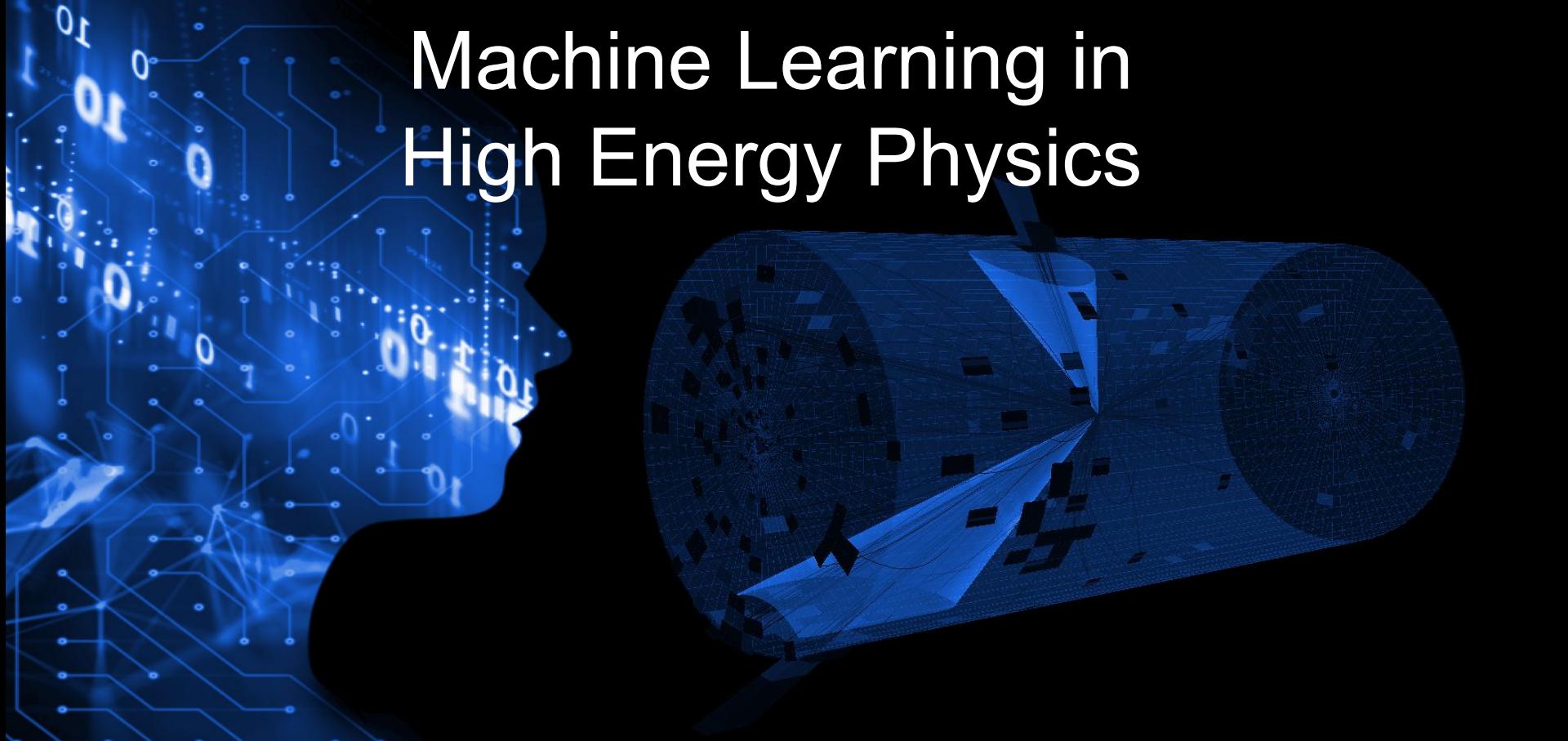


Machine Learning in High Energy Physics



<https://agenda.infn.it/event/28573/timetable/#20211220>

Cristiano Fanelli
cfanelli@mit.edu



Introduction and Scope

- This is a 2 days course for beginners dipping toes into ML for the first time!
 - Goal: introduce basic concepts (a small subset of topics)
 - Starting point for experimenting and playing with ideas
 - Build upon physicist language/background
 - Pre-requisites: linear algebra, multivariate calculus, probability theory, MC-methods, (some) python...
 - Lectures are accompanied by simple hands-on sessions and real-world examples
 - References

- Course indico: <https://agenda.infn.it/event/28573/>
- Lectures, exercises: <https://github.com/cfteach/ml4hep>
- A high bias, low-variance introduction to Machine Learning [hblvi2ML]:
<https://arxiv.org/abs/1803.08823>

Which these slides largely draw from.
- “A Living Review of Machine Learning for Particle Physics”:
<https://iml-wq.github.io/HEPML-LivingReview/> [arXiv:2102.02770](https://arxiv.org/abs/2102.02770) (2021)
- “Artificial Intelligence and Machine Learning in Nuclear Physics”: [arXiv:2112.02309](https://arxiv.org/abs/2112.02309) (2021)

AI

ARTIFICIAL INTELLIGENCE

A program that can sense, reason, act, and adapt

MACHINE LEARNING

Algorithms that learn patterns in data over time

DEEP LEARNING

Multilayered neural networks learn from vast amount of data

I. Goodfellow, Y. Bengio, and A. Courville
(2016), Deep Learning (MIT Press)
<http://www.deeplearningbook.org>.

Bayesian methods
Genetic Algorithms,
Rules-based system,
...

ML

Random Forest,
Support Vector Machines,
XGBoost,
...

DL

CNN, RNN, GAN, ...

DS

Data Science blends data analytics, computer science and business domain expertise to solve problems.

Data Analytics is the practice of using Machine Learning algorithms and visualization to derive insights.

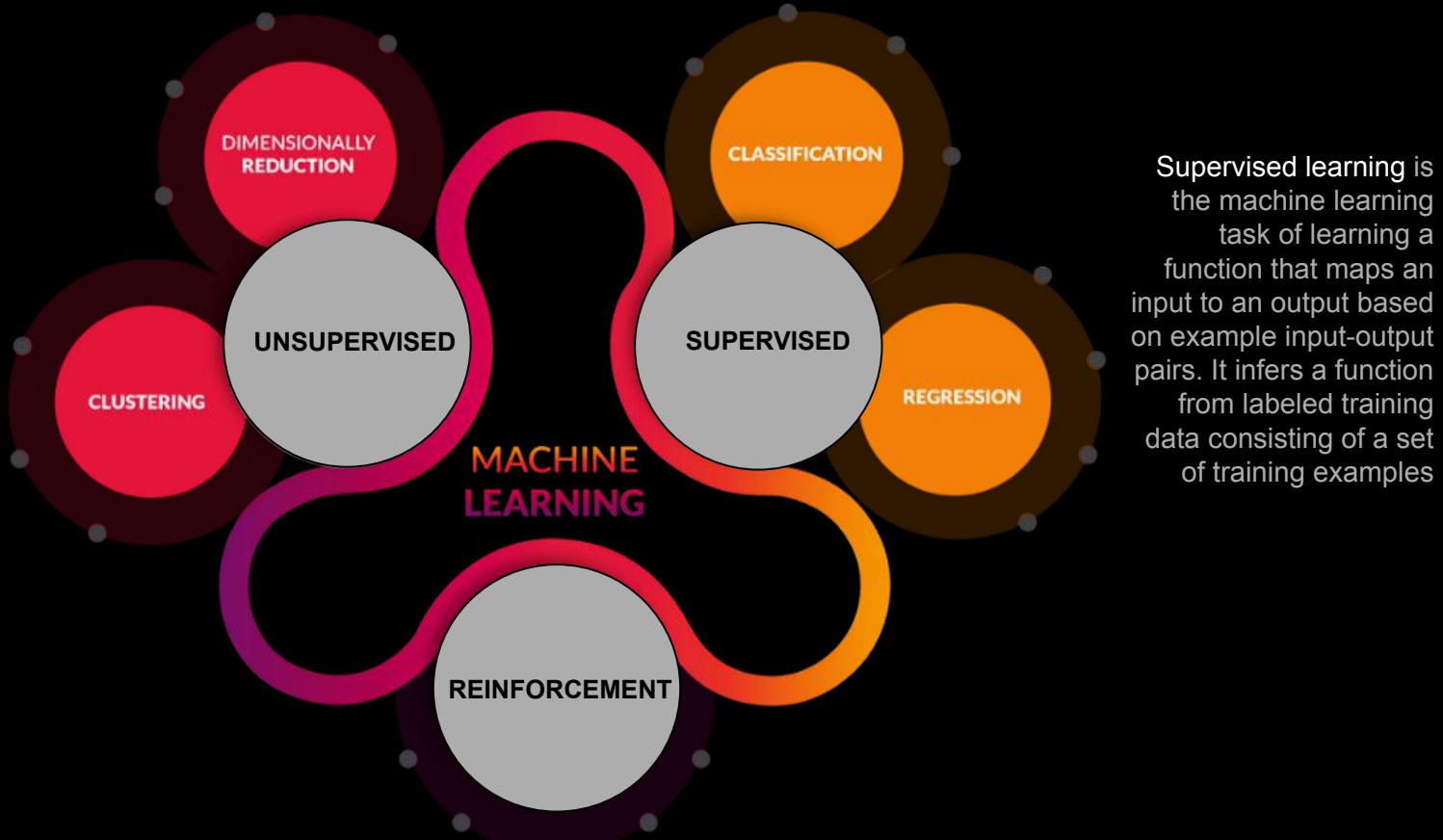
Use of AI/ML/DL in HEP becoming ubiquitous

1950s

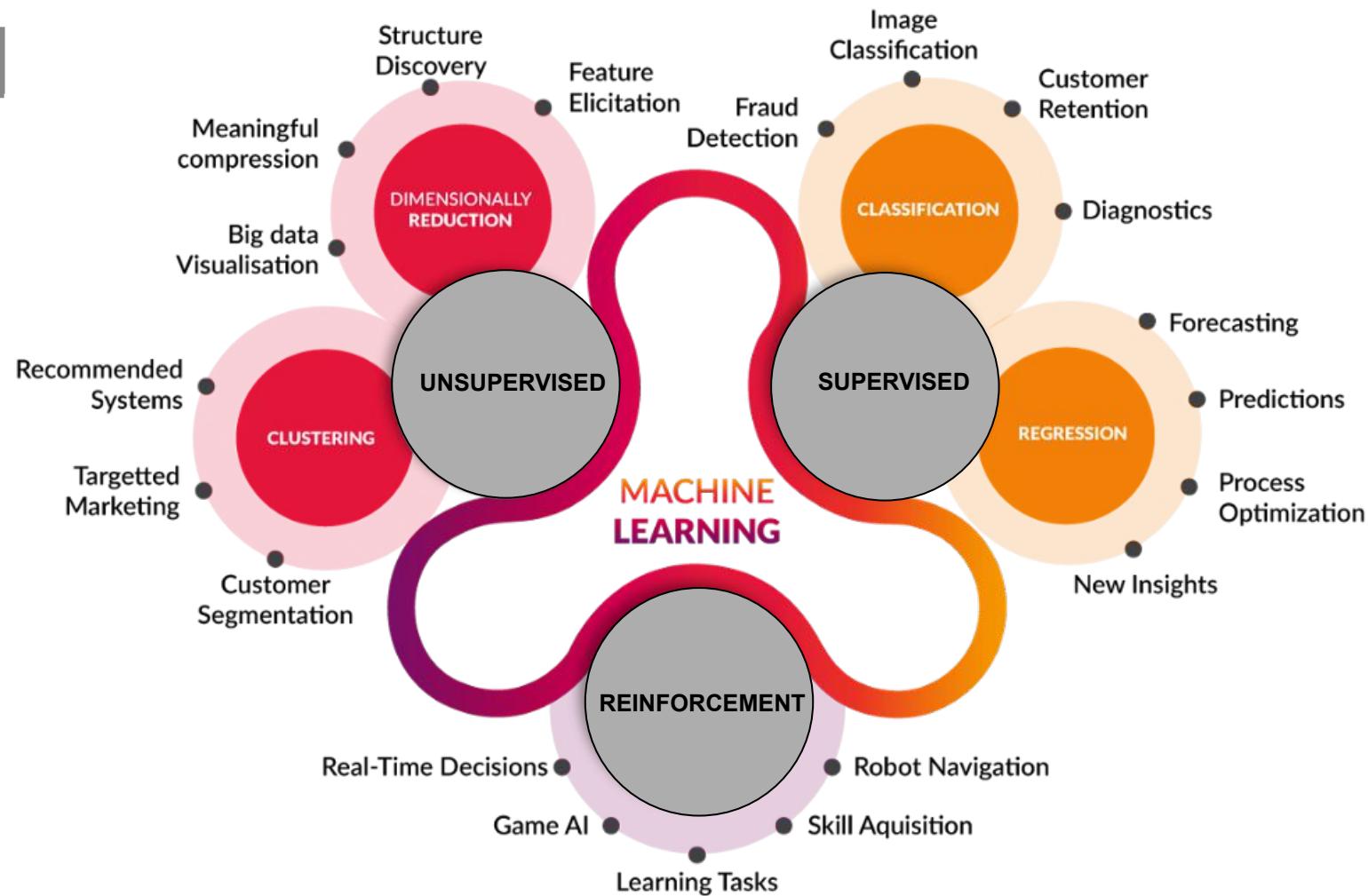
2010s -

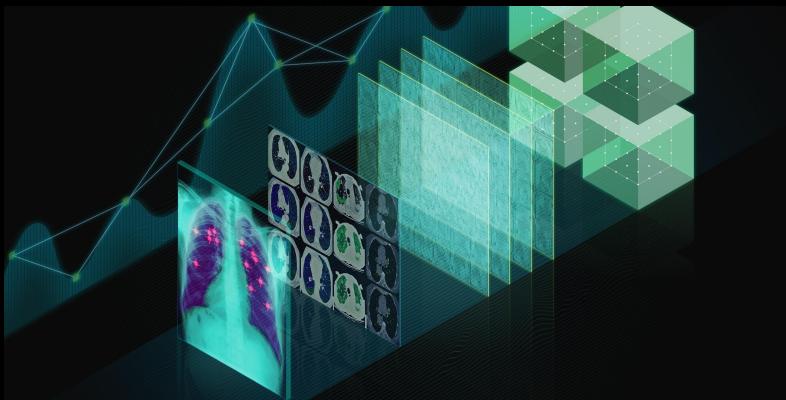
Unsupervised learning is a type of machine learning in which the algorithm is not provided with any pre-assigned labels or scores for the training data.

Unsupervised learning algorithms must first self-discover any naturally occurring patterns in that training data set.

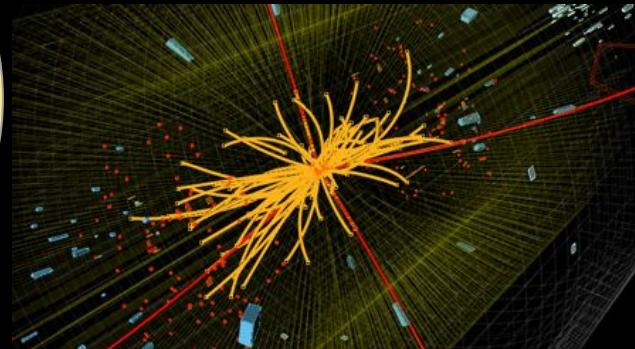
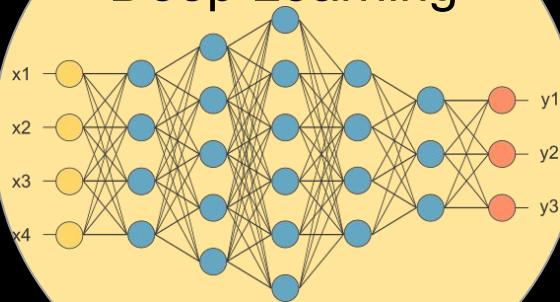


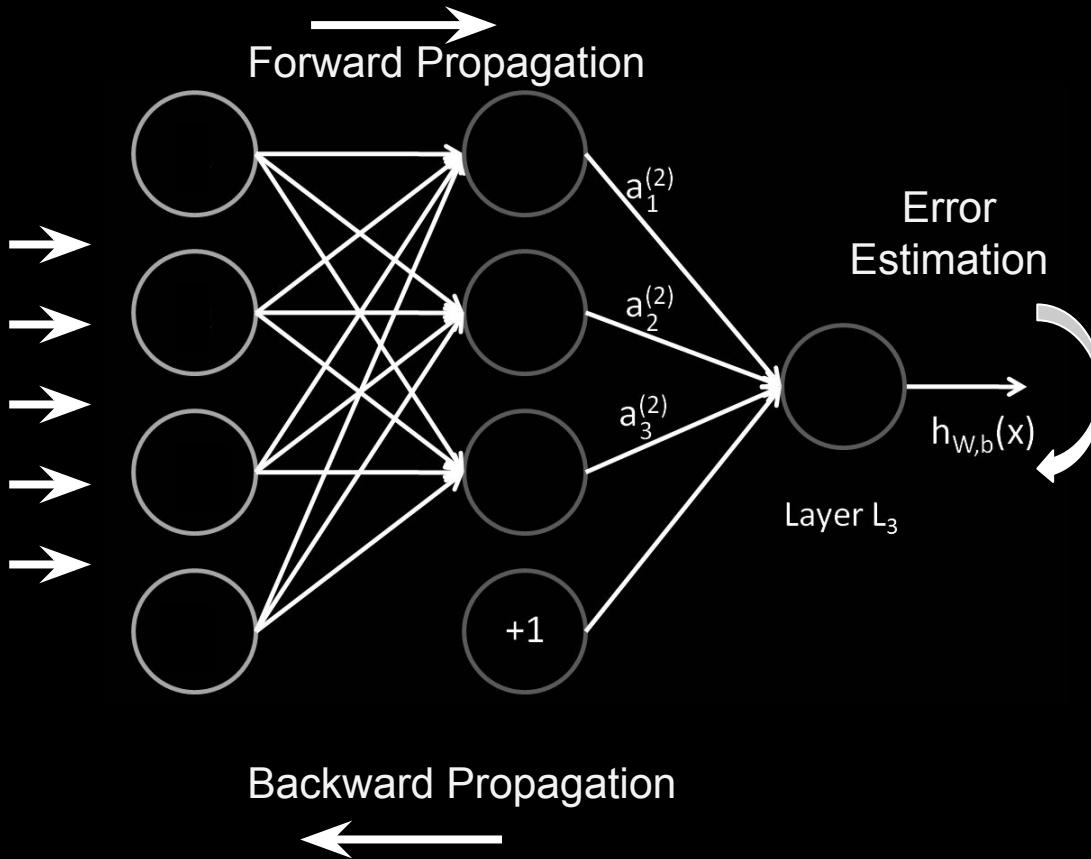
Reinforcement learning is concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward and make informed choices.





Deep Learning





- The real magic about NN is the result of an optimization technique: back-propagation (how a NN works to improve its output over time)
- DL (more hidden layers) nets are good in learning non-linear functions (heavy processing tasks)
- Based on old school NN revitalized by augmented capabilities (e.g. GPU) and a plethora of new architectures (RNN, CNN, autoencoders, GAN, etc.)

Unsupervised/Supervised/Reinforcement

Mnih et al, 1312.5602
Nature, 518.7540 (2015)

NIPS 2016: "If intelligence is a **cake**,
the bulk of the **cake** is **unsupervised learning**,
the icing on the **cake** is **supervised learning**,
and the cherry on the **cake** is **reinforcement learning**"



LeCun, Turing award 2018
VP and Chief AI Scientist, Facebook

Google DeepMind

Deep Q-learning
playing Atari Breakout



Timetable/Overview

timetable can change a bit as needed

09:00	1 - Introduction to the training course 2 - AIML/DL: what it all about	Cristiano Fanelli
10:00	Aula Salvini , LNF 3 - Gradient Descent	09:10 - 09:50_d Cristiano Fanelli
11:00	Aula Salvini , LNF 4 - Regression in a nutshell	09:50 - 10:30_d Cristiano Fanelli
12:00	Aula Salvini , LNF Coffee Break	10:30 - 11:10_d
13:00	5 - Hands-on - Exercise	Cristiano Fanelli
14:00	Aula Salvini , LNF	11:30 - 13:00_d
15:00	Lunch Break	
16:00	Aula Salvini , LNF 6 - Decision Trees: short overview + example	13:00 - 14:30_d Cristiano Fanelli
17:00	Aula Salvini , LNF 7 - Clustering: practical approach with few examples	14:30 - 15:30_d Cristiano Fanelli
18:00	Aula Salvini , LNF 8 - Hands-on - Exercise	15:30 - 16:30_d Cristiano Fanelli
		17:00 - 18:30_d

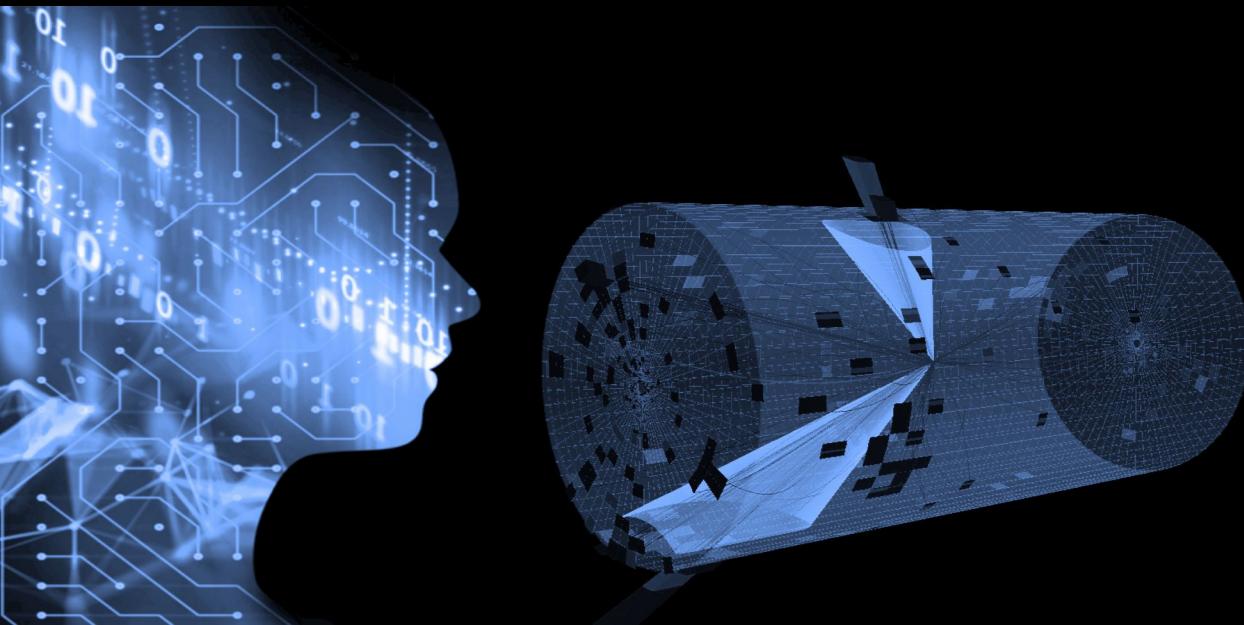
09:00	9 - Deep Neural Networks	Cristiano Fanelli
10:00	Aula Salvini , LNF 10 - Deep Neural Networks	09:00 - 10:00_d Cristiano Fanelli
11:00	Aula Salvini , LNF Coffee Break	10:00 - 11:00_d
12:00	Aula Salvini , LNF 11 - Hands-on - Build a simple NN	11:00 - 11:30_d Cristiano Fanelli
13:00	Aula Salvini , LNF Lunch Break	11:30 - 13:00_d
14:00	Aula Salvini , LNF 12 - Approaches to Multi-Objective Optimization in a nutshell	13:00 - 14:30_d Cristiano Fanelli
15:00	Aula Salvini , LNF 14 - AI-assisted Detector Design	14:30 - 15:30_d
16:00	Aula Salvini , LNF Coffee Break	15:30 - 16:30_d
17:00	Aula Salvini , LNF 15 - Hands-on - MOO Exercise	16:30 - 17:00_d Cristiano Fanelli
18:00	Aula Salvini , LNF	17:00 - 18:30_d

Structure and Disclaimer

- ML is broad and interdisciplinary and draw on ideas from many fields
- In these 2 days we can only cover very few ideas
- In some cases I will just give the gist of the theoretical foundation behind these concepts but I won't have time to go into the details.
- We will try to provide/point to tools to start using ML in practical problems
 - Jupyter notebooks
 - Numerous of great software packages
- Datasets:
 - SUSY dataset (5M MC samples)
 - Higgs dataset (11M)
 - Artificially created datasets

Following [hblvi2ML] we will cover:

- Regression
- “Decision trees”
- Clustering
- (Intro to) Deep Learning
- Optimization



ML4HEP

Notebooks are available at [\[hblvi2ml\]](#).

In these 2 days we will cover explicitly only few exercises, spanning supervised classification tasks, unsupervised approaches for clustering and optimization.

Jupyter Notebooks

- Work with python3
- Install jupyter in a virtual environment
 - python3 -m venv env_ml4hep
 - source ./env_ml4hep/bin/activate
 - pip install jupyter (you can also install with anaconda if you are more familiar)
- How to change/add the kernel of a jupyter notebook in a virtual environment?
 - ipython kernel install --name "env_ml4hep" --user
- To start notebook (from terminal, virtual environment):
 - jupyter notebook



We will also use
colab later

What is ML?

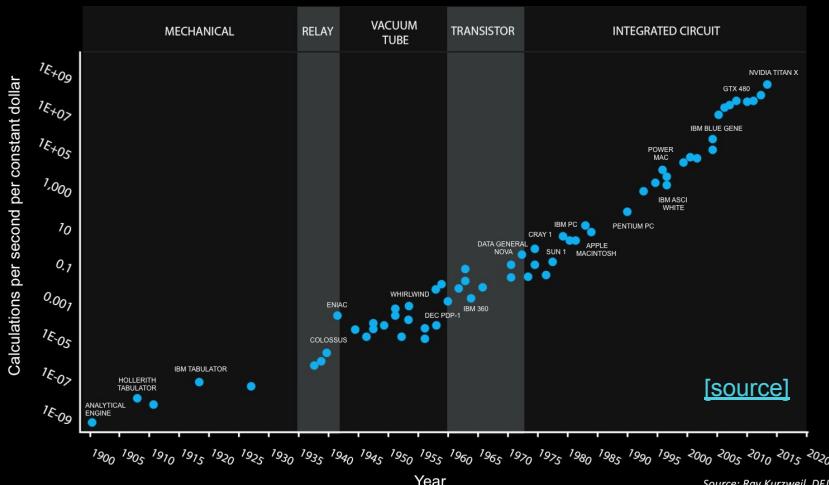
- ML, data science and statistics are fields describing how to learn from and make predictions about data.
- Techniques in ML tend to be more focused on predictions than estimation. Methods from ML tend to be applied to more complex high-dimensional problems.
- Estimation and prediction problems can be cast into a common conceptual framework related to some parameters θ of a model $p(x|\theta)$ that describes the probability of observing x given θ .
- Fitting the model involves finding θ^* providing best explanation for data. If fitting refers to the method of least squares, the estimated parameters maximize $\theta^* = \operatorname{argmax}_{\theta} \{p(x|\theta)\}$.
- Although the goals of estimation and prediction are related they often lead to different approaches:
 - Estimation problems are concerned with the accuracy of θ^* .
 - Prediction problems are concerned with the ability of the model to predict new observations. We will focus on prediction.

Problems in ML typically involve inference about complex systems where we do not know the exact form of the mathematical model that describes the system.

Why is ML ubiquitous

- Last three decades unprecedented ability to generate and analyze large data sets: big data revolution spurred by exponential increase of computing power and memory
- Computations that were unthinkable can now be routinely performed on laptops.
- Specialized computing machines (e.g., GPU-based) are continuing this trend towards cheap, large scale computation.
- Physicists are uniquely situated to benefit from and contribute to ML. Many core concepts in ML have their origin in physics: MC methods, variational methods, simulated annealing, energy based models etc
- HEP has been at the forefront of using big data.
 - LHC experiments produce data at a rate of 1PB/sec; after data reduction (zero suppression, custom compression algorithms) 50TB/s resulting in as much data every hour as Facebook collects globally in a year
 - At LHCb, 70% of all data retained are classified by ML and all charged particle tracks are vetted by NNs.

120 Years of Moore's Law



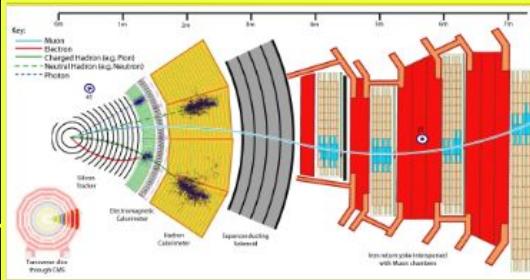
[source]

Source: Ray Kurzweil, 2005

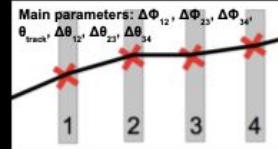
Why is ML ubiquitous

HW Trigger Muon ID
@CMS

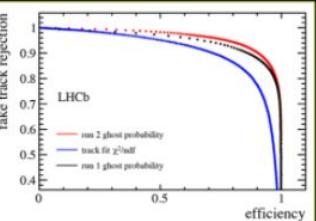
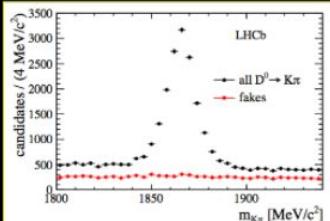
Ghost tracks killer,
HLT2 Topological Trigger @LHCb



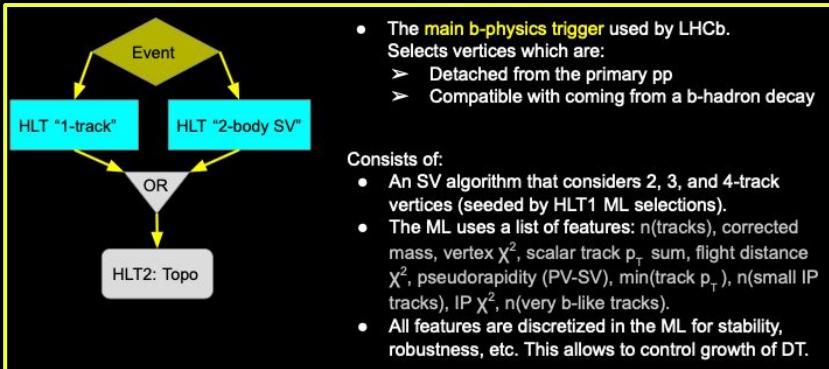
- transverse momentum (p_T) is assigned based on curvature
- The Endcap Muon Track Finder (EMTF) needs to process hits and assign a momentum
- Interesting muons have large p_T

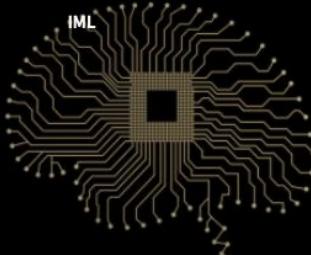


Fake-track (ghost) killing DNN based on 22 features, most important are hit multiplicities and track-segment chi2 values from tracking subsystems. Significantly reduces the rate of events selected in the HLT1.



Run in the trigger on all tracks (it must be very fast). Use of custom activation function and highly-optimized C++ implementation.



[ABOUT IML](#)[FORUM \(MAILING LIST\)](#)[MEETINGS](#)[PEOPLE](#)[PUBLIC DATASETS](#)[SOFTWARE](#)[HARDWARE](#)

IML

Inter-Experimental LHC Machine Learning Working Group

Who we are

The Large Hadron Collider at CERN produces petabytes of data per day at the four large experiments (ALICE, ATLAS, CMS, and LHCb), which altogether have roughly 10.000 collaborators. Processing this amount of data leads to plenty of demanding challenges that require development and deployment of state-of-the-art machine learning solutions. Applications range from small to truly large scales and from very fast (a few μs) to modest inference (many seconds) times. The Inter-experimental Machine Learning (**IML**) Working Group provides a forum for the machine learning community at the LHC. It brings together scientists from the LHC experiments, connects them to the data science community, fosters inter-experimental common solutions, and provides training and benchmarks. Each experiment is represented by an IML coordinator. The IML working group is hosted and supported by the LHC Physics Center at CERN ([LPC](#)). (For a formal definition of the group, please refer to the [Mandate](#).)

What we do

IML organizes monthly [meetings](#) on a variety of subjects. These meetings are often topic-oriented (focusing on a certain ML technique) and may include external experts. Each spring IML organizes an annual workshop typically comprised of roughly 300 participants, which includes invited data scientist's talks, submitted talks, and tutorials.

IML also serves as entry point to find LHC specific machine learning resources, such as [software](#) solutions for machine learning starting from the common ROOT file format. We build a forum for community driven summaries of software solutions, announce LHC tailored trainings/school, and list relevant papers and people involved. We can help finding temporary [hardware](#) resources (GPUs) for tests. We are currently building up a database with [benchmarks](#) datasets and challenges in order to better enable testing new methods in our domain against previous ones.

News

New ALICE coordinator

Gian Michele Innocenti took over from Rüdiger. Welcome to the team!

New TH coordinators

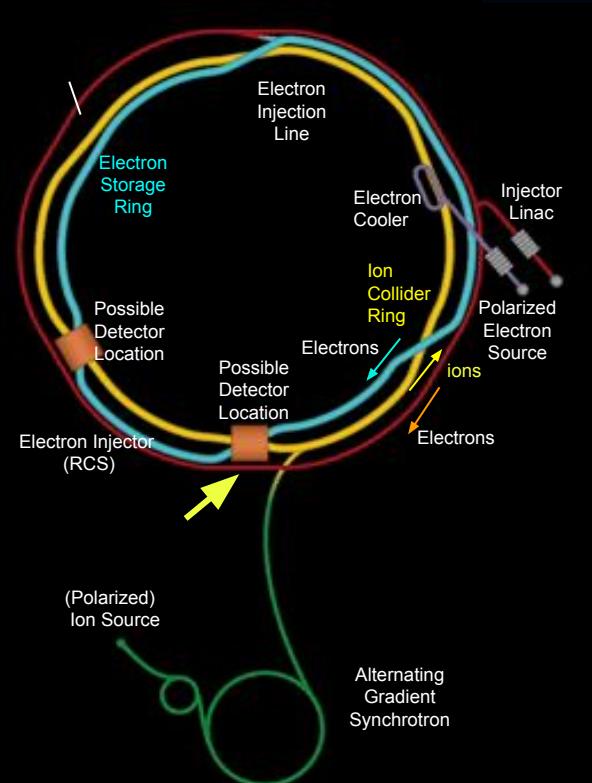
Riccardo Torre and Andrea Wulzer represent now theory. Welcome to the team!

NEW CMS coordinator

Pietro Vischia took over from Loukas. Welcome to the team!

NEW LHCb coordinator

Simon Akar took over from Paul. Welcome to the team!



ARTIFICIAL INTELLIGENCE FOR THE ELECTRON ION COLLIDER

HOME EVENTS WORKSHOPS HACKATHONS REFERENCES HOW TO JOIN

A large graphic on the right side of the page features a blue-toned circular visualization of particle interactions, showing 'e-' and 'e+' particles. To the left of this graphic is a large, stylized white 'AI' logo composed of a grid of blue squares. Below the 'AI' logo is the text '4EIC'. The overall background of the banner is dark blue with a subtle grid pattern.

A.I. for the Electron Ion Collider

[— CONTACTS —](#)

SCOPE

Artificial Intelligence for the Electron Ion Collider

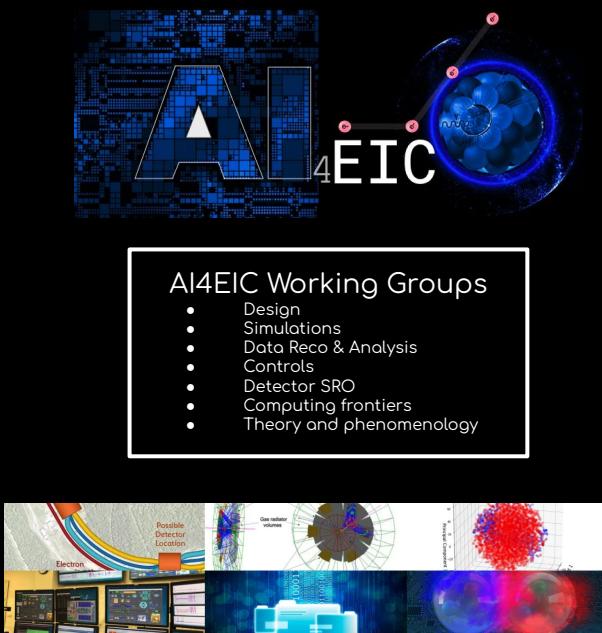
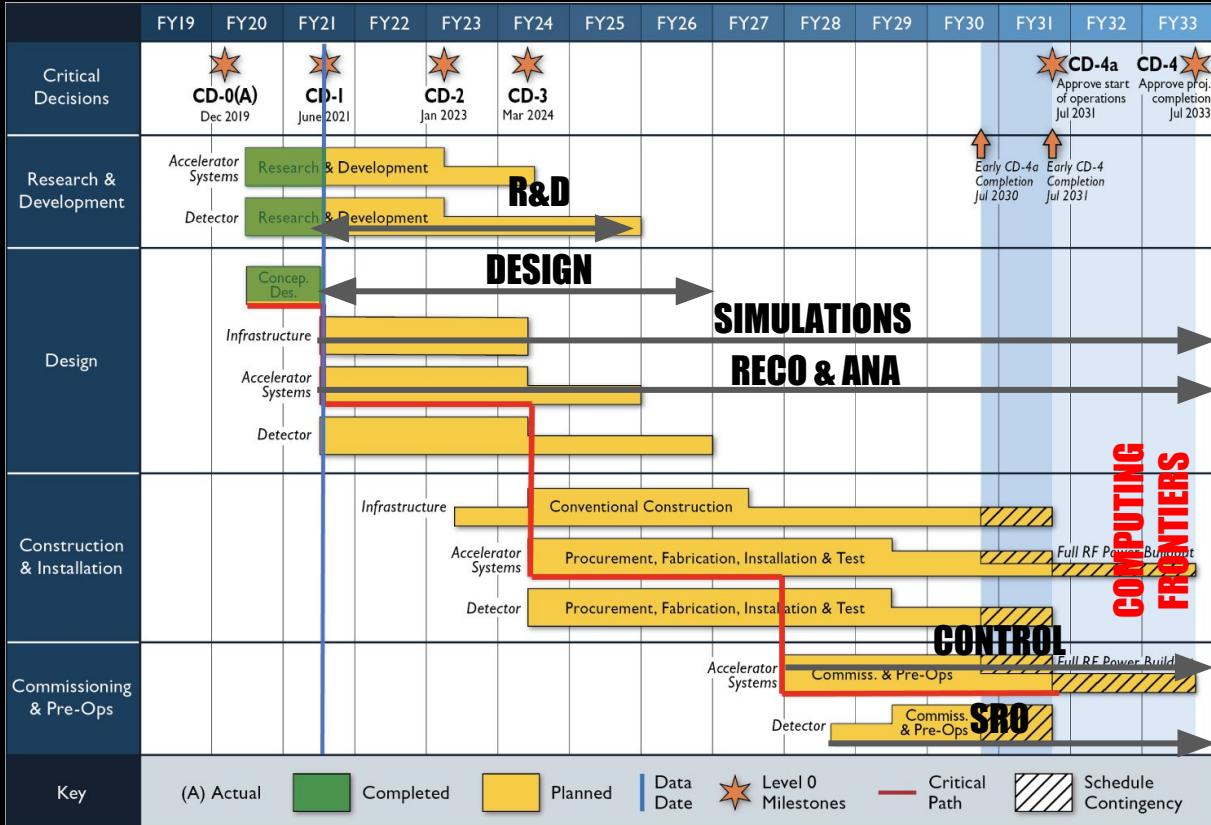
AI will be an essential part of future experiments like the Electron Ion Collider, a new \$2B high-luminosity facility capable to collide high-energy electron beams with high-energy proton and ion beams that will be built at BNL in approximately 10 years from now to unlock the secrets of the "glue" that binds the building blocks of visible matter in the universe.

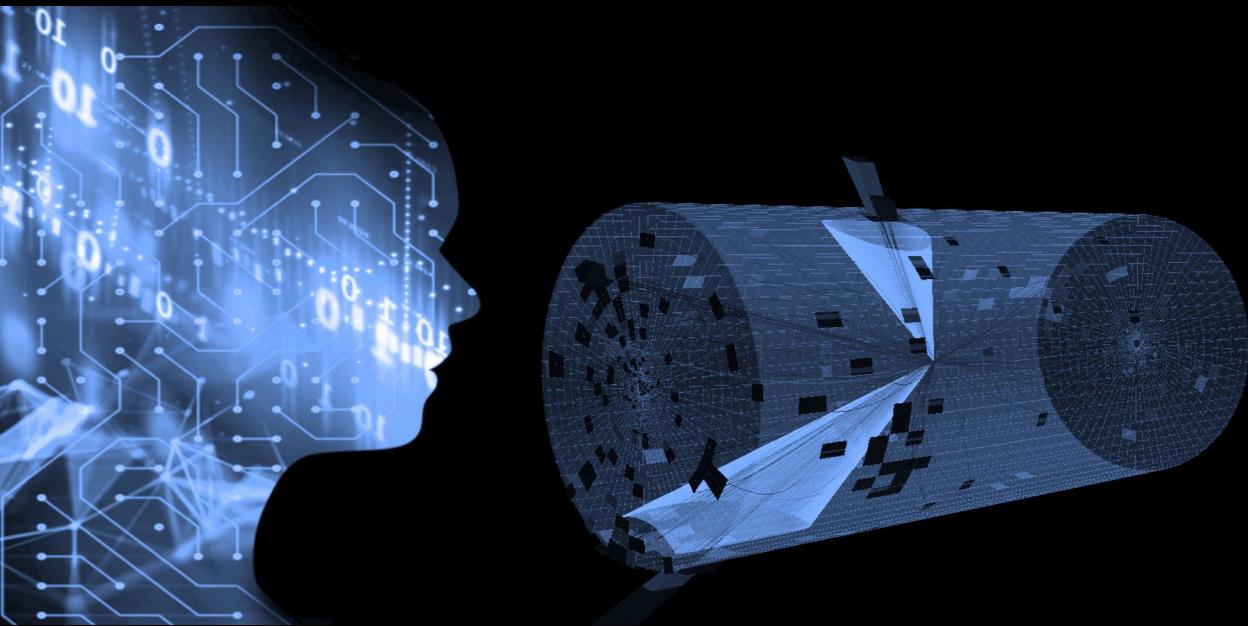
AI can provide new insights and discoveries from both experimental and computational data produced at user facilities.

This website includes AI activities related to EIC that will characterize the different phases of its realization.

EIC Schedule and Role of AI

<https://eic.ai>





Why is ML
difficult

Typical Problem

- Ingredients:
 - Dataset $D(\mathbf{X}, \mathbf{y})$: \mathbf{X} matrix of independent variables, \mathbf{y} dependent variables
 - Model $f(\mathbf{X}; \boldsymbol{\theta})$ where $f: \mathbf{X} \rightarrow \mathbf{y}$ is a function of the parameters $\boldsymbol{\theta}$
 - Cost function $C(\mathbf{y}, f(\mathbf{X}; \boldsymbol{\theta}))$ to judge how well the model performs on the observations \mathbf{y}
- The model is fit to find $\boldsymbol{\theta}$ that minimize the cost function; commonly used cost is squared error (method of least squares)
- Recipe for prediction problems:

1. Randomly divide the dataset D into mutually exclusive D_{train} (typically 90%) and D_{test} (10%)
2. Model is fit on training data $\boldsymbol{\theta}^* = \operatorname{argmin}_{\boldsymbol{\theta}} \{C(\mathbf{y}_{\text{train}}, f(\mathbf{X}_{\text{train}}; \boldsymbol{\theta}))\}$
3. The performance of the model is evaluated on $C(\mathbf{y}_{\text{test}}, f(\mathbf{X}_{\text{test}}; \boldsymbol{\theta}^*))$
 - Splitting data provides an unbiased estimate for the predictive performance (known as **cross-validation**)
 - In-sample error: $E_{\text{in}} = C(\mathbf{y}_{\text{train}}, f(\mathbf{X}_{\text{train}}; \boldsymbol{\theta}))$; out-of-sample error: $E_{\text{out}} = C(\mathbf{y}_{\text{test}}, f(\mathbf{X}_{\text{test}}; \boldsymbol{\theta}))$
 - E_{out} is always larger than E_{in} , $E_{\text{out}} \geq E_{\text{in}}$

Bias/Variance

- The model that provides the best explanation for the current dataset will probably not provide the best explanation for future datasets
- The discrepancy between E_{in} , E_{out} grows with the complexity of our data and of our model (increased model parameters, high dimensional space, curse of dimensionality)
- For these reasons (and for complicated models), predicting and fitting can be different things. Need to pay attention to out-of-sample performance. Fitting existing data well is fundamentally different from making predictions about new data.
- Let's see this starting from simple one-dimensional problem: we want to fit data with polynomials of different orders.
- Our ability to predict depends on the number of data points, the noise in the data, and our prior knowledge about the system

Fitting vs Predicting

- Consider probabilistic process that assigns a label y_i to an observation x_i . Data are generated from
 - $y_i = f(x_i) + \eta_i$ where η_i is a gaussian uncorrelated noise variable such that $\langle \eta_i \rangle = 0$ and $\langle \eta_i \eta_j \rangle = \delta_{ij} \sigma^2$
- To make predictions we consider a family of functions $f_\alpha(x; \theta_\alpha)$ (different model complexity):
 - Polynomial of order 1: $f_1(x; \theta_1) \rightarrow 2$ pars
 - Polynomial of order 3: $f_3(x; \theta_3) \rightarrow 4$ pars
 - Polynomial of order 10: $f_{10}(x; \theta_{10}) \rightarrow 11$ pars
- Using a more complex model class may give us better prediction power but only if we have a large enough sample size to accurately learn the model parameters
- To learn θ_α we use a training dataset and test the effectiveness of the model on the test dataset
- Obviously the more data and less noise we have the better the predictions are

Fitting vs Predicting

- We utilize a test interval $[0, 1.2]$ which is larger than the training interval $[0, 1.0]$
- Data sampled from
 - $f(x) = 2x$
 - $f(x) = 2x - 10x^5 + 15x^{10}$
- In absence of noise, even with a small training set ($N_{\text{train}} = 10 < N_{\text{test}} = 20$) the model class that generated the data provides the best fit and also the best out-of-the sample prediction.



FIG. 1 Fitting versus predicting for noiseless data. $N_{\text{train}} = 10$ points in the range $x \in [0, 1]$ were generated from a linear model (top) or tenth-order polynomial (bottom). This data was fit using three model classes: linear models (red), all polynomials of order 3 (yellow), all polynomials of order 10 (green) and used to make prediction on $N_{\text{test}} = 20$ new data points with $x_{\text{test}} \in [0, 1.2]$ (shown on right). Notice that in the absence of noise ($\sigma = 0$), given enough data points that fitting and predicting are identical.

Fitting vs Predicting

- We utilize a test interval $[0, 1.2]$ which is larger than the training interval $[0, 1.0]$
- Data sampled from
 - $f(x) = 2x$
 - $f(x) = 2x - 10x^5 + 15x^{10}$
- Noise = 1; training set ($N_{\text{train}} = 100 > N_{\text{test}} = 20$); even when the model class that generated the data is a 10 order polynomial, the linear and 3rd order polynomials give better out-of-sample predictions.
- At small sample sizes, noise can create fluctuations in the data that look like genuine patterns.

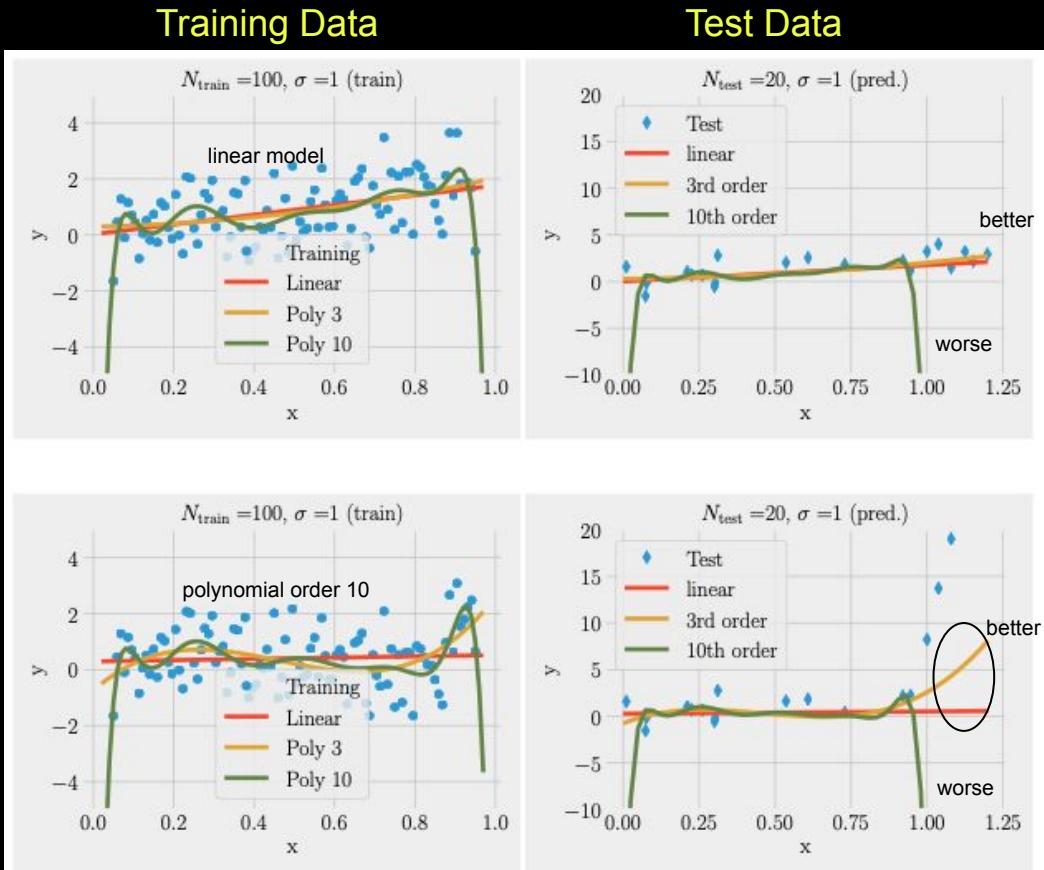


FIG. 2 Fitting versus predicting for noisy data. $N_{\text{train}} = 100$ noisy data points ($\sigma = 1$) in the range $x \in [0, 1]$ were generated from a linear model (top) or tenth-order polynomial (bottom). This data was fit using three model classes: linear models (red), all polynomials of order 3 (yellow), and all polynomials of order 10 (green) and used to make prediction on $N_{\text{test}} = 20$ new data points with $x_{\text{test}} \in [0, 1.2]$ (shown on right). Notice that even when the data was generated using a tenth order polynomial, the linear and third order polynomials give better out-of-sample predictions, especially beyond the x range over which the model was trained.

Bias/Variance Tradeoff

- We utilize a test interval $[0, 1.2]$ which is larger than the training interval $[0, 1.0]$
- Data sampled from
 - $f(x) = 2x$
 - $f(x) = 2x - 10x^5 + 15x^{10}$
- Noise = 1; let's increase the training set to $N_{\text{train}} = 10^4$

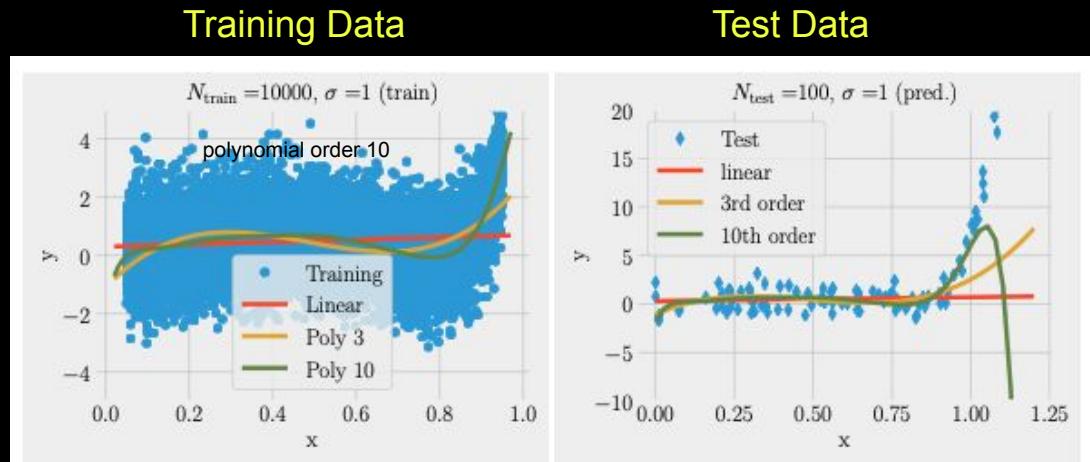


FIG. 3 Fitting versus predicting for noisy data. $N_{\text{train}} = 10^4$ noisy data points ($\sigma = 1$) in the range $x \in [0, 1]$ were generated from a tenth-order polynomial. This data was fit using three model classes: linear models (red), all polynomials of order 3 (yellow), all polynomials of order 10 (green) and used to make prediction on $N_{\text{test}} = 100$ new data points with $x_{\text{test}} \in [0, 1.2]$ (shown on right). The tenth order polynomial gives good predictions but the model's predictive power quickly degrades beyond the training data range.

- The 10 polynomial model gives both the best fit and the most predictive power over the entire range $[0, 1]$ and actually slightly beyond up to ~ 1.05 , but then the predictive power quickly degrades
- This is our first experience with the **bias-variance tradeoff**: where the amount of data is limited, we often get better predictive performance by using a less expressive model (lower order polynomial)
 - The simpler model has more bias but is less dependent on the particular realization of the training set, i.e. has less variance.

ML is difficult

- Last one was a good example where we are good at interpolating but not at extrapolating.
- Fitting is not predicting.
 - Fitting existing data well is fundamentally different from making predictions about new data
- Using a complex model can result in overfitting
 - Better result on training data; when data size is small and the data are noise, this results in overfitting and degrade predictive performance
- For complex datasets and small training sets, simple models can be better at predicting than complex ones due to the bias-variance tradeoff
 - Even though the correct model (less bias) has better predictive performance for an infinite amount of training data, the training errors stemming from finite-size sampling (variance) can cause simpler models to outperform the more complex model
- It is difficult to generalize beyond what seen in the training dataset.

Statistical Learning Theory

- We summarize here the sense in which learning is possible with focus on supervised learning
- We begin from an unknown function $y=f(x)$ and fix a *hypothesis set H* of all functions we want to consider
- $f(x)$ produces a set of pairs (x_i, y_i) , $i=1,\dots,N$, which serve as the observable data. Our goal is to select a function from the hypothesis set $h \in H$ that approximates $f(x)$ as best as possible, such that $h \approx f$
- If that is possible we say we learnt $f(x)$
- But if the function $f(x)$ can, in principle, take any value on unobserved inputs, how is it possible to learn in any meaningful sense?

The relationship between the in-sample error E_{in} and the out-of-sample (or generalization) error is the domain of statistical learning theory

Bias/Variance

Assumptions:

- E_{out} and E_{in} as a function of the size of training data
- We assume data come from a complicated distribution (so we won't exactly learn $f(x)$)

- (1) The more data we train on, the more the sampling noise decreases and the training data becomes representative of the true distribution. For this reason E_{out} and E_{in} approach the same bias.
- The more data we train on, the more the sampling noise decreases and the training data becomes representative of the true distribution. For this reason E_{out} and E_{in} approach the same value called bias. **The bias represents the best our model can do if we had an infinite amount of data.**
- The more complex the model we use, the smaller the bias. However we do not have an infinite amount of data.
- For this reason **the best predictive power is get by minimizing the E_{out} .** E_{out} is decomposed in a bias and a variance, which measures the errors in training our model due to sampling noise
- **The difference between E_{out} and E_{in} measures the difference between fitting and predicting.**

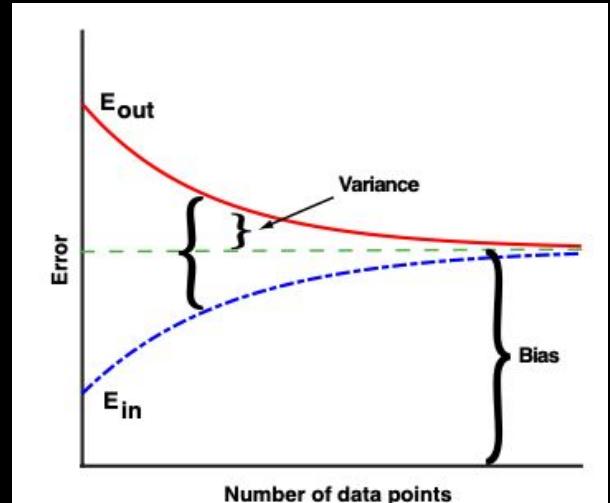


FIG. 4 Schematic of typical in-sample and out-of-sample error as a function of training set size. The typical in-sample or training error, E_{in} , out-of-sample or generalization error, E_{out} , bias, variance, and difference of errors as a function of the number of training data points. The schematic assumes that the number of data points is large (in particular, the schematic does not show the initial drop in E_{in} for small amounts of data), and that our model cannot exactly fit the true function $f(x)$.

Bias/Variance

Assumptions:

- E_{out} and E_{in} as a function of the size of training data
- We assume data come from a complicated distribution (so we won't exactly learn $f(x)$)

- Model complexity is a subtle concept which can in many cases be related to the number of parameters needed to approximate the true function $f(x)$
- E_{out} will be in general a non-monotonic function of the model complexity.
- It is generally minimized for intermediate complexity.
- Even though using a complicated model always reduces bias, at some point the model becomes too complex for the amount of training data that the generalization error becomes large due to high variance
- Thus it may be more suitable to use a more biased model with small variance.

$$E_{\text{out}} = \text{Bias}^2 + \text{Var} + \text{Noise},$$

It can be shown that:

$$\text{Var} = \sum_i \mathbb{E}_{\mathcal{D}}[(f(\mathbf{x}_i; \hat{\boldsymbol{\theta}}_{\mathcal{D}}) - \mathbb{E}_{\mathcal{D}}[f(\mathbf{x}_i; \hat{\boldsymbol{\theta}}_{\mathcal{D}})])^2],$$

$$\text{Bias}^2 = \sum_i (f(\mathbf{x}_i) - \mathbb{E}_{\mathcal{D}}[f(\mathbf{x}_i; \hat{\boldsymbol{\theta}}_{\mathcal{D}})])^2$$

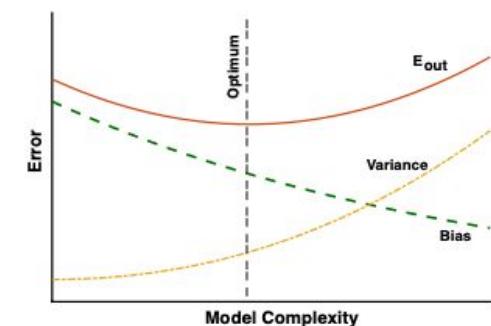
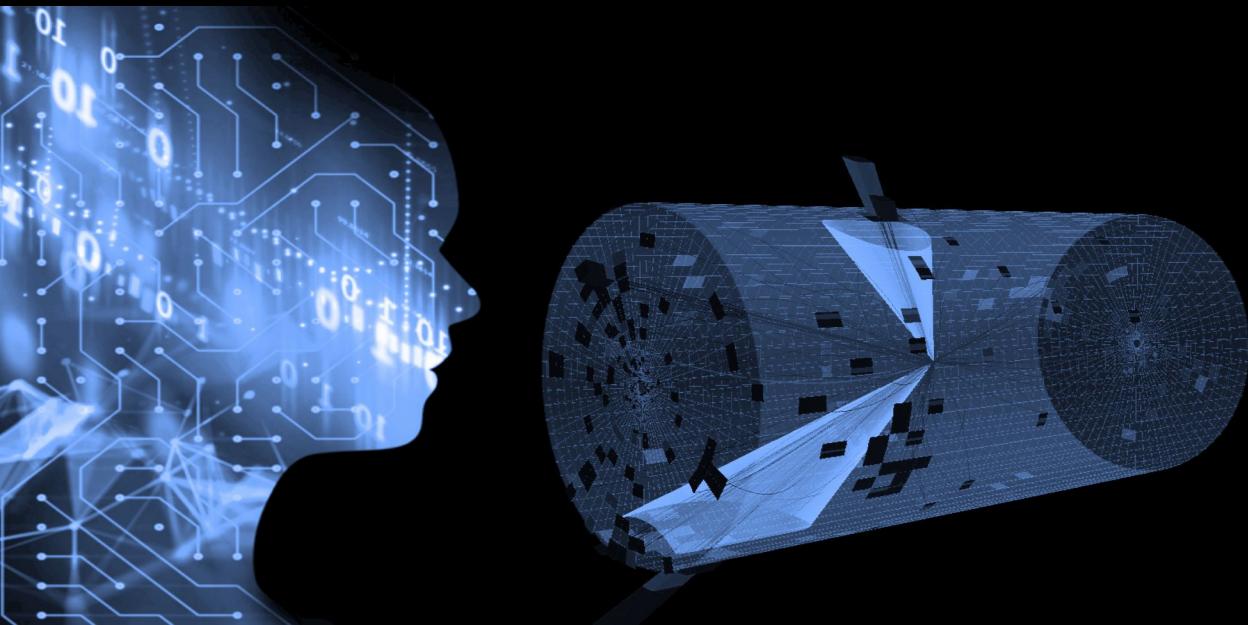


FIG. 5 Bias-Variance tradeoff and model complexity. This schematic shows the typical out-of-sample error E_{out} as function of the model complexity for a training dataset of fixed size. Notice how the bias always decreases with model complexity, but the variance, i.e. fluctuation in performance due to finite size sampling effects, increases with model complexity. Thus, optimal performance is achieved at intermediate levels of model complexity.

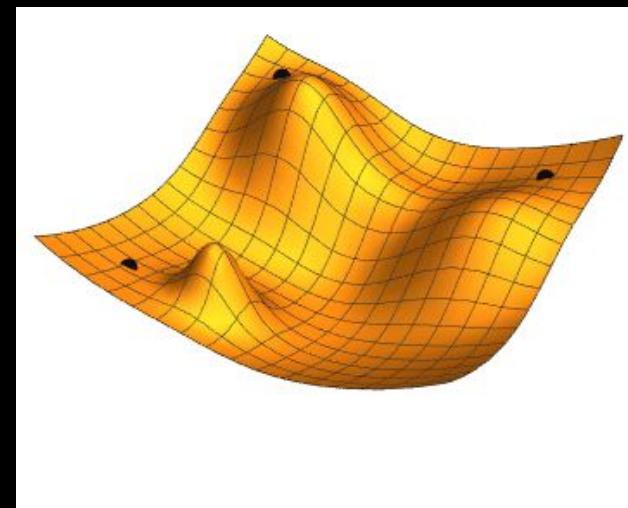
- Bias measures the deviation of the expectation value of our estimator (asymptotic value in the infinite data limit) from the true value
- Variance measures how much our estimator fluctuates due to finite-sample effects



Gradient
Descent

Gradient Descent

- Almost every problem in ML starts with the same ingredients: a dataset X , a model $g(\theta)$ which is function of parameters θ , and a cost function $C(X, g(\theta))$ describing how well the model explains the observations.
- The model is fit by minimizing the cost function.
- Gradient descent is a powerful approach to do so.



- Following this approach the training procedure ensures that the parameters flow towards a minimum of the cost function.
- GD in practice is full of surprises: the cost functions in ML are usually complicated, non-convex functions in a high-dimensional space with many local minima.
- Furthermore we almost never access the true function that we wish to minimize (we do not know it at the ground truth).
- In modern applications the number of parameters to fit is often enormous (millions of parameters and examples).

Gradient Descent

- Let's call the function we want to minimize.
- This energy function can be written in terms of n data points
- In the simplest GD algorithm we iteratively update the parameters as:

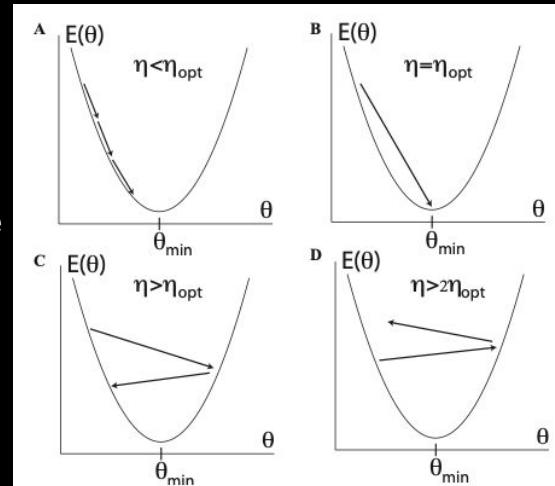
$$\mathbf{v}_t = \eta_t \nabla_{\theta} E(\boldsymbol{\theta}_t),$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

(η_t) learning rate: controls how big a step we should take in the direction of the gradient

- For sufficiently small η_t we would converge to a local minimum, but this comes at a huge computational cost
- If too large we can overshoot the minimum.
- In practice we need to specify a "schedule" that decreases η_t at long times.
- The learning rate could be adapted using the inverse of the Hessian matrix (see Newton's method), so that larger steps are taken in flat directions and smaller steps in steep directions.
- In the case of a single parameter quadratic energy function we can easily identify four regimes depending on an η_{opt} as in the figure. In a multidimensional case, one could determine the largest eigenvalue λ_{max} of the Hessian and use a single learning rate for all parameters. Convergence requires $\eta < 2/\lambda_{max}$

$$E(\boldsymbol{\theta}) = \sum_{i=1}^n e_i(\mathbf{x}_i, \boldsymbol{\theta})$$



Limitations and Alternatives: Stochastic Gradient Descent

- Limitations of the simple GD approach:

- GD finds local minima
- Gradients are computationally expensive to calculate for large datasets (sum over all data points)
- Unlike in Newton's method, GD treats all directions in the parameter uniformly. We need to keep track of Hessian but it is computationally expensive)
- GD is sensitive to initial conditions and can take exponential time to escape saddle points

- Stochastic Gradient Descent (SGD) is one of the most applied variants of the GD. The algorithm is stochastic.
- Where do we introduce stochasticity? It is incorporated by randomly selecting data points at each step to calculate the gradients. In other words, by approximating the gradient on a subset of the data called minibatch of size M (traditionally SGD was reserved for each data point, that is minibatch of size 1), $B_k = 1, \dots, n/M$ size M:(32, 64, 128, 256, ...)
- A full iteration over all n data points, i.e. n/M minibatches, is called an epoch.

SGD
algorithm

$$\begin{aligned}\mathbf{v}_t &= \eta_t \nabla_{\theta} E^{MB}(\boldsymbol{\theta}), \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \mathbf{v}_t.\end{aligned}$$

- Benefits:
 - Stochasticity reduces chance to get stuck in local minima
 - It speeds up calculation

Gradient Descent with Momentum

(*) this is demonstrated by the physical analogy with the equation of motion for a mass m moving in a viscous medium with damping coefficient and potential

- SGD is almost always used with a “momentum” or inertia term (*)

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_{\theta} E(\boldsymbol{\theta}_t) \quad (\gamma) \text{ momentum parameter}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t,$$

- \mathbf{v}_t is a running average of recently encountered gradients. It is possible to demonstrate that the characteristic timescale for the memory used in the averaging procedure is: $(1 - \gamma)^{-1}$
- Why is momentum useful? SGD momentum helps the GD algorithm gain speed in directions with persistent but small gradients even in the presence of stochasticity, while suppressing oscillations in high-curvature directions. Empirical studies show benefits in the transient phase of training, rather than during fine-tuning
- These benefits are sometimes even more pronounced in a slight modification called Nesterov Accelerated Gradient (NAG), which calculates the gradient at the expected value of the parameters

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_{\theta} E(\boldsymbol{\theta}_t + \gamma \mathbf{v}_{t-1})$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t.$$

Gradient Descent with Second Moment

- RMSprop

$$\mathbf{g}_t = \nabla_{\theta} E(\theta)$$

$$\mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1 - \beta) \mathbf{g}_t^2$$

$$\theta_{t+1} = \theta_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}},$$

$$\beta = 0.9$$

learning rate $\eta_t \sim 10^{-3}$ (can be larger than previous methods due to the adaptive step size)
 $\epsilon \sim 10^{-8}$

The learning rate is reduced in directions where the gradient is consistently large

- ADAM

$$\mathbf{g}_t = \nabla_{\theta} E(\theta)$$

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t$$

$$\mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - (\beta_1)^t}$$

$$\hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - (\beta_2)^t}$$

$$\theta_{t+1} = \theta_t - \eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{s}}_t + \epsilon}},$$

$$\begin{aligned}\beta_1 &= 0.9 \\ \beta_2 &= 0.99\end{aligned}$$

ADAM performs an additional bias correction to account for the fact that we are estimating the first two moments of the gradient using a running average (denoted with a hat)

It's possible to rewrite formulas as:

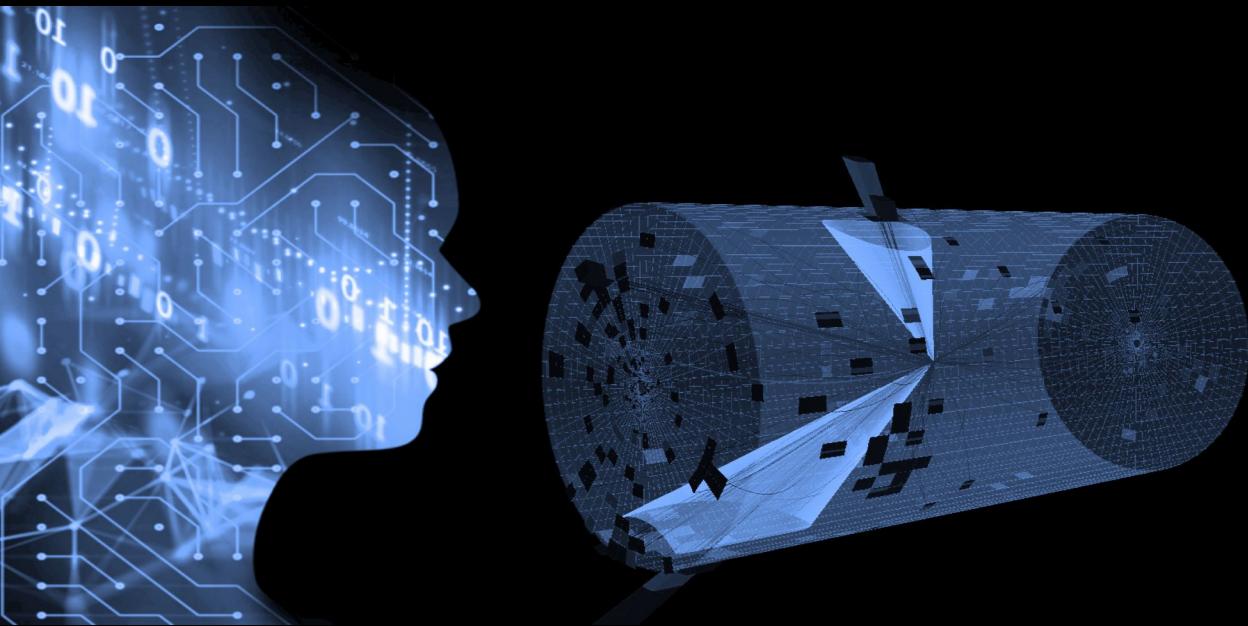
$$\sigma_t^2 = \hat{\mathbf{s}}_t - (\hat{\mathbf{m}}_t)^2$$

$$\Delta \theta_{t+1} = -\eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\sigma_t^2 + \hat{\mathbf{m}}_t^2 + \epsilon}}$$

We adapt learning rate proportional to signal-to-noise ratio, for example if widely fluctuating, $\sigma \gg m_t$

Practical tips and Further Reading

- Randomize data when making mini-batches
- Standardize inputs (learning becomes difficult when it has a mixture of steep and flat directions)
- Monitor the out-of-sample performance — validation set
- Adaptive optimization methods (ADAM, RMSprop) do not always have good generalization
 - Suffer when number of parameters exceeds number of data points [1]
 - Outperform with deep networks such as generative adversarial networks [2]



Due to limited time no exercises on this part. Notebooks are available at [[hblvi2m](#)].

Regression (in a nutshell)

Regression: the problem

scalar response

- Given a dataset with n samples $\mathcal{D} = \{(y_i, \mathbf{x}^{(i)})\}_{i=1}^n$
- Assume every sample has p features $\mathbf{x}^{(i)} \in \mathbb{R}^p$
- Let f be the true function that generated these samples via
where $\mathbf{w}_{\text{true}} \in \mathbb{R}^p$ and ϵ_i is some i.i.d. white noise with zero mean and finite variance.

- One can cast the samples into an $n \times p$ matrix, called the design matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$
with rows $\mathbf{X}_{i,:} = \mathbf{x}^{(i)} \in \mathbb{R}^p$, $i = 1, \dots, n$ being observations
and the columns $\mathbf{X}_{:,j} \in \mathbb{R}^n$, $j = 1, \dots, p$ being the measured features
- The function f is unknown explicitly and we presume its functional form.
- For example, in linear regression we assume $y_i = f(\mathbf{x}^{(i)}; \mathbf{w}_{\text{true}}) + \epsilon_i = \mathbf{w}_{\text{true}}^T \mathbf{x}^{(i)} + \epsilon_i$
for some unknown but fixed $\mathbf{w}_{\text{true}} \in \mathbb{R}^p$

Linear Regression

- We want to find a function g with parameters \mathbf{w} fitting to the data D that can best approximate f

$$g(\mathbf{x}; \hat{\mathbf{w}}) \sim f$$

- When this is done, we can use this g to make predictions about the response y_0 for a new data point \mathbf{x}_0 .
- Let's introduce the L_p norm which is helpful in regression. For any real number $p \geq 1$, we define the L^p norm of a vector $\mathbf{x} = (x_1, \dots, x_d) \in \mathbb{R}^d$ to be

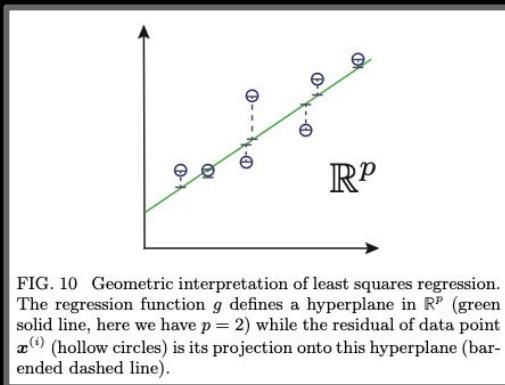
$$\|\mathbf{x}\|_p = (|x_1|^p + \dots + |x_d|^p)^{\frac{1}{p}}$$

Least-square Regression

- Ordinary least squares linear regression (OLS) is defined as the minimization of the L_2 norm of the difference between the response y_i and the predictor $g(x^{(i)}; w) = w^T x^{(i)}$:

$$\min_{w \in \mathbb{R}^p} \| \mathbf{X}w - \mathbf{y} \|_2^2 = \min_{w \in \mathbb{R}^p} \sum_{i=1}^n (w^T x^{(i)} - y_i)^2$$

- We need to determine w that minimizes the L^2 error
- Geometrically speaking, the predictor $g(x^{(i)}; w) = w^T x^{(i)}$ defines an hyperplane in \mathbb{R}^p .



- This leads to the solution $\hat{w}_{LS} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$, where we assume $\mathbf{X}^T \mathbf{X}$ is invertible, which is often the case when $n \geq p$. Formally if $\text{rank}(\mathbf{X}) = p$, the columns of \mathbf{X} are linearly independent, then w_{LS} is unique.
- When $\text{rank}(\mathbf{X}) < p$ $\mathbf{X}^T \mathbf{X}$ is singular, implying there are infinitely many solutions to the least-squares problem (if w_0 is a solution, $w_0 + \eta$ is also a solution for any η such that $\mathbf{X}\eta = 0$).

Least-square Regression

- Having determined the least squares solution, we can calculate the best fit to our data as:

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}_{\text{LS}} = P_{\mathbf{X}}\mathbf{y}, \text{ where } P_{\mathbf{X}} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$$

- Notice that we found the solution \mathbf{w}_{LS} in one shot, without any iterative optimization as seen e.g. for gradient descent.

- As already discussed the difference between **learning** and **fitting** lies in the prediction on unseen data. It is therefore necessary to examine the out-of-sample error. The reader can find Jupyter notebooks in [[hblvi2ml](#)]
- It can be shown that the average errors are:

$$\left\{ \begin{array}{l} \bar{E}_{\text{in}} = \sigma^2 \left(1 - \frac{p}{n}\right) \\ \bar{E}_{\text{out}} = \sigma^2 \left(1 + \frac{p}{n}\right) \end{array} \right. \rightarrow |\bar{E}_{\text{in}} - \bar{E}_{\text{out}}| = 2\sigma^2 \frac{p}{n}$$

(provided we obtain the least squares solution \mathbf{w}_{LS} from i.i.d. samples \mathbf{X} and \mathbf{y} generated through $\mathbf{y} = \mathbf{X}\mathbf{w}_{\text{true}} + \epsilon$)

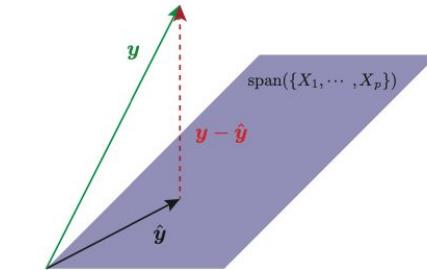


FIG. 11 The projection matrix $P_{\mathbf{X}}$ projects the response vector \mathbf{y} onto the column space spanned by the columns of \mathbf{X} , $\text{span}(\{\mathbf{X}_{:,1}, \dots, \mathbf{X}_{:,p}\})$ (purple area), thus forming a fitted vector $\hat{\mathbf{y}}$. The residuals in Eq. (37) are illustrated by the red vector $\mathbf{y} - \hat{\mathbf{y}}$.

If we have $p \gg n$ (i.e. high-dimensional data) the generalization error is extremely large and the model is not learning. Even when $p \approx n$ we might still not learn well due to the intrinsic noise σ^2 .

To ameliorate this we use regularization and we will briefly mention two forms:
Ridge (L_2 penalty) and LASSO (L_1 penalty).

Ridge-Regression

- We add to the least squares loss function a regularizer defined as L_2 norm of the parameter vector we wish to optimize over. The **penalized Ridge regression** problem is:

$$\hat{\mathbf{w}}_{\text{Ridge}}(\lambda) = \arg \min_{\mathbf{w} \in \mathbb{R}^p} (\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_2^2) \quad (1)$$

- Equivalent to the constrained optimization problem:

$$\hat{\mathbf{w}}_{\text{Ridge}}(\lambda) = (\mathbf{X}^T \mathbf{X} + \lambda I_{p \times p})^{-1} \mathbf{X}^T \mathbf{y}$$

$$\hat{\mathbf{w}}_{\text{Ridge}}(t) = \arg \min_{\mathbf{w} \in \mathbb{R}^p: \|\mathbf{w}\|_2^2 \leq t} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 \quad (2)$$

meaning for any $t \geq 0$ and solution $\mathbf{w}_{\text{Ridge}}$ of (2), there exists a $\lambda \geq 0$ such that $\mathbf{w}_{\text{Ridge}}$ solves (1) — and vice versa. With the regularization term we are constraining the magnitude of the parameter vector learnt from the data.

- What's the difference?

$$\hat{\mathbf{w}}_{\text{Ridge}}(\lambda) = \frac{\hat{\mathbf{w}}_{\text{LS}}}{1 + \lambda}$$

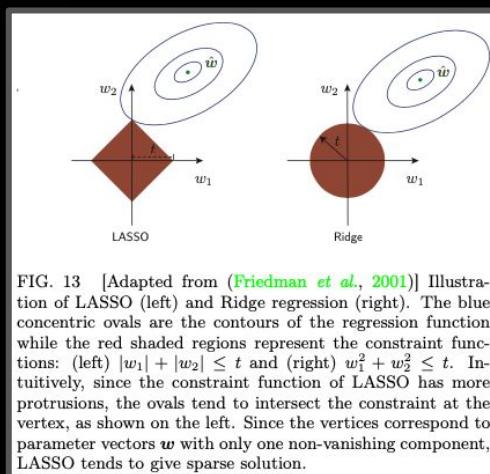
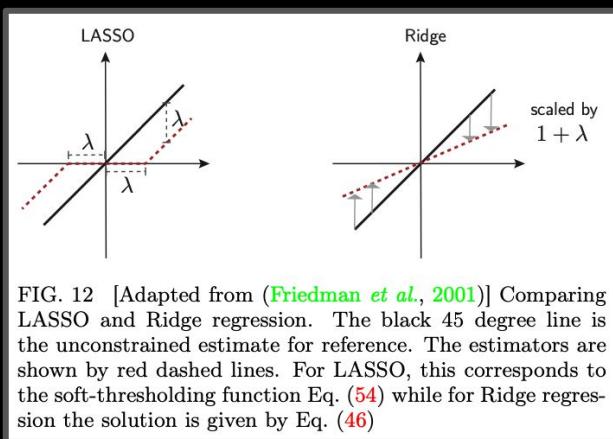
It is possible to demonstrate that both Ridge and LS linear regression have to project \mathbf{y} to the column space of \mathbf{X} . Ridge further shrinks each basis component by a factor $d_j^2/(d_j^2+\lambda^2)$, as can be obtained from singular value decomposition.

- LASSO in the penalized form is defined by

$$\hat{\mathbf{w}}_{\text{LASSO}}(\lambda) = \arg \min_{\mathbf{w} \in \mathbb{R}^p} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_1 \quad (1)$$

- Assuming \mathbf{X} is orthogonal, it is possible to demonstrate that:

$$\hat{w}_j^{\text{LASSO}}(\lambda) = \text{sign}(\hat{w}_j^{\text{LS}})(|\hat{w}_j^{\text{LS}}| - \lambda)_+ \quad \text{where } (\cdot)_+ \text{ indicates the positive part and } w_j^{\text{LS}} \text{ is the } j\text{-th component of the LS solution.}$$



LASSO vs Ridge

LASSO tends to give sparse solutions, i.e. many components of $\mathbf{w}_{\text{LASSO}}$ are zero. The L1 regularizer of LASSO has sharp protrusions, the intersection of the regressor contours tend to occur at the vertex of the feasibility region, implying the solution vector will be sparse.

```
import numpy as np
import matplotlib.pyplot as plt
#import seaborn

from sklearn import datasets, linear_model

# Load Training Data set with 200 examples

number_examples=200
diabetes = datasets.load_diabetes()
X = diabetes.data[:number_examples]
y = diabetes.target[:number_examples]

# Set up Lasso and Ridge Regression models
ridge=linear_model.Ridge()
lasso = linear_model.Lasso()
```

```
n_samples = 150
n_samples_train = 100
X_train, X_test = X[:n_samples_train], X[n_samples_train:]
y_train, y_test = y[:n_samples_train], y[n_samples_train:]
train_errors_ridge = list()
test_errors_ridge = list()

train_errors_lasso = list()
test_errors_lasso = list()

# Initialize coefficients for ridge regression and Lasso

coefs_ridge = []
coefs_lasso=[]
for a in alphas:
    ridge.set_params(alpha=a)
    ridge.fit(X_train, y_train)
    coefs_ridge.append(ridge.coef_)

    # Use the coefficient of determination R^2 as the performance of prediction.
    train_errors_ridge.append(ridge.score(X_train, y_train))
    test_errors_ridge.append(ridge.score(X_test, y_test))

    lasso.set_params(alpha=a)
    lasso.fit(X_train, y_train)
    coefs_lasso.append(lasso.coef_)
    train_errors_lasso.append(lasso.score(X_train, y_train))
    test_errors_lasso.append(lasso.score(X_test, y_test))

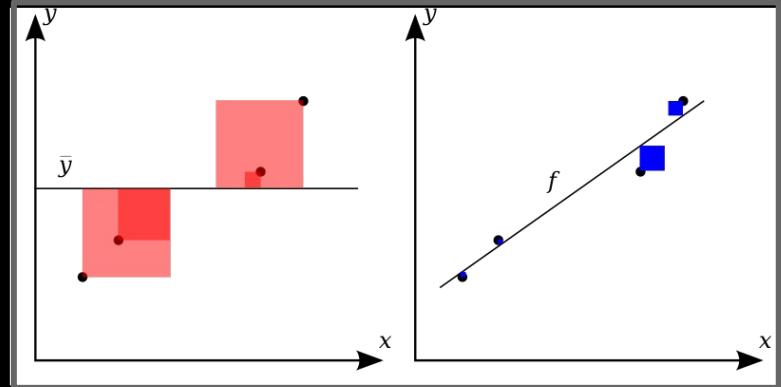
#####
#####
```

Coefficient of Determination

$$R^2 = 1 - \frac{\sum_{i=1}^n |y_i^{\text{true}} - y_i^{\text{pred}}|^2}{\sum_{i=1}^n |y_i^{\text{true}} - \frac{1}{n} \sum_{i=1}^n y_i^{\text{pred}}|^2}$$

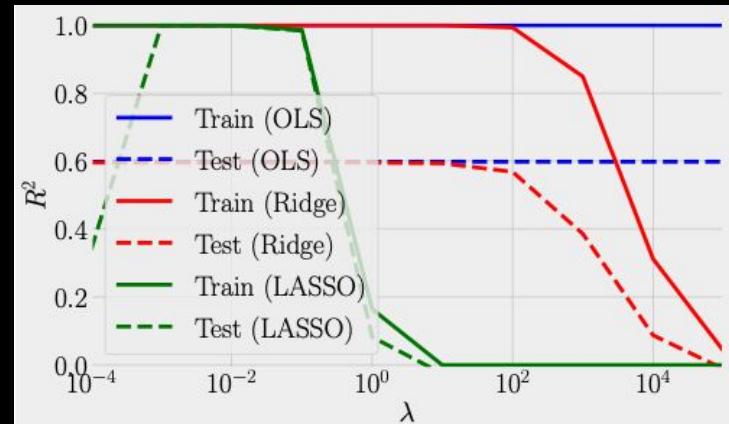
SS_{res} SS_{tot}

- In the best case, the modeled values exactly match the observed values, which results in $\text{SS}_{\text{res}} = 0$ and $R^2=1$.
- A baseline model which always predicts the average y , will have $R^2=0$.
- Models that have worse predictions than this baseline will have a negative R^2 .
- The regularization parameter λ affects Ridge and LASSO regressions. A good practice is check the performance as a function of λ .



The better the linear regression (on the right) fits the data in comparison to the simple average (on the left graph), the closer the value of R^2 is to 1.

The areas of the blue squares represent the squared residuals with respect to the linear regression. The areas of the red squares represent the squared residuals with respect to the average value. Taken from [here](#).



Example taken from [\[hblvi2m\]](#) for the Isospin regression problem. Notice that LASSO test curve is not monotonic and there is a sweet spot.

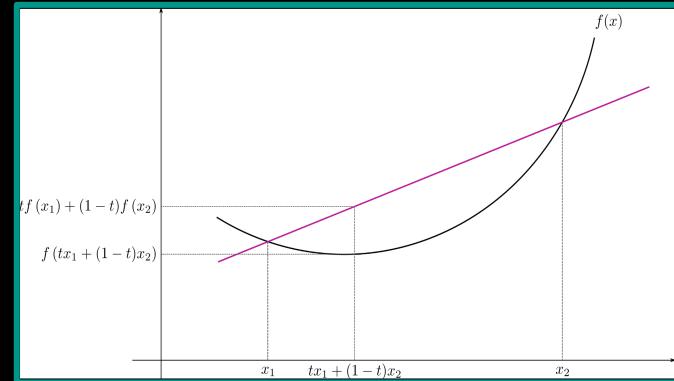
Convexity and elastic net

- A simple inspection reveals that both LASSO and Ridge regression are convex in \mathbf{w} .
- A function $\mathbb{R}^n \rightarrow \mathbb{R}$ is called convex if its domain is a convex set and for any x, y in the domain, and $t \in [0, 1]$:

$$f(tx + (1 - t)y) \leq tf(x) + (1 - t)f(y)$$

- For convex functions, any local minimizer is a global minimizer: as long as we're "going down the hill" and we stop when we can't go any further, then we've hit the global minimum.
- Ridge is actually a *strictly convex* problem (assuming $\lambda > 0$) due to presence of L_2 penalty. In fact, this is always true regardless of X and the solution is always well-defined.
- LASSO is not always strictly convex and hence by convexity theory, it need not have a unique solution. The LASSO solution is unique under general conditions, e.g., when X has columns in *general position* (see [Tibshirani 2013](#)). To mitigate this, one can define a modified problem called the elastic net such that the function we want to minimize is always strictly convex:

$$\min_{\mathbf{w} \in \mathbb{R}^p} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_1 + \delta \|\mathbf{w}\|_2^2$$



The elastic net combines some of the desirable properties of Ridge regression (e.g., prediction) with the sparsity properties of LASSO.

Logistic regression

- Problems like classification are concerned with outcomes taking discrete variables (i.e. categories):
 - Signal vs bkgd, Phase of a system, etc
- Logistic regression deals with dichotomous outcome (True or False, 1 or 0, etc)
- The inner workings of logistic regression are valuable in the study of modern supervised deep neural networks
- In what follows:
 - Define logistic regress
 - Derive cost function (cross entropy) using a Bayesian approach
 - Discuss its minimization
 - Generalize logistic regression in the case of multiple categories (called SoftMax regression)
- We focus on the SUSY dataset of [[hblvi2ml](#)]

Categorize

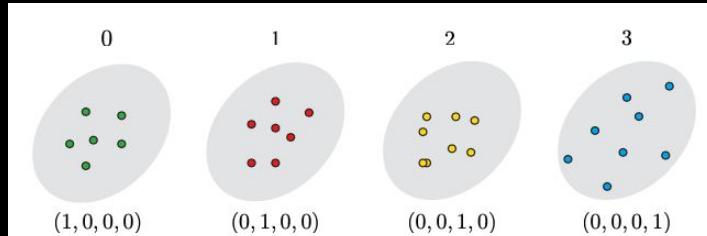
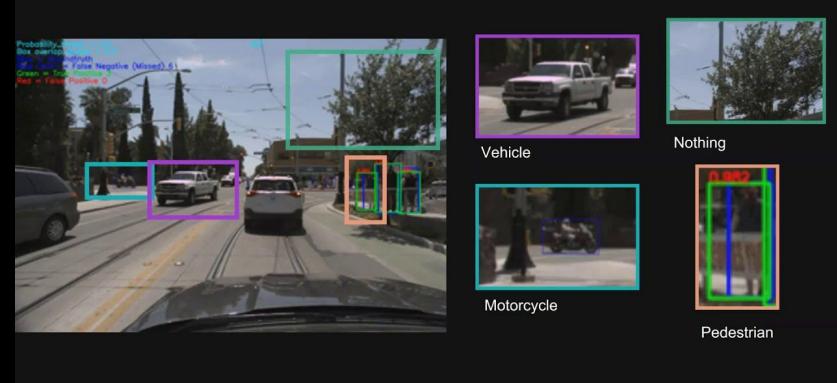


FIG. 18 Pictorial representation of four data categories labeled by the integers 0 through 3 (above), or by one-hot vectors with binary inputs (below).



- Before delving into logistic regression, it is helpful to consider a simple linear classifier that categorizes examples using a weighted linear-combination of the features and an additive offset

$$s_i = \mathbf{x}_i^T \mathbf{w} + b_0 \equiv \mathbf{x}_i^T \mathbf{w}$$

- We use the short-hand notation $\mathbf{x}_i = (1, \mathbf{x}_i) \quad \mathbf{w} = (b_0, \mathbf{w})$

This function takes values on the entire real axis. In the case of logistic regression, however, the labels y_i are discrete variables.

- One could use a sign function for a binary classifier:

$$\{0, 1\}, \sigma(s_i) = \text{sign}(s_i) = 1 \text{ if } s_i \geq 0$$

commonly known as the [Perceptron](#)

Logistic (sigmoid) function

- Perceptron is an example of “hard classification”, i.e. each datapoint is assigned a category (0 or 1)
- It is favorable in many cases to have a “soft classifier”, that outputs the probability of a given category (given x_i , the classifier returns the probability of being in category m)
- Logistic (or sigmoid) function: $\sigma(s) = \frac{1}{1 + e^{-s}}$. where $1 - \sigma(s) = \sigma(-s)$
- In logistic regression, the probability that a point x_i belongs to a category $y_i = \{0,1\}$ is given by:

$$P(y_i = 1 | \mathbf{x}_i, \boldsymbol{\theta}) = \frac{1}{1 + e^{-\mathbf{x}_i^T \boldsymbol{\theta}}},$$

Here $\boldsymbol{\theta}$ are the weights we want to learn

$$P(y_i = 0 | \mathbf{x}_i, \boldsymbol{\theta}) = 1 - P(y_i = 1 | \mathbf{x}_i, \boldsymbol{\theta})$$

Cross-entropy as cost function for logistic regression

- We now use Maximum Likelihood Estimation (MLE) to define the cost function for logistic regression. In MLE we choose parameters to maximize the probability of seeing the observed data.
- Consider a dataset $D = \{(y_i, x_i)\}$ with binary labels $y_i \in \{0, 1\}$ from which the data points are drawn independently. The likelihood of observing the data under our model is:

$$P(D|\mathbf{w}) = \prod_{i=1}^n [\sigma(\mathbf{x}_i^T \mathbf{w})]^{y_i} [1 - \sigma(\mathbf{x}_i^T \mathbf{w})]^{1-y_i}$$

- We can compute the log-likelihood:
$$l(\mathbf{w}) = \sum_{i=1}^n y_i \log \sigma(\mathbf{x}_i^T \mathbf{w}) + (1 - y_i) \log [1 - \sigma(\mathbf{x}_i^T \mathbf{w})]$$
- The maximum likelihood estimator is defined as:
- $$\hat{\mathbf{w}} = \arg \max_{\theta} \sum_{i=1}^n y_i \log \sigma(\mathbf{x}_i^T \mathbf{w}) + (1 - y_i) \log [1 - \sigma(\mathbf{x}_i^T \mathbf{w})]$$
- The **cost (error) function** for logistic regression is defined as the negative log-likelihood:

$$\begin{aligned} \mathcal{C}(\mathbf{w}) &= -l(\mathbf{w}) && \text{commonly known as the Cross-entropy} \\ &= \sum_{i=1}^n -y_i \log \sigma(\mathbf{x}_i^T \mathbf{w}) - (1 - y_i) \log [1 - \sigma(\mathbf{x}_i^T \mathbf{w})] \end{aligned}$$

We note that, just as in linear regression, in practice we usually supplement the cross-entropy with additional regularization terms, usually L_1 and L_2 regularization

Minimization of the cross-entropy

- The cross-entropy is a convex function of the weights w and therefore any local minimizer is a global minimizer.
- Minimizing the cost function leads to:

$$\mathbf{0} = \nabla \mathcal{C}(\mathbf{w}) = \sum_{i=1}^n [\sigma(\mathbf{x}_i^T \mathbf{w}) - y_i] \mathbf{x}_i$$

where we made use of the logistic function identity $\partial \sigma(s) = \sigma(s)[1-\sigma(s)]$

- The cross-entropy is a convex function of the weights w and therefore any local minimizer is a global minimizer.
- The above is a transcendental equation for \mathbf{w} , the solution of which unlike linear regression, cannot be written in closed form. This is the case where we need to utilize the numerical methods previously introduced, such as gradient descent.
- Notice that Scikit's logistic regression solvers have in-built regularizers. Their role is fundamental in general to prevent overfitting.

SoftMax Regression

- So far we focused only on binary classification. We can generalize logistic regression to multi-class classification (multinomial logistic regression). The softmax in machine learning is related to the Boltzmann distribution in physics:

$$p_i = \frac{1}{Q} e^{-\varepsilon_i/(kT)} = \frac{e^{-\varepsilon_i/(kT)}}{\sum_{j=1}^M e^{-\varepsilon_j/(kT)}}$$

p_i is the probability of a state;
M is the number of accessible states

- In ML, can be formulated as:

$$P(y_{im'} = 1 | \mathbf{x}_i, \{\mathbf{w}_k\}_{k=0}^{M-1}) = \frac{e^{-\mathbf{x}_i^T \mathbf{w}_{m'}}}{\sum_{m=0}^{M-1} e^{-\mathbf{x}_i^T \mathbf{w}_m}}$$

where $\mathbf{y}_i = (1, 0, \dots, 0)$ means the sample x_i belongs to class 1, and $y_{im'}$ is the m'-th component of the vector \mathbf{y}_i . From the above, it's possible to build a likelihood and define the cost:

$$\begin{aligned} \mathcal{C}(\mathbf{w}) = & - \sum_{i=1}^n \sum_{m=0}^{M-1} y_{im} \log P(y_{im} = 1 | \mathbf{x}_i, \mathbf{w}_m) \\ & + (1 - y_{im}) \log (1 - P(y_{im} = 1 | \mathbf{x}_i, \mathbf{w}_m)) \end{aligned}$$

Notice that for M=1 we recover the cross-entropy of the logistic regression.

Practical tips and Further Reading

- OLS can be optimized with gradient descent, Newton's method, or in closed form.

Ordinary Least Squares:

- $\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i^\top \mathbf{w} - y_i)^2$.
- Squared loss.
- No regularization.
- Closed form: $\mathbf{w} = (\mathbf{X}\mathbf{X}^\top)^{-1}\mathbf{X}\mathbf{y}^\top$.

Ridge Regression:

- $\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i^\top \mathbf{w} - y_i)^2 + \lambda \|\mathbf{w}\|_2^2$.
- Squared loss.
- $l2$ -regularization.
- Closed form: $\mathbf{w} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I})^{-1}\mathbf{X}\mathbf{y}^\top$.

- Ridge has a closed form too.
- Read scikit-learn documentation to see details on implementation
- “Polynomial” regression
- The most commonly used loss function for Linear Regression is **Least Squared Error**, and its cost function is also known as Mean Squared Error(MSE) [The terms cost and loss functions almost refer to the same meaning. But, loss function mainly applies for a single training set as compared to the cost function which deals with a penalty for a number of training sets or the complete batch]

[1] <https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote08.html>

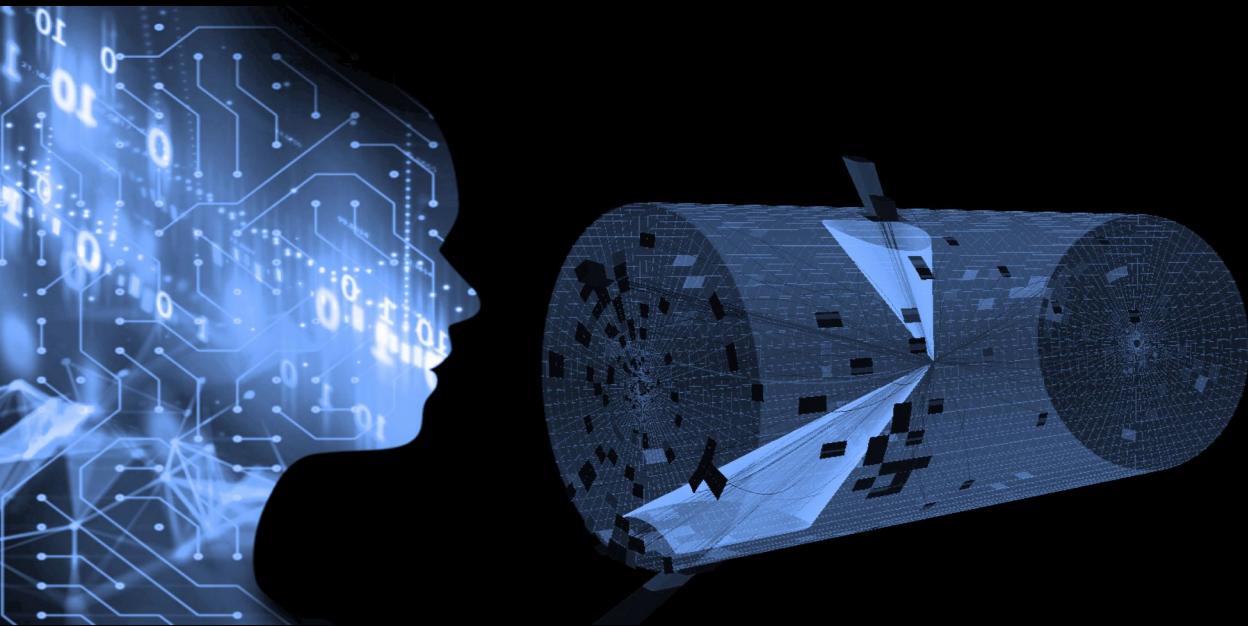
[2] <https://towardsdatascience.com/optimization-of-supervised-learning-loss-function-under-the-hood-df1791391c82>

Practical tips and Further Reading

- Categories: one-hot encoder One-hot encoding **ensures that machine learning does not assume that higher numbers are more important**. For example, the value '8' is bigger than the value '1', but that does not make '8' more important than '1'. The same is true for words: the value 'laughter' is not more important than 'laugh'.
- <https://pytorch.org/docs/master/generated/torch.nn.CrossEntropyLoss.html> Therefore, there's no need to one-hot encode your data if you have the labels already provided
- Logistic regression is easier to implement, interpret, and very efficient to train. **If the number of observations is lesser than the number of features**, Logistic Regression should not be used, otherwise, it may lead to overfitting. Logistic regression can be thought as a special case of NN with no hidden layer, that uses the sigmoid activation function and softmax with cross-entropy loss.
- A full fledged NN with hidden layers (deep network) and non-linear activation functions allows to capture highly complex functions of the features that could be characteristic of several problems.
- There are other techniques (e.g., SVMs not covered here) which can capture nonlinear functions. But NNs are popular because there are highly evolved and scalable platforms to capture more and more complex relationships by easily constructing a deep network and feeding in a lot of data

[1] <https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote08.html>

[2] <https://towardsdatascience.com/optimization-of-supervised-learning-loss-function-under-the-hood-df1791391c82>



Combining
Models

Ensemble Methods

- One of the most powerful and widely-applied ideas in modern ML. Ensemble methods correspond to combining predictions from multiple — often weak — classifiers to improve the predictive performance.
- Also in the context of NN is common to combine the predictions from multiple NNs, e.g., in complicated image classification problems.
- The key to determining when ensemble methods work is the degree of correlation between the models [1].
- Largely used ensemble methods are:
 - Boosting
 - Random forest
 - Gradient boosted trees (e.g., XGBoost)

Importance/role of correlation* between models

1. Holding the ensemble size fixed, averaging the predictions of correlated models reduces the variance less than averaging uncorrelated models.
2. In some cases, correlations between models within an ensemble can result in an increase in bias, offsetting any potential reduction in variance gained from ensemble averaging.

One of the most dramatic examples of increased bias from correlations is the catastrophic predictive failure of all derivative models used by Wall Street during the 2008 financial crisis.

[1] Louppe, Gilles (2014). “Understanding random forests: From theory to practice,” arXiv preprint arXiv:1407.7502

*E.g., the correlation coefficient between the predictions made by two randomized models based on the same training set but with different random seeds [hblvi2ml]

Bias-Variance Decomposition for Ensembles

- The bias-variance tradeoff is discussed in the context of continuous predictions like regression, but many intuitions apply also for classification tasks.
- Let's recall the bias-variance tradeoff for a single model first.
- Consider the dataset $\mathbf{X}_{\mathcal{L}} = \{(y_j, \mathbf{x}_j), j = 1 \dots N\}$ and let's assume is generated from a noisy model $y = f(\mathbf{x})$ where noise is normally distributed with mean zero and st. deviation σ_{ϵ} .
- Assume we have a predictor $\hat{g}_{\mathcal{L}}(\mathbf{x})$ that gives a prediction of our model for a data point \mathbf{x} . The estimator can be chosen by minimizing the cost function squared error:

$$\mathcal{C}(\mathbf{X}, g(\mathbf{x})) = \sum_i (\mathbf{y}_i - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2$$

- We already showed that the expected generalization error can be decomposed as:

$$\mathbb{E}_{\mathcal{L}, \epsilon}[\mathcal{C}(\mathbf{X}, g(\mathbf{x}))] = Bias^2 + Var + Noise.$$

- What happens now if we have an **aggregate ensemble predictor**?

$$\hat{g}_{\mathcal{L}}^A(\mathbf{x}_i, \{\theta\}) = \frac{1}{M} \sum_{m=1}^M \hat{g}_{\mathcal{L}}(\mathbf{x}_i, \theta_m)$$

M is the dimension of the ensemble
We assume that θ parametrize members of the ensemble

θ can be seen as the hyper-parameters of models, they introduce stochasticity in the ensemble

Bias-Variance Decomposition for Ensembles

- What happens now if we have an **aggregate ensemble predictor**?

$$\hat{g}_{\mathcal{L}}^A(\mathbf{x}_i, \{\theta\}) = \frac{1}{M} \sum_{m=1}^M \hat{g}_{\mathcal{L}}(\mathbf{x}_i, \theta_m)$$

M is the dimension of the ensemble

- For a thorough derivation see [[hblvi2ml](#), Sec. VIII].

$$Var(\mathbf{x}) = \mathbb{E}_{\mathcal{L}, \theta}[(\hat{g}_{\mathcal{L}}^A(\mathbf{x}, \{\theta\}) - \mathbb{E}_{\mathcal{L}, \theta}[\hat{g}_{\mathcal{L}}^A(\mathbf{x}, \{\theta\})])^2]$$

Intuitively this derives from the variance of n correlated variables:

$$\begin{aligned} Var(X) &= \sigma^2/n + (n-1)/n \rho \sigma^2 \\ &= \rho \sigma^2 + (1-\rho)/n \sigma^2 \end{aligned}$$

$$\begin{aligned} &= \frac{1}{M^2} \left[\sum_{m, m'} \mathbb{E}_{\mathcal{L}, \theta}[\hat{g}_{\mathcal{L}}(\mathbf{x}, \theta_m) \hat{g}_{\mathcal{L}}(\mathbf{x}, \theta_{m'})] - M^2 \sum_i [\mu_{\mathcal{L}, \theta}(\mathbf{x})]^2 \right] \\ &= \rho(\mathbf{x}) \sigma_{\mathcal{L}, \theta}^2 + \frac{1 - \rho(\mathbf{x})}{M} \sigma_{\mathcal{L}, \theta}^2. \end{aligned}$$

$$\begin{aligned} Bias^2(\mathbf{x}) &= (f(\mathbf{x}) - \mathbb{E}_{\mathcal{L}, \theta}[\hat{g}_{\mathcal{L}}^A(\mathbf{x}, \{\theta\})])^2 \\ &= (f(\mathbf{x}) - \frac{1}{M} \sum_{m=1}^M \mathbb{E}_{\mathcal{L}, \theta}[\hat{g}_{\mathcal{L}}(\mathbf{x}, \theta_m)])^2 \\ &= (f(\mathbf{x}) - \mu_{\mathcal{L}, \theta})^2. \end{aligned}$$

Expectations are computed over the joint distribution of datasets \mathcal{L} and hyperparameters θ

- Variance of the aggregate estimator depends on their correlation
- For large ensemble ($M \rightarrow \infty$) variance is significantly reduced, and for completely random ensembles ($\rho(\mathbf{x})=0$) it is maximally suppressed!
- Bias of the aggregate predictor is just the expected bias of a single model

Intuitions behind ensembles

Ensembles are successful for the following reasons:

1. Statistical: provided their predictions are uncorrelated, averaging several models reduces the risk of choosing the wrong hypothesis.
2. Computational: many learning algorithms rely on some greedy assumption or local search that may get stuck in local optima. An ensemble made of individual models built from many different starting points may provide a better approximation of the true unknown function
3. Representational power (expressivity): for a learning set of finite size, the true function cannot be represented by any of the candidate models in the hypothesis H. By combining several models in an ensemble, it may be possible to expand the space of representable functions and to better model the true function.

Using an ensemble allows one to reduce the variance by averaging the result of many independent classifiers. This procedure works best for unstable predictors for which errors are dominated by variance due to finite sampling.

Bagging

Bootstrap AGGregation

- Imagine to have a large dataset \mathcal{L} that we could partition into M smaller data sets $\{\mathcal{L}_1, \mathcal{L}_2, \dots, \mathcal{L}_M\}$. If each is sufficiently large to create a predictor, we can create an ensemble aggregate predictor:

$$\hat{g}_{\mathcal{L}}^A(\mathbf{x}) = \frac{1}{M} \sum_{i=1}^M g_{\mathcal{L}_i}(\mathbf{x})$$

We know this can significantly reduce variance without increasing bias.
But we need to have enough data in each partition.

- For classification tasks where each predictor predicts a class label $j \in \{1, \dots, J\}$, this corresponds to a majority vote:

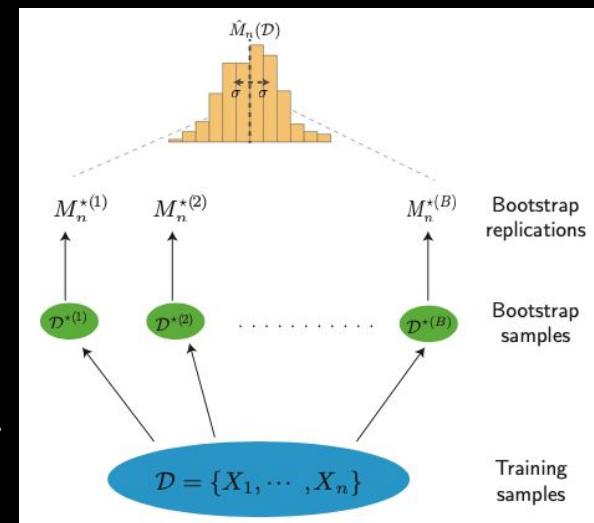
$$\hat{g}_{\mathcal{L}}^A(\mathbf{x}) = \arg \max_j \sum_{i=1}^M I[g_{\mathcal{L}_i}(\mathbf{x}) = j]$$

- This can be circumvented with empirical bootstrapping, that is with new bootstrapped datasets

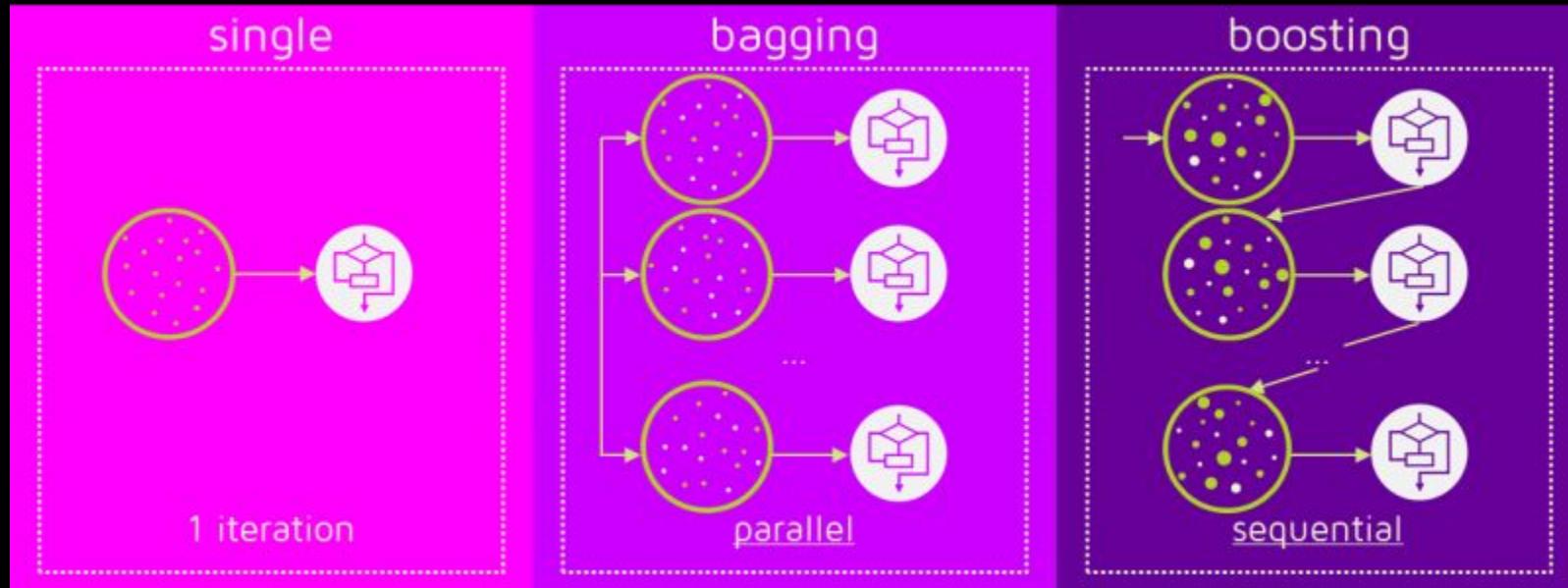
$$\{\mathcal{L}^{BS}_1, \mathcal{L}^{BS}_2, \dots, \mathcal{L}^{BS}_M\}$$

- Resampling with replacement from the original dataset

- The price we pay for using bootstrapped training datasets as opposed to really partitioned dataset is an increase in the bias of our bagged estimators.



Boosting



Boosting is another powerful ensemble method.

In bagging, the contribution of all predictors is weighted equally in the bagged (aggregate) predictor. In some problems instead one might prefer to use an “autocratic approach” that emphasizes the best predictors.

Boosting

- In boosting an ensemble of weak classifiers is combined into a boosted classifier. Each classifier is associated with a weight α_k (such that $\sum_k \alpha_k = 1$) that indicates how much it contributes.

$$g_A(\mathbf{x}) = \sum_{K=1}^M \alpha_k g_k(\mathbf{x})$$

- AdaBoost (Adaptive Boosting) is a popular technique. The aggregate classifier is formed in an iterative process.

We construct the boosted classifier as follows:

- Initialize $w_{t=1}(\mathbf{x}_n) = 1/N, n = 1, \dots, N$.
- For $t = 1 \dots, T$ (desired termination step), do:
 1. Select a hypothesis $g_t \in \mathcal{H}$ that minimizes the weighted error

$$\epsilon_t = \sum_{i=1}^N w_t(\mathbf{x}_i) \mathbb{1}(g_t(\mathbf{x}_i) \neq y_i) \quad (105)$$

2. Let $\alpha_t = \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$, update the weight for each data \mathbf{x}_n by

$$w_{t+1}(\mathbf{x}_n) \leftarrow w_t(\mathbf{x}_n) \frac{\exp[-\alpha_t y_n g_t(\mathbf{x}_n)]}{Z_t},$$

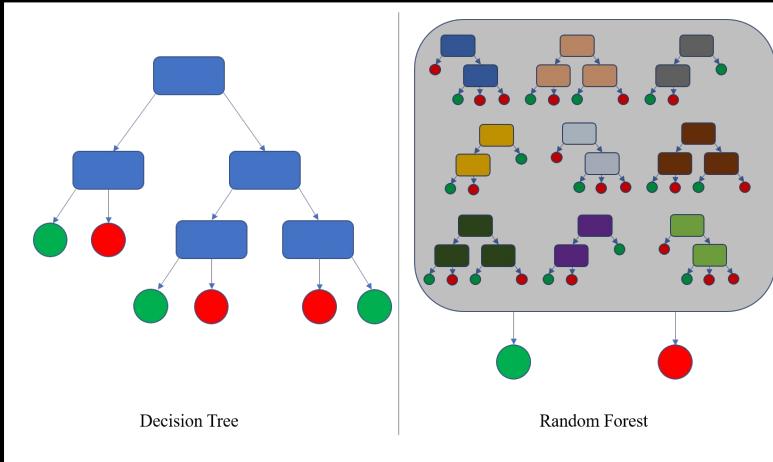
where $Z_t = \sum_{n=1}^N w_t(\mathbf{x}_n) e^{-\alpha_t y_n g_t(\mathbf{x}_n)}$ ensures all weights add up to unity.

- Output $g_A(\mathbf{x}) = \text{sign} \left(\sum_{t=1}^T \alpha_t g_t(\mathbf{x}) \right)$

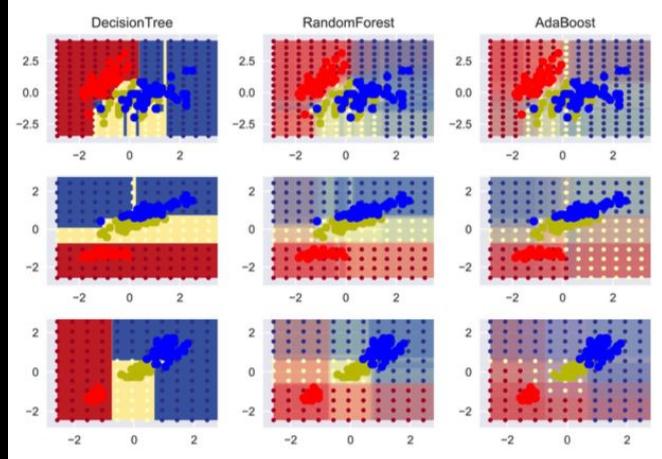
pseudo-code

Random Forest

- A decision tree uses a series of questions to hierarchically partition the data.
- A random forest is composed of a family of (randomized) tree-based classifier decision trees
- It is clear that more complex decision trees lead to finer partitions that give improved performance on the training set. However, this generally leads to overfitting, limiting the out-of-sample performance.
- In order to create an ensemble of decision trees, we must introduce a randomization procedure (the power of ensembles to reduce variance only manifests when randomness reduces correlations). Three ways for randomness
 - (i) Bag, (ii) feature bagging, (iii) extremized random forests (combination)



Comparison of decision surfaces
(each row a subset of 2 features, Iris problem)



Gradient Boosted Trees and XGBoost

- Gradient boosted trees combine boosting and gradient descent (in particular Newton's method) to construct ensemble of decision trees
- Ensemble are created iteratively. A cost function measures the performance of the ensemble. At each step we compute the gradient of the ensemble and add trees that move in the direction of the gradient.
- Extreme Gradient Boosting (XGBoost) is a particular technique

aggregate prediction $\hat{y}_i = g_A(\mathbf{x}_i) = \sum_{j=1}^M g_j(\mathbf{x}_i), \quad g_j \in \mathcal{F}$ space of tree

cost function $\mathcal{C}(\mathbf{X}, g_A) = \sum_{i=1}^N l(y_i, \hat{y}_i) + \sum_{j=1}^M \Omega(g_j)$

Goodness of prediction (convex and differentiable)

Regularization term
Penalizes large weights on the leaves and large partitions with many leaves

$$\Omega(g) = \gamma T + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

Ensemble is formed

$$\hat{y}_i^{(t)} = \sum_{j=1}^t g_j(\mathbf{x}_i) = \hat{y}_i^{(t-1)} + g_t(\mathbf{x}_i)$$

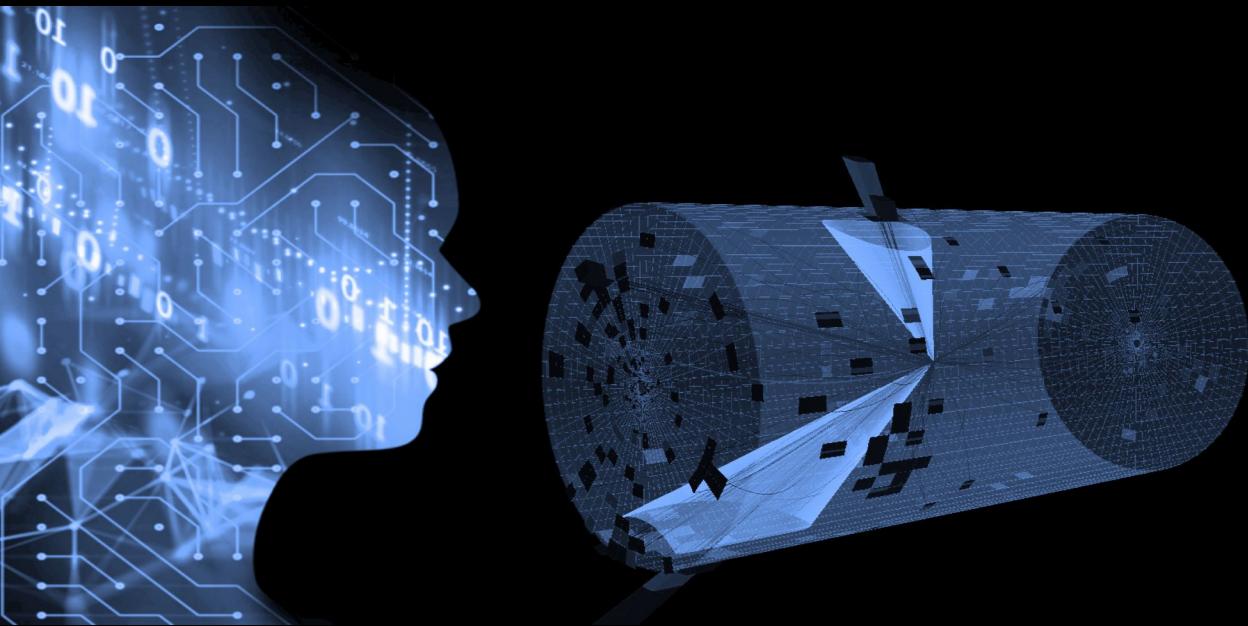
Ensemble Idea: for large t , each decision tree is a small perturbation to the predictor

An approximate greedy algorithm is run to optimize one level of the tree at a time to find optimal splits of the data. Additional regularization such as shrinkage and feature subsampling are also used.

Further Reading

- **Structured vs unstructured data:** Unstructured data can be information that is not arranged according to a pre-set data model or schema. Examples of structured data include names, dates, addresses, credit card numbers, stock information, geolocation, and more. Structured data is highly organized. Ensemble methods perform well especially on structured datasets. Neural networks generally perform better than ensemble methods on unstructured data, images, and audio.
- Feature importance

<https://machinelearningmastery.com/calculate-feature-importance-with-python/>



Clustering
in a nutshell

K-Means

hard-clustering

STEP 1: Choose the number K of clusters

STEP 2: Select at random K points, the centroids
(not necessarily from your dataset)

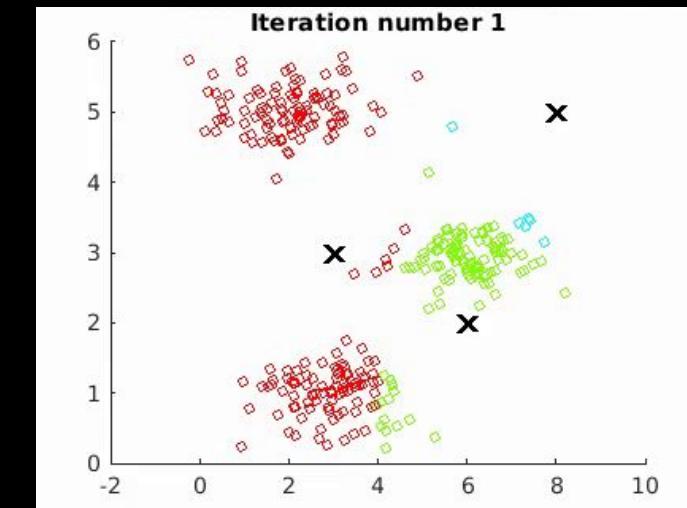
STEP 3: Assign each data point to the closest centroid
(That forms K clusters)

STEP 4: Compute and place the new centroid of each cluster

STEP 5: Reassign each data point to the new closest centroid
If any reassignment took place, go to STEP 4,
otherwise go to FIN.



Your Model is Ready



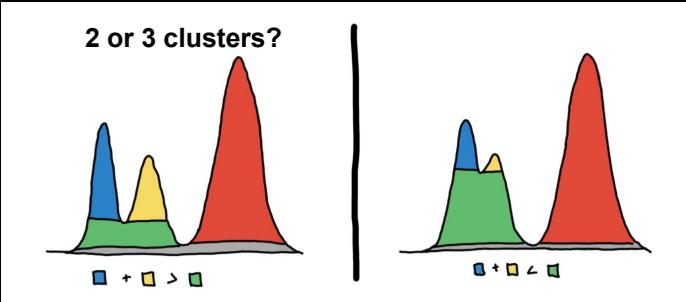
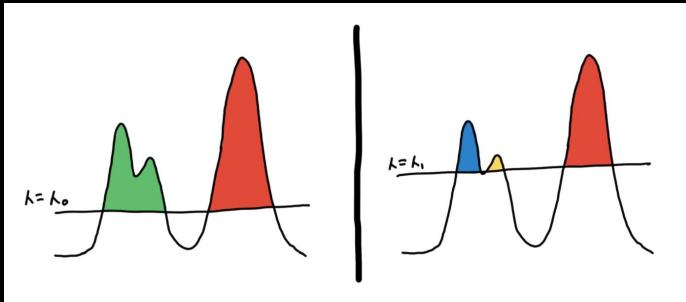
K-means clustering can be formulated as: given a fixed integer K , find the cluster means $\{\mu\}$ and the data point assignments in order to minimize the following objective function:

$$C(\{x, \mu\}) = \sum_{k=1}^K \sum_{n=1}^N r_{nk} (\mathbf{x}_n - \boldsymbol{\mu}_k)^2,$$

$r_{nk} \in \{0,1\}$ binary assignment

Density Based Clustering

Two different clusterings based on two different level-sets



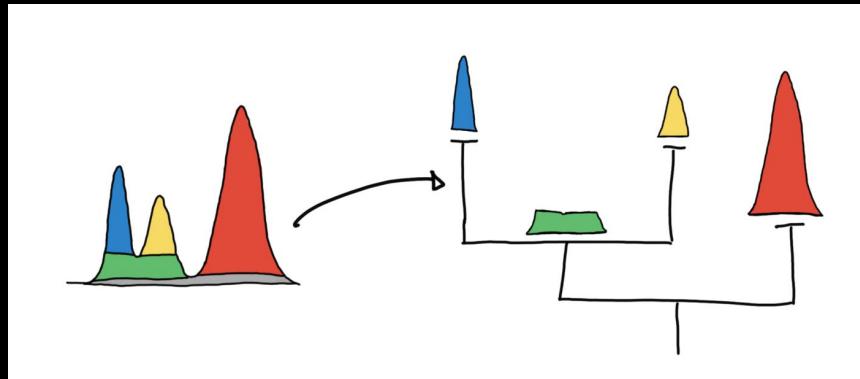
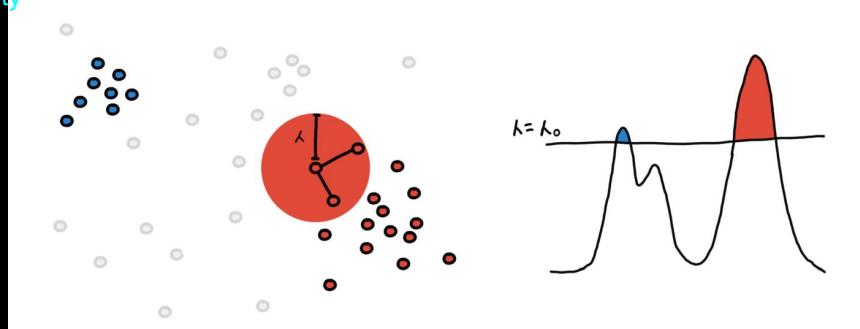
The area of the regions is the measure of “**persistence**”.

Maximize the persistence of the clusters under the constraint that they do not overlap.

soft-clustering

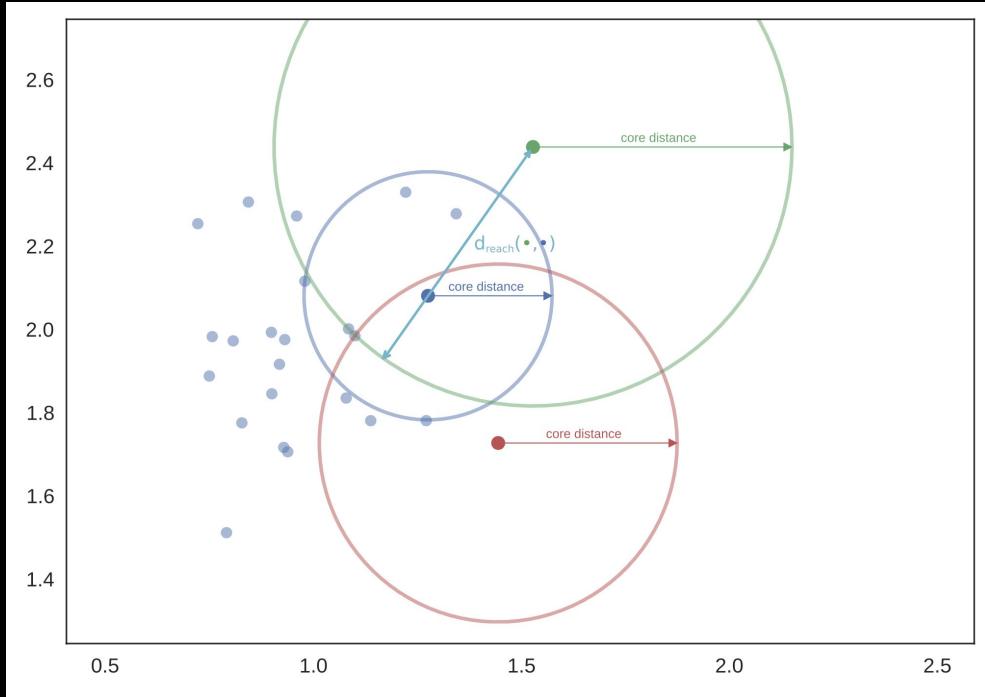
Core distance (defined by a required # of neighbors) as estimate of density

Points have to be in a high density region and close to each other (“**mutual reachability**”)



clusters as more likely “regions” separated by less likely regions -> densities

Density Based Clustering



Mutual reachability

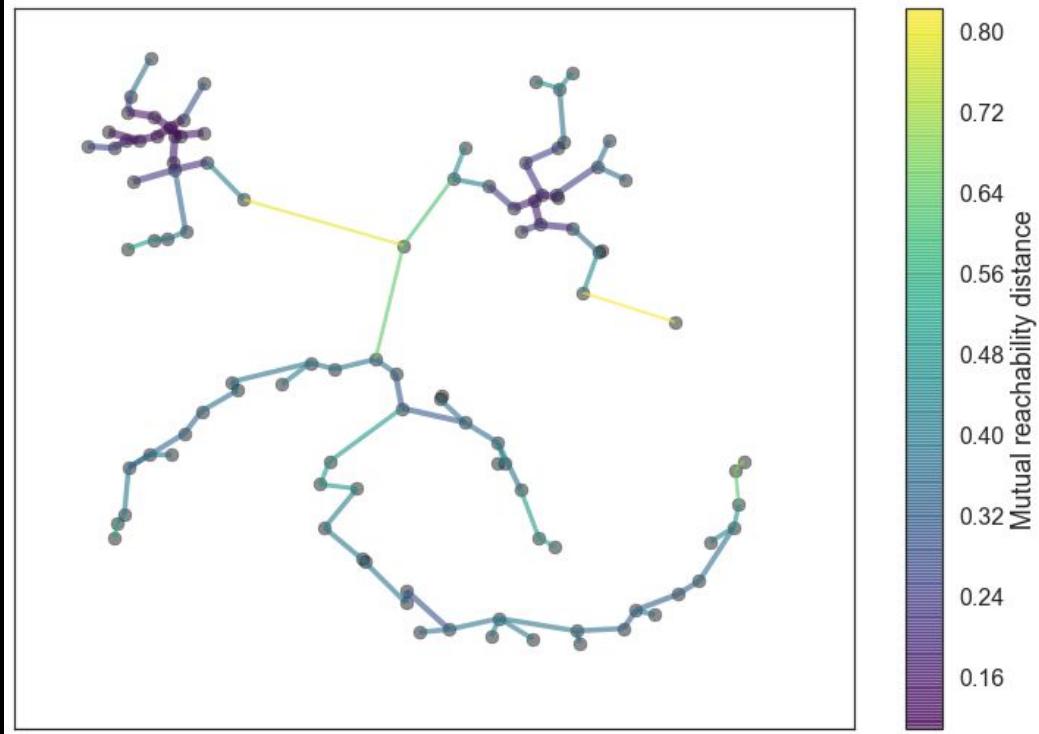
$$d_{mreach-k}(a, b) = \max\{core_k(a), core_k(b), d(a, b)\}$$

The **mutual reachability** distance is a summary at what level of " λ " two points together will connect. This is what we use as a new metric.

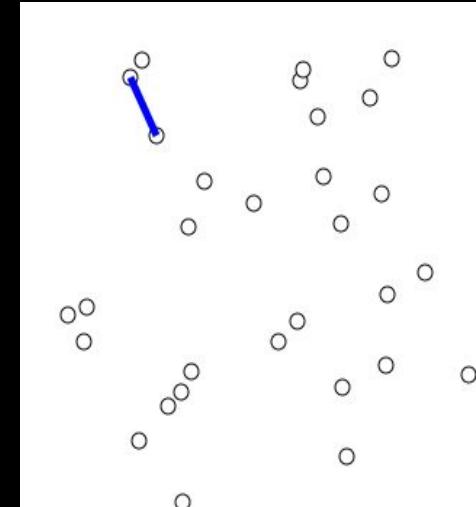
[1] DBSCAN, or density-based spatial clustering of applications with noise [[Khan, 2014](#)]

[2] HDBSCAN hierarchical DBSCAN [[McInnes, 2017](#)]

Hierarchical Clustering



Minimum spanning tree
https://en.wikipedia.org/wiki/Prim%27s_algorithm

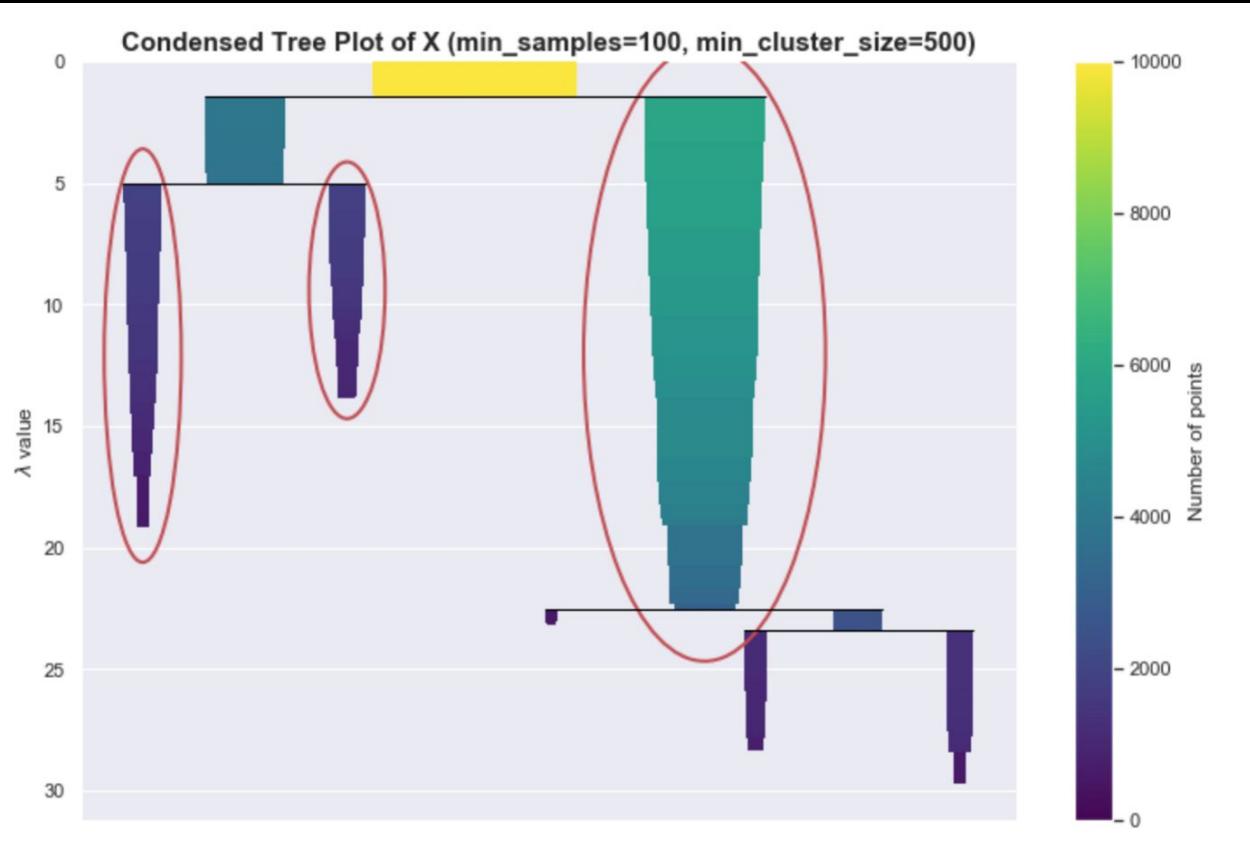


After mutual reachability... consider a threshold value, starting high, and steadily being lowered. Drop any edges with weight above that threshold. As we drop edges we will start to disconnect the graph into connected components. Eventually we will have a hierarchy of components (from completely connected to completely disconnected) at varying threshold levels). In practice this is very expensive: many edges... Fortunately graph theory furnishes us with just such a thing: the minimum spanning tree of the graph.

Hierarchical Clustering

Minimum spanning tree

https://en.wikipedia.org/wiki/Prim%27s_algorithm



$O(n^2)$

A hierarchy of multiple level-sets is obtained by varying the density threshold

Visualization of the tree top-down as in the literature

hdbSCAN vs k-means

K-means is a semi-supervised parametric algorithm parameterized by the *K cluster centroids* (if you want K seeds). Can perform if the underlying assumptions on the shape of the clusters are not met. Clusters have to be:

- “round” or spherical
- equally sized
- equally dense
- most dense in the center of the sphere
- not contaminated by noise/outliers

Hdbscan on the other hand is an unsupervised hierarchical clustering which excels when data has:

- Arbitrarily shaped clusters
- Clusters with different sizes and densities
- Noise

PCA

- Ubiquitous method for dimensional reduction
- Inspired by the observation that in many cases, the relevant information in a signal is contained in the directions with largest variance
- Use to reduce dimensionality
- Principal components are orthogonal to each other
- Calculate the Eigenvectors and Eigenvalues of the Covariance Matrix
- We will utilize useful and easy to use python packages for **Principal Component Analysis** to create insightful plots.

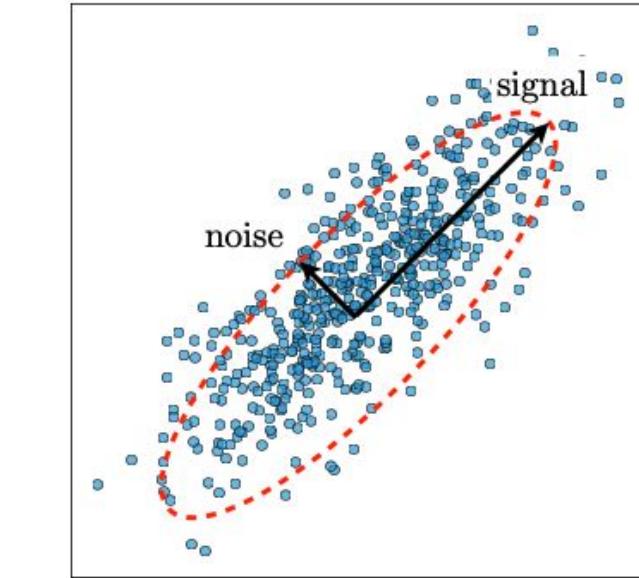


FIG. 50 PCA seeks to find the set of orthogonal directions with largest variance. This can be seen as “fitting” an ellipse to the data with the major axis corresponding to the first principal component (direction of largest variance). PCA assumes that directions with large variance correspond to the true signal in the data while directions with low variance correspond to noise.

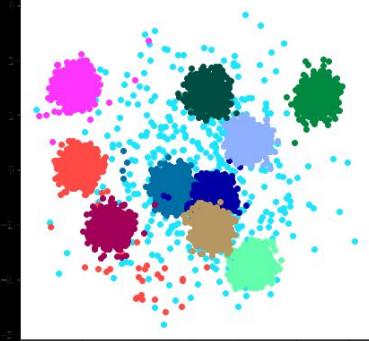
[1] Levina, Elizaveta, and Peter J. Bickel. "Maximum likelihood estimation of intrinsic dimension." *Advances in neural information processing systems*. 2005.

[2] PCA for dimensionality reduction: Abdi, Hervé, and Lynne J. Williams. "Principal component analysis." *Wiley interdisciplinary reviews: computational statistics* 2.4 (2010): 433-459.

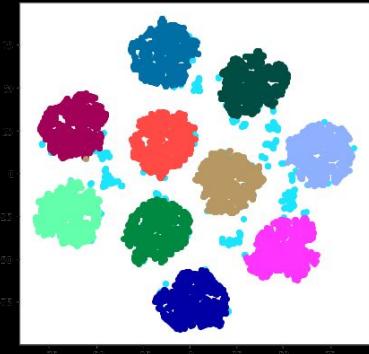
t-SNE

- t-SNE is also a method to reduce the dimension. One of the most major differences between PCA and t-SNE is it preserves only local similarities whereas PA preserves large pairwise distance maximize variance.
- Recently, t-stochastic neighbor embedding (t-SNE) has emerged as one of the go-to methods for visualizing high-dimensional data.
- Each high-dimensional training point is mapped to low-dimensional embedding coordinates, which are optimized in a way to preserve the local structure in the data.
- The idea of stochastic neighbor embedding is to associate a probability distribution to the neighborhood of each data

- t-SNE can rotate data
- t-SNE results are stochastic (depends on initial seed)
- t-SNE preserves short distance information
- Scales are deformed
- Computationally intensive



PCA



t-SNE

[1] Maaten, Laurens van der, and Geoffrey Hinton (2008), “Visualizing data using t-sne,” Journal of machine learning research 9 (Nov), 2579–2605.

Tips, further reading

- **Hard Clustering:** In hard clustering, **each data point either belongs to a cluster completely or not.**
- **Soft Clustering:** In soft clustering, instead of putting each data point into a separate cluster, a probability or likelihood of that data point to be in those clusters is assigned
- **Since clustering algorithms including kmeans use distance-based measurements to determine the similarity between data points, it's recommended to standardize the data to have a mean of zero and a standard deviation of one since almost always the features in any dataset would have different units of measurements** [[ref](#)]
- PCA is at a disadvantage if the data has not been standardized before applying the algorithm to it.
- PCA vs t-SNE: which one to use? [[ref](#)]