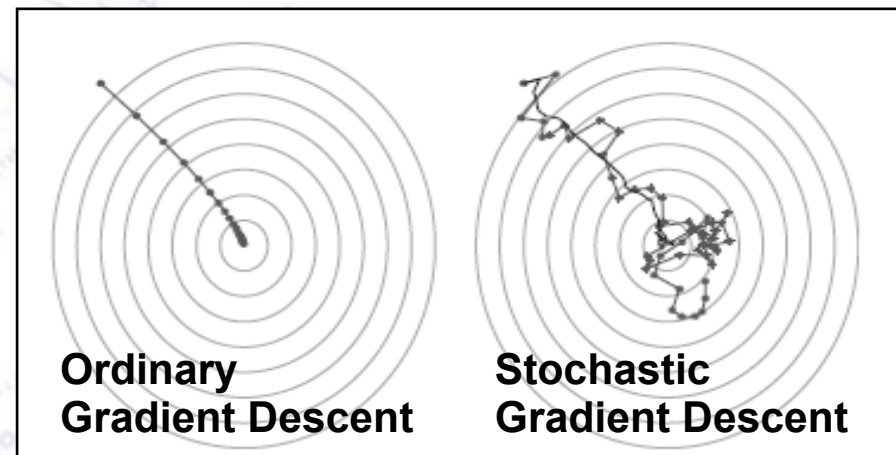
Stochastic Gradient Descent

The way to obtain the parameters / weights of ML algorithms, is generally by **Stochastic Gradient Descent**.

This “back propagation” algorithm works by computing the gradient of the loss function (to be optimised) with respect to each weight using the chain rule.

One thus computes the gradient one layer at a time, iterating backwards from the last layer (avoiding redundancies). See Goodfellow et al. for details.

The gradient descent is made stochastic (and fast) by only considering a fraction (called a “batch”) of the data, when calculating the step in the search for optimal parameters for the algorithm. This allow for stochastic jumping, that avoids local (false) minima.



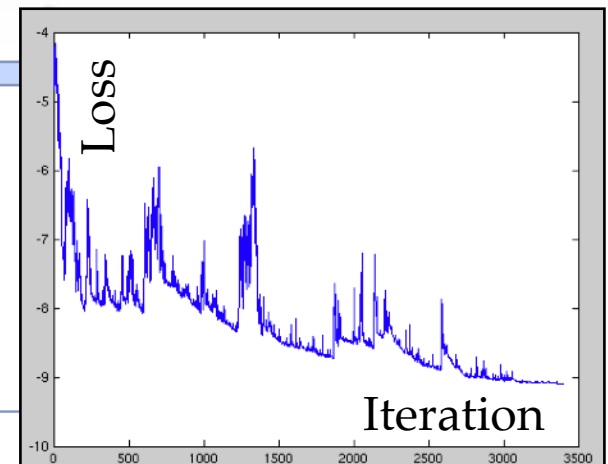
Stochastic Gradient Descent

In order to give the gradient descent some degree of “randomness” (stochastic), one evaluates the below function **for small batches** instead of the full dataset.

$$\theta_{j+1} = \theta_j - \eta \nabla L(\theta) = \theta_j - \frac{\eta}{N} \sum_i^N \nabla L_i(\theta)$$

The algorithm thus becomes:

- Choose an initial vector of parameters w and learning rate η .
- Repeat until an approximate minimum is obtained:
 - Randomly shuffle examples in the training set.
 - For $i = 1, 2, \dots, n$, do:
 - $w := w - \eta \nabla Q_i(w)$.



Not only does this vectorise well and gives smoother descents, but with decreasing learning rate, it **“almost surely” finds the global minimum** (Robbins-Siegmund theorem).

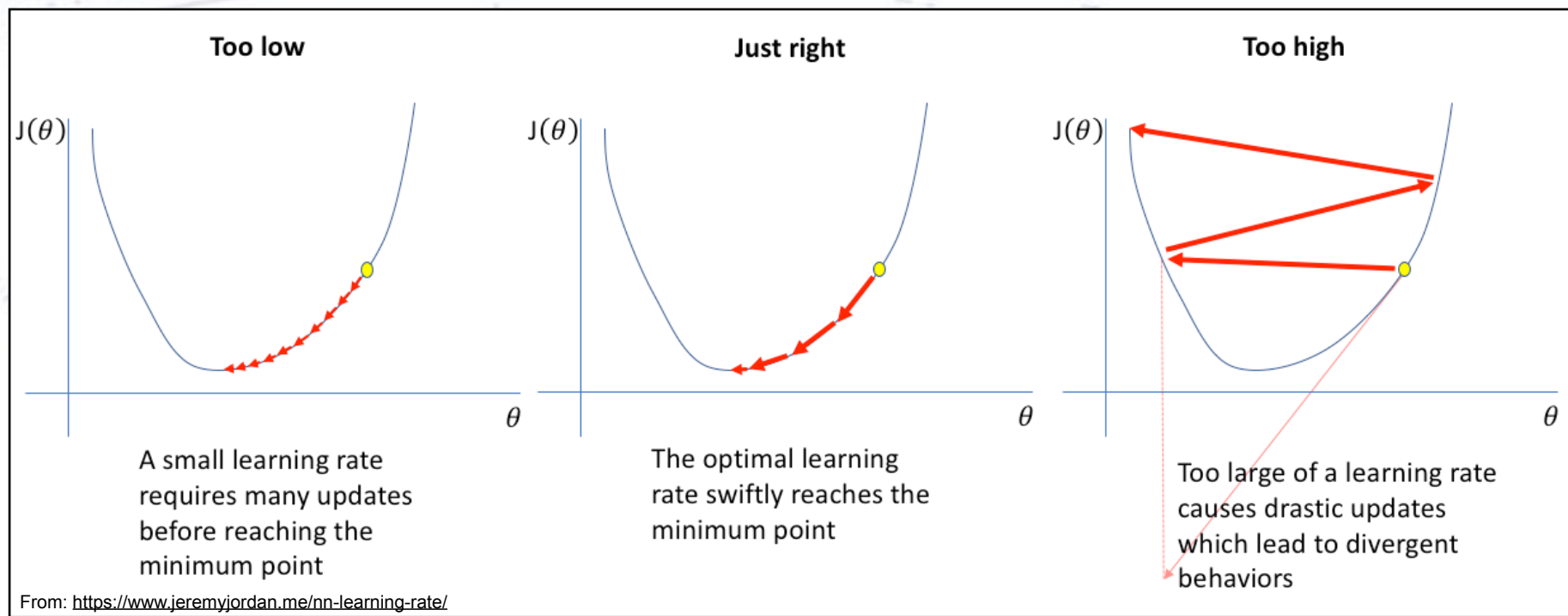
Learning Rate Schedulers

But, there is **no reason to consider a fixed value for the learning rate!**

More practically, one would typically adapt the learning rate to the situation:

- When exploring: Use larger learning rate.
- When exploiting: Use lower learning rate (when converging).

Below is illustrated what happens, when the learning rate is right/wrong.



Choosing Learning Rate

Too low learning rate: Convergence very (too) slow.

Too high learning rate: Random jumps and no convergence.

You want to increase it until it fails and then just below...

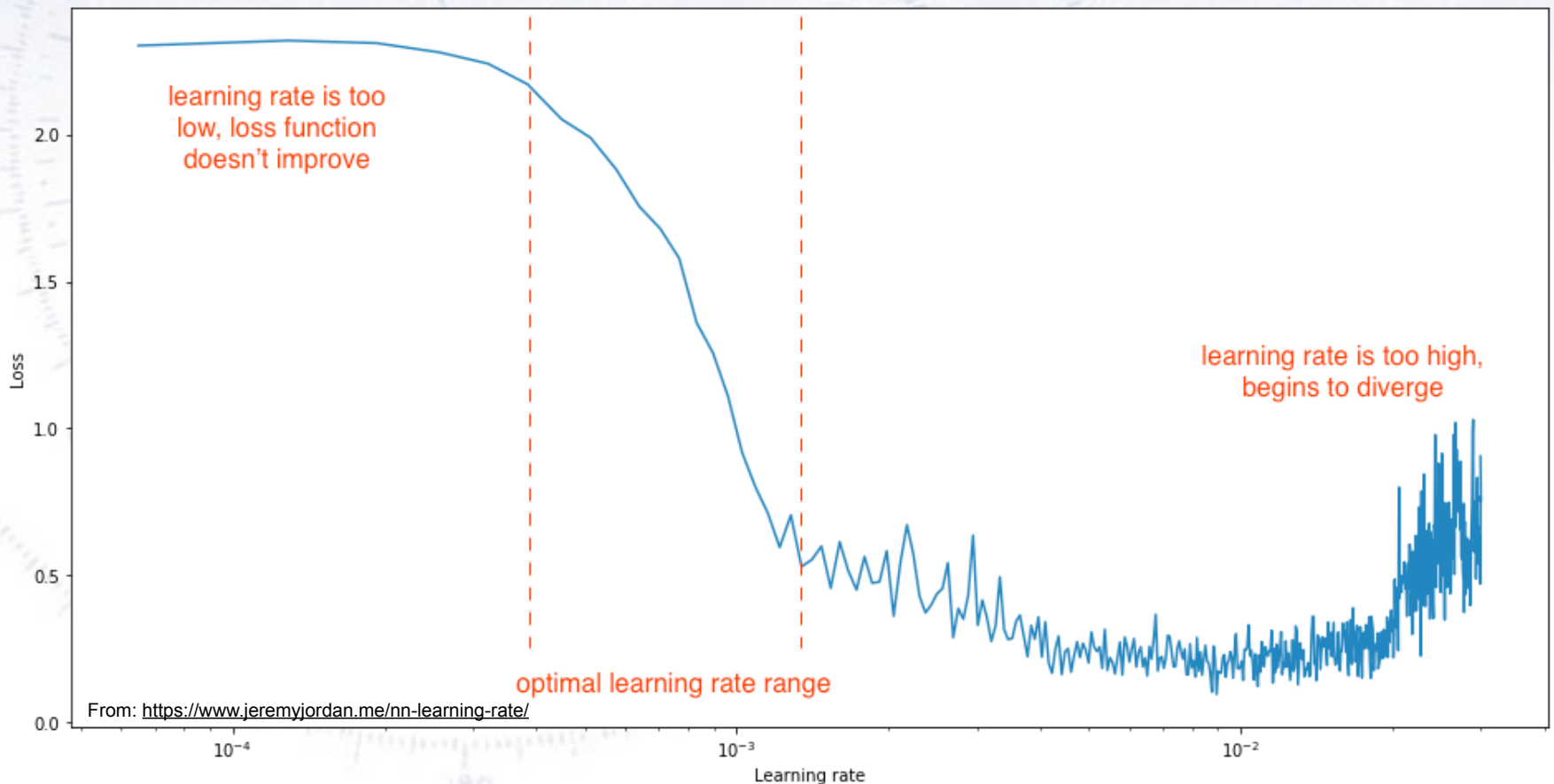


*Ristet brød er let at lave
blot man vil erindre:
når det oser, skal det have
to minutter mindre.*

Piet Hein

Learning Rate Schedulers

First, we want to investigate, which learning rates are relevant (and best) for our problem. **The best learning rate is when the loss decreases the fastest.** Thus we look for the greatest slope of the loss as a function of learning rate:

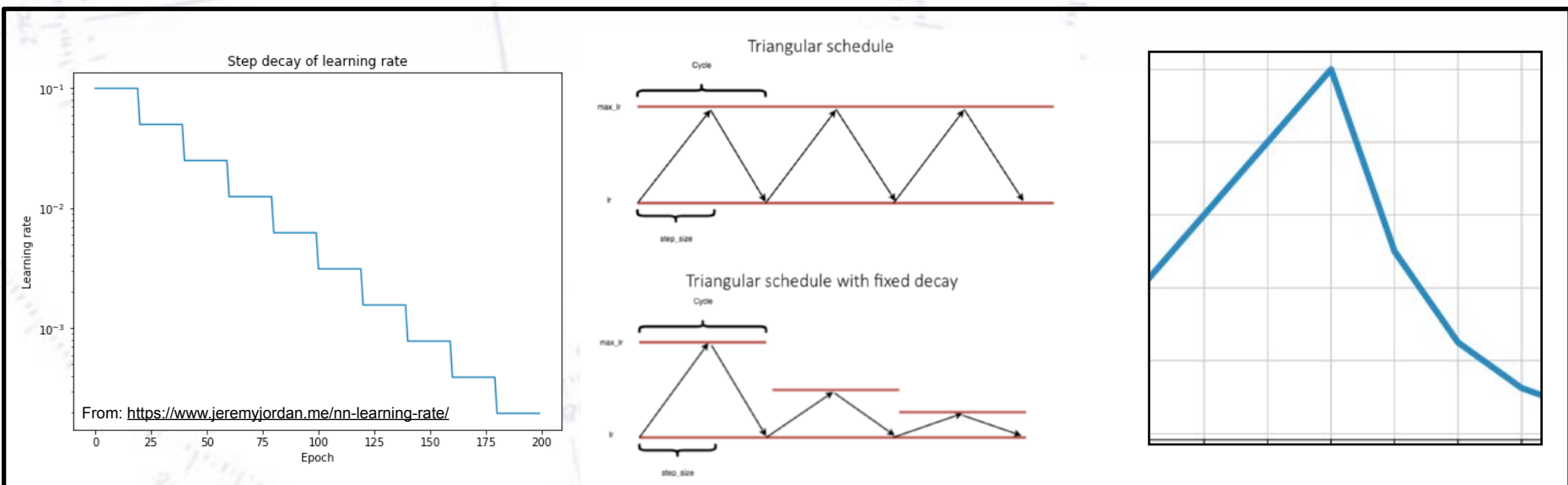


Learning Rate Schedulers

For this reason, Learning Rate Schedulers have been “invented”.

There are MANY different types, and as usual, there is no “right answer”.

However, it is fair to say, that the learning rate is (especially for NNs) the most important Hyper Parameter, and thus it **requires attention**.



Ingredients for ML

So now we know that at least in principle:

- a solution exists (Universal Approximation Theorem) and
- that it can be found (Stochastic Gradient Descent).

But this does not in reality make us capable of getting ML results.

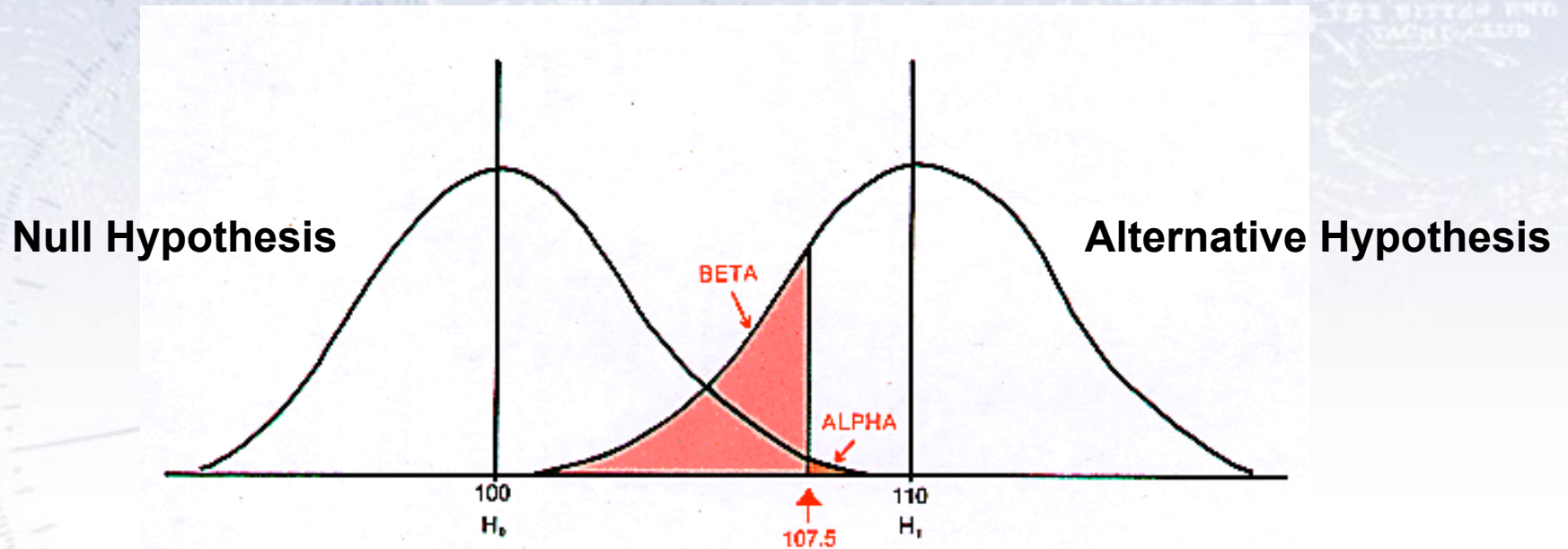
We (at least) also need:

- actual functions/ algorithms for making approximations
Boosted Decision Trees (BDTs) & Neural Networks (NNs)
- knowledge about how to tell them what to learn
Loss functions (and how to minimise these)
- a scheme for how to use the data we have available
Training, validation, and testing samples & Cross Validation



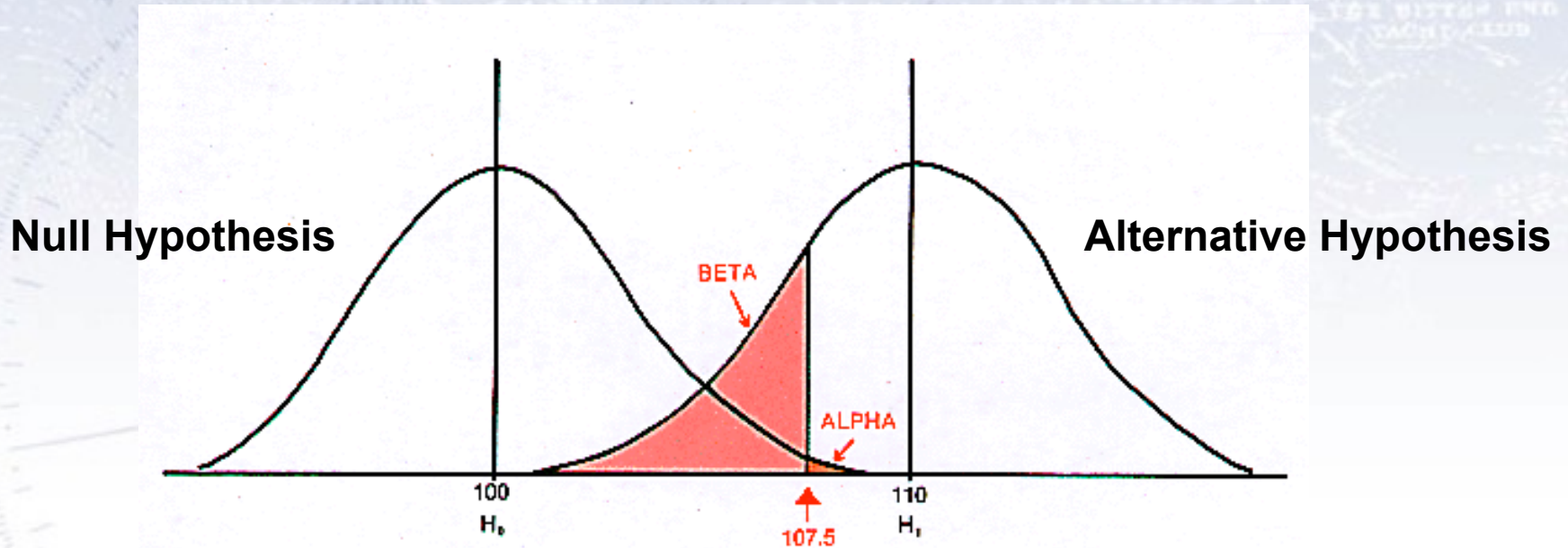
Target of ML

Classification



		REALITY	
		Null is True	Null is False
STATISTICAL DECISION:	Do Not Reject Null	$1 - \alpha$ Correct	β Type II error
	Reject Null	α Type I error	$1 - \beta$ Correct

Classification



Machine Learning typically enables a better separation between hypothesis

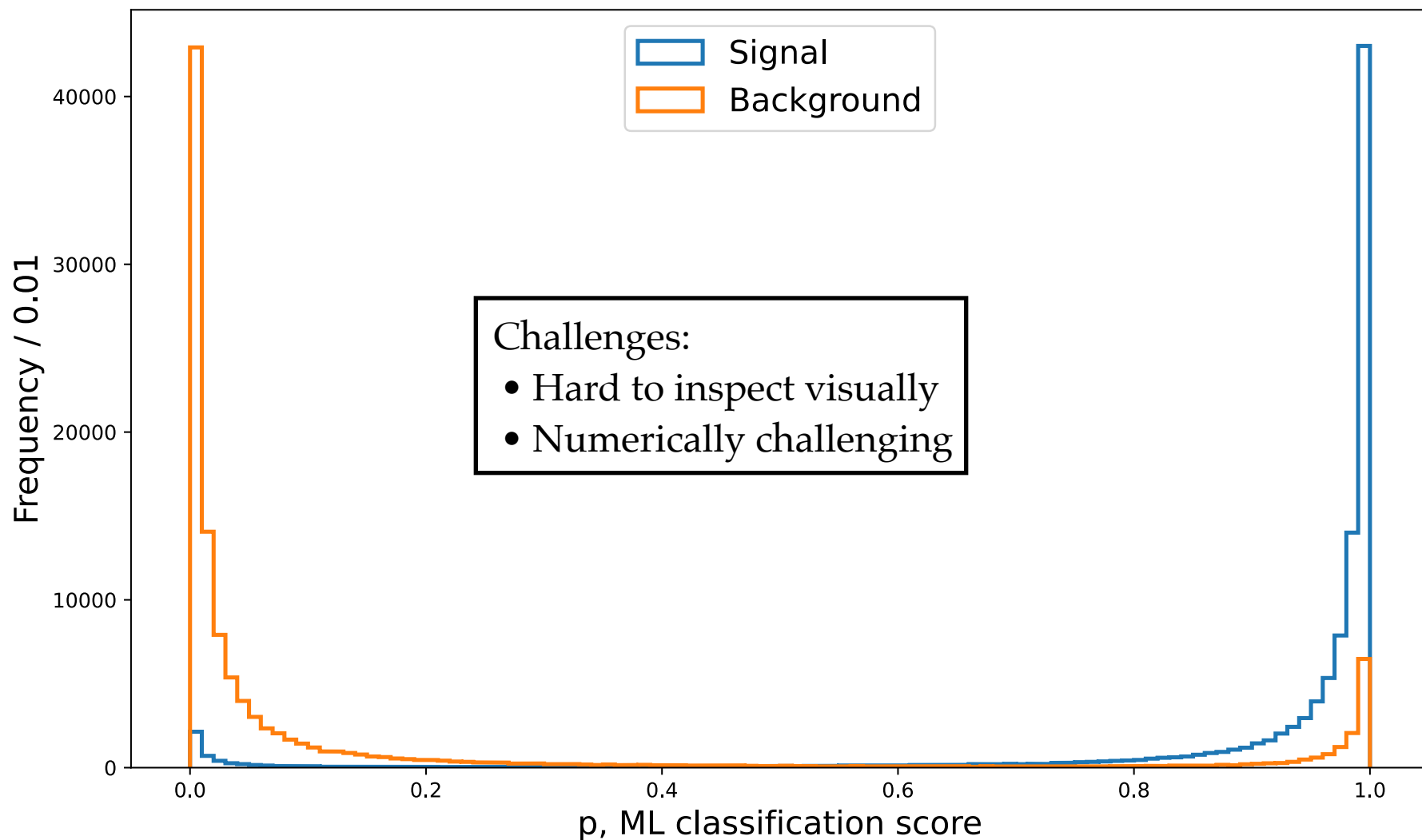
DECISION:

Reject Null

α Type I error	$1 - \beta$ Correct
--------------------------	------------------------

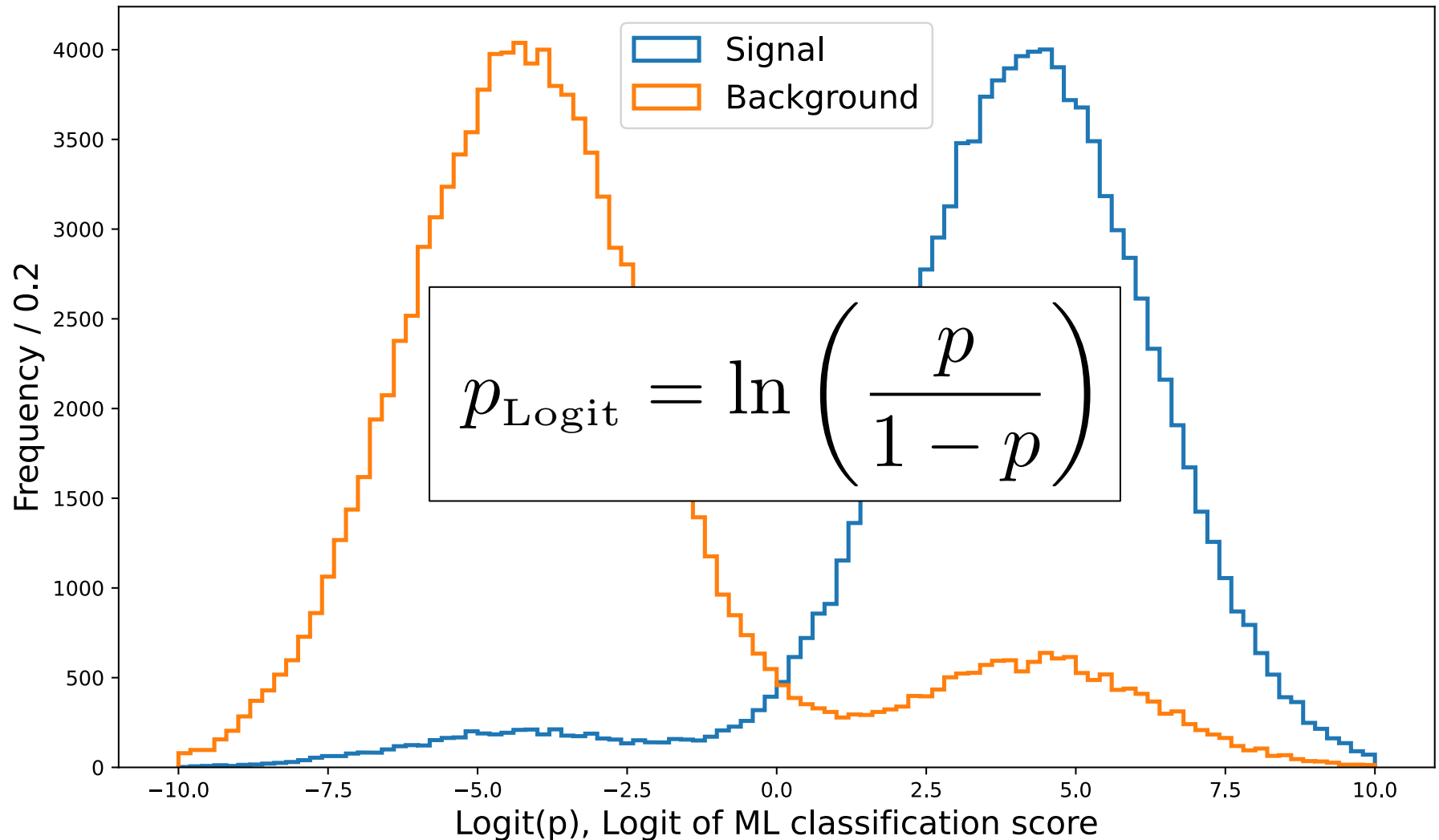
Typical ML Distribution

An ML score distribution from binary classification typically looks as follows:



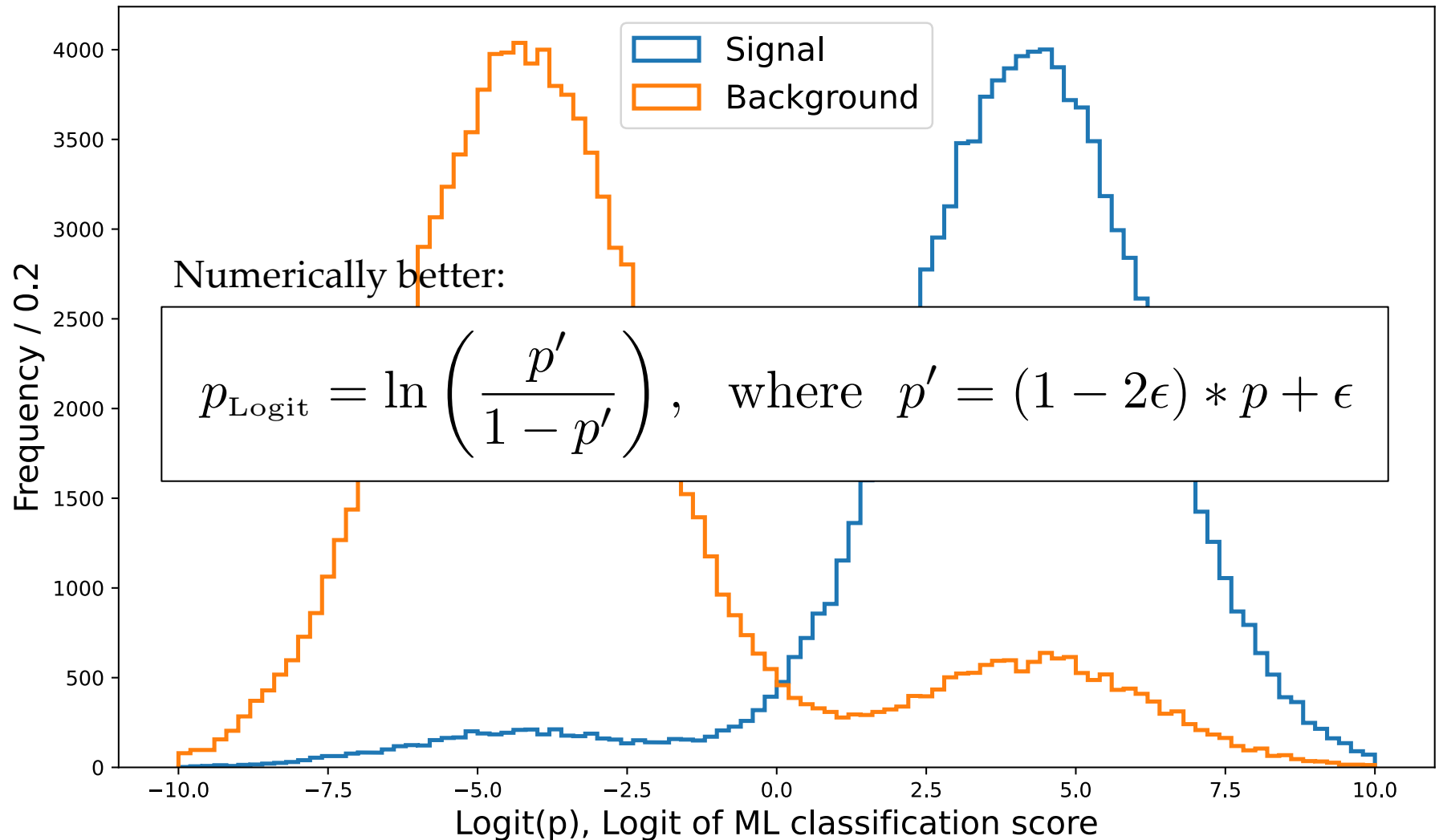
Logit transformation

Once logit transformed, it takes on a “nicer” (and numerically friendly) shape:



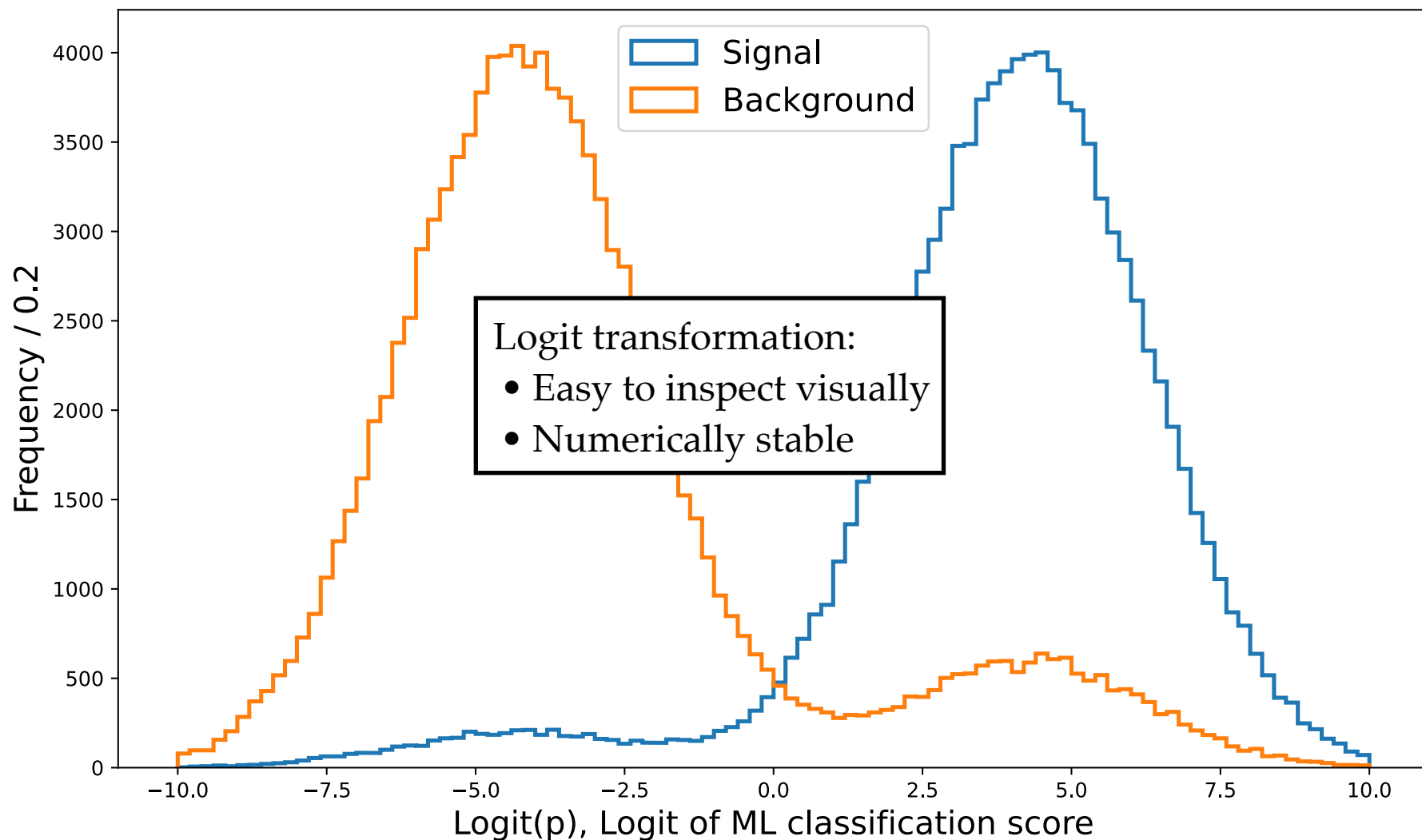
Logit transformation

Once logit transformed, it takes on a “nicer” (and numerically friendly) shape:



Logit transformation

Once logit transformed, it takes on a “nicer” (and numerically friendly) shape:



Hypothesis testing

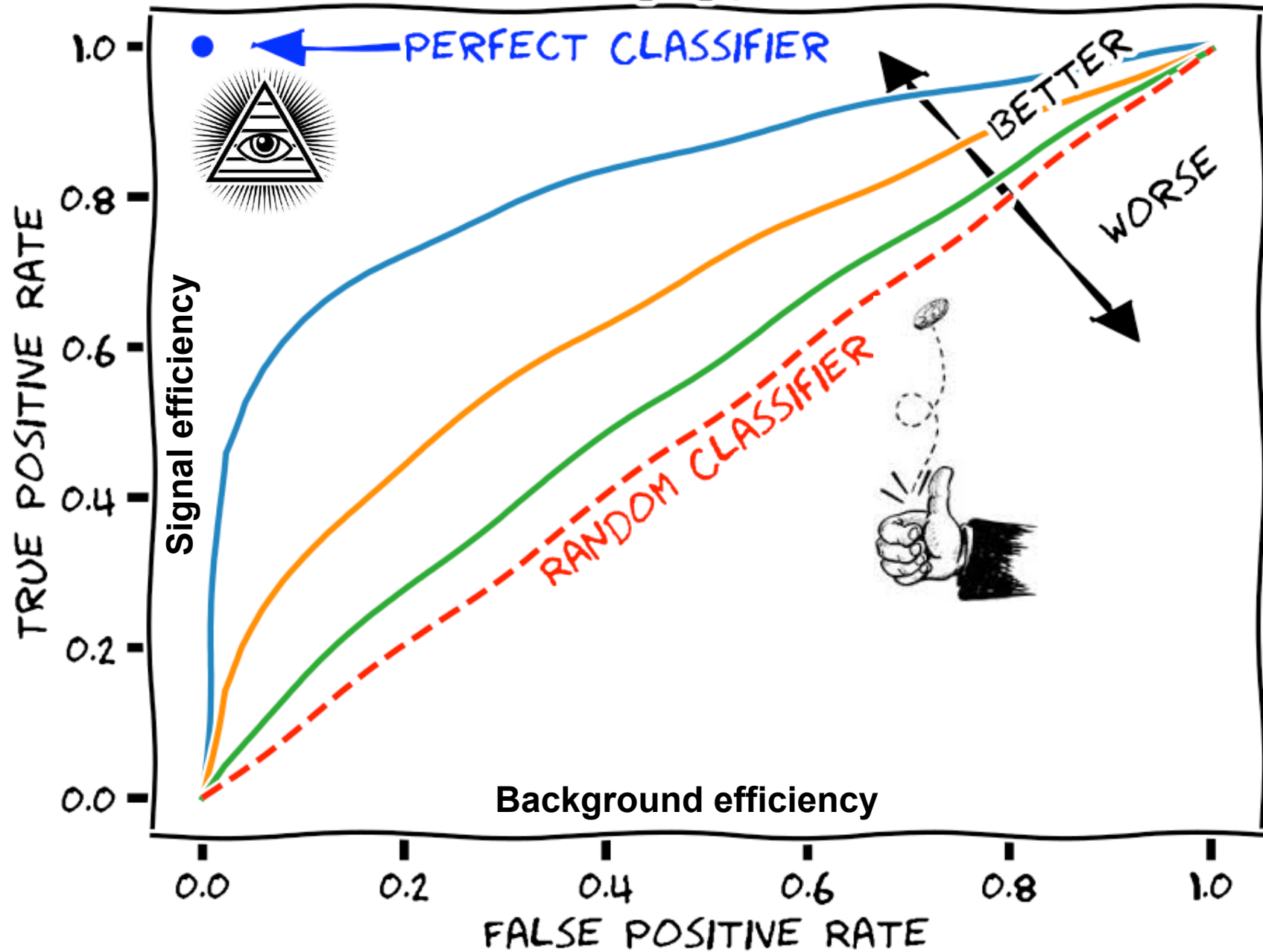
Hypothesis testing is like a criminal trial. The basic “null” hypothesis is **Innocent** (called H_0) and this is the hypothesis we want to test, compared to an “alternative” hypothesis, **Guilty** (called H_1).

Innocence is initially assumed, and this hypothesis is only rejected, if enough evidence proves otherwise, i.e. that the probability of innocence is very small (“beyond reasonable doubt”). This is summarised in a **Contingency Table**:

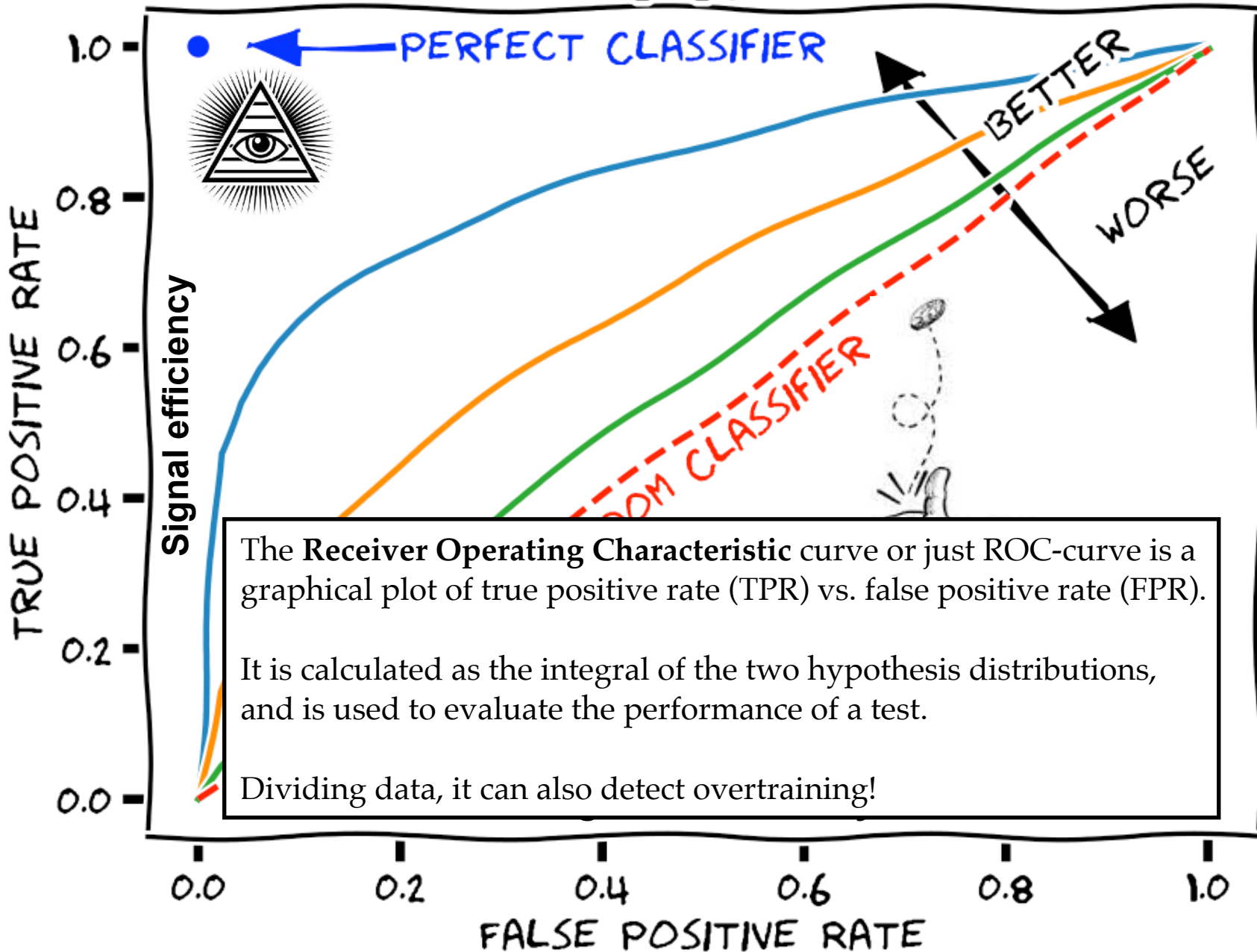
	Truly innocent (H_0 is true)	Truly guilty (H_1 is true)
Acquittal (Accept H_0)	Right decision True Positive (TP)	Wrong decision False Negative (FN)
Conviction (Reject H_0)	Wrong decision False Positive (FP)	Right decision True Negative (TN)

The rate of FP and FN are correlated, and one can only choose one of these!

ROC CURVE



ROC CURVE



Which metric to use?

There are a ton of metrics in hypothesis testing, see below. However, those in the boxes below are the most central ones.

One metric - not mentioned here - is the Area Under the Curve (AUC), which is simply an integral of the ROC curve (thus 1 is perfect score). This is sometimes used to optimise performance (loss), but not great!

		True condition			
Total population		Condition positive	Condition negative	Prevalence = $\frac{\Sigma \text{ Condition positive}}{\Sigma \text{ Total population}}$	Accuracy (ACC) = $\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$
Predicted condition	Predicted condition positive	True positive	False positive, Type I error	Positive predictive value (PPV), Precision = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Predicted condition positive}}$
	Predicted condition negative	False negative, Type II error	True negative	False omission rate (FOR) = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Predicted condition negative}}$
		True positive rate (TPR), Recall, Sensitivity, probability of detection, Power $= \frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm $= \frac{\Sigma \text{ False positive}}{\Sigma \text{ Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR+}}{\text{LR-}}$
		False negative rate (FNR), Miss rate $= \frac{\Sigma \text{ False negative}}{\Sigma \text{ Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) $= \frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$	Negative likelihood ratio (LR-) = $\frac{\text{FNR}}{\text{TNR}}$	
				F ₁ score = $2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$	

Matthew's Correlation Coefficient

Given a Contingency Table:

	Got well	Remained ill
Medicin	28	5
No Medicin	19	9

One of the commonly used measures of separation the MCC, which (in this case) is the Pearson ϕ , and related to the ChiSquare:

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Read more at:

https://en.wikipedia.org/wiki/Phi_coefficient

However, when optimising an algorithm and giving continuous scores in the range $]0,1[$, there are other things to consider (see talk on Loss Functions).



Tree based models

Decision tree learning

“Tree learning comes closest to meeting the requirements for serving as an off-the-shelf procedure for data mining”,

because it:

- is invariant under scaling and various other transformations of feature values,
- is robust to discontinuous, categorical, and irrelevant features,
- produces inspectable models.

HOWEVER... they are seldom accurate (i.e. most performant)!

[**Trevor Hastie**, Prof. of Mathematics & Statistics, Stanford]

For tabular data, I tend to disagree with the two last statements!

Decision Trees

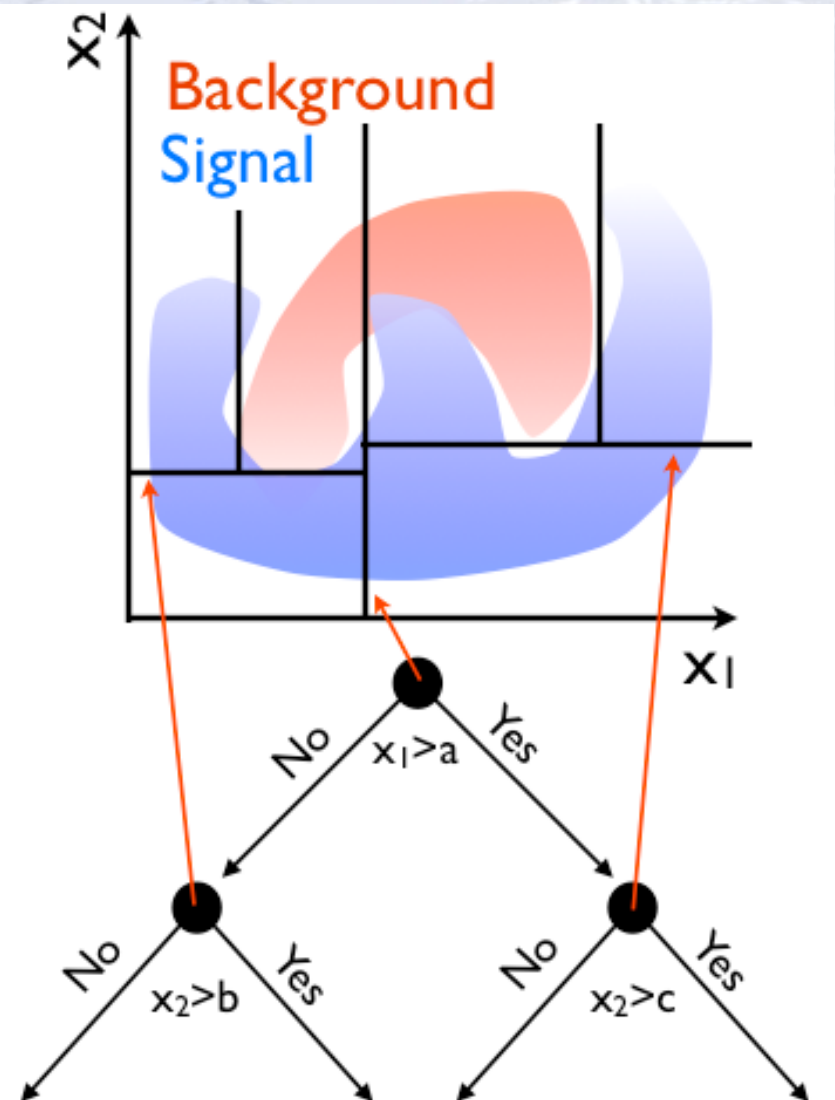
A decision tree divides the parameter space, starting with the maximal separation. In the end each part has a probability of being signal or background.

- Works in 95+% of all problems!
- Fully uses non-linear correlations.

But BDTs require a lot of data for training, and is sensitive to overtraining.

Overtraining can be reduced by limiting the number of nodes and number of trees.

Decision trees are from before 1980!!!



Boosting...

There is no reason, why you can not have more trees. Each tree is a simple classifier, but many can be combined!

To avoid N identical trees, one assigns a higher weight to events that are hard to classify, i.e. boosting:

First classifier

$$y_{\text{Boost}}(\mathbf{x}) = \frac{1}{N_{\text{collection}}} \cdot \sum_i^{N_{\text{collection}}} \ln(\alpha_i) \cdot h_i(\mathbf{x})$$

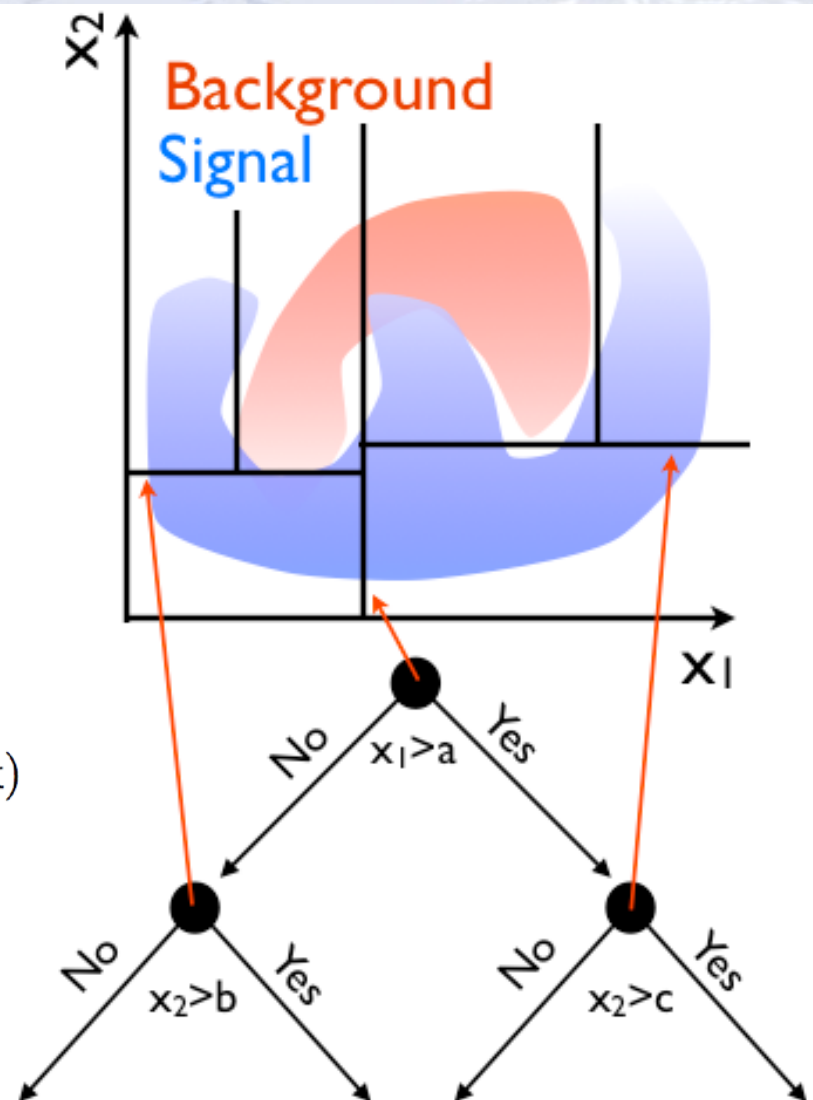
Parameters in event N

Boost weight

$$\alpha = \frac{1 - \text{err}}{\text{err}}$$

Individual tree

Boosting is from 1997 (AdaBoost).



Boosting...

There is no reason, why you can not have more trees. Each tree is a simple classifier, but many

To avoid N identical trees, assign a higher weight to misclassified entries, i.e. boost

Rerun...
increasing the weight of misclassified entries

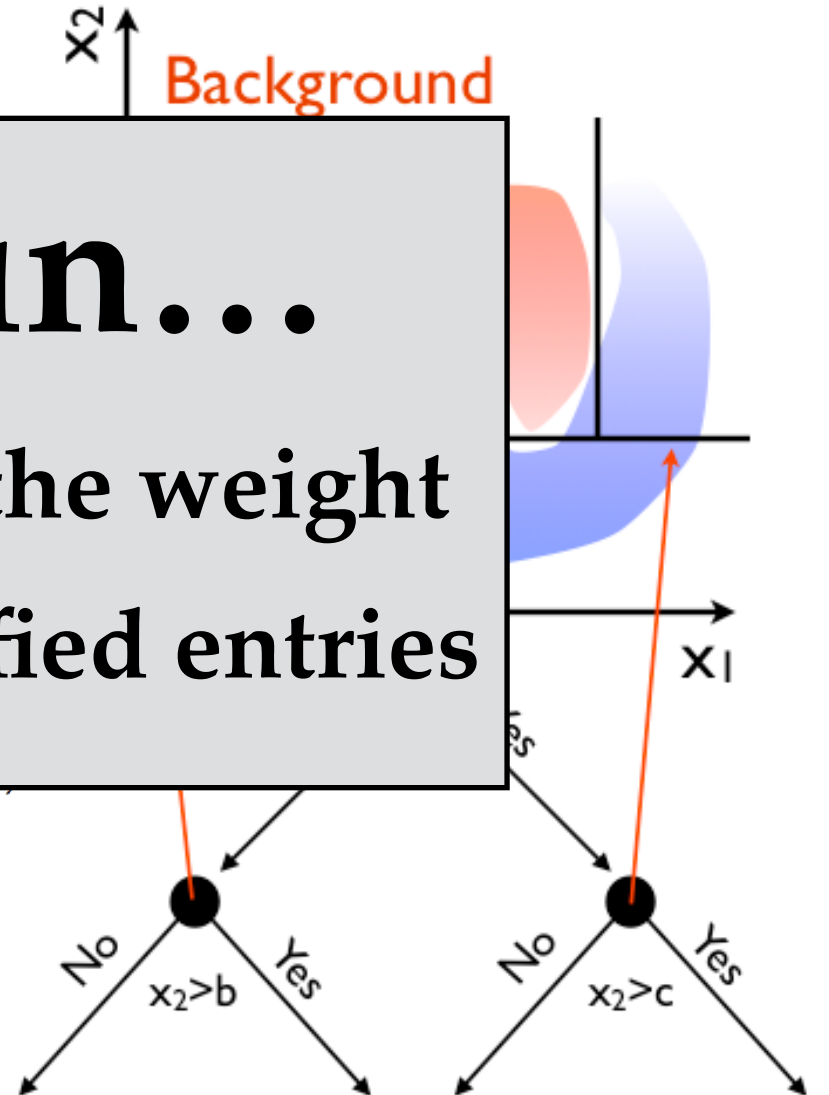
First classifier

$$y_{\text{Boost}}(\mathbf{x}) = \sum_{i=1}^N \frac{w_i}{N_{\text{collection}}} y_i(\mathbf{x})$$

Parameters in event N

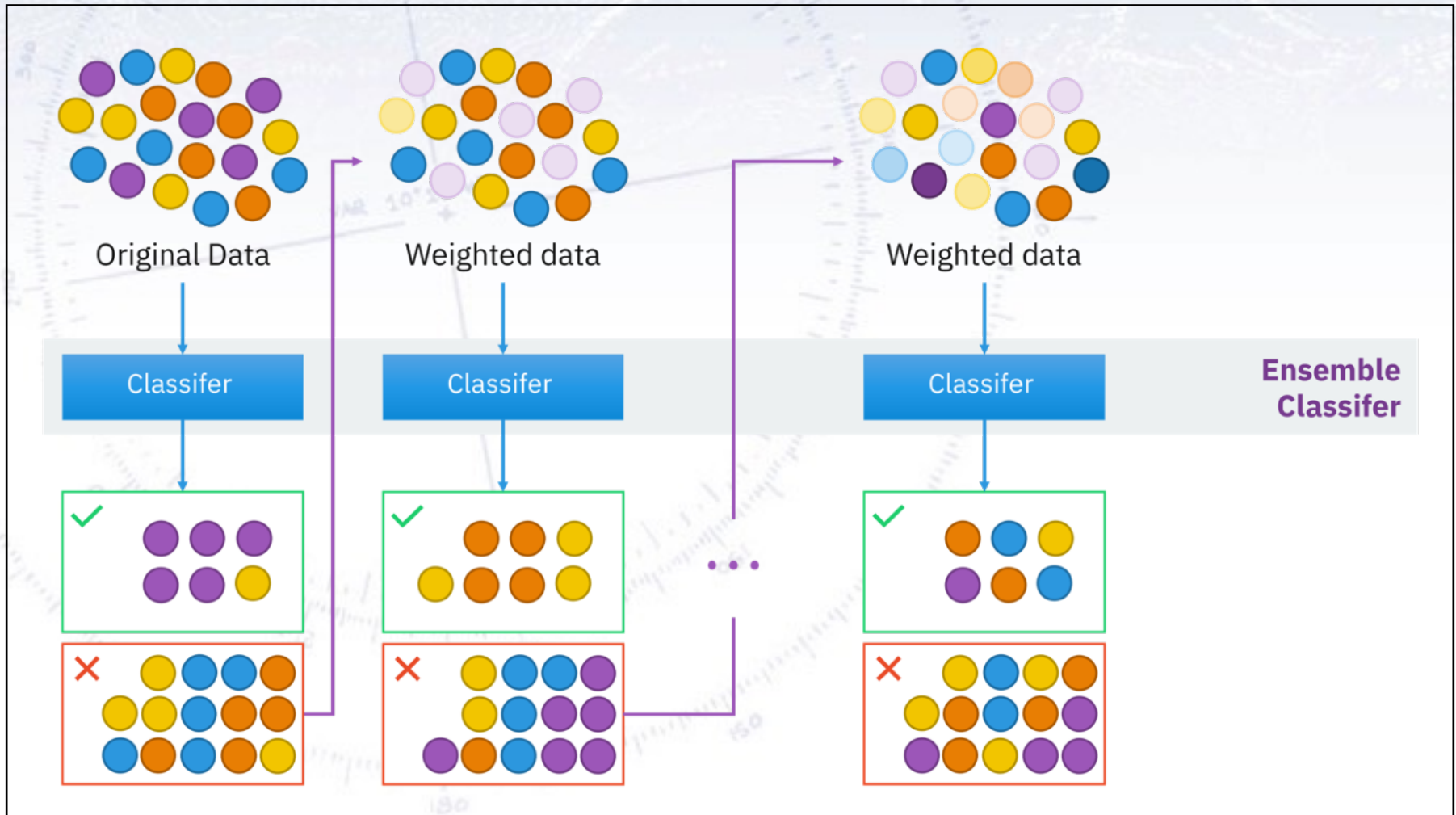
Individual tree

Boosting is from 1997 (AdaBoost).



Boosting illustrated

Boosting provides a reweighing scheme giving harder cases higher weights. At the end of training, the trees are collected into an “ensemble classifier”.



Where to split?

How does the algorithm decide which variable to split on and where to split?

There are several ways in which this can be done, and there is a difference between how to do it for classification and regression. But in general, one would like to **make the split, which maximises the improvement gained by doing so.**

In **classification**, one often uses the average binary cross entropy (aka. “log-loss”):

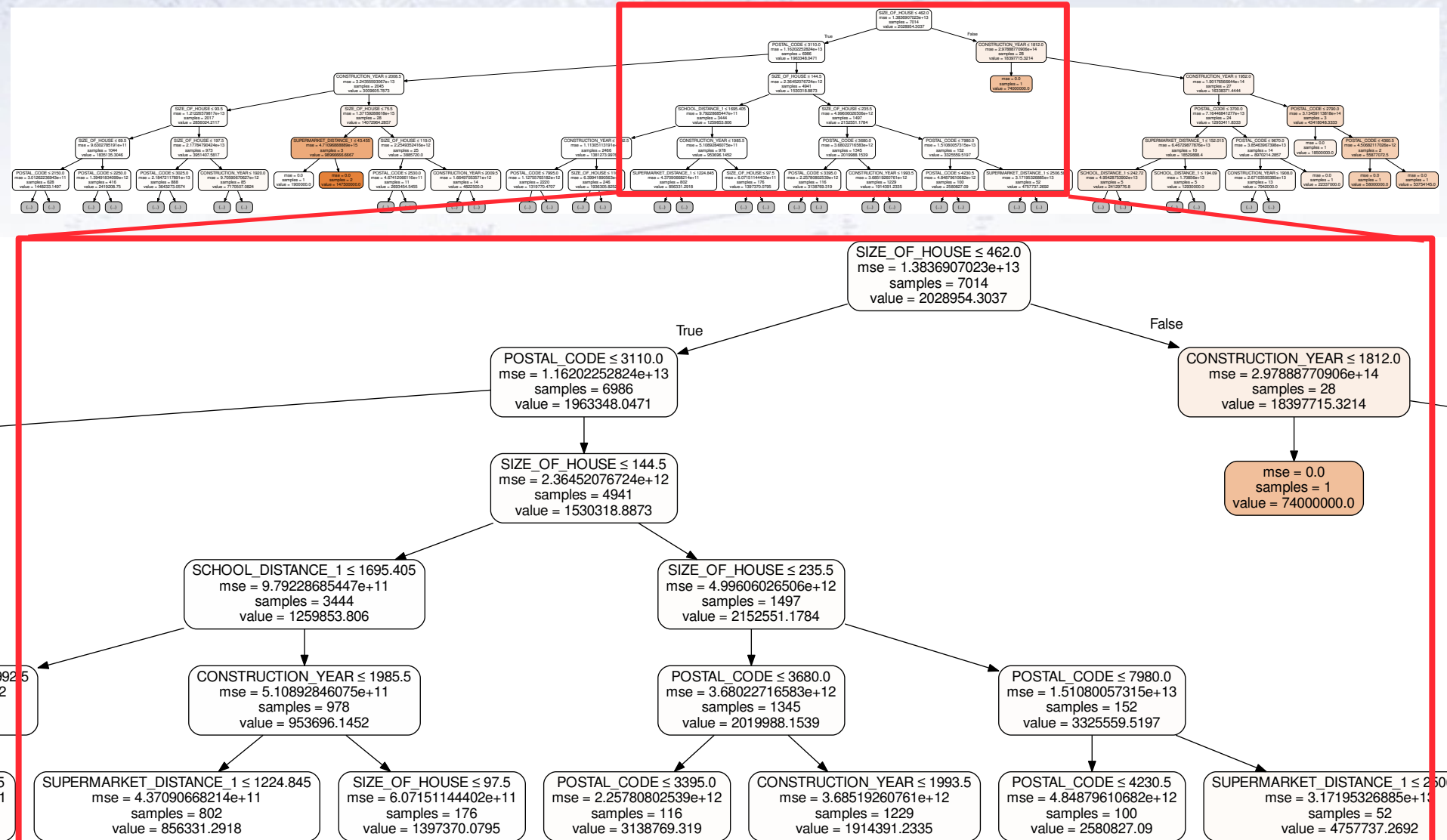
$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^N [y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)]$$

Here, y_n is the truth, while \hat{y}_n is the estimate (in $[0,1]$).

Other alternatives include using Gini coefficients, Variance reduction, and even ChiSquare. However, in classification the above is somewhat “standard”.

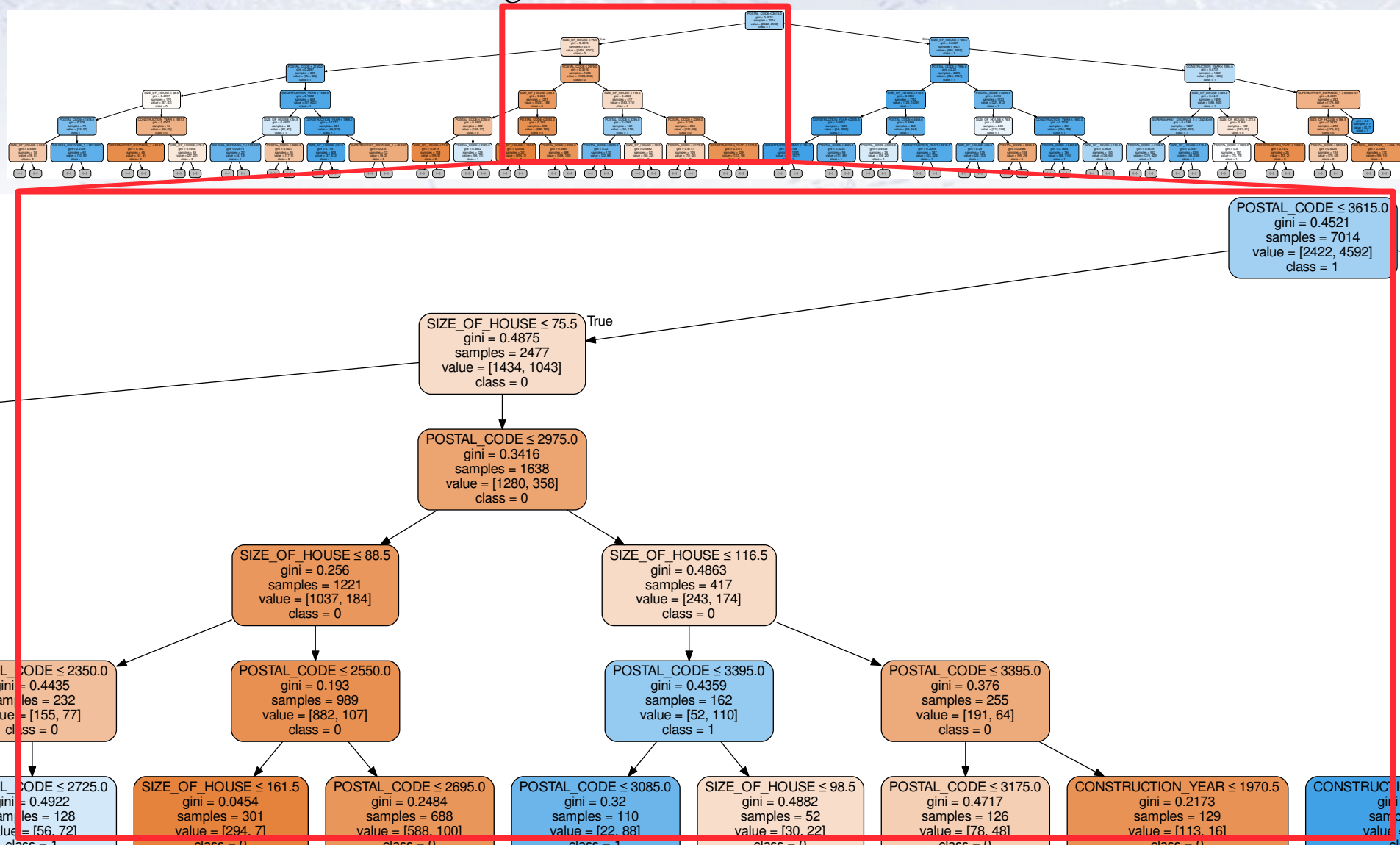
Housing Prices decision tree

Decision tree for estimating the price in the housing prices data set:



Housing Type decision tree

Decision tree for determining, if a house will be sold for more or less than 2Mkr.





The HiggsML Kaggle Challenge

CERN analyses its data using a vast array of ML methods. CERN is thus part of the community that developpes ML!

After 20 years of using Machine Learning it has now become very widespread (NN, BDT, Random Forest, etc.)

A prime example was the Kaggle “HiggsML Challenge”. Most popular challenge of its time! (1785 teams, 6517 downloads, 35772 solutions, 136 forums)




XGBoost history

History [\[edit \]](#)

XGBoost initially started as a research project by Tianqi Chen^[8] as part of the Distributed (Deep) Machine Learning Community (DMLC) group. Initially, it began as a terminal application which could be configured using a libsvm configuration file. After winning the Higgs Machine Learning Challenge, it became well known in the ML competition circles. Soon after, the Python and R packages were built and now it has packages for many other languages like Julia, Scala, Java, etc. This brought the library to more developers and became popular among the [Kaggle](#) community where it has been used for a large number of competitions.^[7]

While Tianqi Chen did not win himself, he provided a method used by about half of the teams, the second place among them!

For this, he got a special award and XGBoost became instantly known in the community.



Higgs Boson Machine Learning Challenge

Use the ATLAS experiment to identify the Higgs boson
\$13,000 · 1,785 teams · 3 years ago

[Overview](#) [Data](#) [Discussion](#) [Leaderboard](#) [Rules](#)

Overview

Description	First Place:
Evaluation	• Gábor Melis - Diósd, Hungary, with this code and model documentation
Prizes	Second Place:
About The Sponsors	• Tim Salimans - Utrecht, The Netherlands, with this code and model documentation
Timeline	Third Place:
Winners	• Pierre C. - Kremlin-bicêtre, France, with this code and model documentation

XGBoost algorithm

The algorithm is documented on the arXiv: 1603.02754

XGBoost: A Scalable Tree Boosting System

Tianqi Chen
University of Washington
tqchen@cs.washington.edu

Carlos Guestrin
University of Washington
guestrin@cs.washington.edu

ABSTRACT

Tree boosting is a highly effective and widely used machine learning method. In this paper, we describe a scalable end-to-end tree boosting system called XGBoost, which is used widely by data scientists to achieve state-of-the-art results on many machine learning challenges. We propose a novel sparsity-aware algorithm for sparse data and weighted quantile sketch for approximate tree learning. More importantly, we provide insights on cache access patterns, data compression and sharding to build a scalable tree boosting system. By combining these insights, XGBoost scales beyond billions of examples using far fewer resources than existing systems.

Keywords

Large-scale Machine Learning

problems. Besides being used as a stand-alone predictor, it is also incorporated into real-world production pipelines for ad click through rate prediction [15]. Finally, it is the de-facto choice of ensemble method and is used in challenges such as the Netflix prize [3].

In this paper, we describe XGBoost, a scalable machine learning system for tree boosting. The system is available as an open source package². The impact of the system has been widely recognized in a number of machine learning and data mining challenges. Take the challenges hosted by the machine learning competition site Kaggle for example. Among the 29 challenge winning solutions³ published at Kaggle's blog during 2015, 17 solutions used XGBoost. Among these solutions, eight solely used XGBoost to train the model, while most others combined XGBoost with neural nets in ensembles. For comparison, the second most popular method, deep neural nets, was used in 11 solutions. The success

XGBoost algorithm

The algorithm is an extension of the decision tree idea (tree boosting), using regression trees with weighted quantiles and being “sparsity aware” (i.e. knowing about lacking entries and low statistics areas of phase space).

Unlike decision trees, each regression tree contains a continuous score on each leaf:

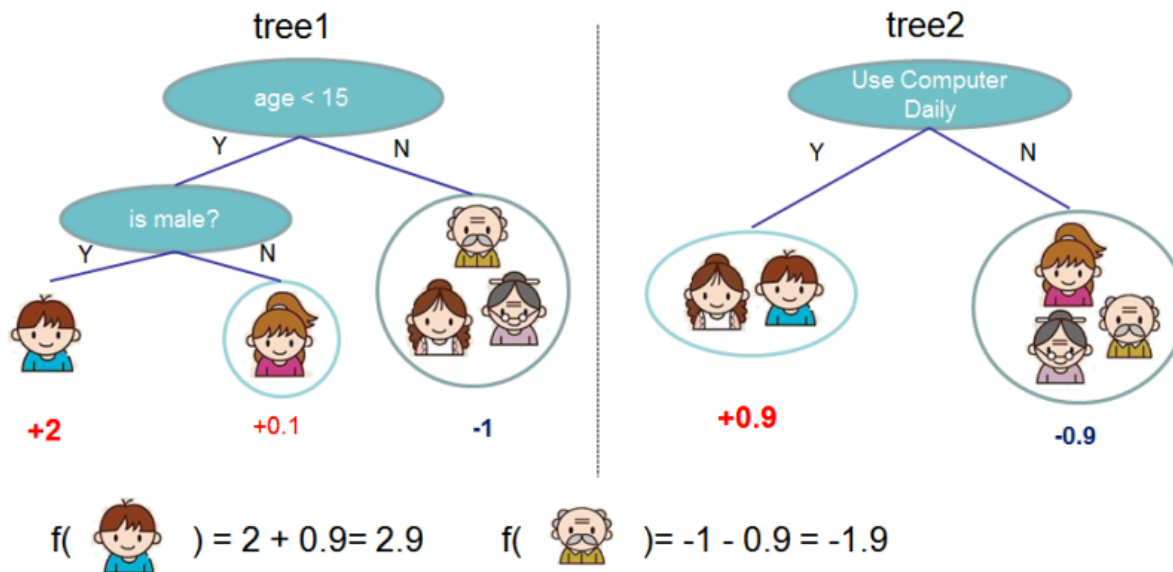
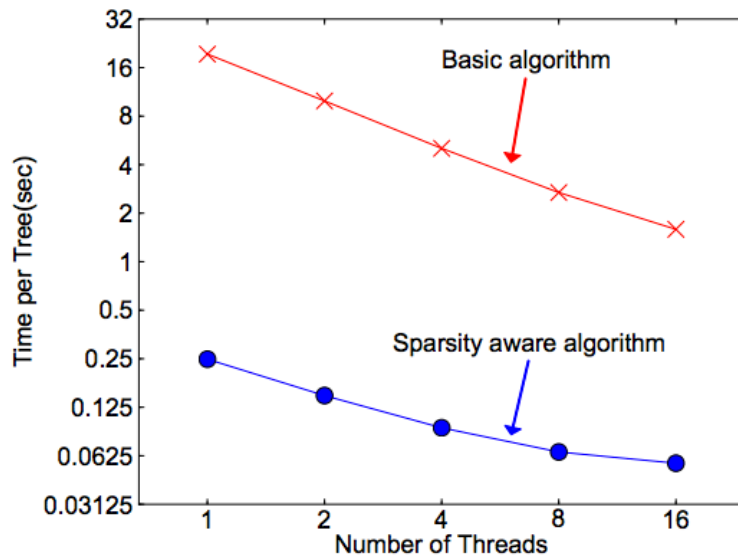


Figure 1: Tree Ensemble Model. The final prediction for a given example is the sum of predictions from each tree.

XGBoost algorithm

The method's speed is partly due to an approximate but fast algorithm to find the best splits.



Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I , instance set of current node

Input: d , feature dimension

$gain \leftarrow 0$

$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$

for $k = 1$ **to** m **do**

$G_L \leftarrow 0, H_L \leftarrow 0$

for j in sorted(I , by \mathbf{x}_{jk}) **do**

$G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$

$G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

$score \leftarrow \max(score, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

end

end

Output: Split with max score

Algorithm 2: Approximate Algorithm for Split Finding

for $k = 1$ **to** m **do**

 Propose $S_k = \{s_{k1}, s_{k2}, \dots, s_{kl}\}$ by percentiles on feature k .

 Proposal can be done per tree (global), or per split(local).

end

for $k = 1$ **to** m **do**

$G_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$

$H_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$

end

Follow same step as in previous section to find max score only among proposed splits.

XGBoost algorithm

In order to “punish” complexity, the cost-function has a regularised term also:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2$$

Table 1: Comparison of major tree boosting systems.

System	exact greedy	approximate global	approximate local	out-of-core	sparsity aware	parallel
XGBoost	yes	yes	yes	yes	yes	yes
pGBRT	no	no	yes	no	no	yes
Spark MLlib	no	yes	no	no	partially	yes
H2O	no	yes	no	no	partially	yes
scikit-learn	yes	no	no	no	no	no
R GBM	yes	no	no	no	partially	no

XGBoost

As it turns out, XGBoost is not only very performant but also very fast...

The most important factor behind the success of XGBoost is its scalability in all scenarios. The system runs more than ten times faster than existing popular solutions on a single machine and scales to billions of examples in distributed or memory-limited settings. The scalability of XGBoost is due to several important systems and algorithmic optimizations.

But this will of course only last for so long - new algorithms see the light of day every week... day?

— — — — — — — — — — shortly after — — — — — — — — — —

Meanwhile, LightGBM has seen the light of day, and it is even faster...

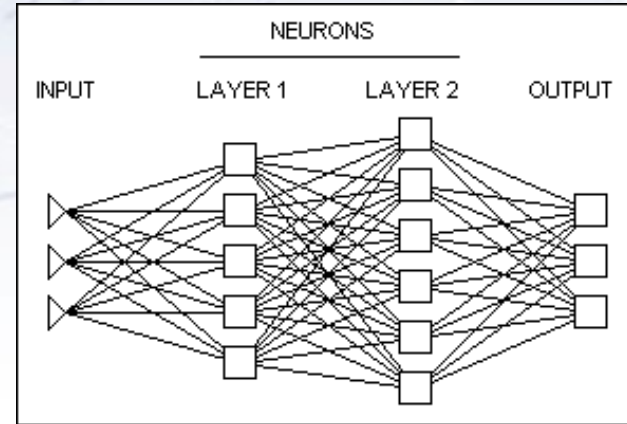
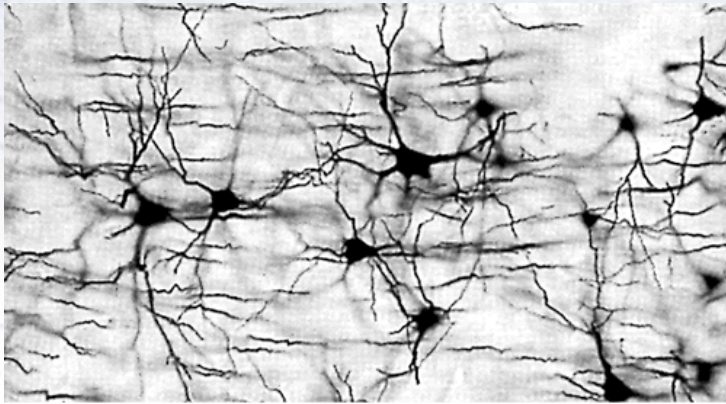
Which algorithm takes the crown: Light GBM vs XGBOOST?

Very good blog with introduction to tree based learning



Neural Network models

Neural Networks (NN)



*In machine learning and related fields, artificial neural networks (ANNs) are computational models inspired by an animal's central nervous systems (in particular the brain) which is capable of **machine learning** as well as **pattern recognition**.*

***Neural networks** have been used to solve a wide variety of tasks that are hard to solve using ordinary rule-based programming, including **computer vision** and **speech recognition**.*

[Wikipedia, Introduction to Artificial Neural Network]

A “Linear Network”

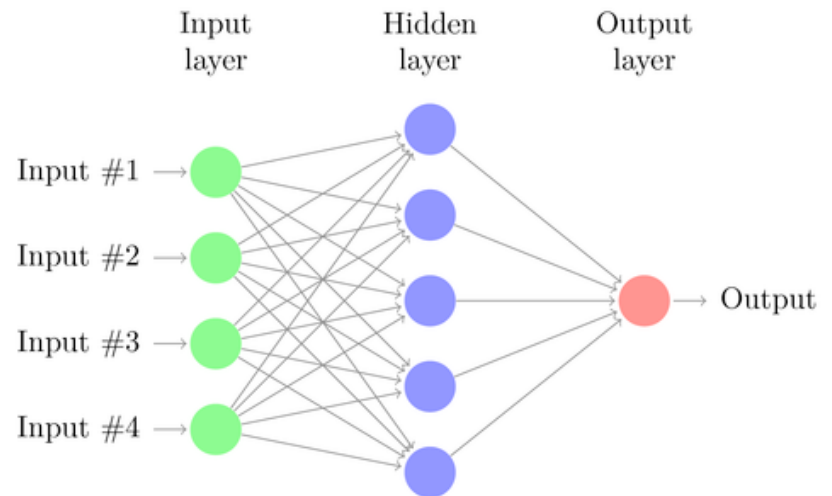
Imagine that we consider a “Linear Network”, and use the (simplest) architecture:
A single layer (linear) perceptron:

$$t(x) = a_0 + \sum a_i x_i$$

As can be seen, this is simply a **linear regression in multiple dimensions** or the (linear) Fisher Discriminant.

Well, then we could consider putting in a hidden (linear) layer:

$$tt(x) = t(a_0 + \sum a_i x_i)$$

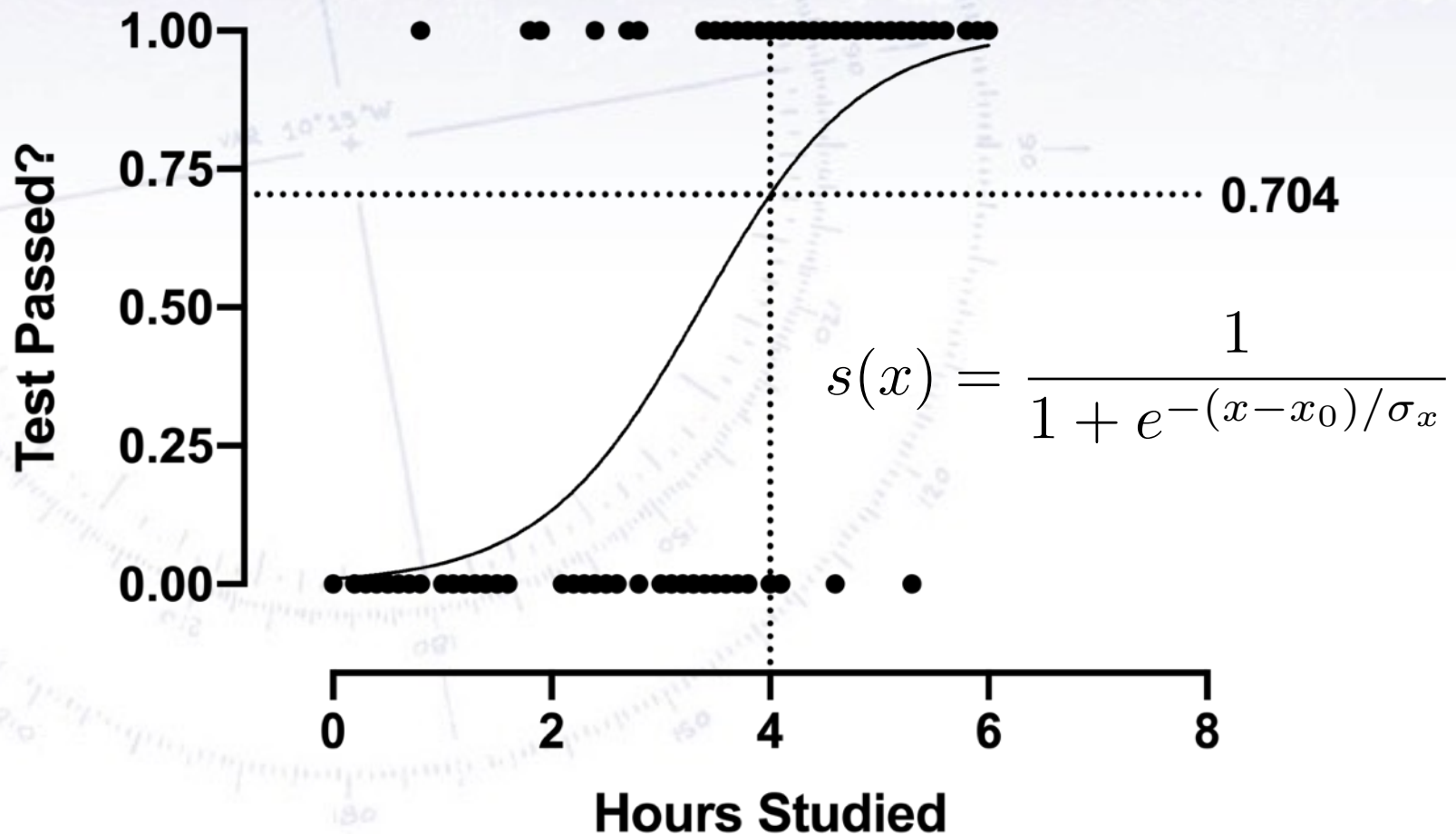


However, this doesn't help anything
as combination of linear functions remain linear. It boils down to the Fisher again!

What we need is something non-linear in the function...

Logistic Regression

Though the word “regression” suggests otherwise, this is in fact a way of doing classification, as the “regression” is usually for a score (s) in the interval [0,1].

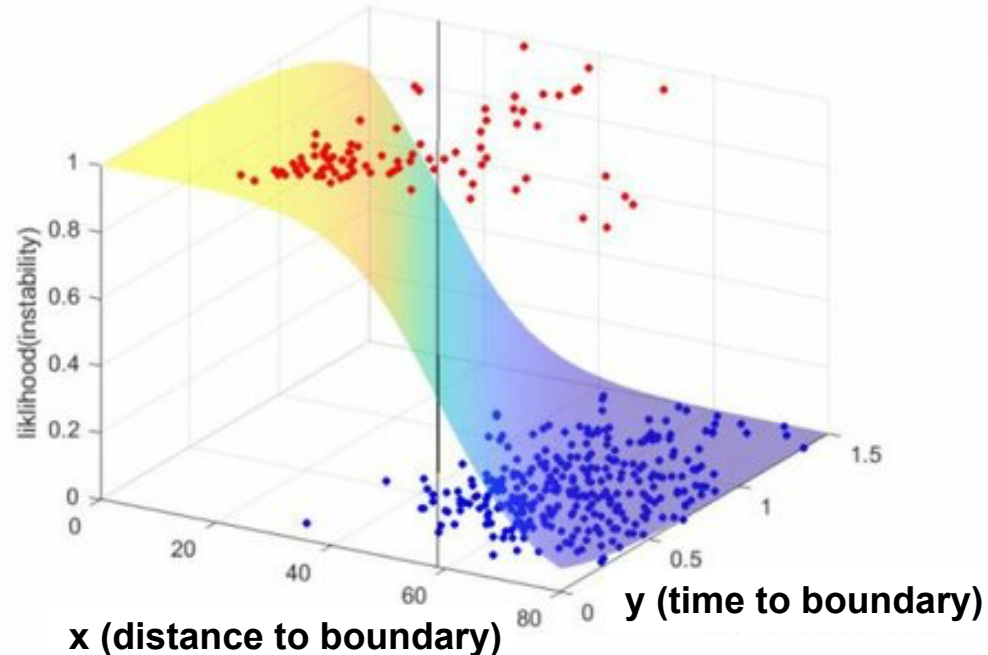
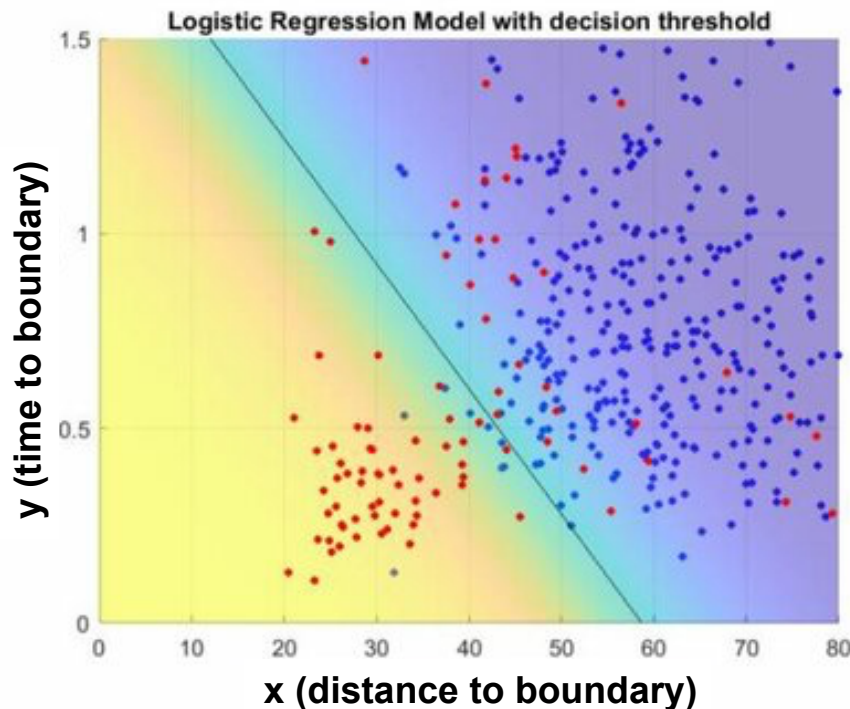


Logistic Regression

Though the word “regression” suggests otherwise, this is in fact a way of doing classification, as the “regression” is usually for a score (s) in the interval $[0,1]$.

The model expands naturally with more parameters:

$$s(x) = \frac{1}{1 + e^{-(x-x_0)/\sigma_x - (y-y_0)/\sigma_y}}$$



Neural Networks

Neural Networks combine the input variables using a “activation” function $s(x)$ to assign, if the variable indicates signal or background.

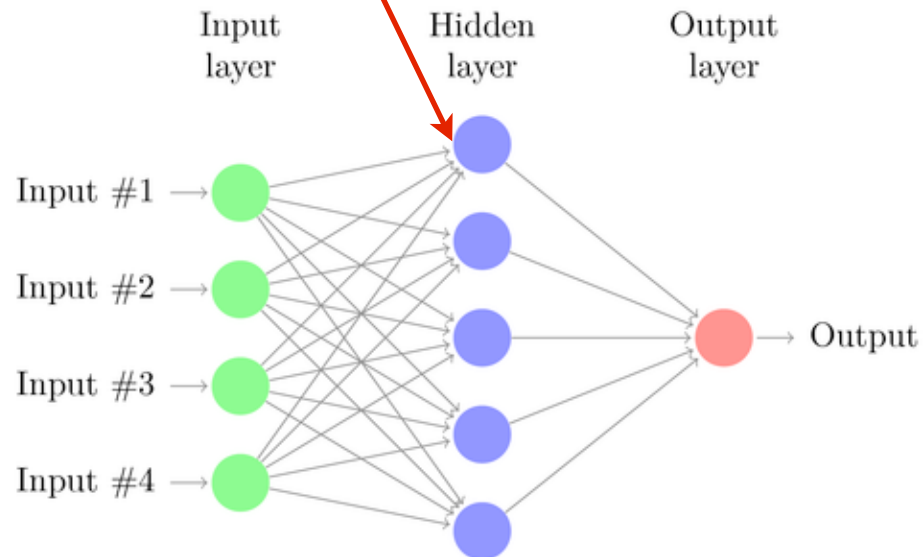
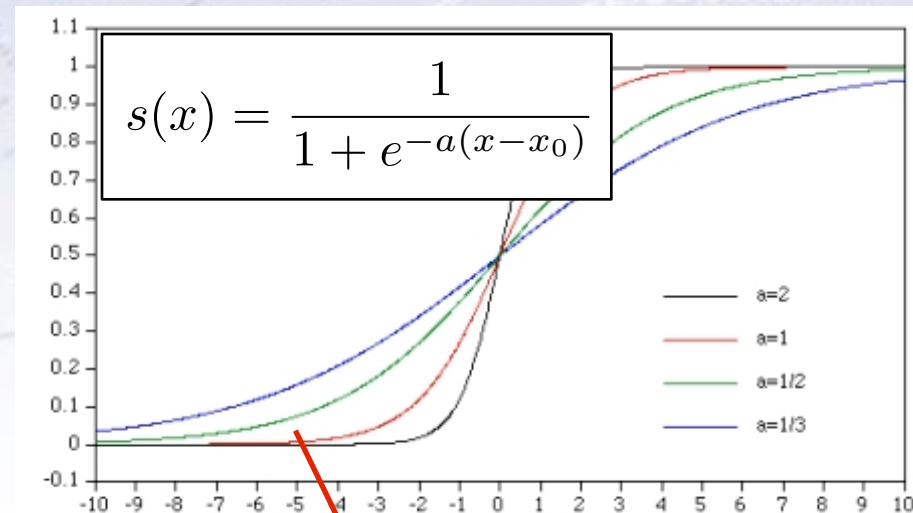
The simplest is a single layer perceptron:

$$t(x) = s \left(a_0 + \sum a_i x_i \right)$$

This can be generalised to a multilayer perceptron (shown right, 1 hidden layer):

$$t(x) = s \left(a_i + \sum a_i h_i(x) \right)$$
$$h_i(x) = s \left(w_{i0} + \sum w_{ij} x_j \right)$$

Activation function can be any “sigmoidal” function.

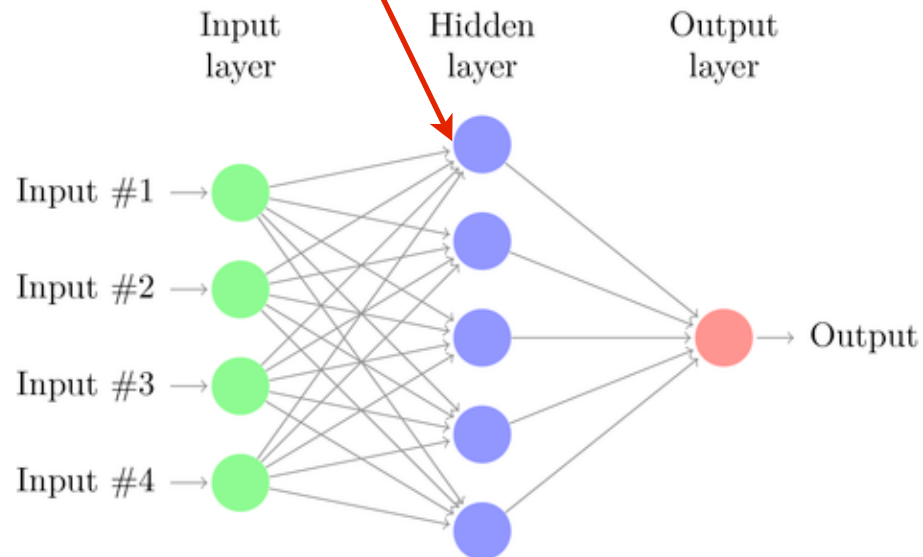
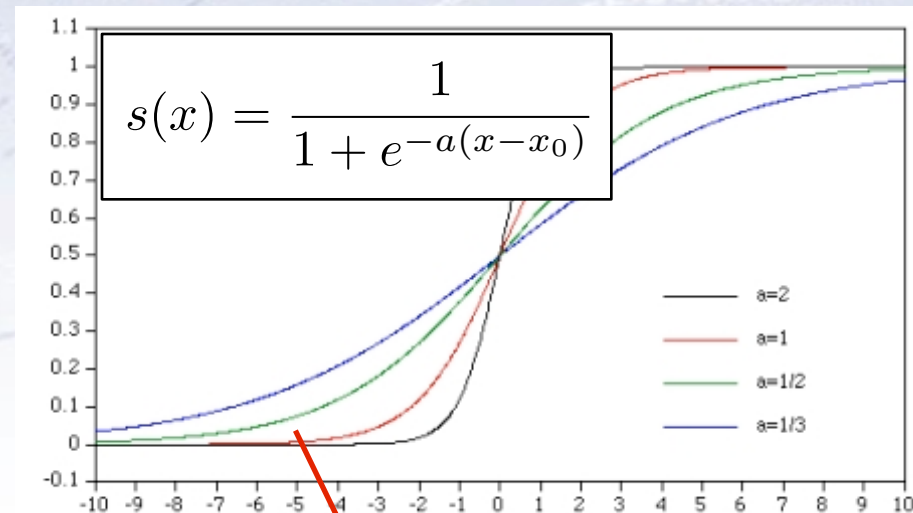
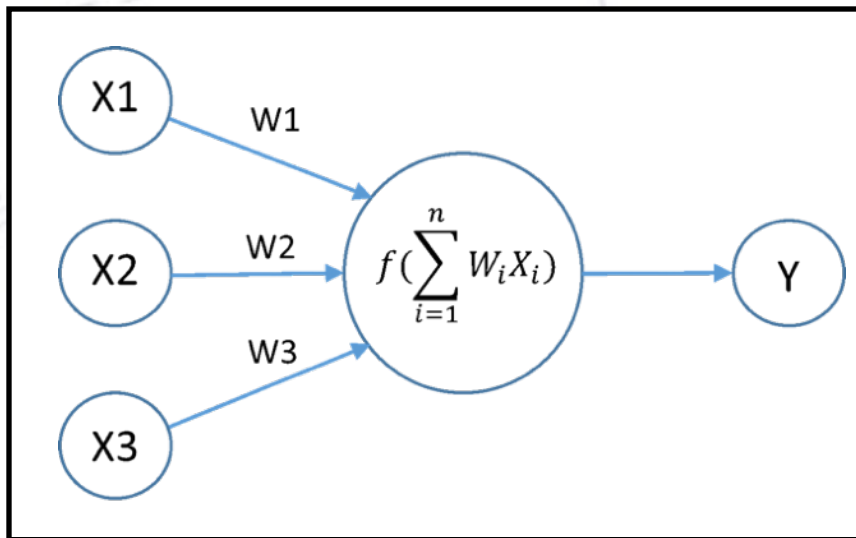


Neural Networks

Neural Networks combine the input variables using a “activation” function $s(x)$ to assign, if the variable indicates signal or background.

The simplest is a single layer perceptron:

$$t(x) = s\left(a_0 + \sum a_i x_i\right)$$



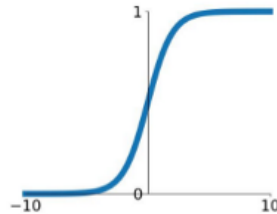
Activation Functions

There are many different activation functions, some of which are shown below. They have different properties, and can be considered a HyperParameter.

Activation Functions

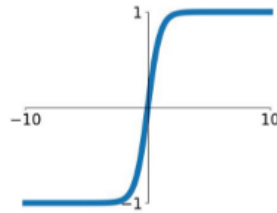
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



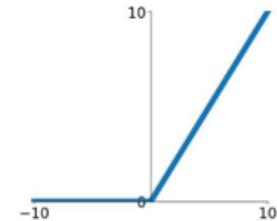
tanh

$$\tanh(x)$$



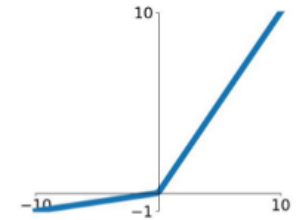
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

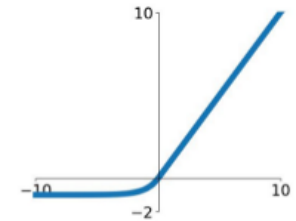


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

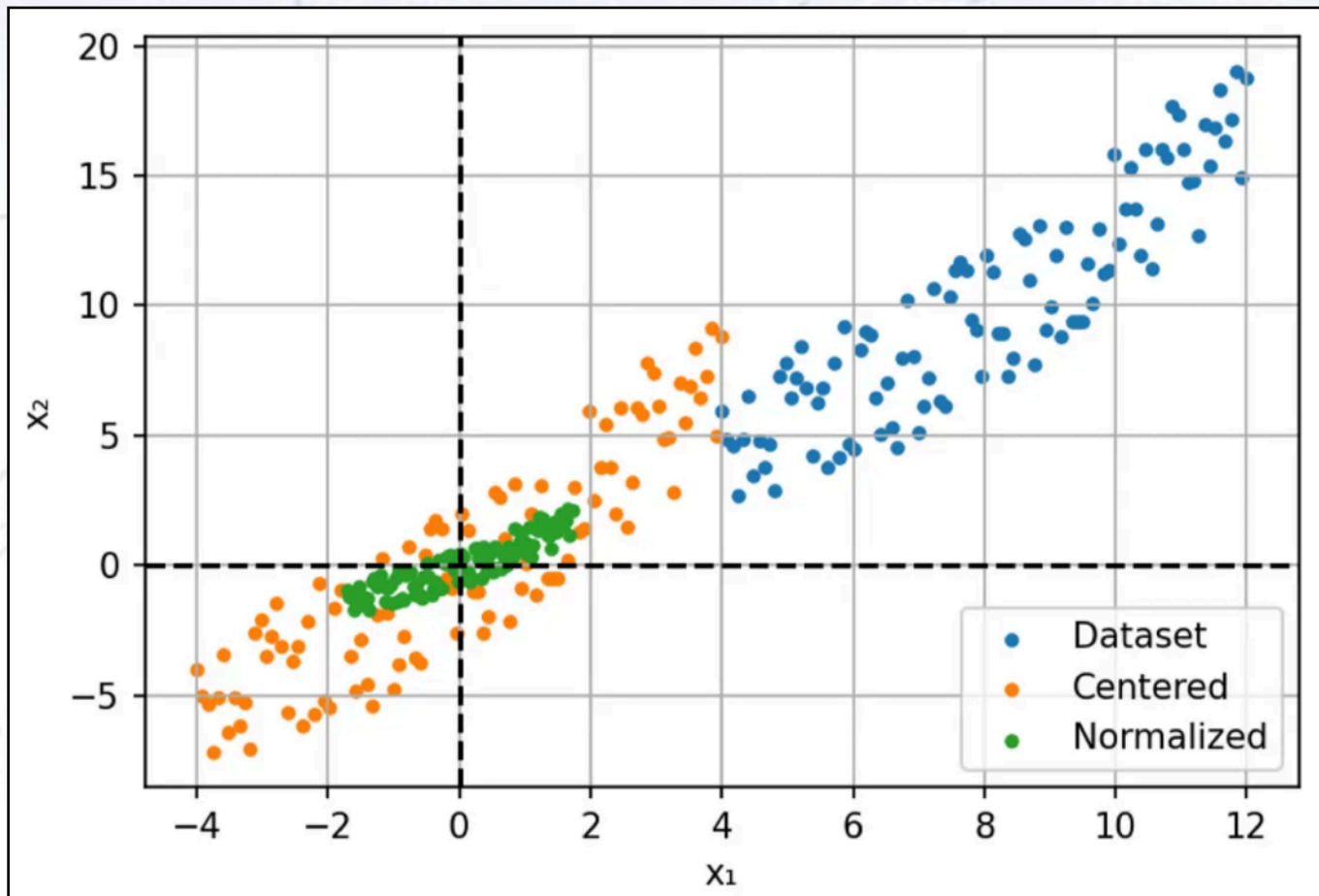
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



For a more complete list, check: https://en.wikipedia.org/wiki/Activation_function

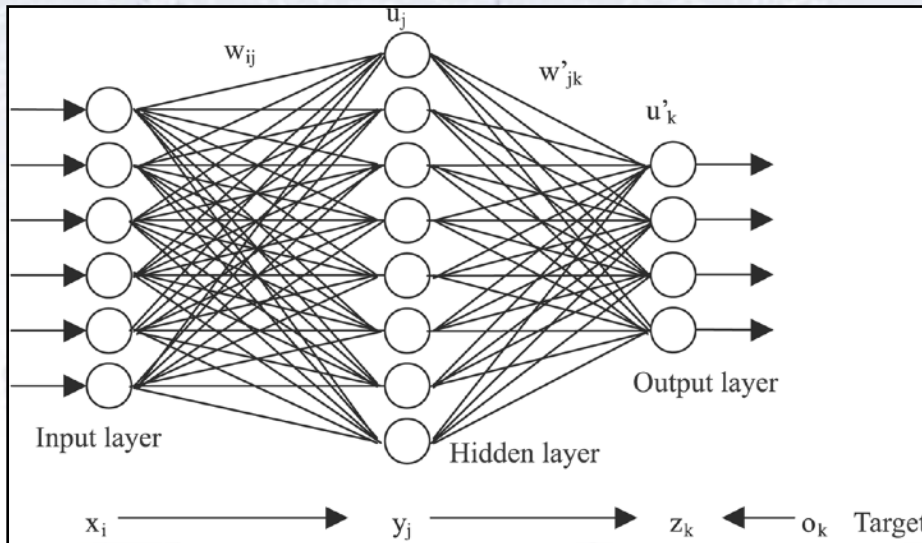
Normalising Inputs

While tree based learning is invariant to (transformations of) distributions, Neural Networks are not. To avoid hard optimisation, vanishing/exploding gradients, and differential learning rates, one should normalise the input:



Deep Neural Networks

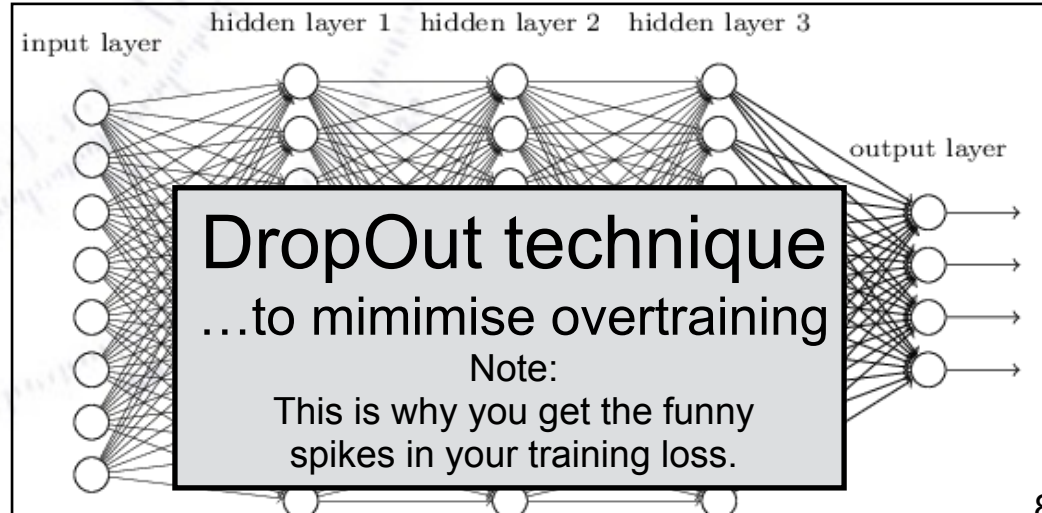
Deep Neural Networks (DNN) are simply (much) extended NNs in terms of layers!



Instead of having just one (or few) hidden layers, many such layers are introduced.

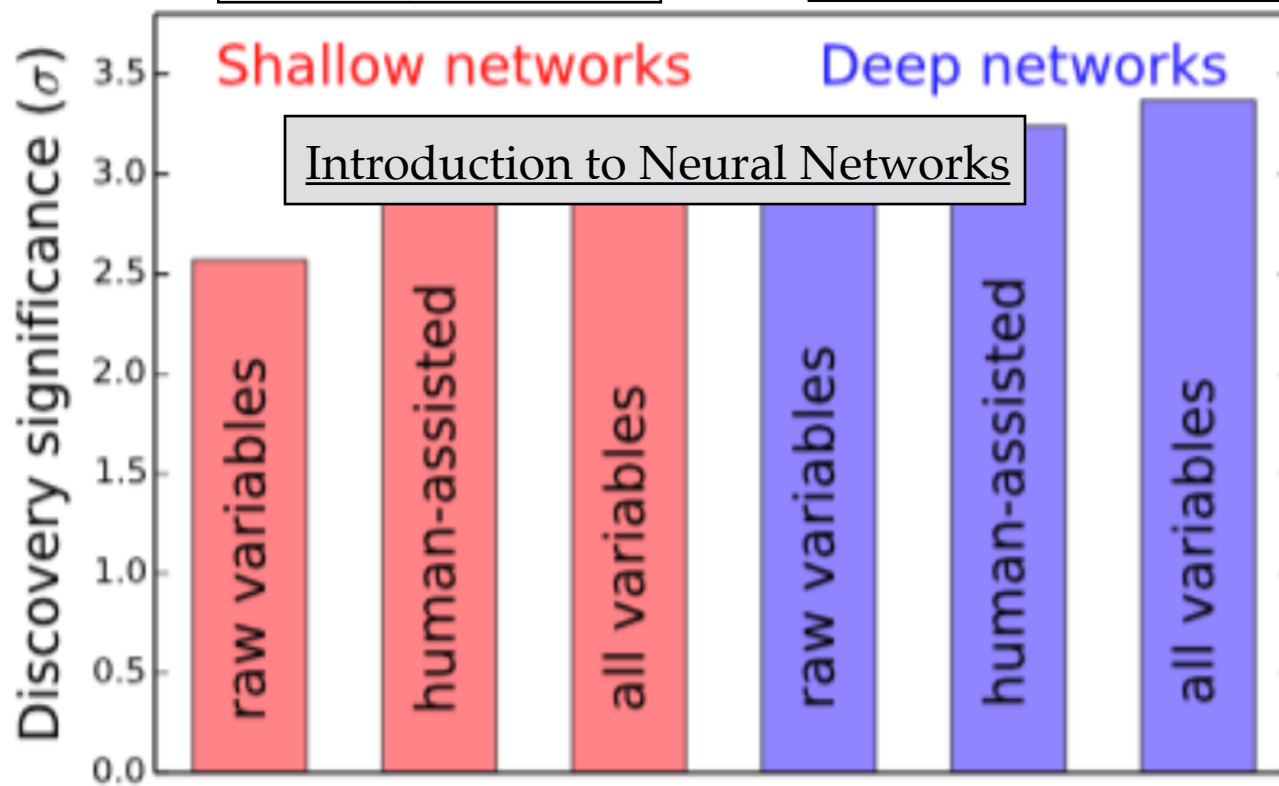
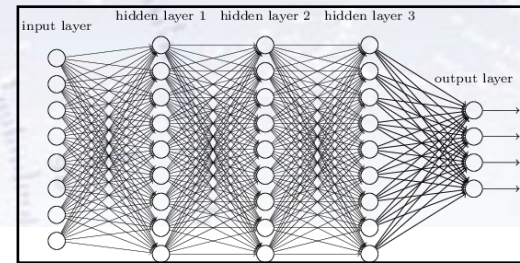
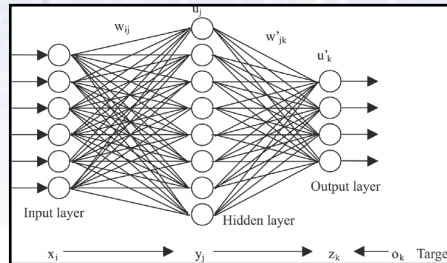
This gives the network a chance to produce key features and use them for many different specialised tasks.

Currently, DNNs can have up to millions of neurons and connections, which compares to about the **brain of a worm**.



Deep Neural Networks

Deep Neural Networks likes to get both raw and “assisted” variables:



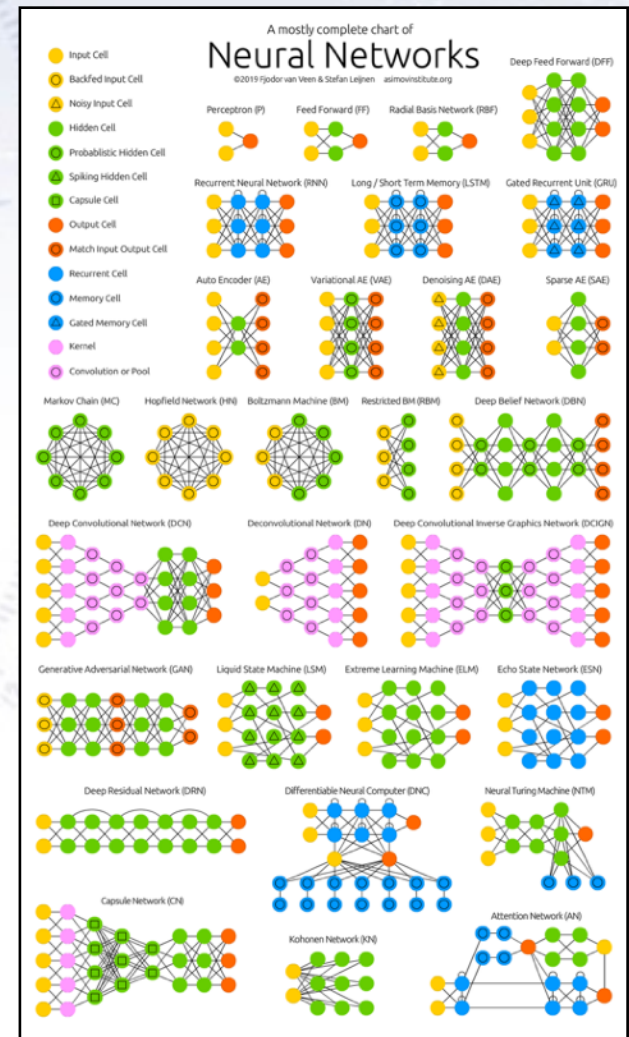
The role of NNs

The reason why NNs play such a central role is that they are versatile:

- Recurrent NNs (for time series)
- Convolutional NNs (for images)
- Adversarial NNs (for simulation)
- Graph NNs (for geometric data)
- etc.

Unlike trees, NNs typically make the “foundation” of all the more advanced ML paradigms. However, they are harder to optimise! This is why trees are great for simpler tasks (i.e. data that typically fits into an excel sheet [2110.01889]), while NNs are typically used for the more advanced.

Have this in mind, when you attack problems with ML - and like any other project or analysis, it is typically good to get a “rough result” fast, and then to refine it from there.



Method's (dis-)advantages

Another comparison is done in Elements of Statistical Learning II (ESL II), where linear methods are not included.

As can be seen, Neural Networks are “difficult” in almost all respects, but performant.

For trees, the case is almost the opposite.

However, I don't agree with the evaluation of the predictive power of trees.

At least not for normal structured data.

For tabular data, I disagree!

Characteristic	Neural Nets	SVM	Trees	MARS	k-NN, Kernels
Natural handling of data of “mixed” type	▼	▼	▲	▲	▼
Handling of missing values	▼	▼	▲	▲	▲
Robustness to outliers in input space	▼	▼	▲	▼	▲
Insensitive to monotone transformations of inputs	▼	▼	▲	▼	▼
Computational scalability (large N)	▼	▼	▲	▲	▼
Ability to deal with irrelevant inputs	▼	▼	▲	▲	▼
Ability to extract linear combinations of features	▲	▲	▼	▼	◆
Interpretability	▼	▼	◆	▲	▼
Predictive power	▲	▲	▼	◆	▲

...and others do too [<https://arxiv.org/abs/2110.01889>]

From ESL II, Chapter 10.7



Loss functions

What loss function to use?

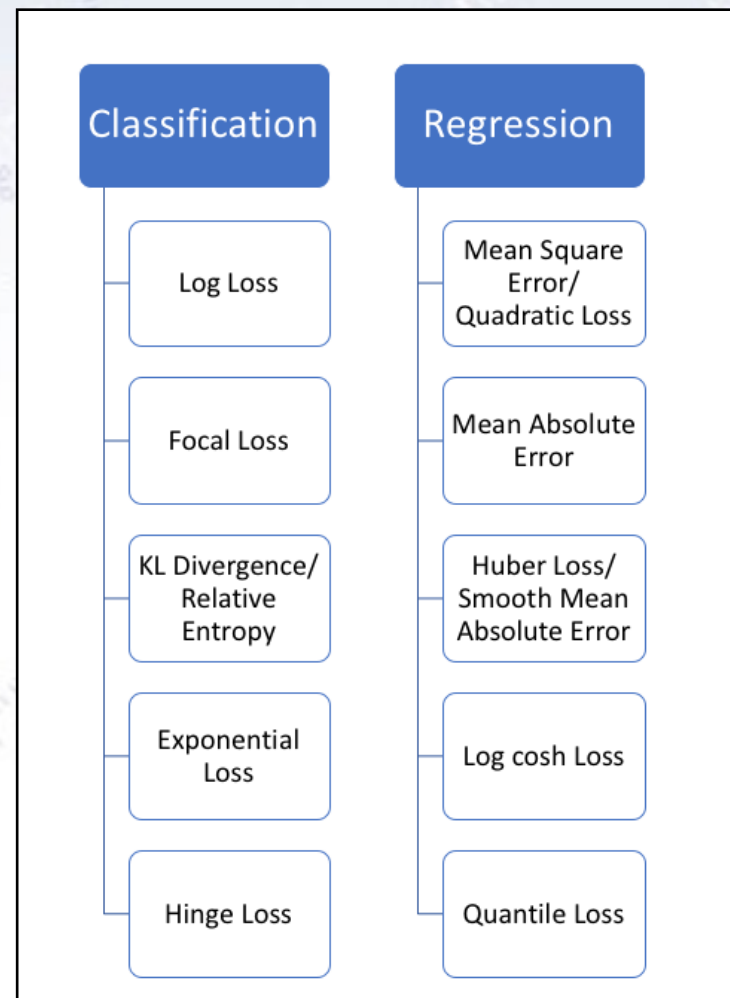
The choice of loss function depends on the problem at hand, and in particular what you find important!

In classification:

- Do you care how wrong the wrong are?
- Do you want pure signal or high efficiency?
- Does it matter what type of errors you make?

In regression:

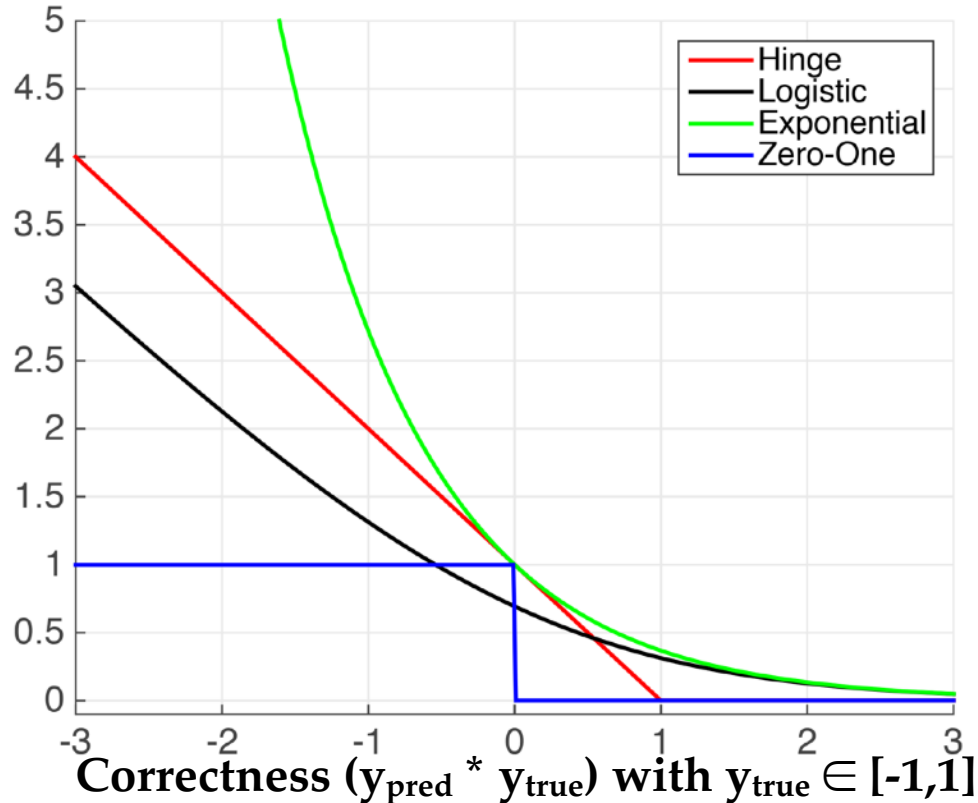
- Do you care about outliers?
- Do you care about size of outliers?
- Is core resolution vital?



What loss function to use?

The choice of loss function depends on the problem at hand, and in particular what you find important!

Loss functions for classification



Classification

Log Loss

Focal Loss

KL Divergence/
Relative
Entropy

Exponential
Loss

Hinge Loss

Regression

Mean Square
Error/
Quadratic Loss

Mean Absolute
Error

Huber Loss/
Smooth Mean
Absolute Error

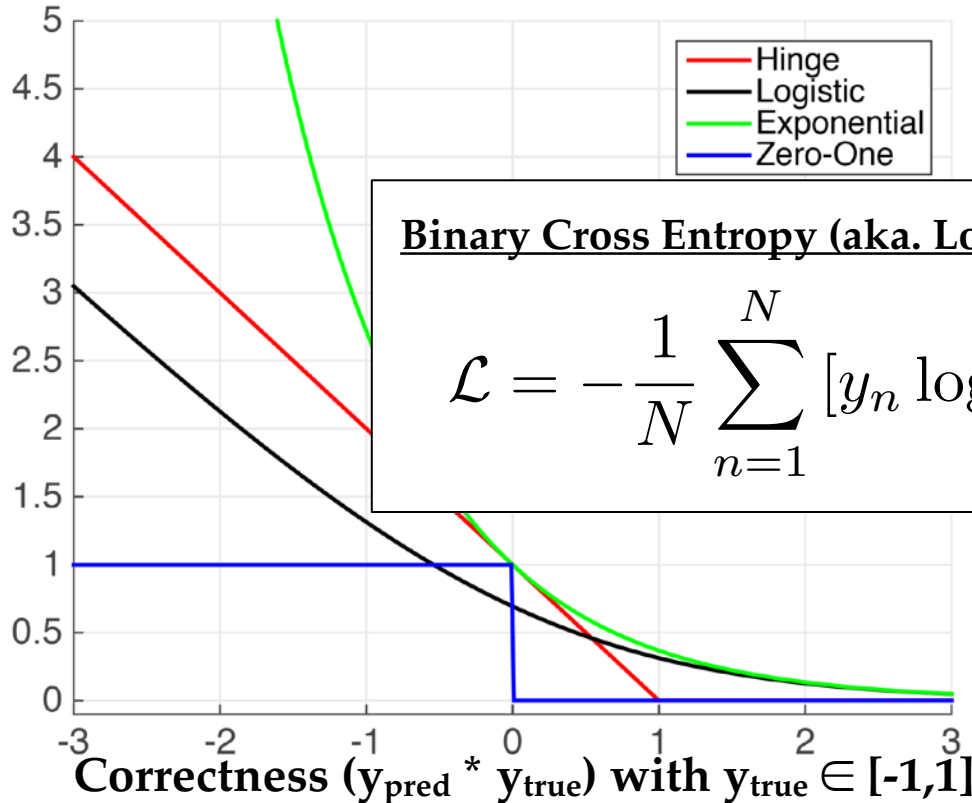
Log cosh Loss

Quantile Loss

What loss function to use?

The choice of loss function depends on the problem at hand, and in particular what you find important!

Loss functions for classification



Binary Cross Entropy (aka. LogLoss or Logistic Loss):

$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^N [y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)]$$

Classification

Log Loss

Regression

Mean Square
Error/
Quadratic Loss

Exponential
Loss

Log cosh Loss

Hinge Loss

Quantile Loss

Unbalanced data

If the data is unbalanced, that is if one outcome/target is much more abundant than the alternative, case has to be taken.

Example: You consider data with 19600 (98%) healthy and 400 (2%) ill patients. An algorithm always predicting “healthy” would get an accuracy score of 98%!

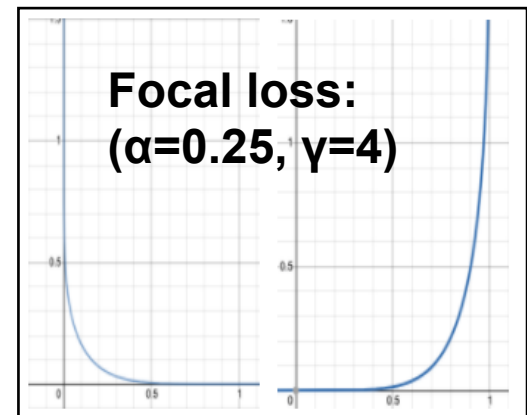
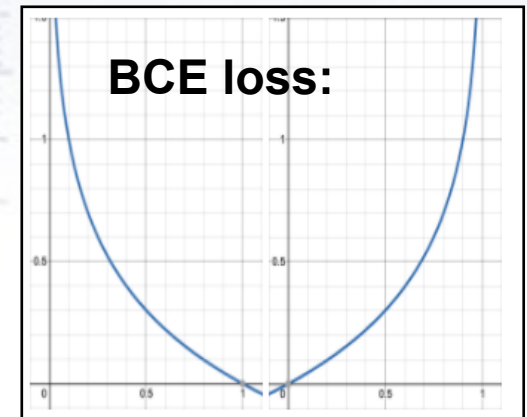
In this case, using Area Under Curve (AUC) or F1 for loss is better. An alternative is “focal loss”, which focuses on the lesser represented cases:

Binary Cross Entropy loss:

$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^N [y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)]$$

Focal loss:

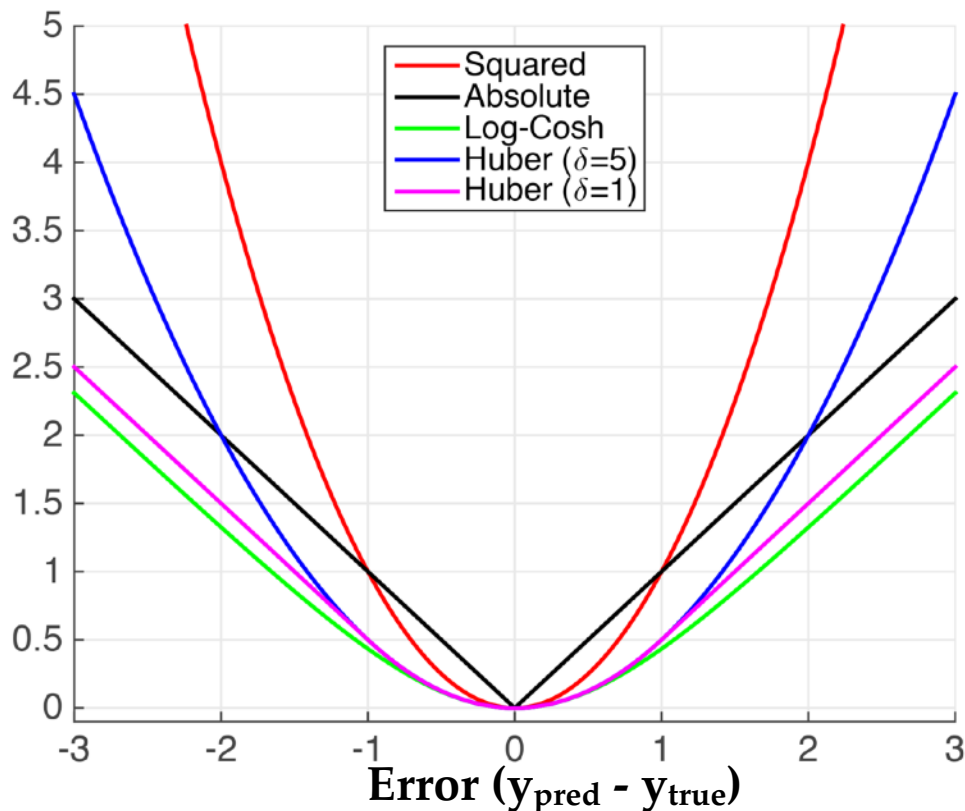
$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^N [(1 - \alpha) y_n^\gamma \log \hat{y}_n + (1 - y_n)^\gamma \log \alpha(1 - \hat{y}_n)]$$



What loss function to use?

The choice of loss function depends on the problem at hand, and in particular what you find important!

Loss functions for regression



Classification

Log Loss

Focal Loss

KL Divergence/
Relative Entropy

Exponential Loss

Hinge Loss

Regression

Mean Square Error/
Quadratic Loss

Mean Absolute Error

Huber Loss/
Smooth Mean Absolute Error

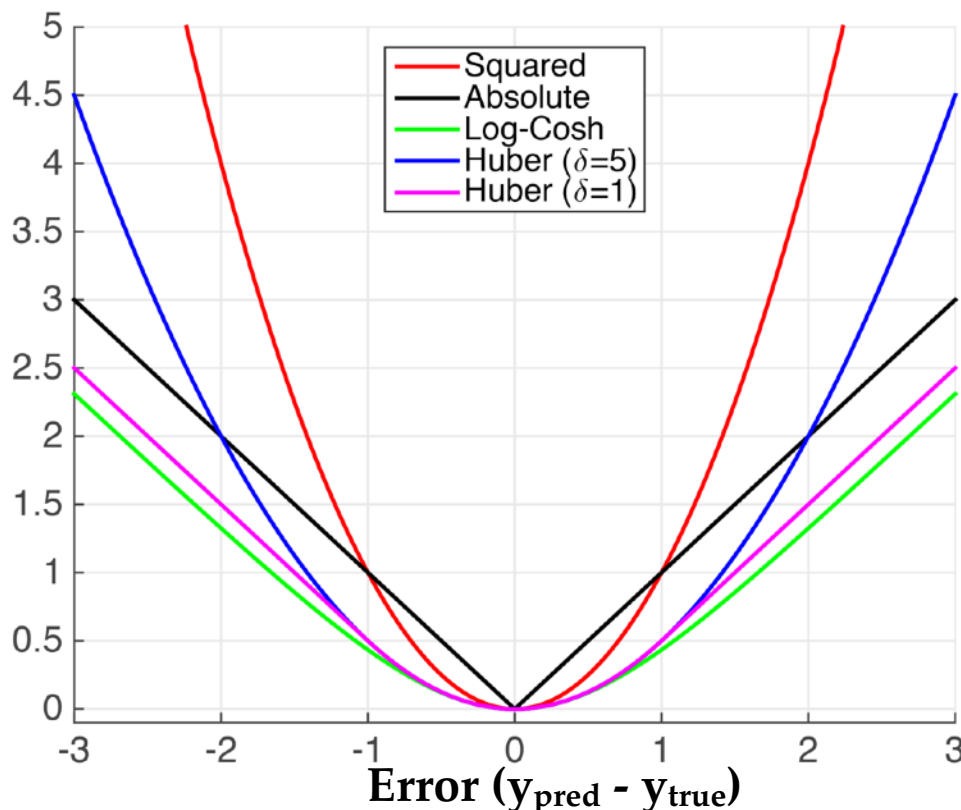
Log cosh Loss

Quantile Loss

What loss function to use?

The choice of loss function depends on the problem at hand, and in particular what you find important!

Loss functions for regression



Squared Loss:

- Most popular regression loss function
- Estimates Mean Label
- ADVANTAGE: Differentiable everywhere
- DISADVANTAGE: Sensitive to outliers

Absolute Loss:

- Also a very popular loss function
- Estimates Median Label
- ADVANTAGE: Less sensitive to noise
- DISADVANTAGE: Not differentiable at 0

Huber Loss:

- ADVANTAGE: "Best of Both Worlds" of Squared and Absolute Loss.
- DISADVANTAGE: Only once-differentiable

LogCosh Loss:

- ADVANTAGE: "Best of Both Worlds" of Squared and Absolute Loss.
- ADVANTAGE: Similar to Huber Loss, but twice differentiable everywhere.

What loss function to use?

The choice of loss function depends on the problem at hand, and in particular what you find important!

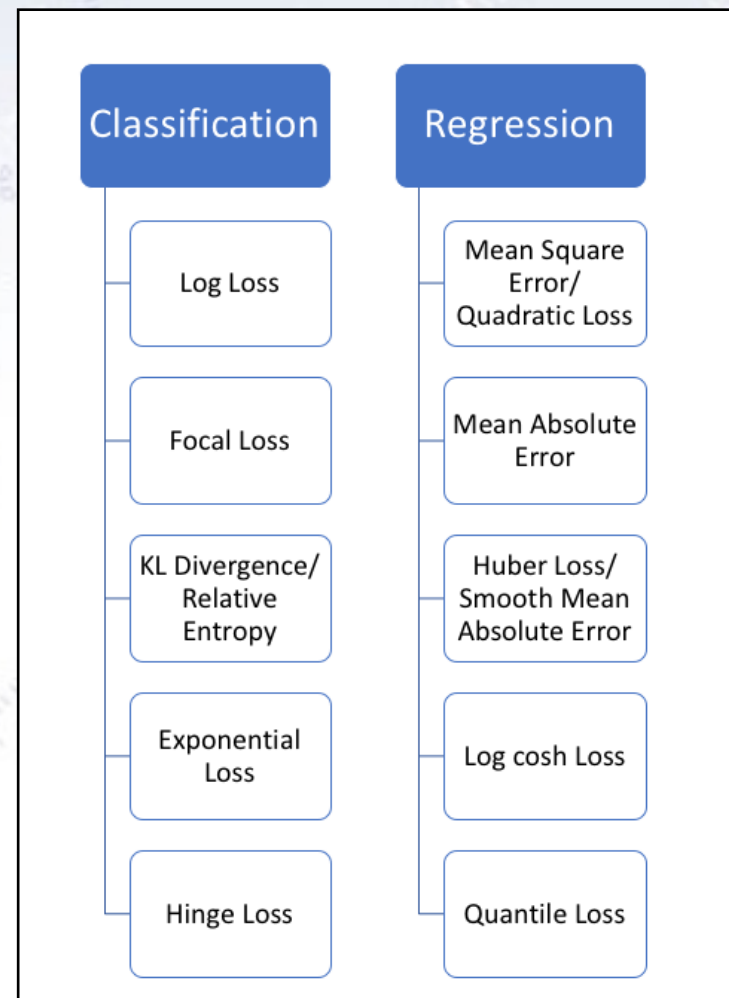
In classification:

- Do you care how wrong the wrong are?
- Do you want pure signal or high efficiency?
- Does it matter what type of errors you make?

In regression:

- Do you care about outliers?
- Do you care about size of outliers?
- Is core resolution vital?

Ultimately, the loss function should be tailored to match the wishes of the user. This is however not always that simple, as this might be hard to even know!



XGBoost algorithm

In order to “punish” complexity, the cost-function has a regularised term also:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2$$

Table 1: Comparison of major tree boosting systems.

System	exact greedy	approximate global	approximate local	out-of-core	sparsity aware	parallel
XGBoost	yes	yes	yes	yes	yes	yes
pGBRT	no	no	yes	no	no	yes
Spark MLlib	no	yes	no	no	partially	yes
H2O	no	yes	no	no	partially	yes
scikit-learn	yes	no	no	no	no	no
R GBM	yes	no	no	no	partially	no

XGBoost algorithm

In order to “punish” complexity, the cost-function has a regularised term also:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2$$

Table 1: Comparison of major tree boosting systems.

System	exact greedy	approximate global	approximate local	out-of-core	sparsity aware	parallel
XGBoost						
pGBRT						
Spark MLlib						
H2O						
scikit-learn	yes	no	no	no	no	no
R GBM	yes	no	no	no	partially	no

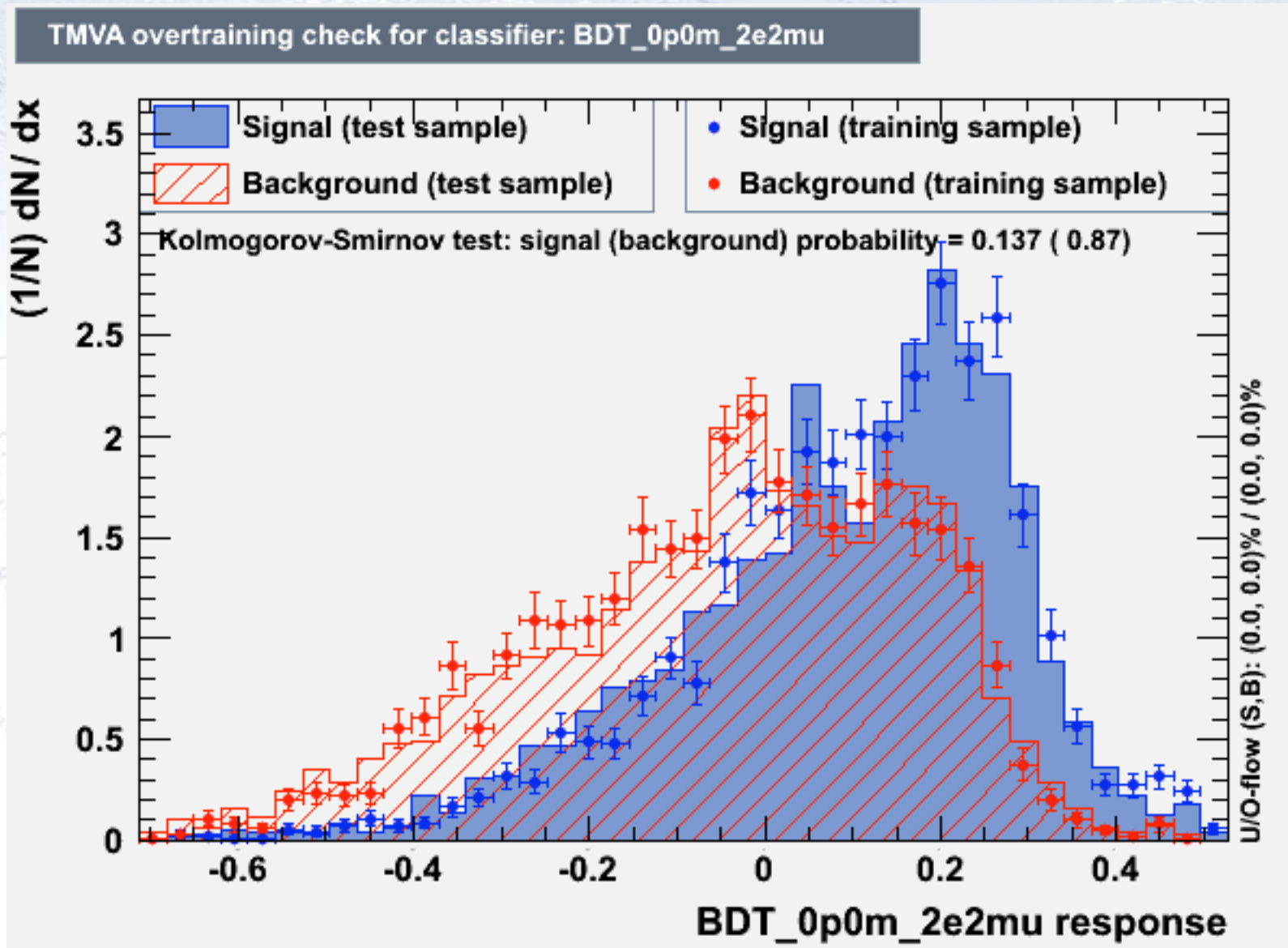
Generally, all constraints or priors should be included into the model through additions to the loss function.



Train, Validation & Test

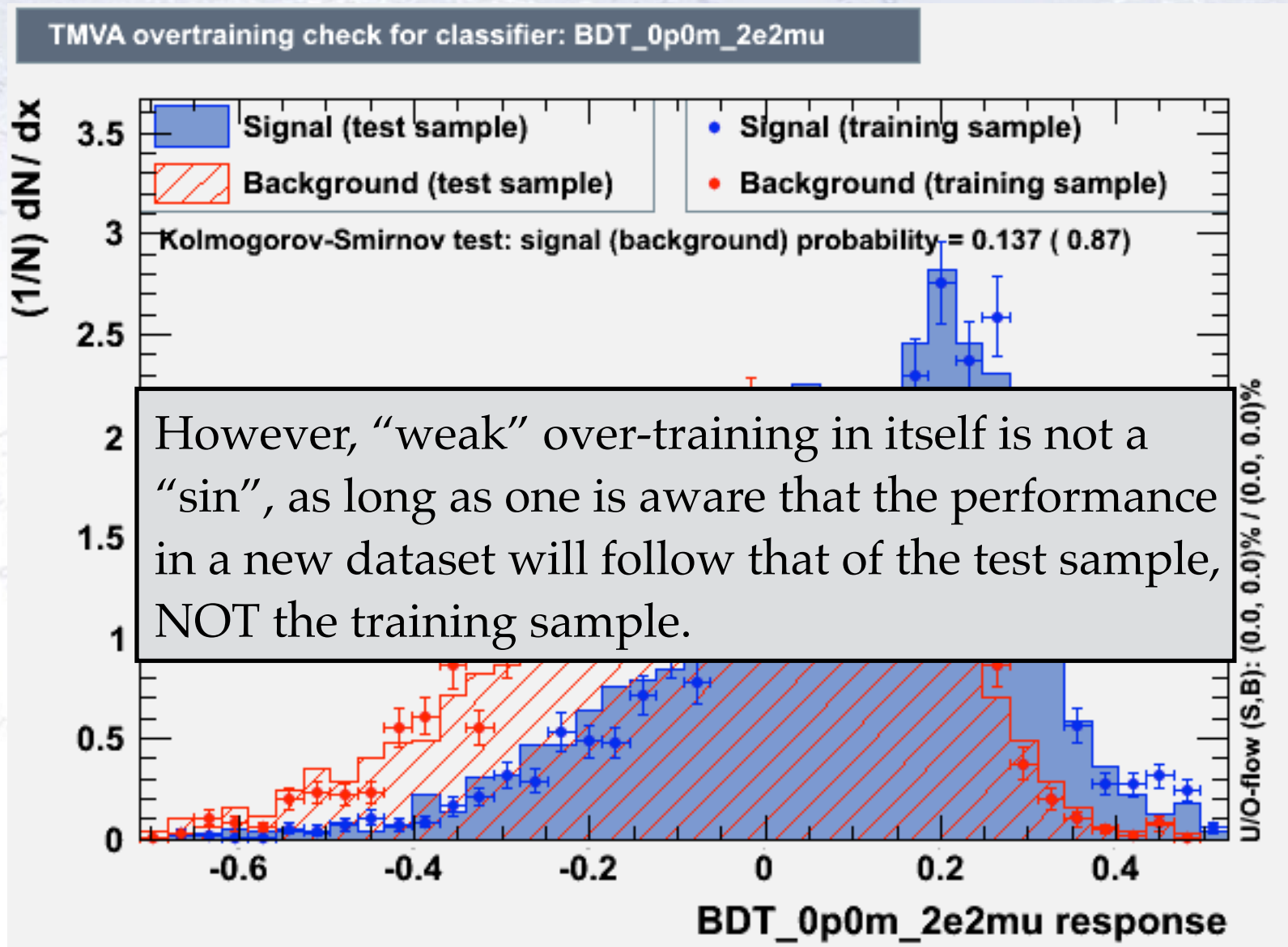
Test for simple over-training

In order to test for overtraining, half the sample is used for training, the other for testing:



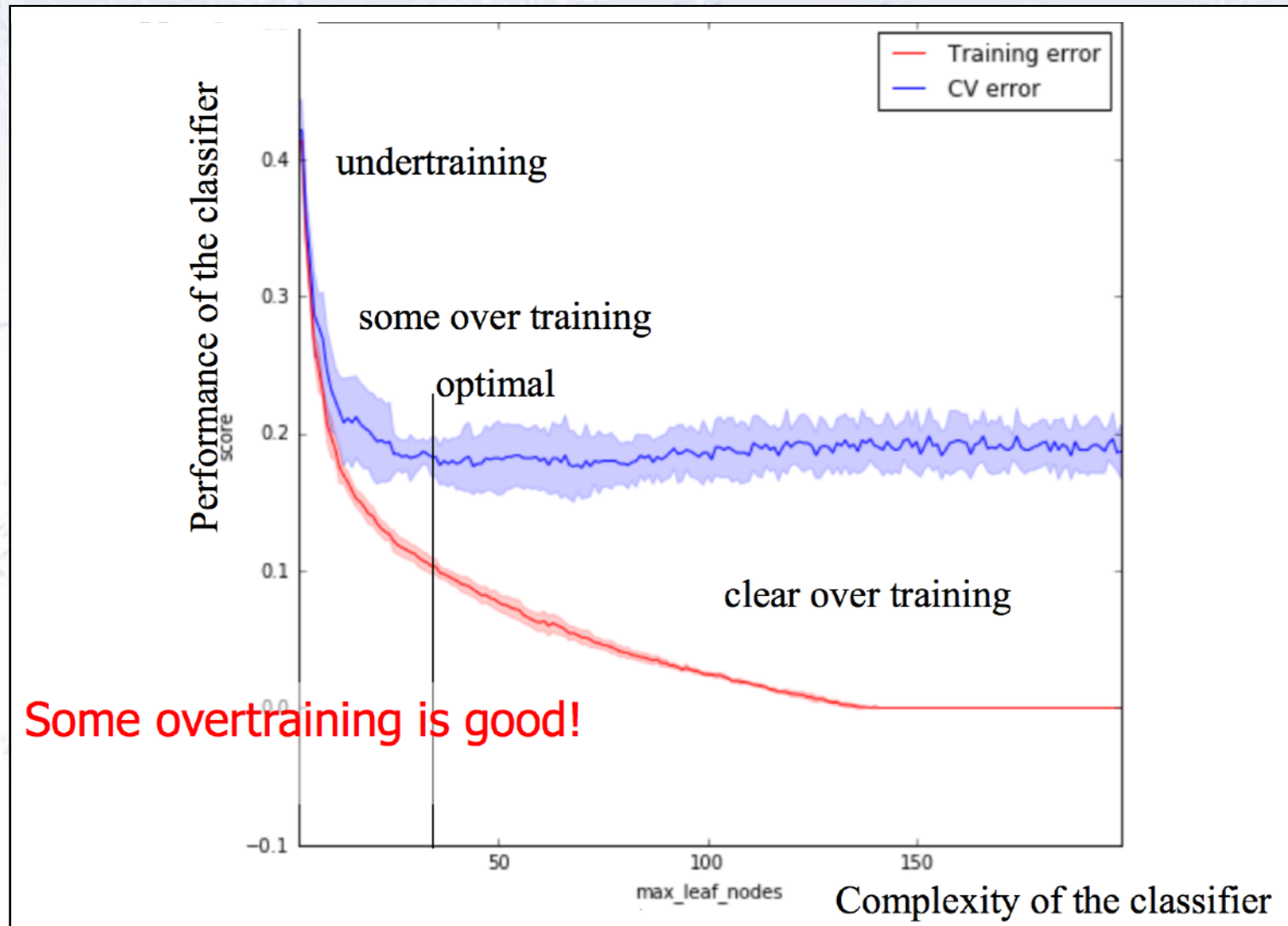
Test for simple over-training

In order to test for overtraining, half the sample is used for training, the other for testing:



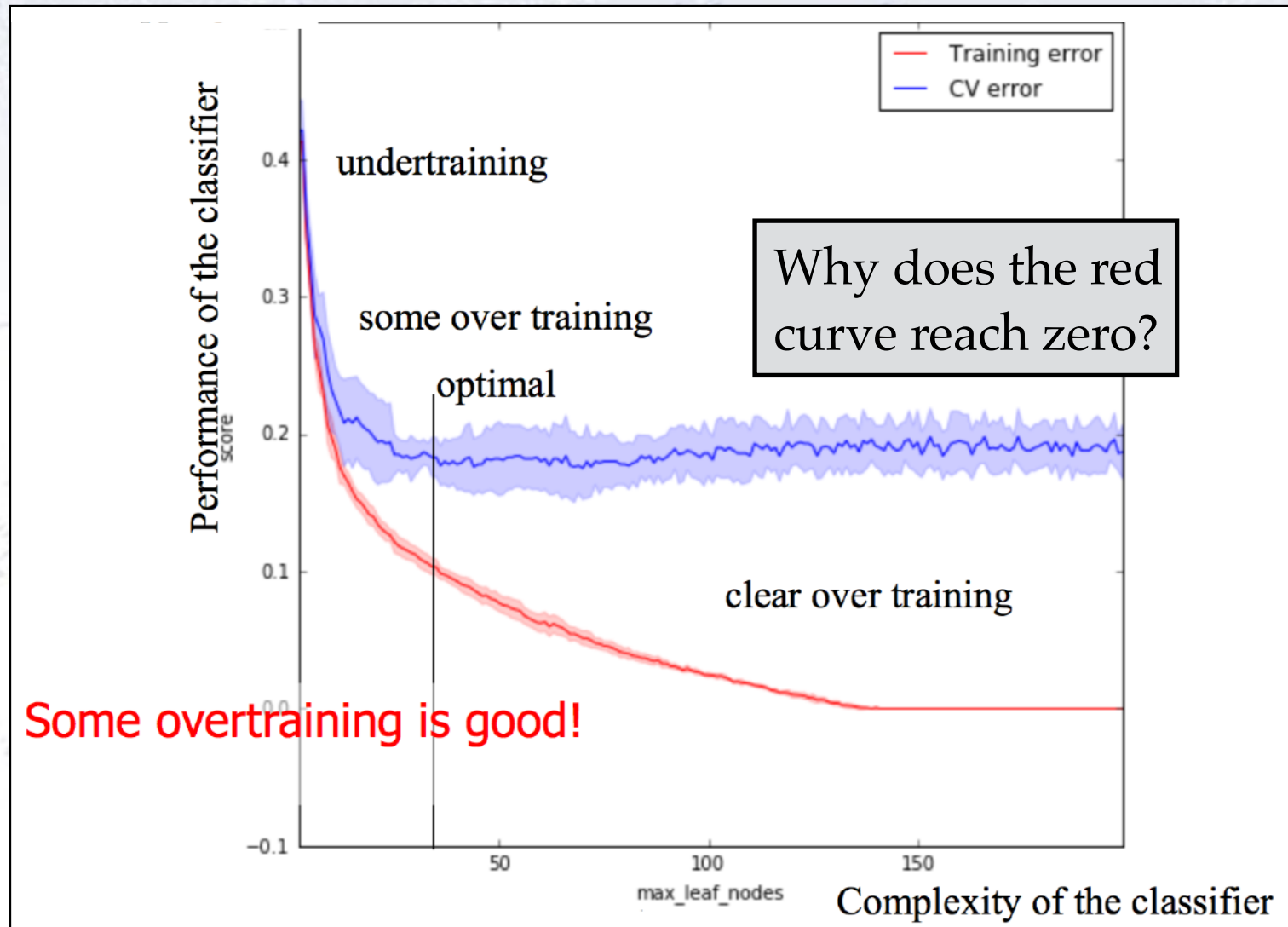
Real overtraining

The “real” limit of overtraining, is when the (Cross) Validation (CV) error starts to grow!



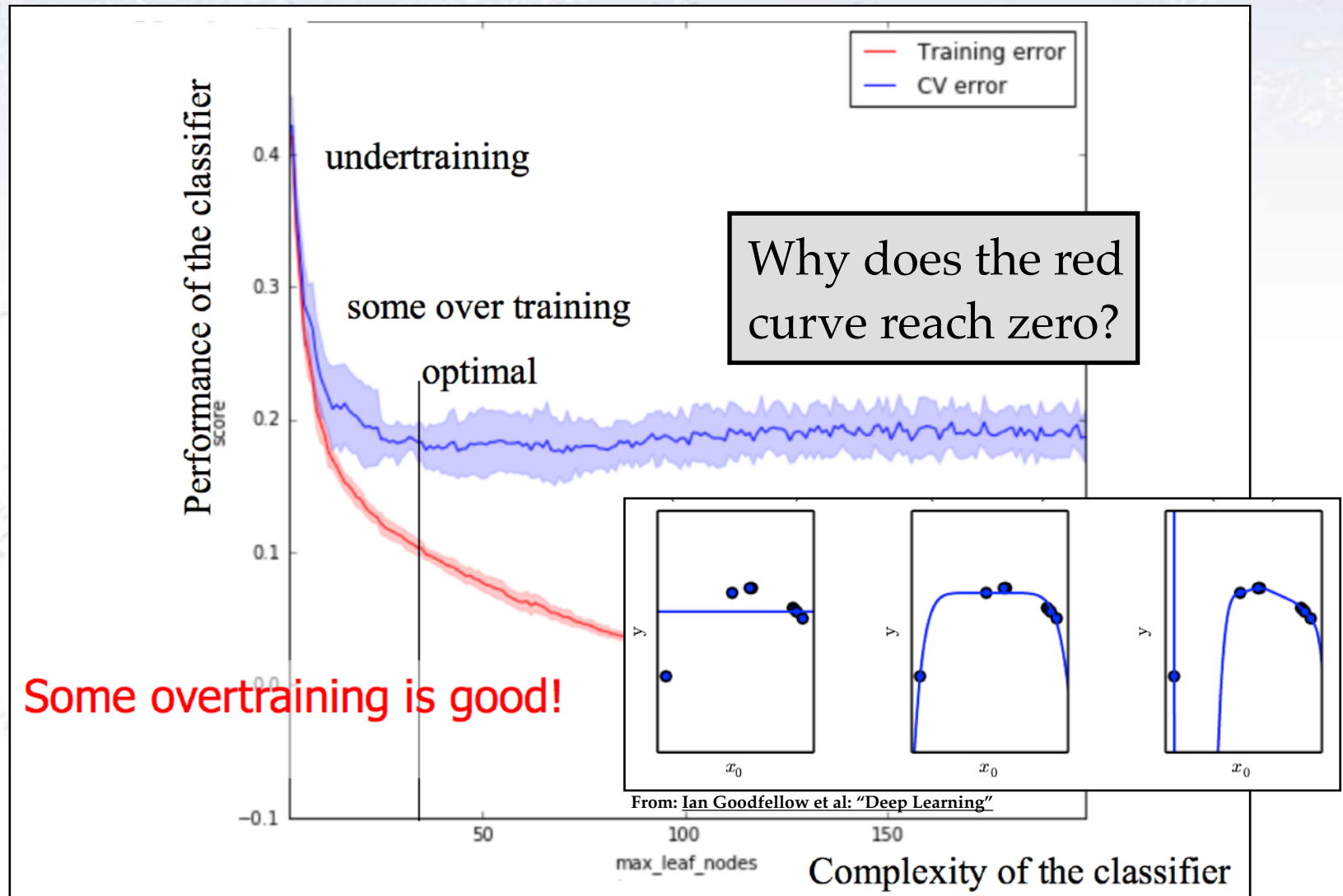
Real overtraining

The “real” limit of overtraining, is when the (Cross) Validation (CV) error starts to grow!



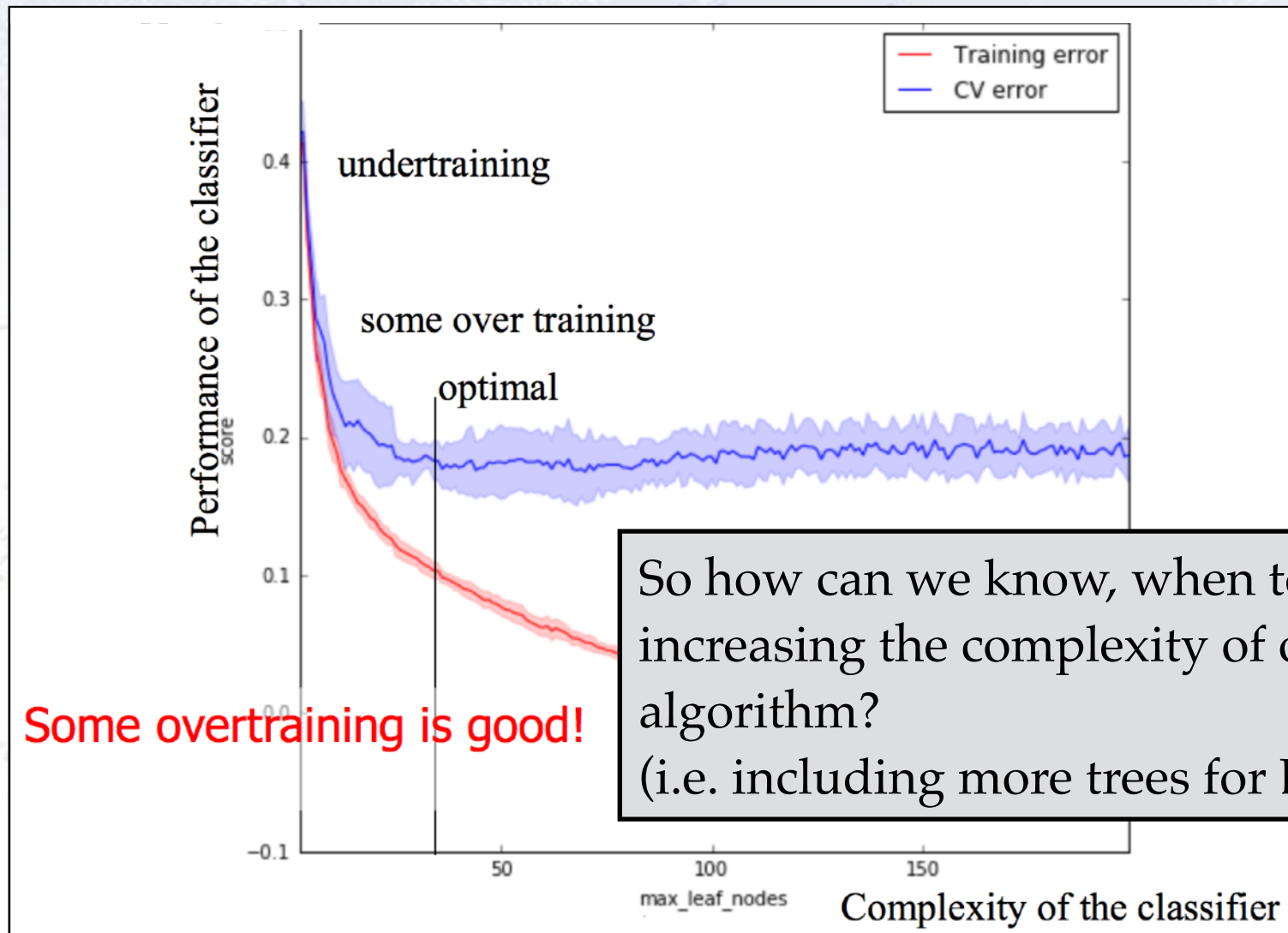
Real overtraining

The “real” limit of overtraining, is when the (Cross) Validation (CV) error starts to grow!



Real overtraining

The “real” limit of overtraining, is when the (Cross) Validation (CV) error starts to grow!





Dividing Data

How to “use” your data?

If you train your algorithm on all data, you will not recognise overtrain, nor what the expected performance on new data will be. Thus we divide the data into:

Train Dataset

- Set of data used for **learning** (by the model), that is, to fit the parameters to the machine learning model using **stochastic gradient descent**.

Valid Dataset

- Set of data used to provide an **unbiased evaluation of intermediate models** fitted on the training dataset while tuning model parameters and hyperparameters, and also for selecting input features.

Test Dataset

- Set of data used to provide an **unbiased evaluation of a final model** fitted on the training dataset.



How to do the division?

You can of course do this yourself with your own code, but there are specially prepared functions for the task:

Scikit-Learn method:

```
from sklearn.model_selection import train_test_split
X_train, X_rem, y_train, y_rem = train_test_split(X, y, train_size=0.8)
X_valid, X_test, y_valid, y_test = train_test_split(X_rem, y_rem, test_size=0.5)
```

Fast ML method:

```
from fast_ml.model_development import train_valid_test_split
X_train, y_train, X_valid, y_valid, X_test, y_test =
train_valid_test_split(df, target = '?', train_size=0.8, valid_size=0.1, test_size=0.1)
```

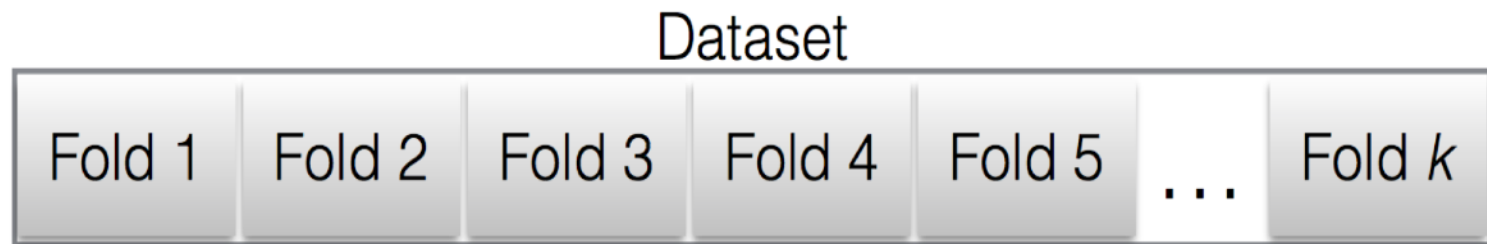
There are a few important things to remember:

- Always do the data cleaning, selecting, weighting, etc. **before** splitting!
- If there is “more than enough” data, then use **less than the total**.
- If there is “a little too little” data, then use **cross validation (next)**.

k-fold Cross Validation

In case your data set is not that large (and perhaps anyhow), one can train on most of it, and then test on the remaining $1/k$ fraction.

This is then repeated for each fold... CPU-intensive, but easily parallelisable and smart especially for small data samples.

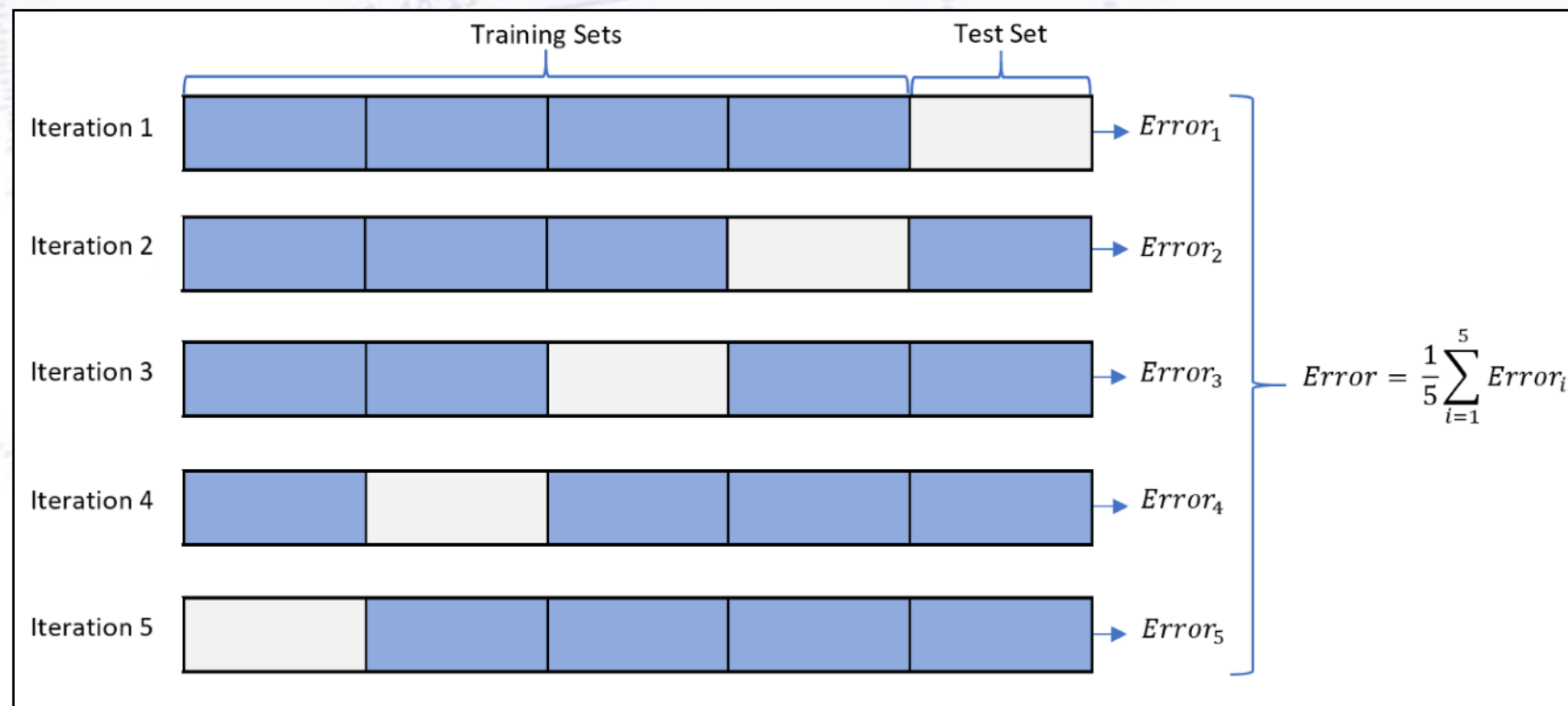


- ▶ Split the dataset into k randomly sampled independent subsets (folds).
- ▶ Train classifier with k-1 folds and test with remaining fold.
- ▶ Repeat k times.

Getting an uncertainty estimate

The k-fold cross validation (CV) does not only allow you to train on almost all $(1 - (1/k))$ and test on all the data, but also has two additional advantages:

- If you consider the performance (“Error” below) on each fold, then you can also calculate the uncertainty on the performance.
- Since you can test on all data, the uncertainty on the loss estimate goes down.



Why use CV?

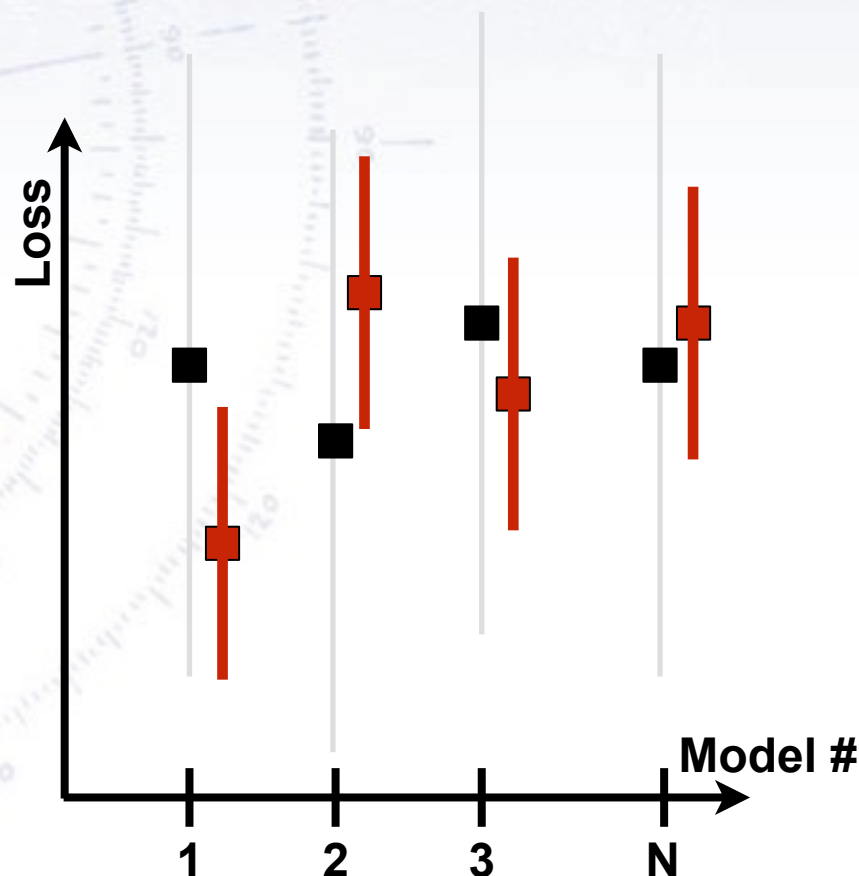
The k-fold cross validation (CV) allows you to get a better error estimate and knowledge of the uncertainty.

Imagine that you train N different models (different type, HPs, training, etc.), and that you get results as shown:

You conclude that model #2 is best. However, you don't know, that the uncertainties are rather large, because your test sample (20%) is small!

Then you do 5-fold CV... and get a more accurate evaluation with smaller uncertainties (by factor $1/\sqrt{5}$).

Now you conclude, that model #1 is the best... and that model #2 is worst!



Why use CV?

The k-fold cross validation (CV) allows you to get a better error estimate and knowledge of the uncertainty.

Imagine that you train N different models (different type, HPs, training, etc.), and that you get results as shown:

You conclude three cases:

- However, you have large uncertainties on your test set performance.
- When there is little (test) data.
 - When you want uncertainty on performance.
 - When accurate performance measure is wanted, e.g. to find the very best model.

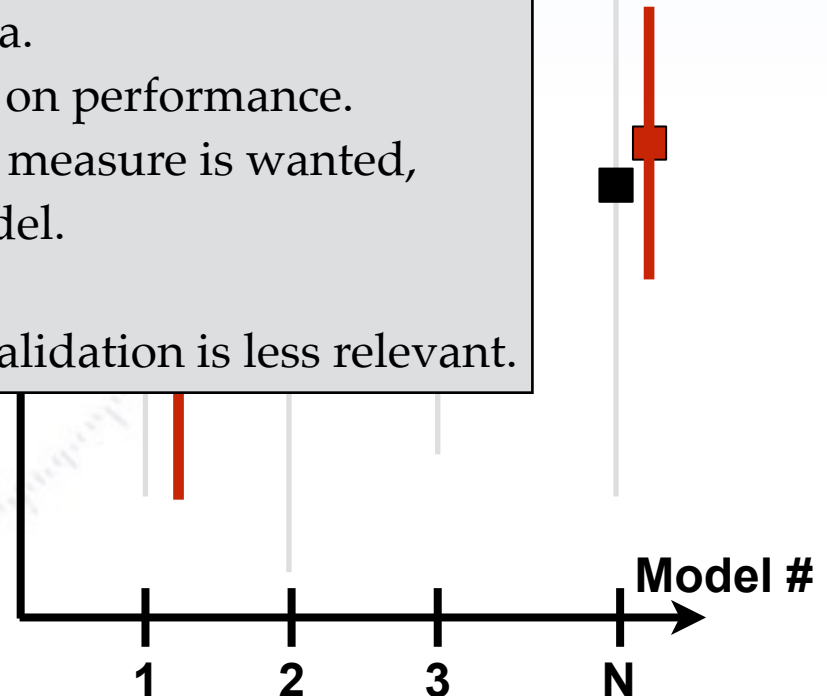
Then you do

a more accurate analysis with smaller uncertainties (by factor $1/\sqrt{5}$).

Now you conclude, that model #1 is the best... and that model #2 is worst!

Note that Cross Validation especially applies mostly to

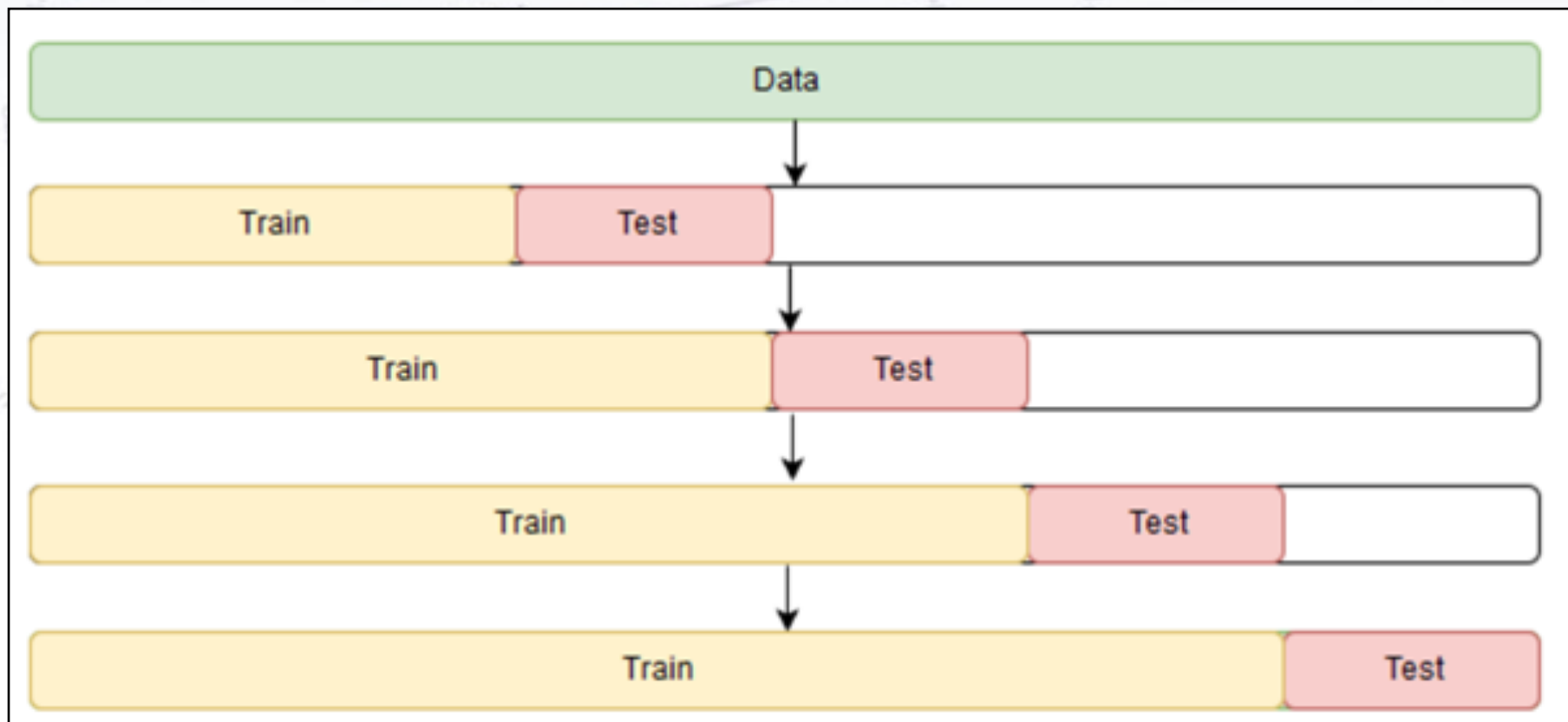
At very high statistics, Cross Validation is less relevant.



CV for time series

Special care has to be taken, when doing Cross Validation for time series, as one should ensure that **training is done only on data from the past, not the future!**

The figure below illustrates the principle. One should choose a certain data period, and put the test period immediately after, and then shift this setup.





Preprocessing Data

When data is imperfect

So far, we have looked at “perfect” data, i.e. data without any flaws in it. However, real world datasets are hardly ever “perfect”, but contains flaws that makes preprocessing imperative.

Effects may be (non-exhaustive list):

- NaN-values and "Non-values" (i.e. -9999)
- Wild outliers (i.e. values far outside the typical range)
- Shifts in distributions (i.e. part of data having a different mean/width/etc.)
- Mixture of types (i.e. numerical and text, from something humans filled out)

It is also important to consider, if entries are missing...

1. **Randomly** (in which case there should be no bias from omitting) or
2. **Following some pattern** (in which case there could be problems!).

In case of NaN values, we might simply decide to drop the variable column or entry row, requiring that all variables/entries have reasonable values.

Alternatively, we might insert “imputed” values instead, saving statistics.

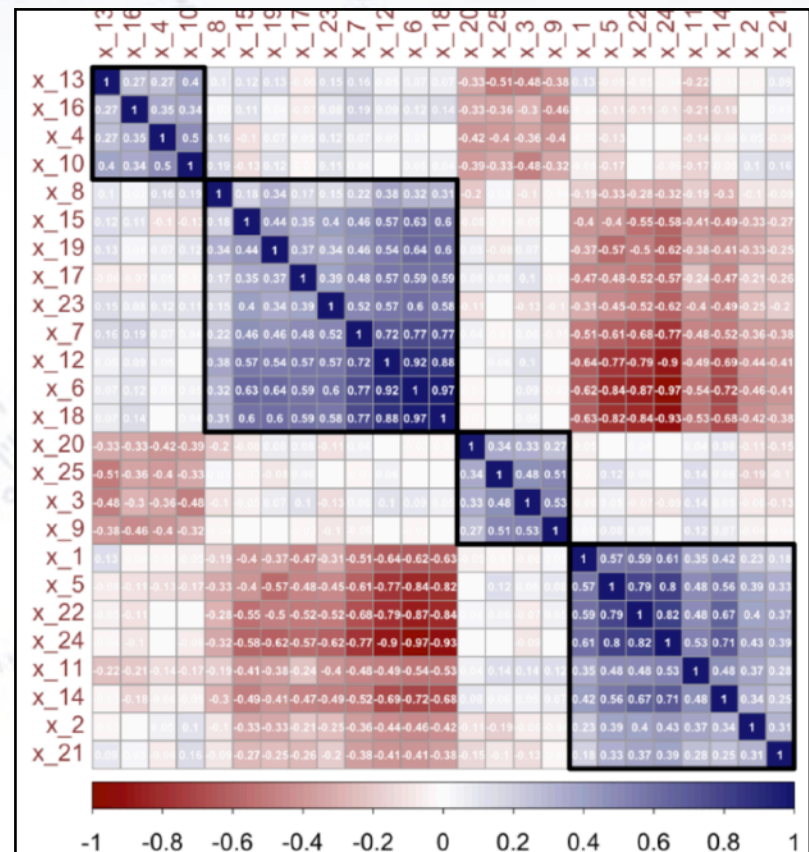
NaN-values tend to correlate

It is often seen, that several variables have the same source, and thus their NaN occurrence might be correlated with each other.

This can be tested by substituting 0's for numerical values and 1's for NaN values. By considering the correlation matrix of these substitute 0/1 values, one gets a pretty clear picture.

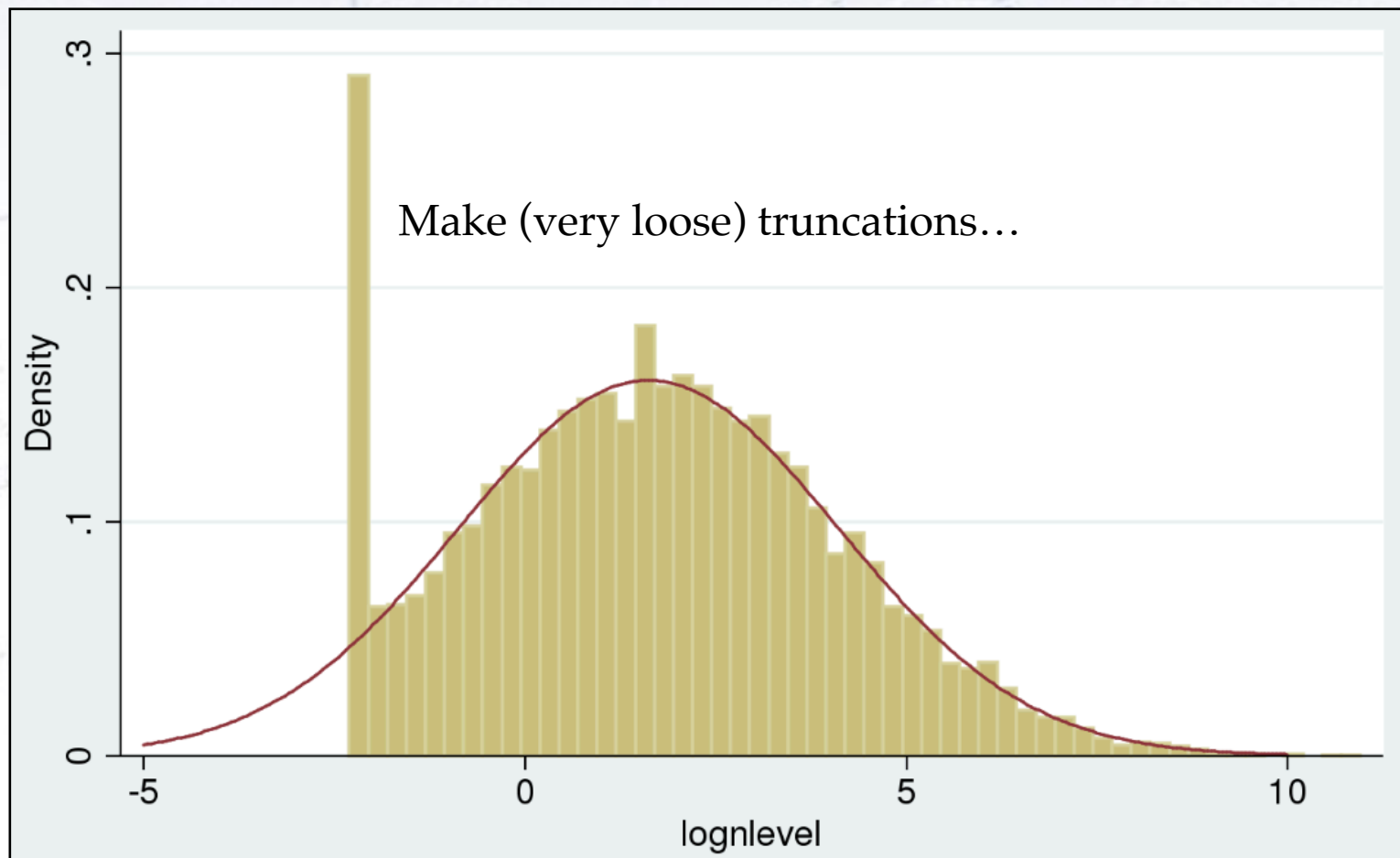
Typically, some entries are 100% correlated, as the source of these variables is shared.

Based on this information, one can better decide how to deal with these variables.



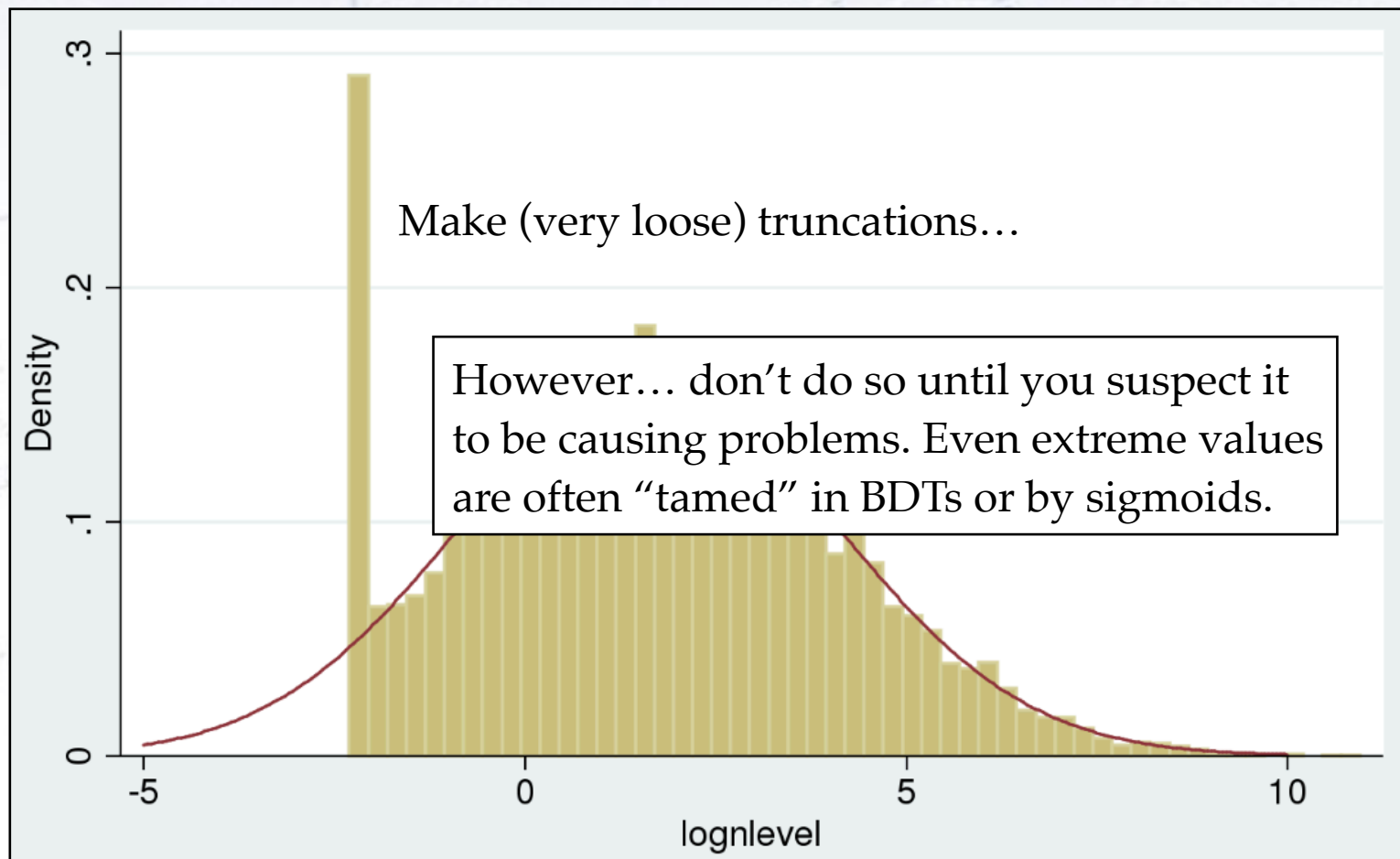
How to deal with outliers?

Sometimes, (a few?) entries take on extreme values, which ruin either the NN performance, or the transformation applied first (and then the performance). How to deal with that?



How to deal with outliers?

Sometimes, (a few?) entries take on extreme values, which ruin either the NN performance, or the transformation applied first (and then the performance). How to deal with that?



Conclusions

No matter what you plan to do with data, my first advice is always:

Print & Plot

This is your first assurance, that you even remotely know what the data contains, and your first guard against nasty surprises.

Also, working with others (from know-nothings to domain experts) you will be required to show the input, and assuring that it is valid and makes sense.

Remember to do so in all your ML work...



Feature Ranking

ML as a science

While Machine Learning is fantastic, it is a black box, and thus unsatisfactory both regarding understanding it, and as a science in itself.

"As a data scientist, I can predict what is likely to happen, but I cannot explain why it is going to happen. I can predict when someone is likely to attrite, or respond to a promotion, or commit fraud, or pick the pink button over the blue button, but I cannot tell you why that's going to happen. And I believe that the inability to explain why something is going to happen is why I struggle to call 'data science' a science."

[Bill Schmarzo, Author of "Big Data: Understanding How Data Powers Big Business"]

However, there are ways of "opening the box", and the most common one is to find out, which input features are important and which are not.

For more info, see:

Interpretable Machine Learning
A Guide for Making Black Box Models Explainable.

Christoph Molnar

Input Feature Ranking

It is of course useful to know, which of your input features/variables are useful, and which are not. Thus a **ranking of the features** is desired.

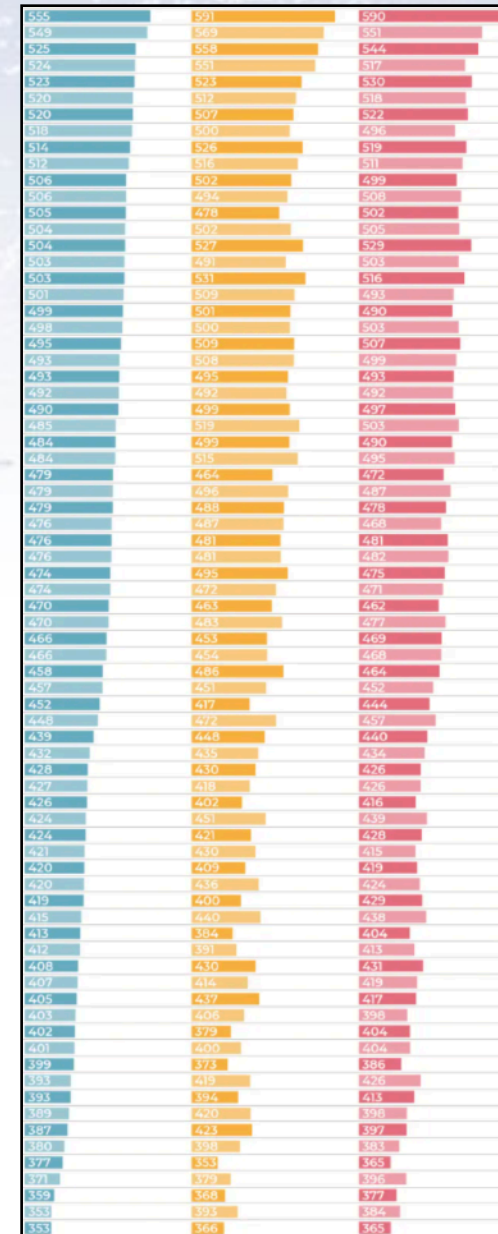
This is not only possible, but actually a general nice feature of ML and feature ranking:

It works as an automation of the detective work behind finding relations.

In principle, one could obtain a variables ranking by testing **all combinations** of variables. But that is not feasible in most situation ($N \text{ features} > 5-7$)...

Most algorithms have a build-in input feature ranking, which is based on various approximations.

A very simple idea (next slide) that works quite well is **“permutation importance”**.



Input Feature Ranking

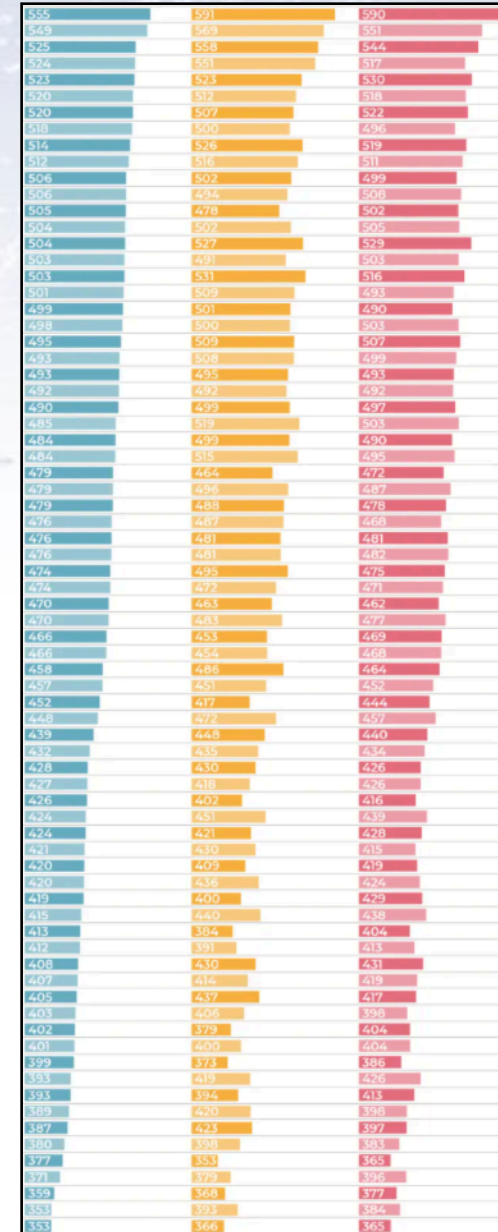
There are many different ways of ranking input features. Three (simple) implementations into XGBoost are:

- **Weight.** The number of times a feature is used to split the data across all trees.
- **Cover.** The number of times a feature is used to split the data across all trees weighted by the number of training data points that go through those splits.
- **Gain.** The average training loss reduction gained when using a feature for splitting.

These have different pro's and con's.

Personally, I much like the idea of...

“permutation invariance”



A nautical chart background with a compass rose. The compass rose shows magnetic variation with a label "VAR 10° 15' W" and a plus sign. The word "MAGNETIC" is also visible. The chart includes various navigational lines and labels like "ICE BITTER END YACHT CLUB".

Permutation Importance

Permutation Importance

One of the most used methods is “permutation importance” (below quoting Christoph M.: ["Interpretable ML" chapter 5.5](#)). The idea is really simple:

We measure the importance of a feature **by calculating the increase in the model's loss function after permuting the feature.**

- A feature is **“important”** if shuffling its values increases the model error, because in this case the model relied on the feature for the prediction.
- A feature is **“unimportant”** if shuffling its values leaves the model error unchanged, because the model thus ignored the feature for the prediction.


Permutation Importance

One of the most used methods is “permutation importance” (below quoting Christoph M.: "Interpretable ML" chapter 5.5). The idea is really simple:

We measure the importance of a feature **by calculating the increase in the model's loss function after permuting the feature.**

- A feature is **“important”** if shuffling its values increases the model error, because in this case the model relied on the feature for the prediction.
- A feature is **“unimportant”** if shuffling its values leaves the model error unchanged, because the model thus ignored the feature for the prediction.

Height at age 20 (cm)	Height at age 10 (cm)	...	Socks owned at age 10
182	155	...	20
175	147	...	10
...
156	142	...	8
153	130	...	24



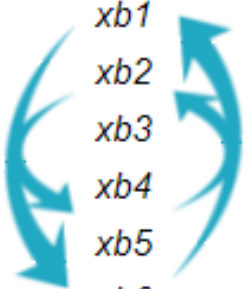
Permutation Importance

Input: Trained model f , feature matrix X , target vector y , loss function $L(y, f)$.

[Fisher, Rudin, and Dominici (2018)]

- Estimate the original model error $e_{\text{orig}} = L(y, f(X))$
- For each feature $j = 1, \dots, p$ do:
 - Generate feature matrix $X_{\text{perm},j}$ by permuting feature j in the data X .
This breaks the association between feature j and true outcome y .
 - Estimate error $e_{\text{perm},j} = L(Y, f(X_{\text{perm},j}))$ based on the predictions of $X_{\text{perm},j}$.
 - Calculate permutation feature importance $FI_j = e_{\text{perm},j} / e_{\text{orig}}$ (or $e_{\text{perm},j} - e_{\text{orig}}$).
- Sort features by descending FI_j .

X_A	X_B	X_C	Y
xa1	xb1	xc1	y1
xa2	xb2	xc2	y2
xa3	xb3	xc3	y3
xa4	xb4	xc4	y4
xa5	xb5	xc5	y5
xa6	xb6	xc6	y6



Note: Permutation Importance calculations are computationally fast. (why?)

Feature importance with Neural Networks (Towards Data Science)

The background is a nautical chart of the Bitter End Yacht Club area. A large circular compass rose is overlaid on the chart, showing magnetic variation lines and degree markings. The text "MAGNETIC" is visible on the compass rose. A specific variation is noted as "VAR 10° 15' W". The text "BITTER END YACHT CLUB" is visible on the right side of the chart. The title "SHAP Values" is centered over the compass rose in a large, bold, black font.

SHAP Values

The background is a faded nautical chart of the Bitter End Yacht Club area. It features a compass rose with concentric circles representing magnetic variation. A line points to a specific variation of 10° 15' W. The text 'MAGNETIC' is visible near the top of the compass rose. The words 'BITTER END YACHT CLUB' are printed in the upper right quadrant of the chart.

SHAP Values

SHAP is a technique for deconstructing a machine learning model's predictions into a sum of contributions from each of its input variables.

The result is an evaluation of the input variables for each single case!

Shapley values

Shapley values is a concept from **corporate game theory**, where they are used to provide a possible answer to the question:

“How important is each player to the overall cooperation, and what payoff can each player reasonably expect?”

The Shapley values are considered “fair”, as they are the only distribution with the following properties:

- **Efficiency:** Sum of Shapley values of all agents equals value of grand coalition.
- **Linearity:** If two coalition games described by v and w are combined, then the distributed gains should correspond to the gains derived from the sum of v and w .
- **Null player:** The Shapley value of a null player is zero.
- **Stand alone test:** If v is sub/super additive, then $\phi_i(v) \leq / \geq v(\{i\})$, where ϕ is the Shapley value for agent i , and v is the worth function (of a coalition). Also called “Monotonicity”: A consistently more contributing feature much a get higher v .
- **Anonymity:** Labelling of agents doesn't play a role in assignment of their gains.
- **Marginalism:** Function uses only marginal contributions of player i as arguments.

From such values, one can determine which variables contribute to a final result. And summing the values, one can get an overall idea of which variables are important.

Shapley value calculation

Consider a set N (of n players) and a (characteristic or worth) function v that maps any subset of players to real numbers:

$$v : 2^N \rightarrow \mathbb{R}, \quad v(\emptyset) = 0$$

If S is a coalition of players, then $v(S)$ yields the total expected sum of payoffs the members of S can obtain by cooperation.

The Shapley values are calculated as:

$$\varphi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(n - |S| - 1)!}{n!} [v(S \cup \{i\}) - v(S)]$$

To formula can be understood, if we imagine a coalition being formed one actor at a time, with each actor demanding their contribution $v(S \cup \{i\}) - v(S)$ as a fair compensation, and then for each actor take the average of this contribution over the possible different permutations in which the coalition can be formed.

Shapley value calculation

Consider a set \mathbf{N} (of n players) and a (characteristic or worth) function v that maps any subset of players to real numbers:

$$v : 2^N \rightarrow \mathbb{R}, \quad v(\emptyset) = 0$$

If S is a coalition of players, then $v(S)$ yields the total expected sum of payoffs the members of S can obtain by cooperation.

The Shapley values can also be calculated as:

$$\varphi_i(v) = \frac{1}{n!} \sum_R [v(P_i^R \cup \{i\}) - v(P_i^R)]$$

where the sum ranges over all $n!$ orders R of the players and P_i^R is the set of players in \mathbf{N} which precede i in the order R . This has the interpretation:

$$\varphi_i(v) = \frac{1}{N_{\text{players}}} \sum_{C \setminus i} \frac{\text{marginal contribution of } i \text{ to coalition } C}{\text{number of coalitions excluding } i \text{ of this size}}$$

Shapley value calculation example

Example 1:

Two friends (F1 and F2) make a business: **Payoff 600\$** (i.e. $v(F1, F2) = 600$).

If F1 or F2 did not participate, payoff would be 0\$ (i.e. $v(F1) = v(F2) = 0$).

Result: F1 and F2 each gets 300\$.

Example 2:

Two friends (F1 and F2) make a business: **Payoff 600\$** (i.e. $v(F1, F2) = 600$).

If F1 did not participate, payoff would be 0\$ (i.e. $v(F1) = 0$).

If F2 did not participate, payoff would be 200\$ (i.e. $v(F2) = 200$).

Cases:

F1 1st gets 0\$. With F2 also they get 600\$. F2's marginal contribution: 600\$.

F2 1st gets 200\$. With F1 also they get 600\$. F1's marginal contribution: 400\$.

Result:

F1 should have: $0.5 \times 0\$ + 0.5 \times 400\$ = 200\$$

F2 should have: $0.5 \times 200\$ + 0.5 \times 600\$ = 400\$$

Note that the number of cases quickly expands!

SHAP Values

A great approximation was developed by Scott Lundberg with **SHAP values**:

SHAP (SHapley Additive exPlanations):

<https://github.com/slundberg/shap>

This algorithm provides - **for each entry** - a ranking of the input variables, i.e. a sort of explanation for the result.

One can also sum of the SHAP values over all entries, and then get the overall ranking of feature variables. **They are based on Shapley values.**

Note: SHAP values are computationally “heavy”.

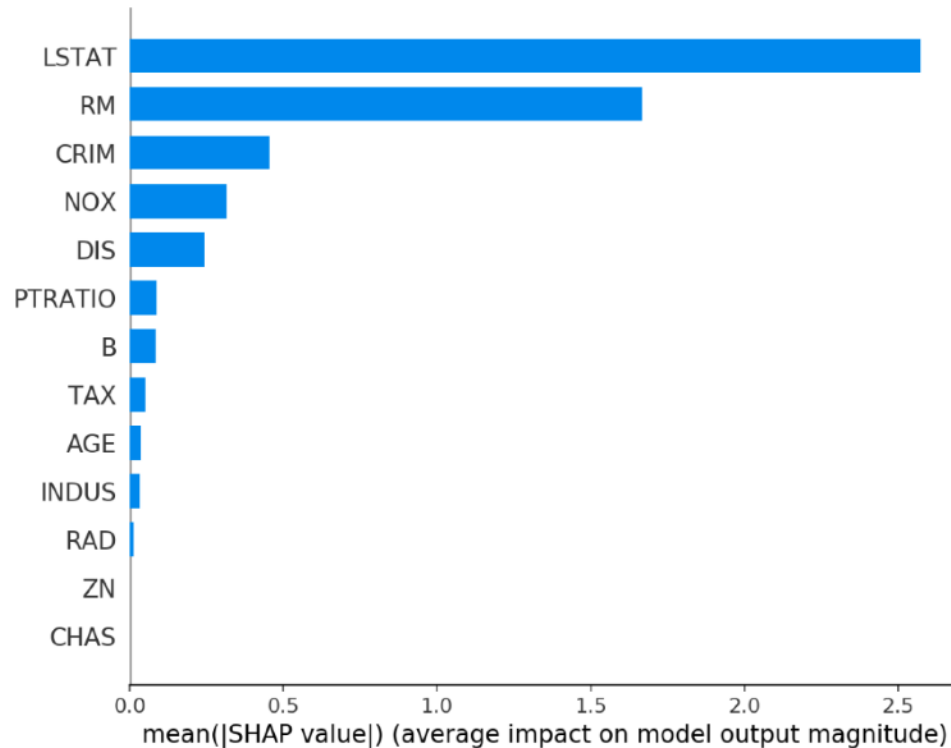
Note2: Lately, this seems to have been improved in 2023!

Input Feature Ranking

Here is an example from SHAP's [github](#) site.

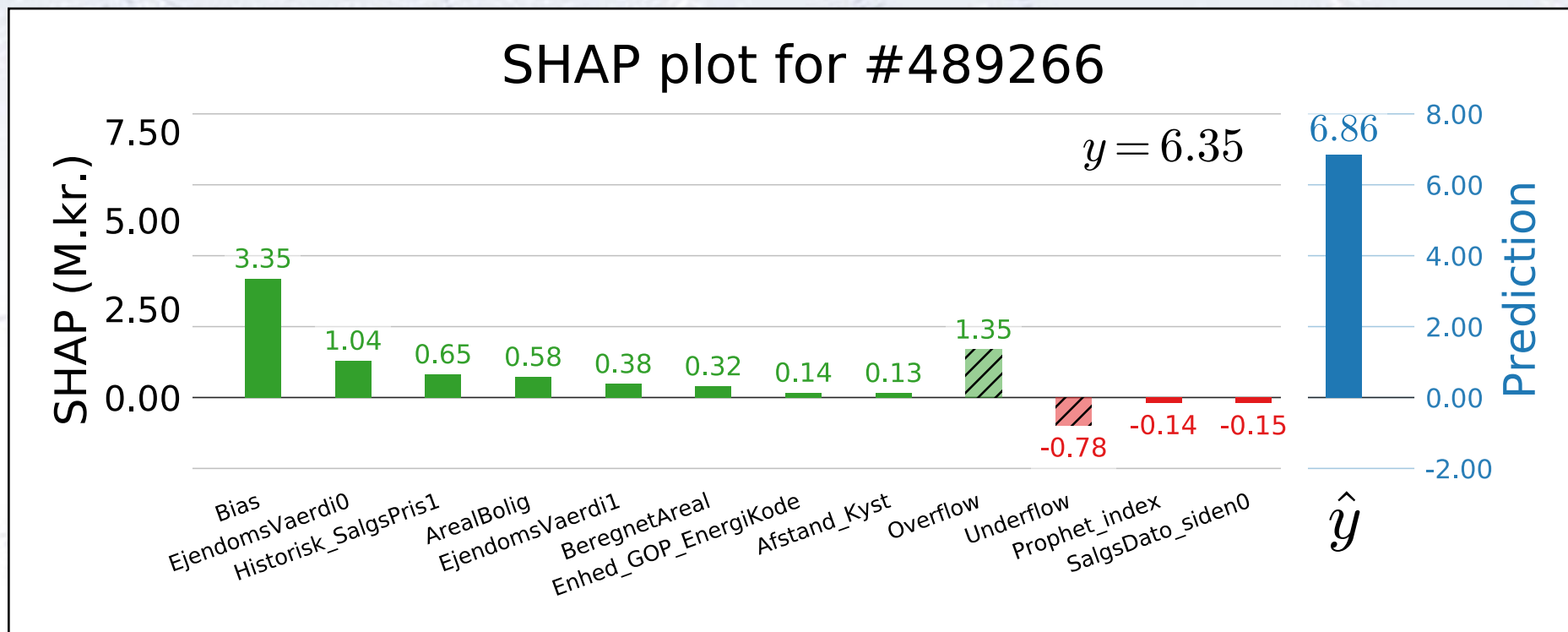
Clearly, LSTAT and RM are the best variables (whatever they are!).

```
shap.summary_plot(shap_values, X, plot_type="bar")
```



Individuel estimates

Shapley-values also gives the possibility to see the reason behind **individueel estimates**. Below is an example, illustrating this point.



Above is shown which factors that influences the final estimate of the sales price (and how much). The estimate is the sum of the contributions (here 6.86 MKr.).

This is a fantastic tool to get insight into the ML workings!!!

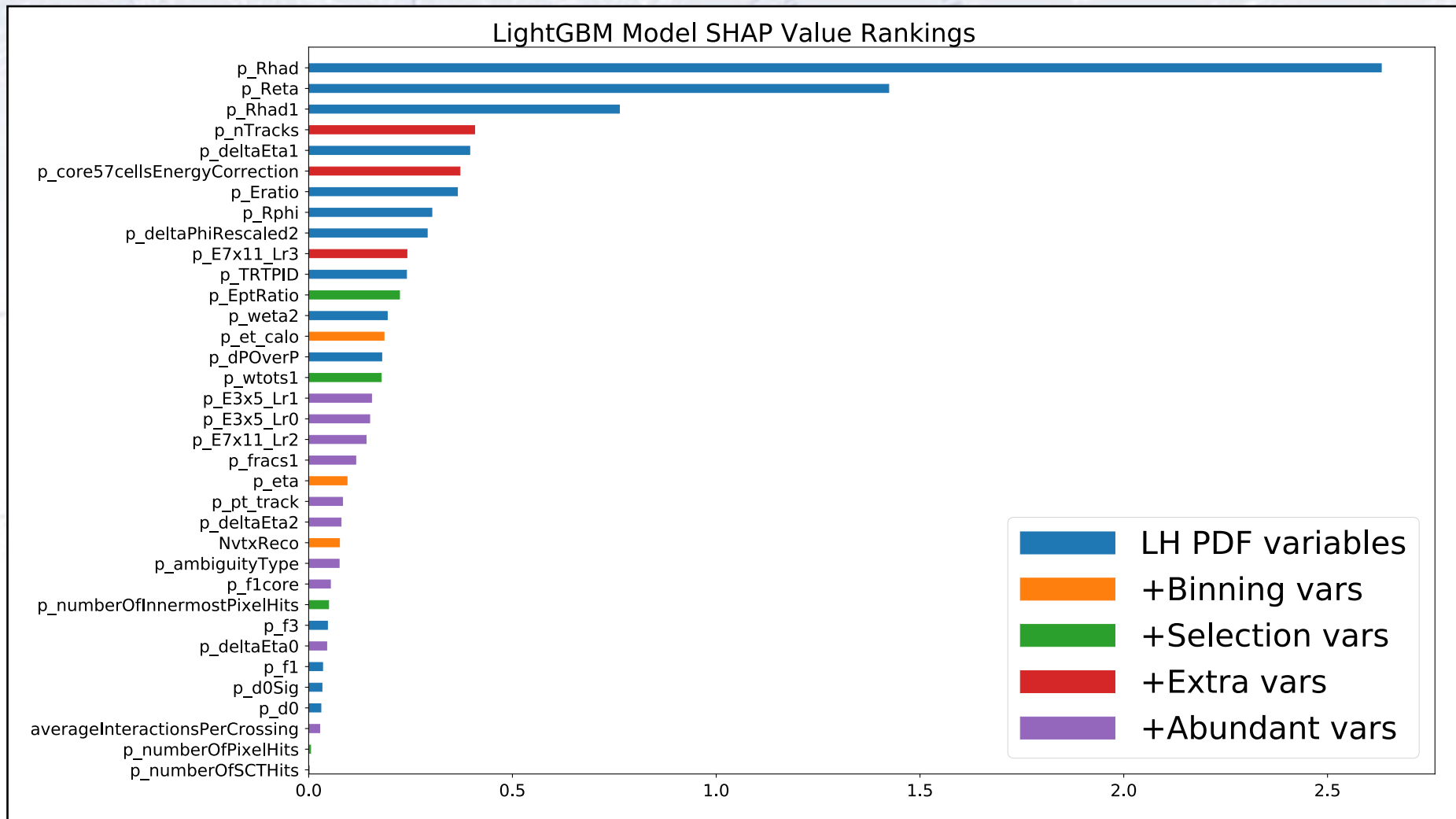


The background image is a nautical chart. It features several compass roses with concentric circles representing magnetic variation. One prominent rose is labeled 'MAGNETIC' and 'VAR 10° 15' W'. Another rose to the right is labeled '152 BITTER END YACHT CLUB'. The chart also shows various navigational lines, including a dashed line labeled '20' and a solid line labeled '30'. The text 'Example of usage' is overlaid in the center.

Example of usage

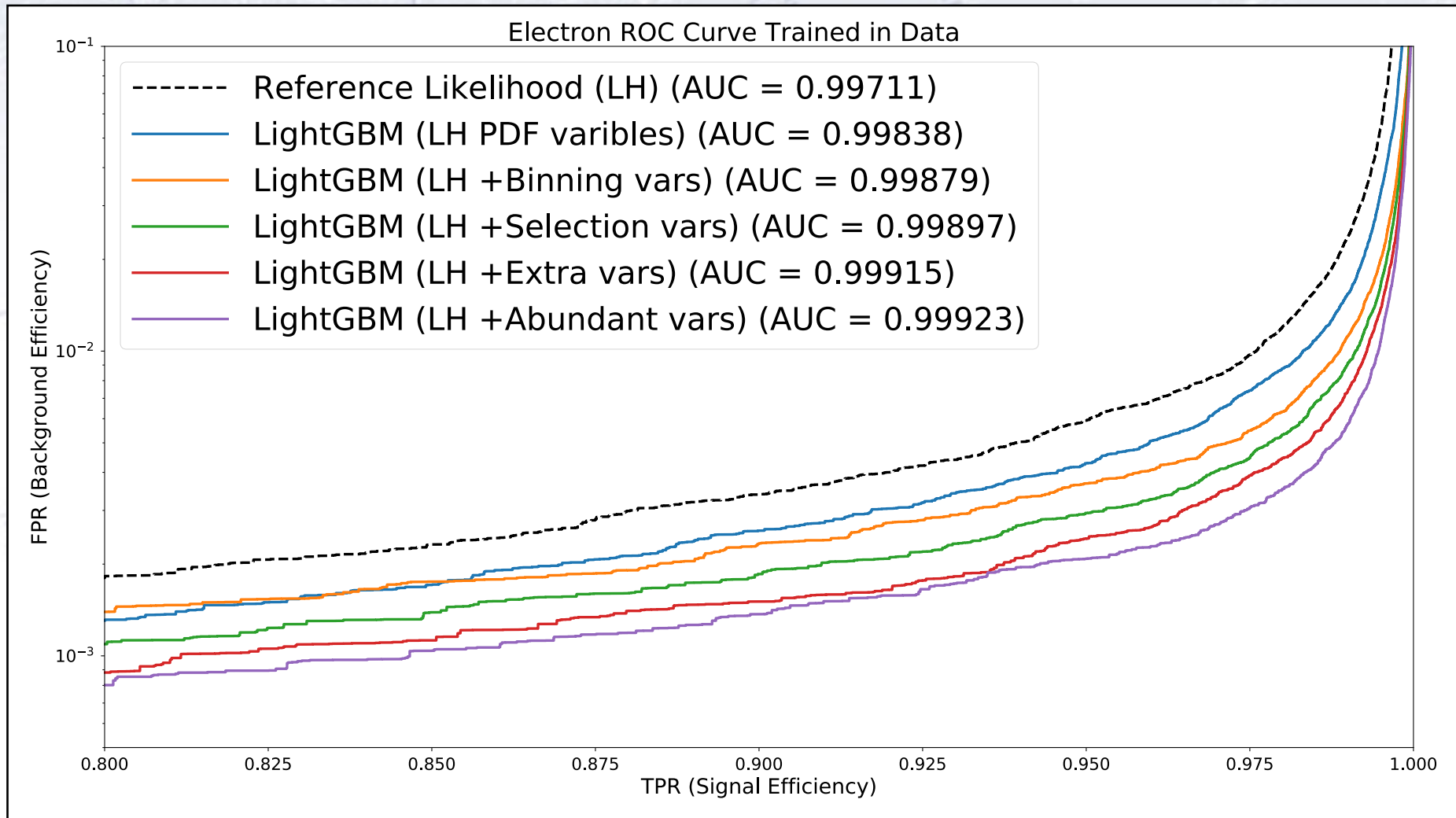
Input Feature Ranking

Here is an example from particle physics. The blue variables were “known”, but with SHAP we discovered three new quite good variables in data.



Input Feature Ranking

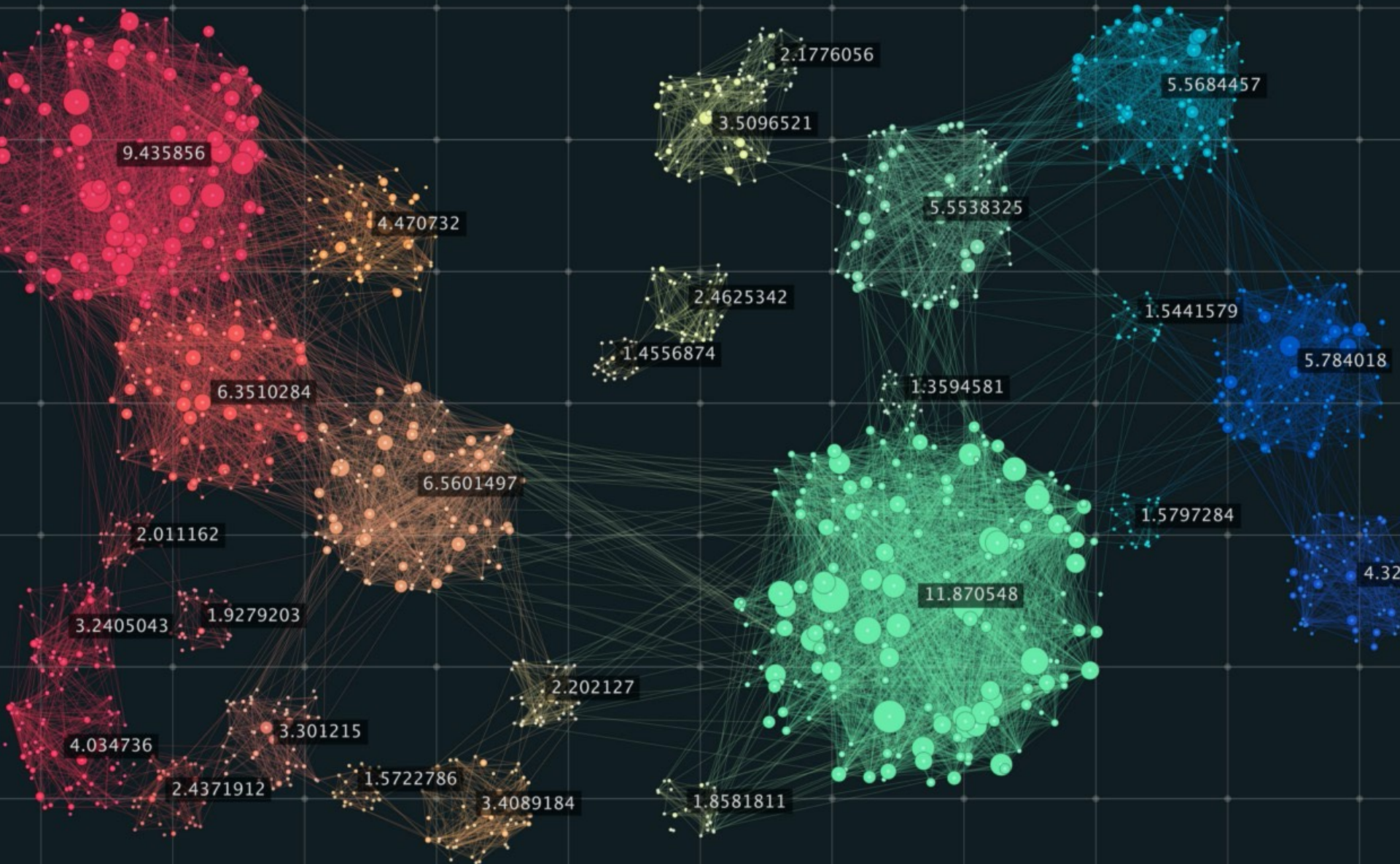
We could of course just add all variables, but want to stay simple, and training the models, we see that the three extra variables gives most of gain.





Unsupervised Learning: Clustering

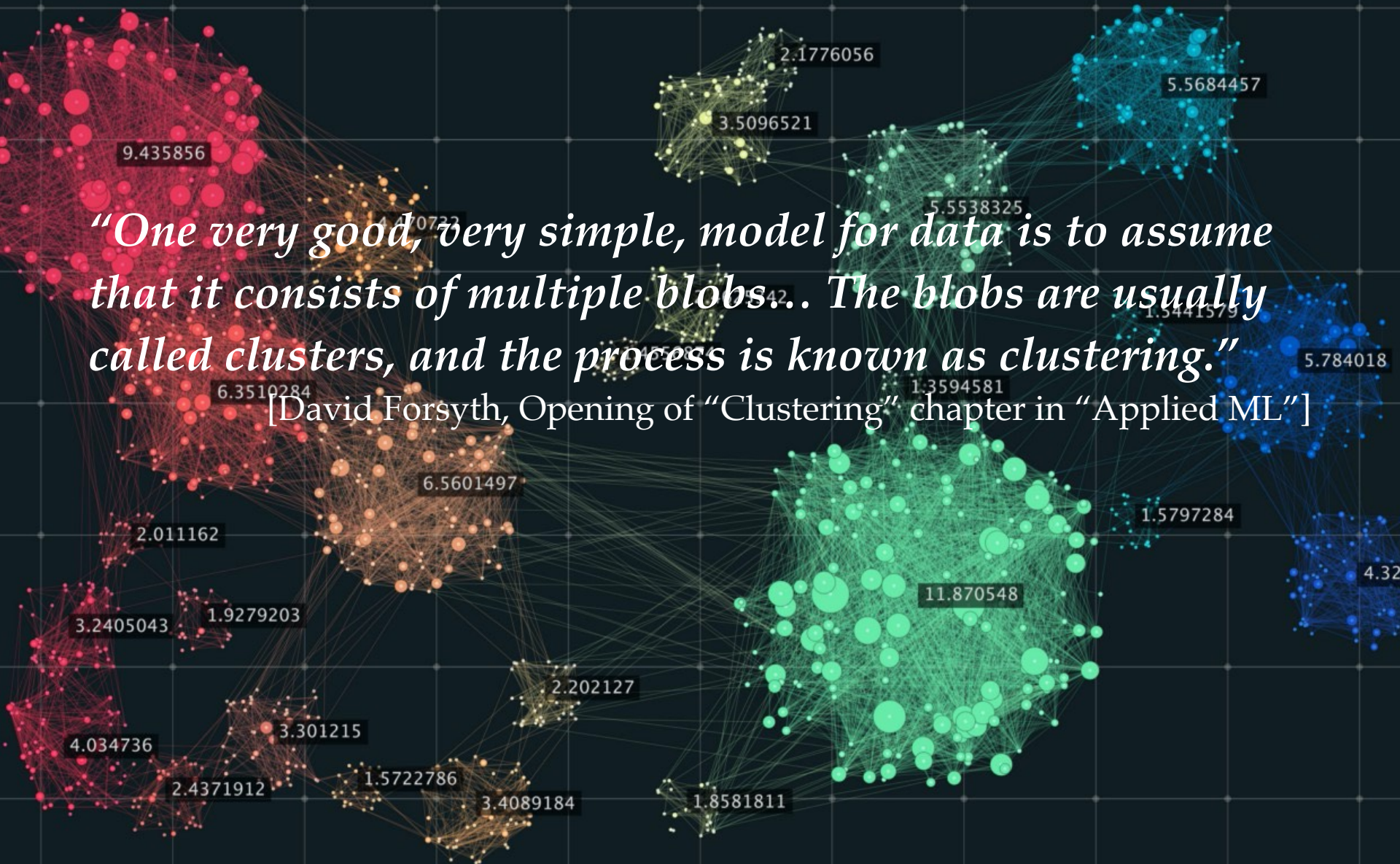
Clustering... is an art!



Clustering... is an art!

"One very good, very simple, model for data is to assume that it consists of multiple blobs... The blobs are usually called clusters, and the process is known as clustering."

[David Forsyth, Opening of "Clustering" chapter in "Applied ML"]



Evaluating clustering

Evaluation of identified clusters is subjective and may require a domain expert, although many clustering-specific quantitative measures do exist.

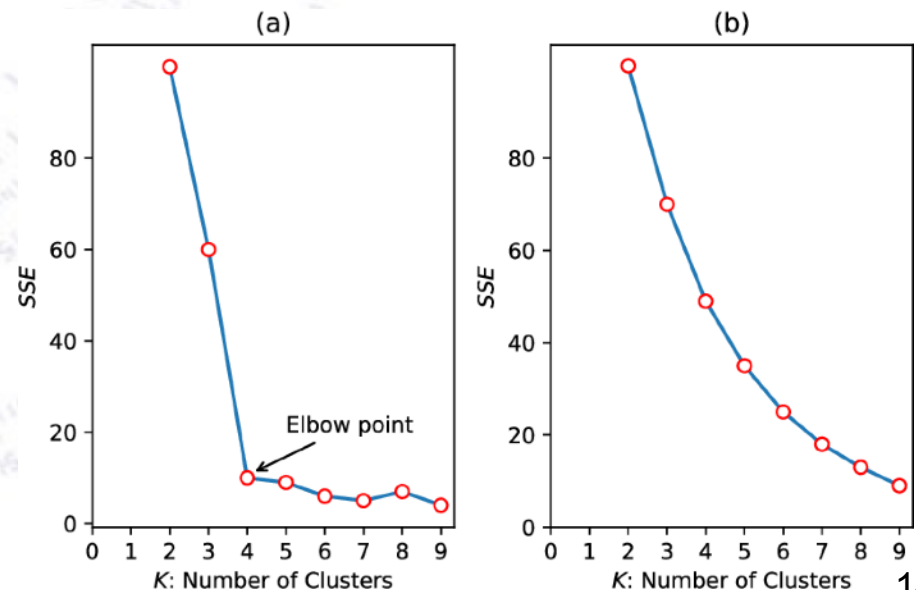
Typically, clustering algorithms are compared on synthetic datasets with pre-defined clusters, which an algorithm is expected to discover.

“Clustering is an unsupervised learning technique, so it is hard to evaluate the quality of the output of any given method.”

[Page 534, Machine Learning: A Probabilistic Perspective, 2012.]

One of the simple principles is that of the “Elbow Method”.

If the loss function shows an “elbow” (sudden stop in rate of improvement), then that probably reflects some structure in the data.



Evaluating clustering

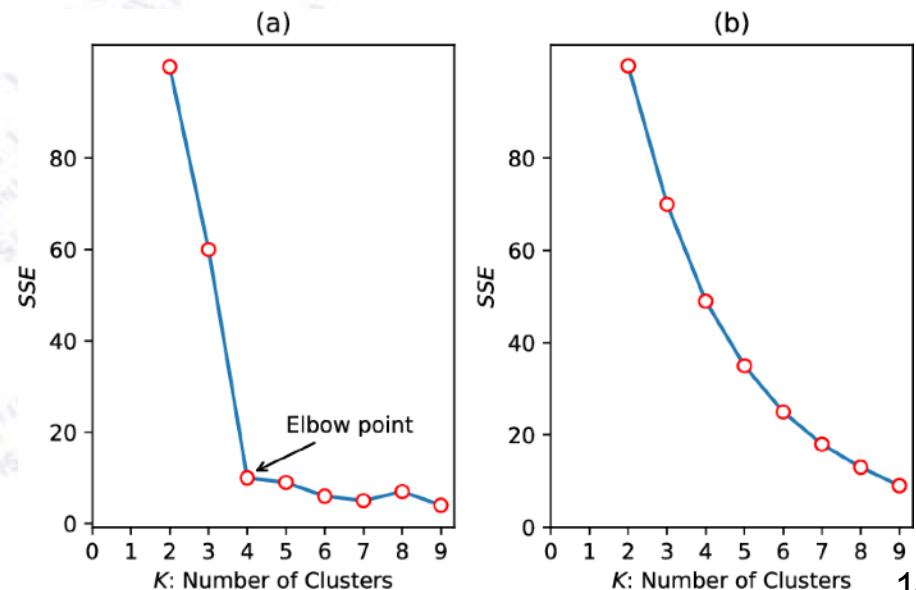
Evaluation of identified clusters is subjective and may require a domain expert, although many clustering-specific quantitative measures do exist.

Typically, clustering algorithms are compared on synthetic datasets with pre-defined clusters, which an algorithm is expected to discover.

One way of visually evaluating a clustering algorithm is to combine it with a dimensionality reduction, though one then observes the combined performance of the two.

One of the simple principles is that of the “Elbow Method”.

If the loss function shows an “elbow” (sudden stop in rate of improvement), then that probably reflects some structure in the data.

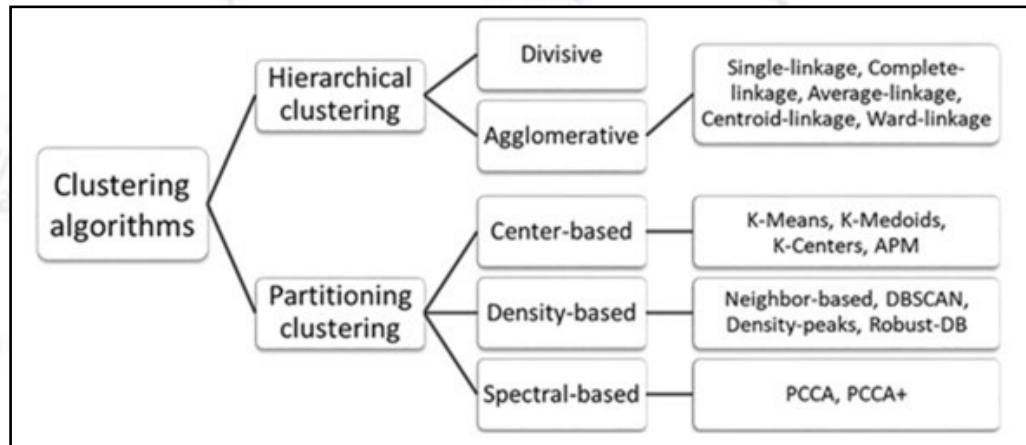


Clustering algorithms

Clustering is an “old field” and many philosophies (and algorithms) have been developed. They can roughly be reduced to two approaches:

- **Hierarchical clustering** algorithms are based on recursively either merging smaller clusters in to larger ones or dividing larger clusters to smaller ones.
- **Partitioning clustering** algorithms generate various partitions and then iteratively place each instance best in one of k mutually exclusive clusters.

Hierarchical clustering does not require any input parameters, while partitioning clustering algorithms require the number of clusters to start running. Hierarchical clustering returns a much more meaningful and subjective division of clusters but partitioning clustering results in exactly k clusters.

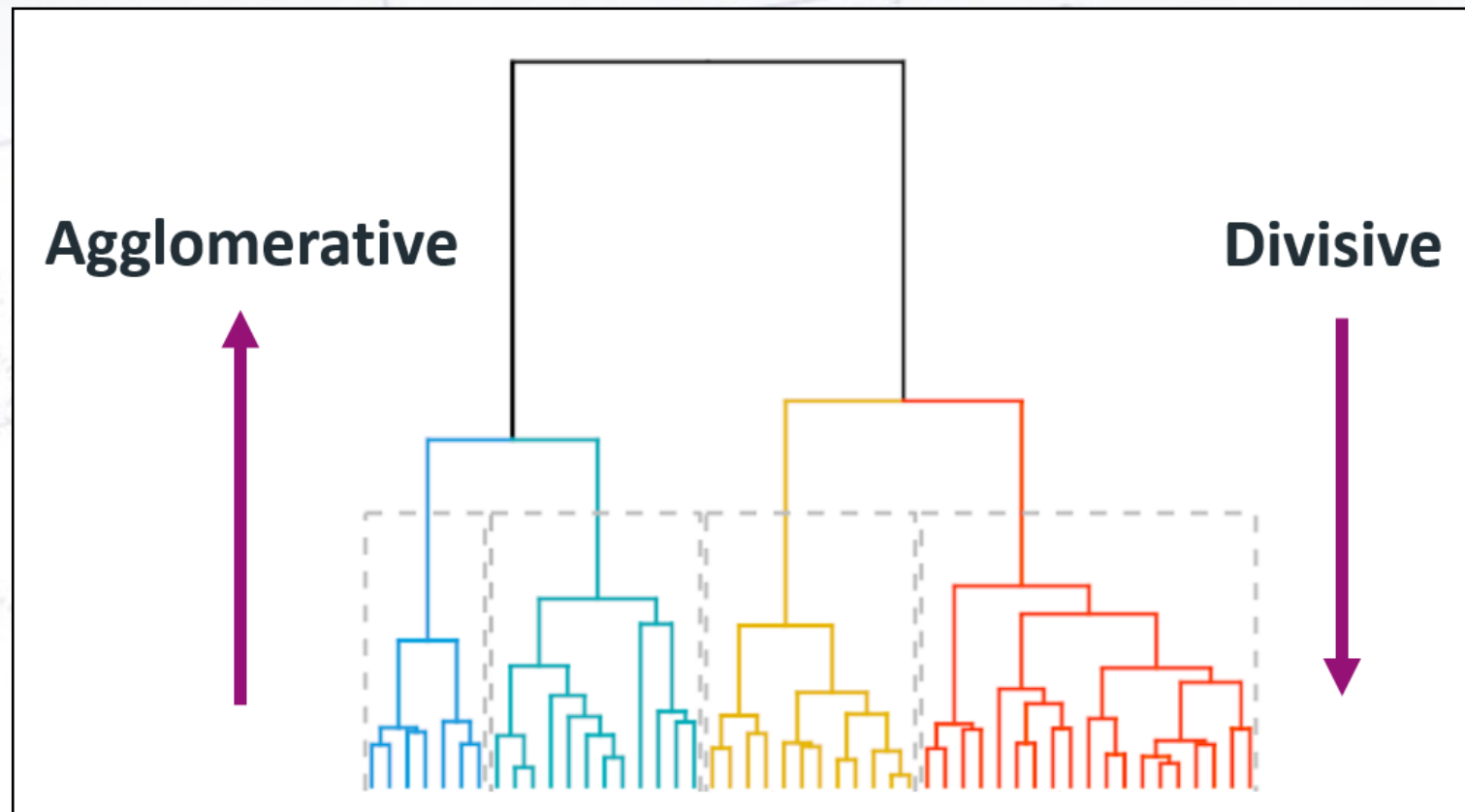


Hierarchical clustering algorithms

Hierarchical clustering algorithms can be further divided:

- **Agglomerative:** Merge smaller clusters into larger ones
- **Divisive:** Divide larger clusters into smaller ones.

The only requirement is a **similarity measure** to decide distance between cases.



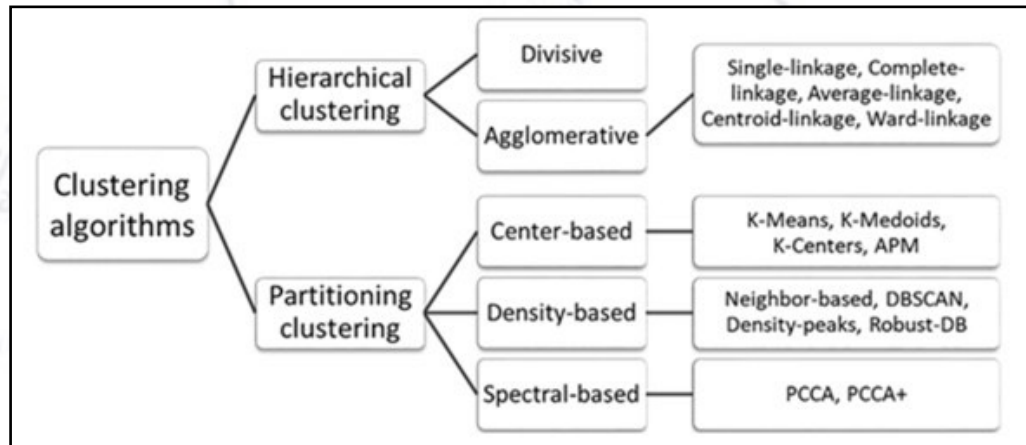
Partitioning clustering algorithms

Partitioning clustering algorithms can (also) be further divided:

- **Center-based:** Build clusters around (random?) centers (**k-Means**).
- **Density-based:** Build clusters around (high) densities (**DBSCAN**).
- **Spectral-based:** Uses eigenvalues of the similarity matrix to perform dimensionality reduction before clustering (**PCCA+**).

“k-Means clustering is the “go-to” clustering algorithm. You should see it as a basic recipe from which many algorithms can be concocted.”

[David Forsyth, “Applied ML” chapter 8.2.6]

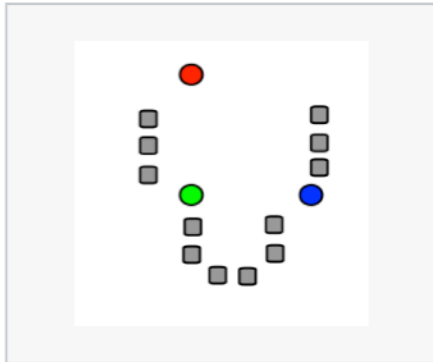


k-Means clustering

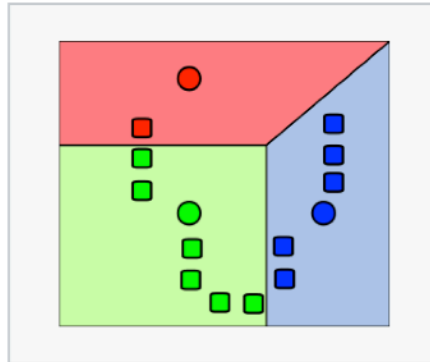
The recipe is to iterate the below points, until movements are “small”:

- Allocate each data point to the closest cluster center
- Re-estimate cluster centers from their data points.

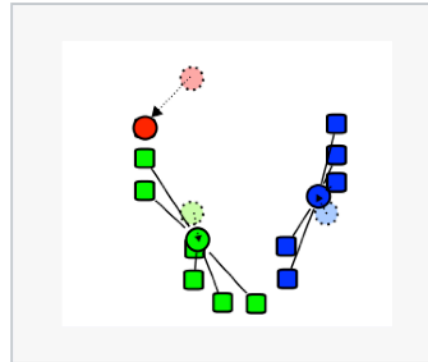
Demonstration of the standard algorithm



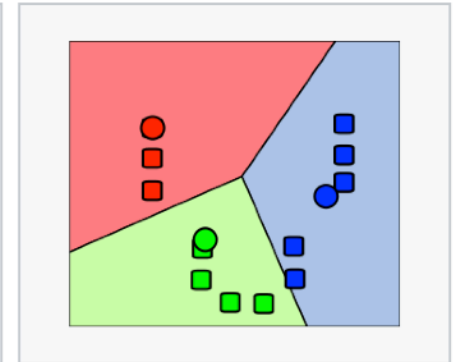
1. k initial "means" (in this case $k=3$) are randomly generated within the data domain (shown in color).



2. k clusters are created by associating every observation with the nearest mean. The partitions here represent the **Voronoi diagram** generated by the means.



3. The **centroid** of each of the k clusters becomes the new mean.



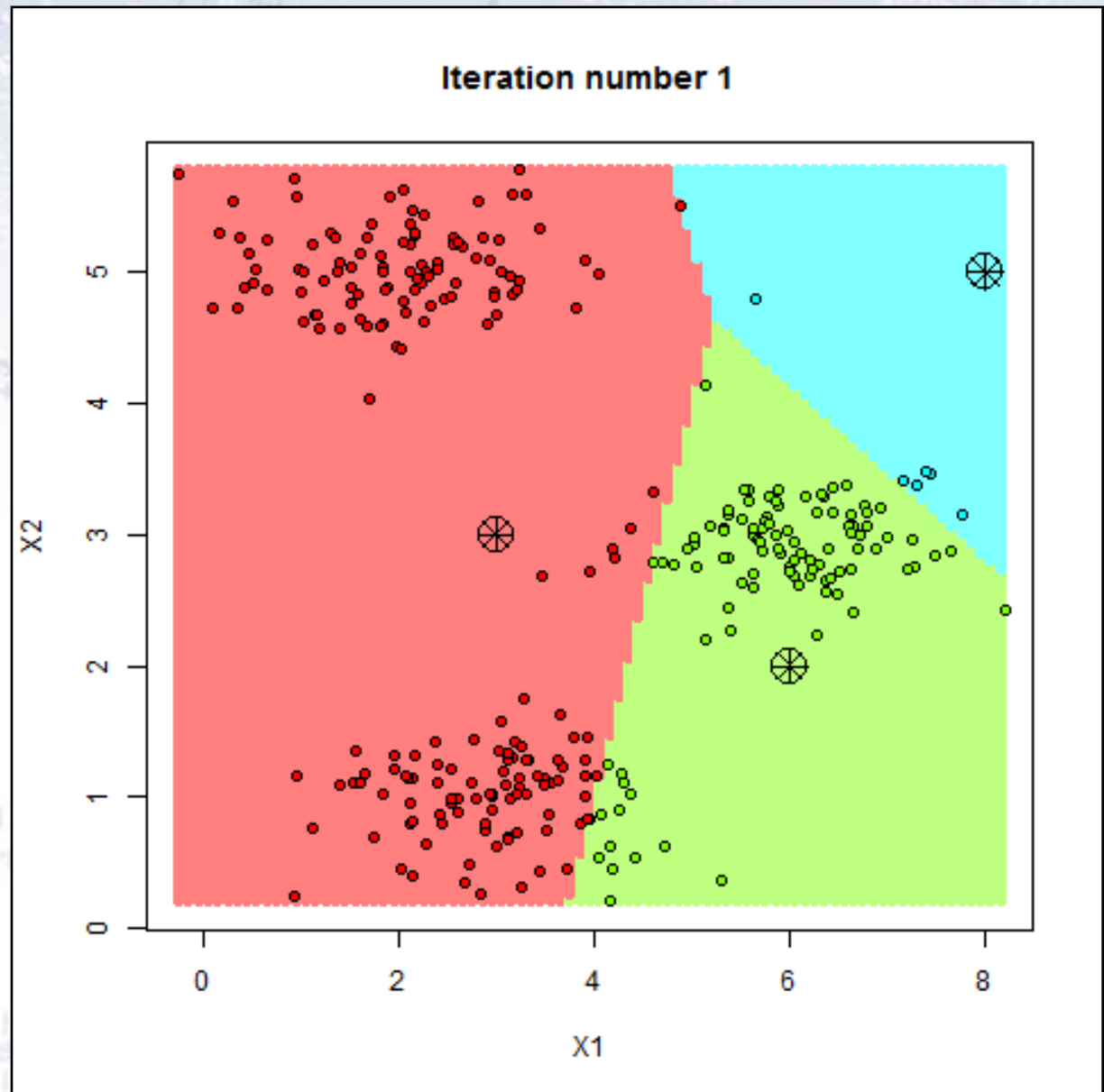
4. Steps 2 and 3 are repeated until convergence has been reached.

There are many variations, improvements, etc. that refine this algorithm. Most notably are the k-means++ (better initial points) and k-medoids methods.

k-Means clustering

The recipe:

- Allocate each data point to the closest cluster center
- Re-estimate cluster centers from their data points.



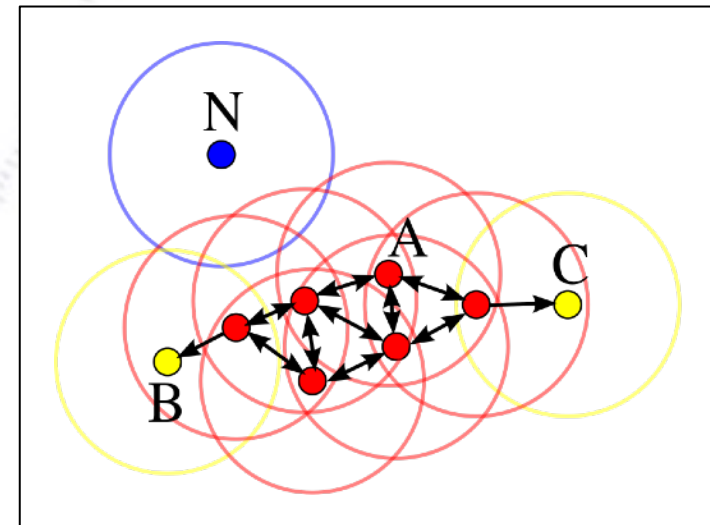
DBSCAN algorithm

DBSCAN classifies points as core points, reachable points and outliers:

- A point p is a **core point** if at least minPts points are within distance ϵ of it.
- A point q is **directly reachable** from p if point q is within distance ϵ from core point p . Points are only said to be directly reachable from core points.
- A point q is **reachable** from p if there is a path p_1, \dots, p_n with $p_1 = p$ and $p_n = q$, where each p_{i+1} is directly reachable from p_i . Note that this implies that the initial point and all points on the path must be core points, with the possible exception of q .
- All points not reachable from any other point are outliers or noise points.

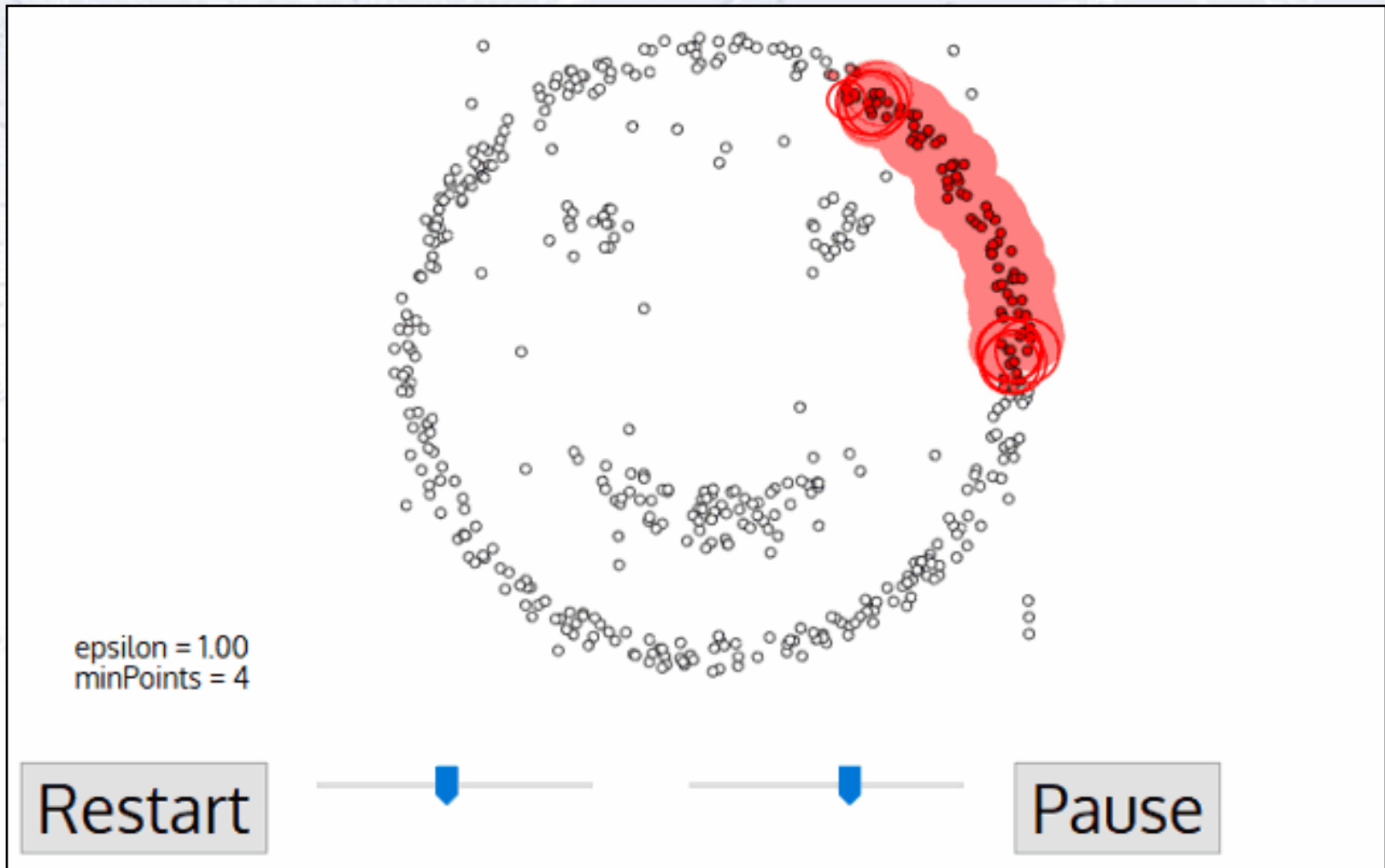
DBSCAN has two parameters: minPts and ϵ .

If p is a core point, then it forms a cluster together with all points (core or non-core) that are reachable from it. Each cluster contains at least one core point; non-core points can be part of a cluster, but they form its “edge”, since they cannot be used to reach more points.



DBSCAN algorithm

As can be seen, DBSCAN is a rather generic algorithm, capable of handling a large variety of data.

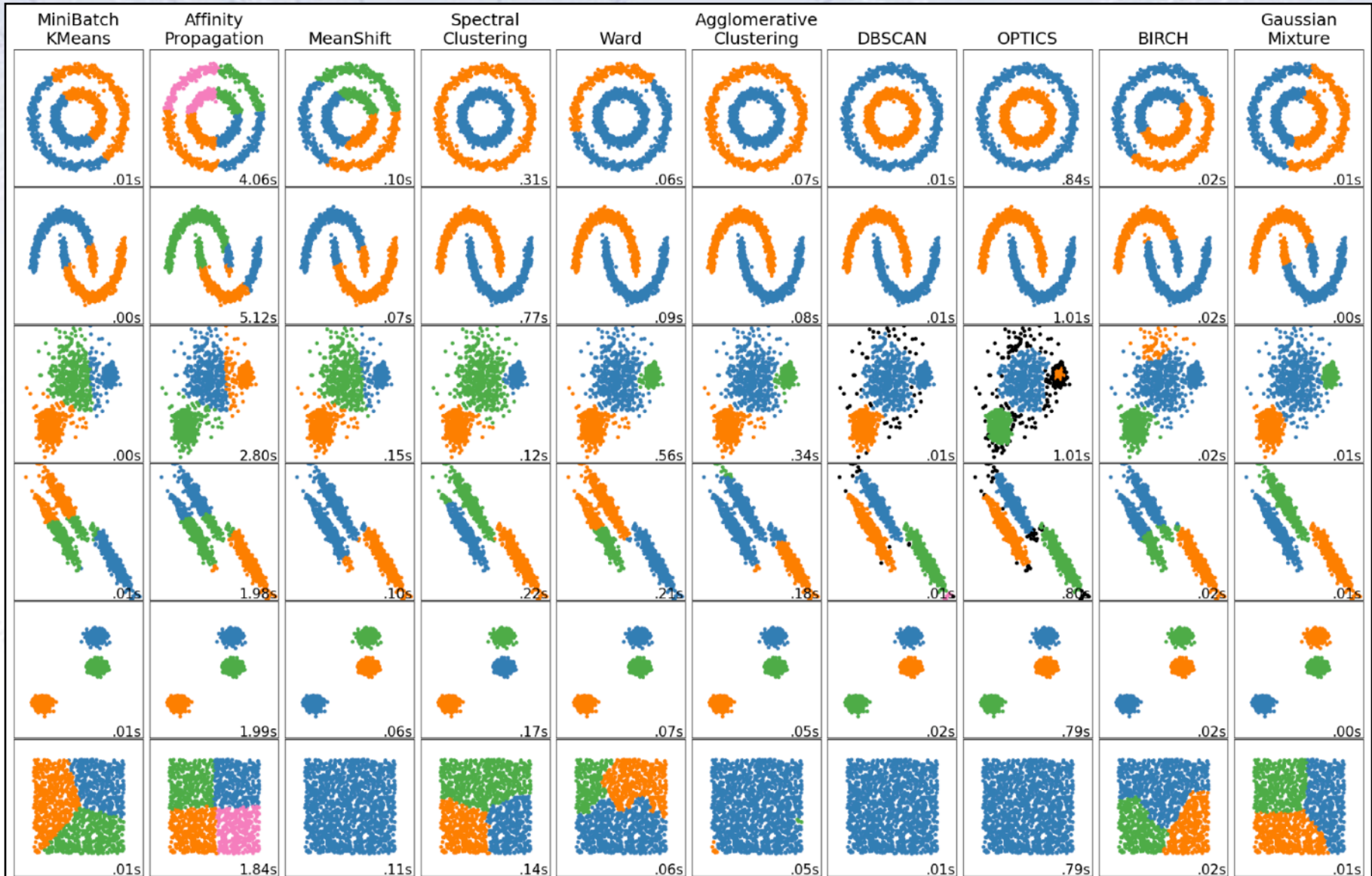


Clustering algorithms in scikit-learn

Scikit-Learn has a rather good selection of clustering algorithms:

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large <code>n_samples</code> , medium <code>n_clusters</code> with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters, inductive	Distances between points
Affinity propagation	damping, sample preference	Not scalable with <code>n_samples</code>	Many clusters, uneven cluster size, non-flat geometry, inductive	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with <code>n_samples</code>	Many clusters, uneven cluster size, non-flat geometry, inductive	Distances between points
Spectral clustering	number of clusters	Medium <code>n_samples</code> , small <code>n_clusters</code>	Few clusters, even cluster size, non-flat geometry, transductive	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters or distance threshold	Large <code>n_samples</code> and <code>n_clusters</code>	Many clusters, possibly connectivity constraints, transductive	Distances between points
Agglomerative clustering	number of clusters or distance threshold, linkage type, distance	Large <code>n_samples</code> and <code>n_clusters</code>	Many clusters, possibly connectivity constraints, non Euclidean distances, transductive	Any pairwise distance
DBSCAN	neighborhood size	Very large <code>n_samples</code> , medium <code>n_clusters</code>	Non-flat geometry, uneven cluster sizes, outlier removal, transductive	Distances between nearest points
OPTICS	minimum cluster membership	Very large <code>n_samples</code> , large <code>n_clusters</code>	Non-flat geometry, uneven cluster sizes, variable cluster density, outlier removal, transductive	Distances between points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation, inductive	Mahalanobis distances to centers
BIRCH	branching factor, threshold, optional global clusterer.	Large <code>n_clusters</code> and <code>n_samples</code>	Large dataset, outlier removal, data reduction, inductive	Euclidean distance between points

Clustering algorithms in scikit-learn



A comparison of the clustering algorithms in scikit-learn

Conclusions

Clustering is an “old” art form, for which there is a vast ocean of methods.

The K-means (and further developments) is the standard algorithm, if there is one such. DBSCAN is also an old (and awarded!) classic.

Note that like in dimensionality reduction, it is important to transform the input variables first, so that mean and variances are of order zero and unity.

It is HARD to evaluate the performance, and visual inspection and testing on similar (typically simulated) cases are some of few methods.



Example Use Cases

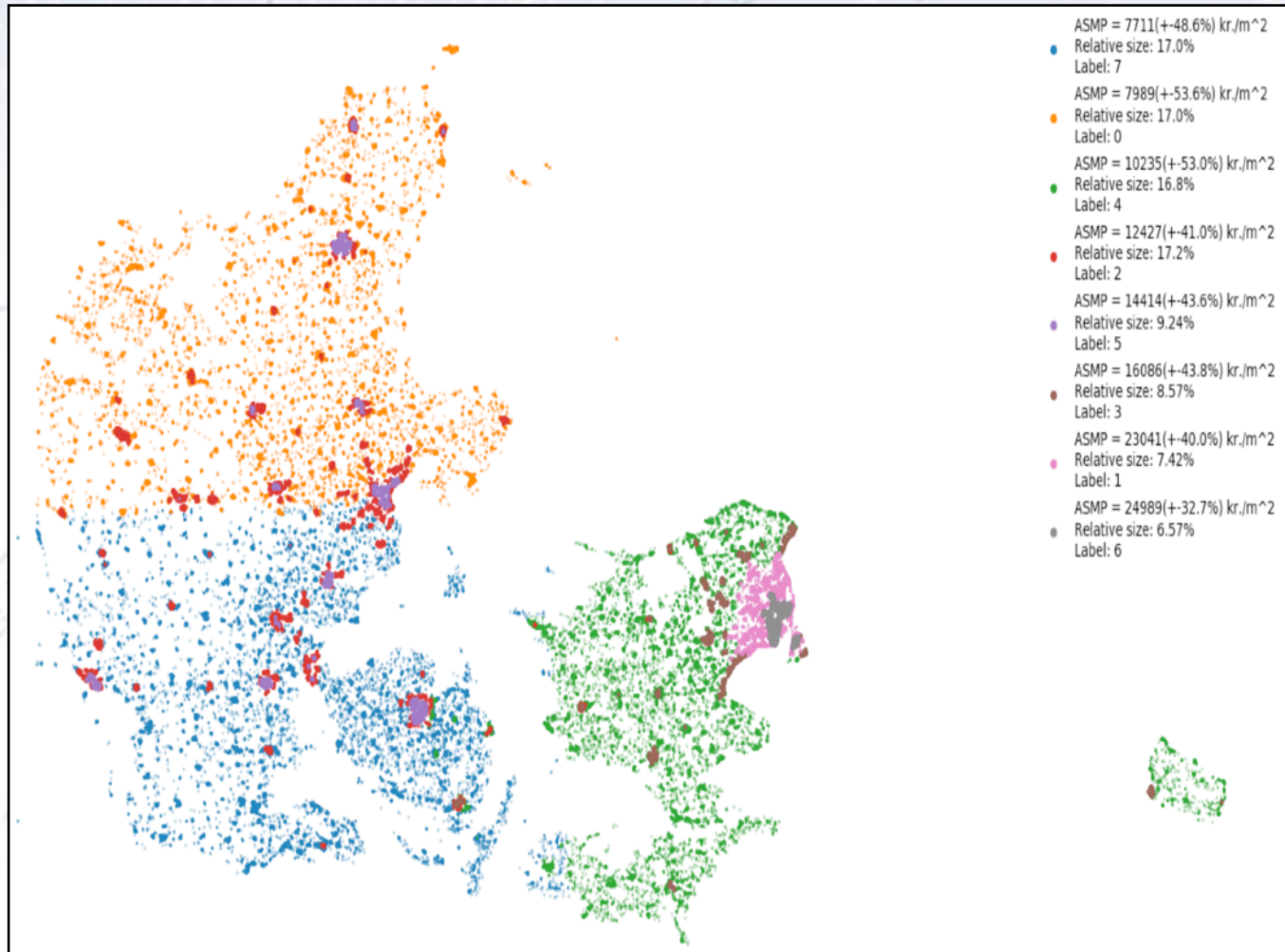
OK - what is it good for?

Clustering is used for several things:

- Market segmentation:
Dividing costumers or products into similar classes is used in advertising.
- Production quality assurance:
Clustering of images is used for detecting faulty productions automatically.
- DNA analysis:
The ability to cluster very high dimensional data (DNA) to groups / families.
- Medical imaging:
Classification of medical images without labels.
- Image segmentation:
Dividing an image into its parts is used in e.g. self-driving and security.
- Anomaly detection:
Quick detection of e.g. credit card fraud saves large amounts of money.

Clustering of Danish housing

Show is a simple clustering of the Danish housing market, based on position (x,y) and price/m². In this way, one can see developments for each market.



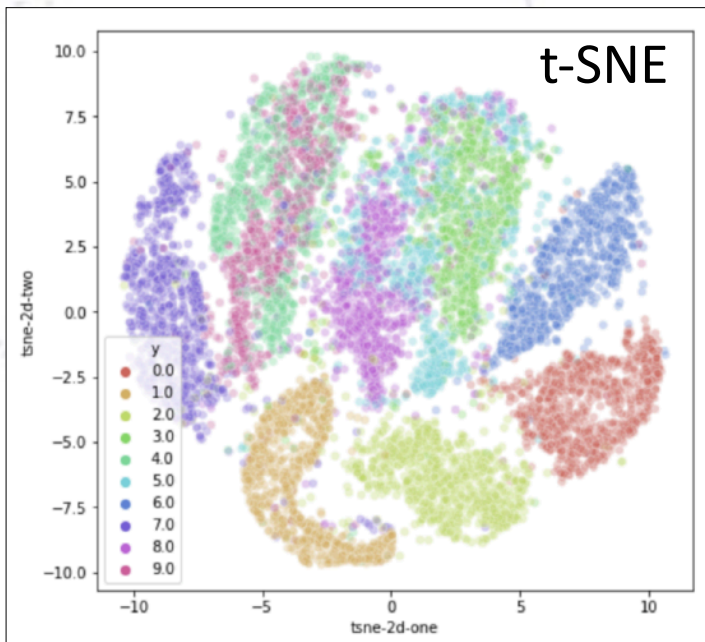


Unsupervised Learning: Dimensionality Reduction

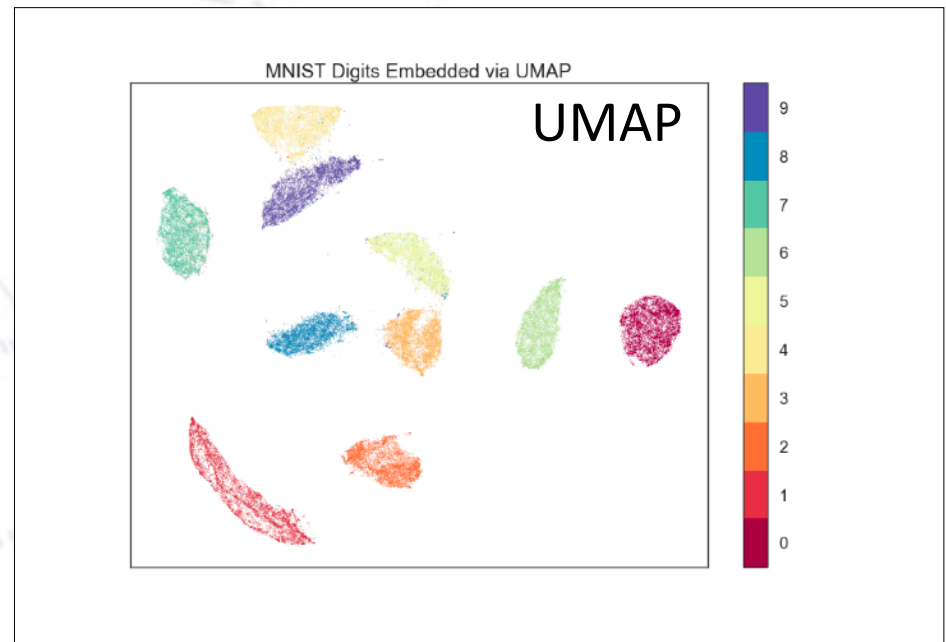
PCA, t-SNE & UMAP

High dimensionality has always been a curse - it is extremely hard to make sense of, and requires a lot of work and domain knowledge to boil down to few dimensions without losing a lot of information.

PCA has long reigned the linear case, and k-means the clustering, but two new(er) non-linear and powerful candidates are around: t-SNE and UMAP. Below are their performance on the MNIST data set.



Source: Towards data science (PCA and t-SNE)



Source: UMAP GitHub page: <https://github.com/lmcinnes/umap>

t-SNE Pro's and Con's

Pro: In the words of the t-Distributed stochastic neighbour embedding (t-SNE) paper, the t-SNE algorithm... *“...minimises the divergence between two distributions: a distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points in the embedding”*.

The great thing about this is, that there are no assumptions about distributions, relationships, or number of clusters. The algorithm is non-linear, which gives it a clear edge over e.g. PCA.

Con: However, computationally it is a “heavy” (ugly?) algorithm, since t-SNE scales **quadratically** in the number of objects N . This limits its applicability to data sets with only a few thousand input objects; beyond that, learning becomes too slow to be practical (and the memory requirements become too large). ”.

In real life, the t-SNE algorithm has especially had its impact in (a)DNA research, where the number of cases is typically not that large.

UMAP

UMAP builds on using **Riemannian manifolds**! Within differential geometry, this allows the definition of angles, hyper-area, and curvature in high dimensionality.

Abstract

UMAP (Uniform Manifold Approximation and Projection) is a novel manifold learning technique for dimension reduction. UMAP is constructed from a theoretical framework based in Riemannian geometry and algebraic topology. The result is a practical scalable algorithm that is applicable to real world data. The UMAP algorithm is competitive with t-SNE for visualization quality, and arguably preserves more of the global structure with superior run time performance. Furthermore, UMAP has no computational restrictions on embedding dimension, making it viable as a general purpose dimension reduction technique for machine learning.

UMAP paper, arXiv 1802.03426, Sep. 2020

The paper is quite mathematical with (10) definitions, lemmas, and proofs in the appendix. I find it a bit hard to read, but like their discussion of scaling and cons.

UMAP

As in the t-SNE case, UMAP tries to find a metric in both the original (large) space X , and the lower dimension output space Y , which can be (topologically) matched:

At a high level, UMAP uses local manifold approximations and patches together their local fuzzy simplicial set representations to construct a topological representation of the high dimensional data. Given some low dimensional representation of the data, a similar process can be used to construct an equivalent topological representation. UMAP then optimizes the layout of the data representation in the low dimensional space, to minimize the cross-entropy between the two topological representations.

UMAP paper, arXiv 1802.03426, Sep. 2020

However, the metrics in X and Y used by UMAP and t-SNE differ:

For t-SNE these metrics are as follows:

$$v_{j|i} = \exp(-\|x_i - x_j\|_2^2 / 2\sigma_i^2)$$

$$w_{ij} = \left(1 + \|y_i - y_j\|_2^2\right)^{-1}$$

For UMAP they are:

$$v_{j|i} = \exp[(-d(x_i, x_j) - \rho_i) / \sigma_i]$$

$$w_{ij} = \left(1 + a \|y_i - y_j\|_2^{2b}\right)^{-1}$$

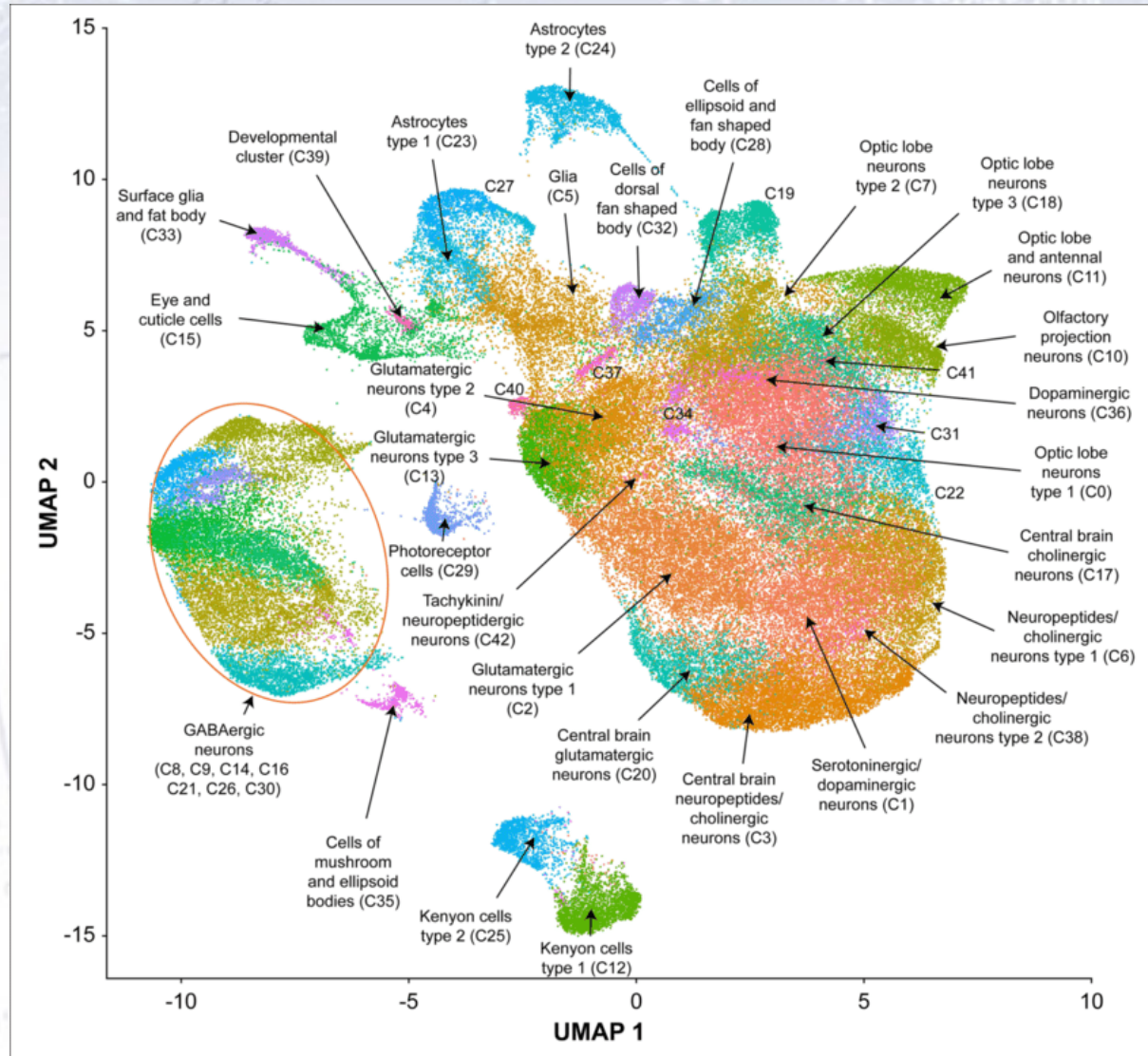


Example use cases...

Differentiating cells types

UMAP of different cell types.
The labelling comes from known cells, but might be based on very little data.

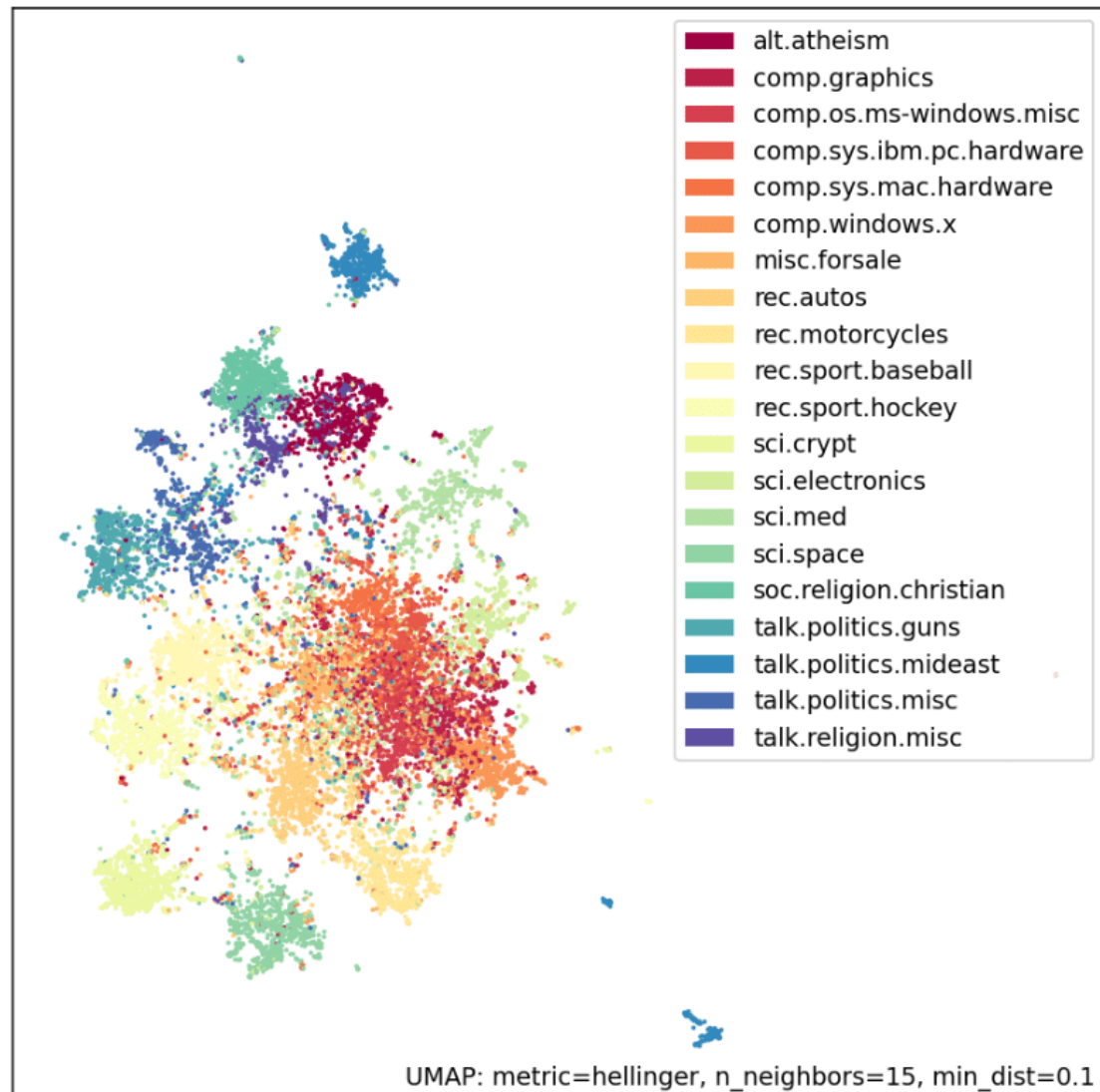
The unsupervised clustering gives a quite clear pattern, and ability to determine cell type without having a large training sample.



Mapping news group discussions

UMAP showing the differences between different news group discussion fora.

The ability to cluster fairly well would allow editors to direct text to the relevant news group.





Summary

Summary

The main ingredients in ML are:

- Solutions exists (Universal Approximation Theorems)
- Solutions can be found (Stochastic Gradient Descent)
- Algorithms that are implemented:
 - Boosted Decision Trees
 - Neural Networks
- Knowledge about how to tell them what to learn (Loss function)
- A scheme for how to use the data (Splitting / Cross Validation)

When applying ML to HEP data, there are several challenges:

- Data and MC do not follow the same distributions!!!
 - Sometimes it is clear: Spells disaster
 - But when it is not clear, then the impact is unknown.
 - Therefore, always think in terms of control channels.
- Loss functions are important
 - They are your way of telling the algorithm what to do (i.e. optimise for).
- Training is great, but stopping when it is done is also important.
- Good control of dividing dataset is important
 - Use Cross Validation (CV) when there is little data or errors are needed.
- Print and plot your data as the first thing!
- **Work hard on getting MC to match data.**



Bonus slides



Coding examples

Example analysis

I've produced a HEP example of classification based on the Aleph data from LEP times (with BDTs and NNs applied).

It runs out of the box, and you are welcome to copy it for your own use :-)

