Lectures on Machine Learning

INFN Istituto Nazionale di Fisica Nucleare

Tommaso Dorigo, INFN-Padova

Braga, March 25-27, 2019

Lecture 2
Classification and Decision Trees

Contents - 2

Lecture 2: Classification and decision trees

- Classification
- The statistical approach: hypothesis testing
  - Neyman-Pearsons lemma
- Loss minimization
- A couple of methods
  - Fisher linear discriminants
  - Support vector machines
- Decision trees
  - random forests
  - boosting techniques

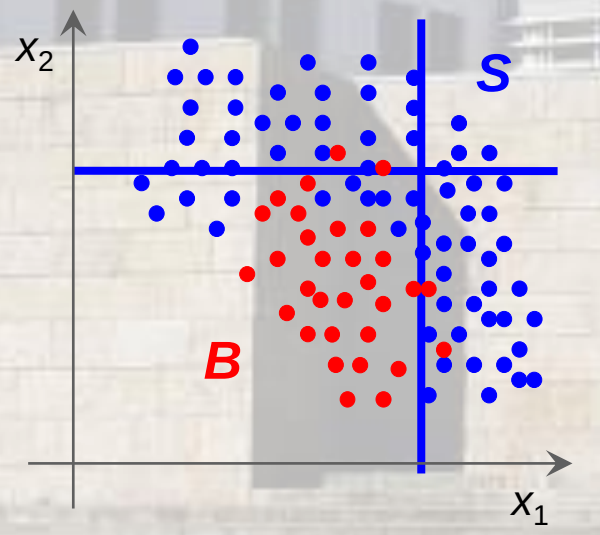
CLASSIFICATION

Binary Classification

Signal and **Background:** a common problem in many setups

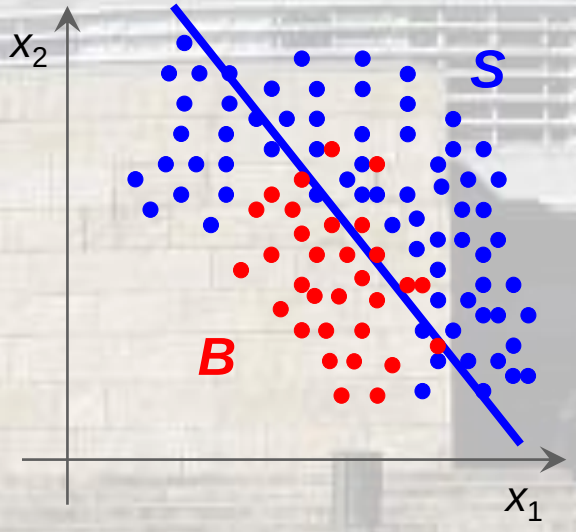
Generally we know the characteristics of two classes of events, and we wish to use them to find the best possible distinction with a fixed rule → a "decision boundary" in the feature space of the event characteristics

A "rectangular" rule

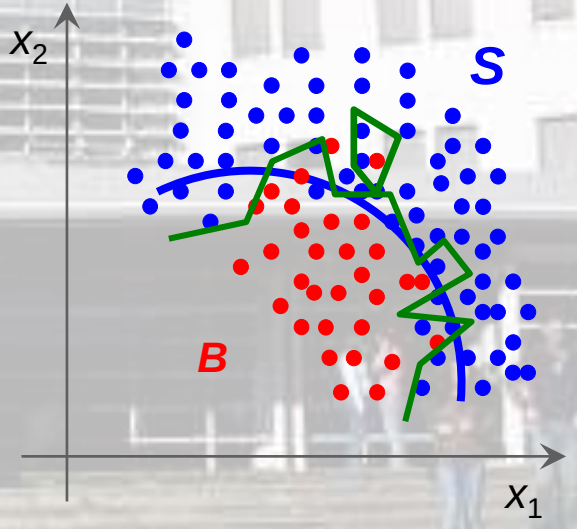


Low variance (stable), high bias methods

A linear rule

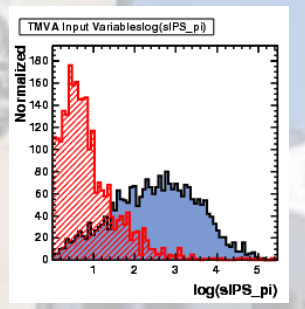


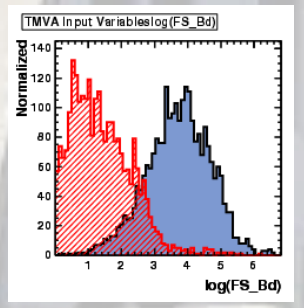
A non linear rule

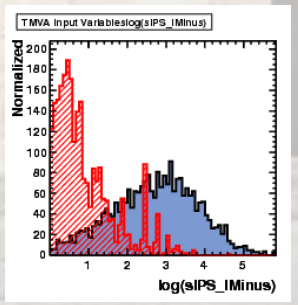


High variance, small bias methods

Feature Space and Output Function







- D-dimensional "feature space"

Every signal or background event has "D" measured variables

Test statistic:

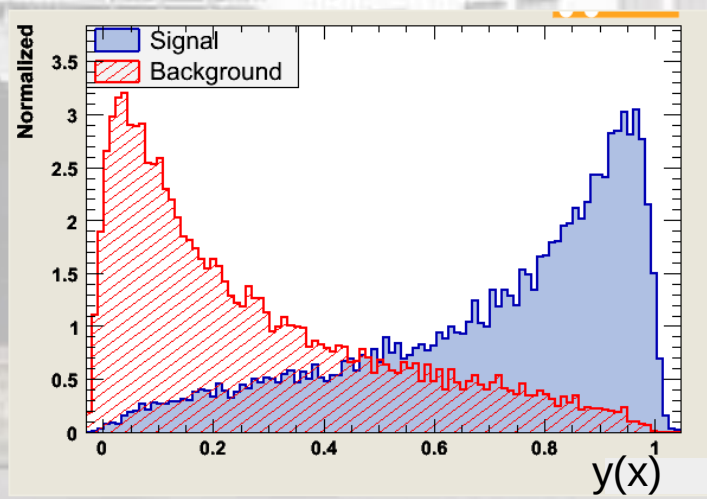
$y(x)$: $R^D \rightarrow R$:

$y = y(\mathbf{x})$; $\mathbf{x} \in R^D$

$\mathbf{x}=\{x_1,\ldots,x_D\}$: input variables

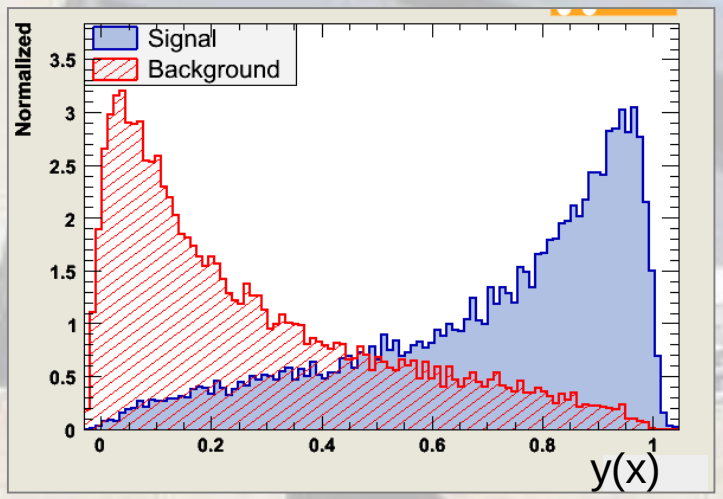
One wishes to find a map of the multi-D space of features, into a real variable that separates in a pseudo-optimal way the two classes

One can construct a histogram of the resulting values of y(x) taken by **S** and **B**

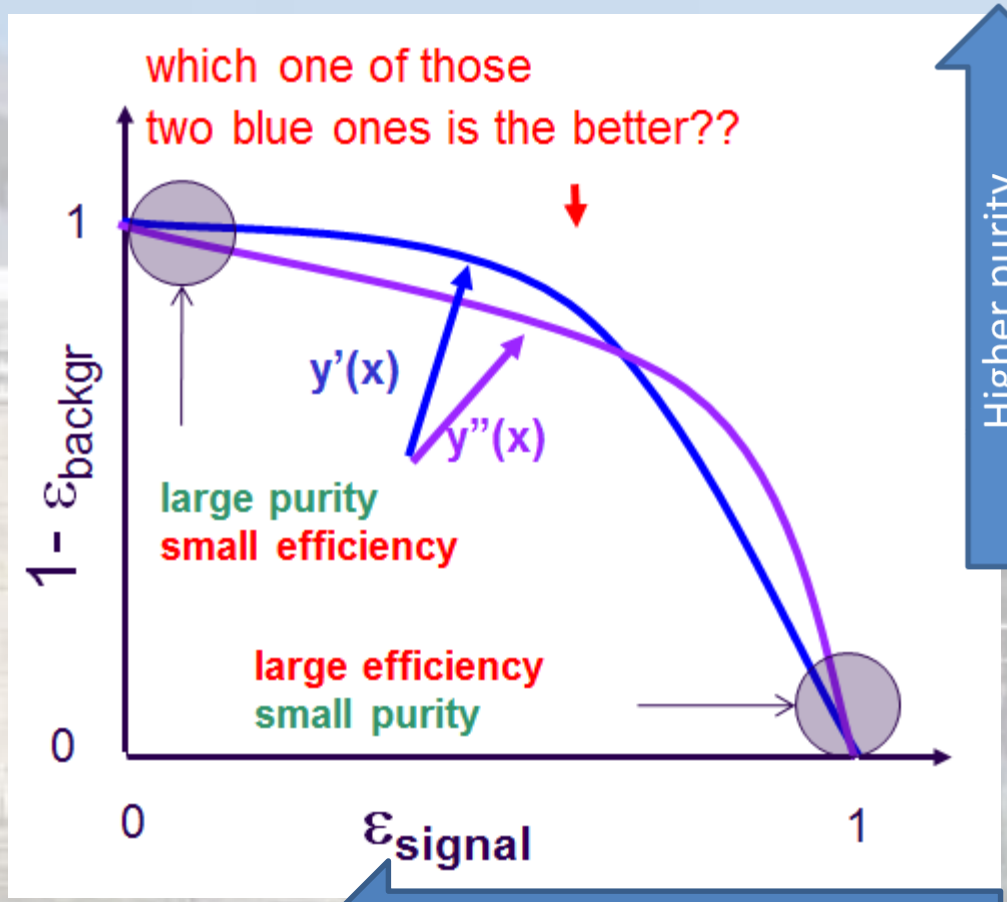


The "ROC" Curve

The ROC curve (receiver operating characteristic) is a way to summarize how well you are doing with your estimated y(x) in the classification problem



Cutting harder on y(x)



How to quantify ?

- look at rejection at fixed eff
- compute AUC
- many other metrics available

The AUC metric

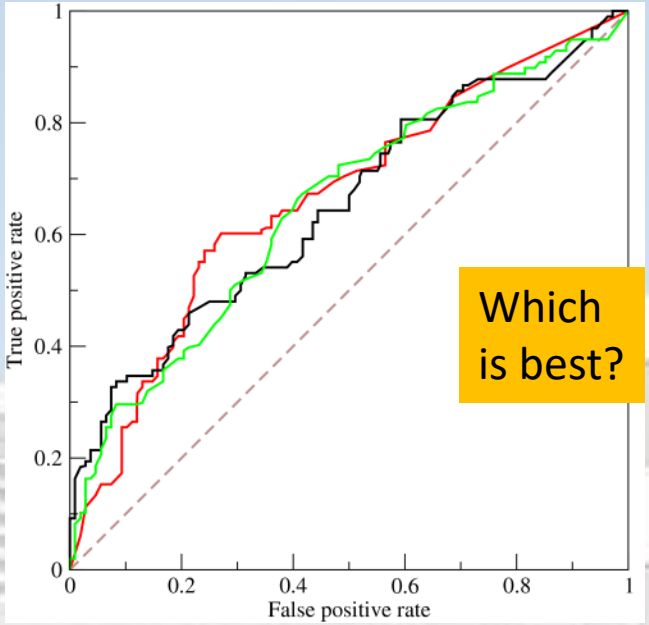
In some cases it is not trivial to rank classifiers by their performance.

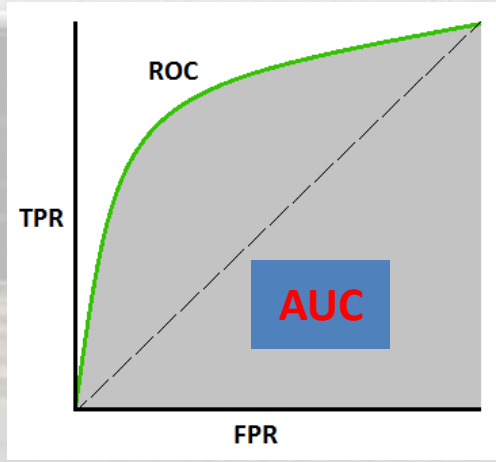
In particular if one does not (yet) have an estimate of the proportion of signal and background (class probabilities), one cannot decide!

A simple criterion is to compute the **area under the ROC curve** (AUC).

The AUC has a clean statistical interpretation: taken two events at random, one from each of the two classes, AUC is the probability that the signal event has higher score than the background event.

AUC is a coherent measure of predictive power of y(x) if we have no information on the relative misclassification cost of the two classes – i.e. if we do not know the operating point. The more we know of that, the less useful AUC is.





A parenthesis: Generalities on Hypothesis Testing

We are often concerned with proving or disproving a theory, or comparing and choosing between different hypotheses.
In general this is a different problem from that of estimating a parameter, but the two are tightly connected.

If nothing is known *a priori* about a parameter, naturally one uses the data to **estimate** it; if however a theoretical prediction exists of a particular value, the problem is more proficuously formulated as a **test of hypothesis**.

A hypothesis is **simple** if it is completely specified; otherwise (e.g. if it depends on the unknown value of a parameter, $H(\theta)$) it is called **composite**.

Nuts and bolts of HT

Be given

- i.i.d. (Identical and Independently Distributed) data $X=\{x_i\}_{i=1,\ldots,N}$, assumed without loss of generality multi-dimensional: $x_i=(x_{i,1},x_{i,2},\ldots,x_{i,D})$

- Two fully specified PDFs:

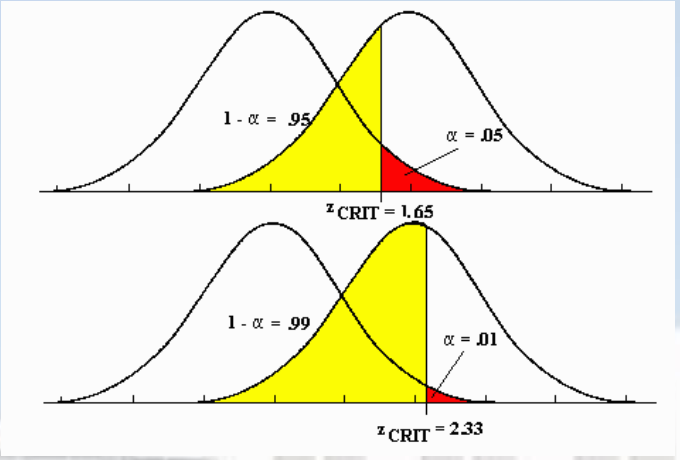
- H_0: null hypothesis $P(X|H_0)$
- H_1: alternate hypothesis $P(X|H_1)$

Three main parameters in the game:

- α: **type-I error rate**; probability that H_0 is true although you allow yourself to accept the alternative hypothesis
- β: **type-II error rate**; probability that you fail to claim a discovery (accept H_0) when in fact H_1 is true
- θ, parameter of interest (describes a continuous hypothesis, for which H_0 is a particular value).

  E.g. θ=0 might be a zero cross section for a new particle

- Common for H_0 to be **nested** in H_1

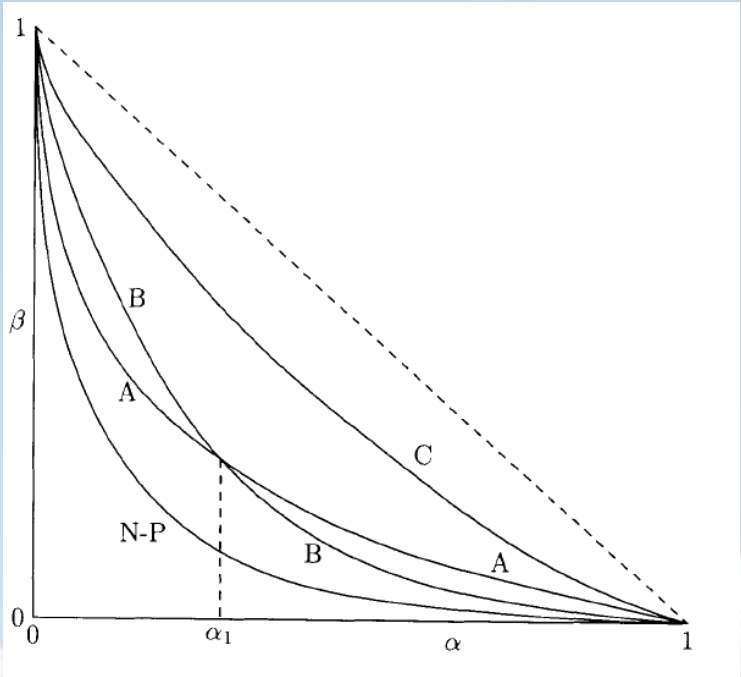


Above, a smaller α is paid for with a larger type-II error rate (yellow area)
→ smaller power 1-β

if one specifies p(H0), p(H1), and α the problem is solved...

Alpha vs Beta and power graphs

- Choice of α and β is conflicting: where to stay in the curve provided by your analysis method highly depends on habits in your field
- What makes a difference is the test statistic: note how the N-P likelihood-ratio test outperforms others in the figure [James 2006] – reason is N-P lemma
- As data size increases, power curve becomes closer to step function



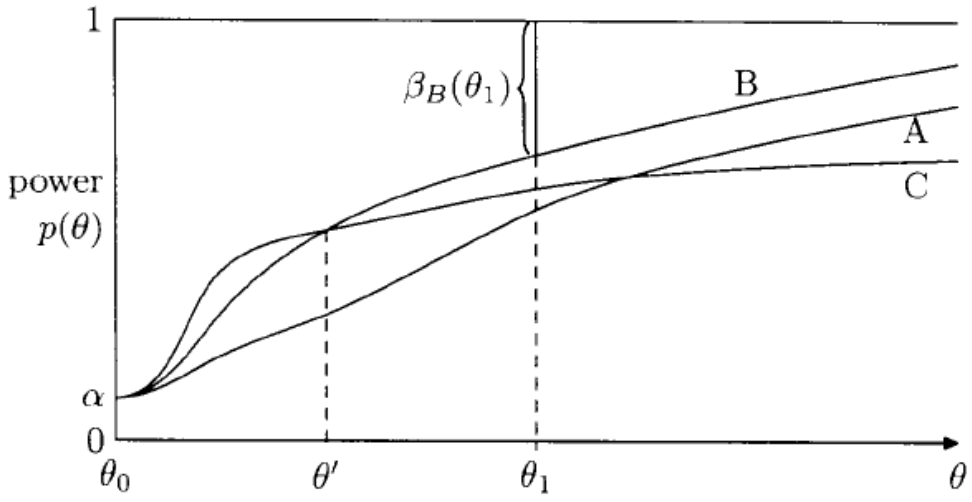


Fig. 10.3. Power functions of tests A, B, and C at significance level α. Of these three tests, B is the best for $\theta > \theta'$. For smaller values of θ, C is better.

The power of a test usually also depends on the parameter of interest: different methods may have better performance in different parameter space points UMP (uniformly most powerful): has the highest power for any θ

Statistical significance

Statistical significance reports the probability that an experiment obtains data at least as discrepant as those actually observed, under a given "null hypothesis“ H_0

- In physics H_0 *usually* describes the currently accepted and established theory

- Given **data X** and a **test statistic T** (a function of X), one may obtain a **p**-value as the probability of obtaining a value of T **at least as extreme** as the one observed, if H_0 is true.

p can then be converted into the corresponding number of "sigma," *i.e.* standard deviation units from a Gaussian mean. This is done by finding **x** such that the integral from **x** to infinity of a unit Gaussian equals **p**:

$$\frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{t^2}{2}} dt = p$$

According to the above recipe, a **15.9%** probability is a one-standard-deviation effect; a **0.135%** probability is a three-standard-deviation effect; and a **0.0000285%** probability corresponds to five standard deviations - "**five sigma**" in jargon.

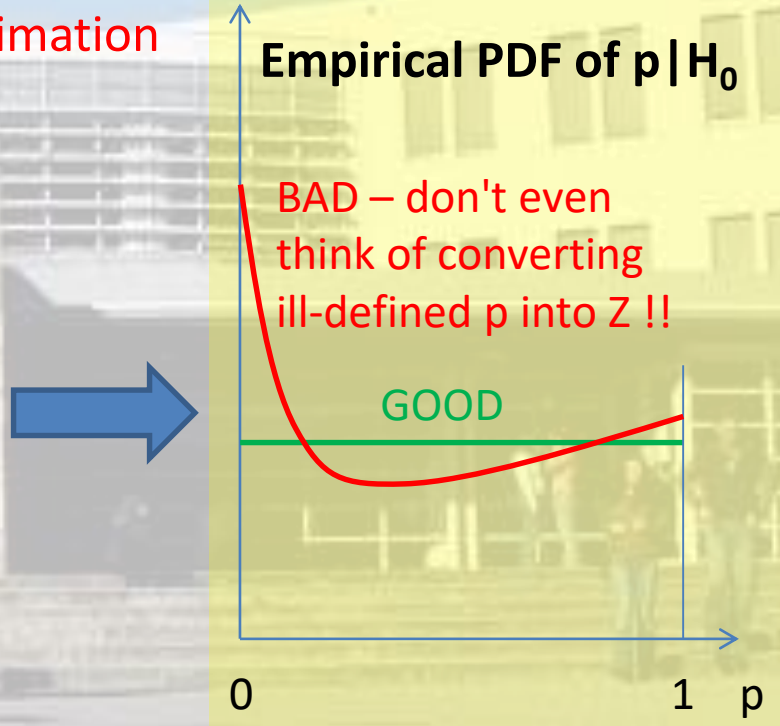
Notes

The convention is to use a "one-tailed" Gaussian: we do not care about departures of x from the mean in the *un-interesting direction*

The conversion of p into σ is independent of experimental detail. Using Nσ rather than p is just **a shortcut, nothing more** !

In particular, using "sigma" units does in no way mean we are operating some kind of Gaussian approximation anywhere in the problem

The whole construction rests on a proper definition of the p-value. Any shortcoming of the properties of p (*e.g.* a tiny non-flatness of its PDF under the null hypothesis) totally invalidates the meaning of the derived Nσ



A note on the notes: How a non-flat p-value arises

The graph in the previous page is probably cryptic unless we consider how we could construct it

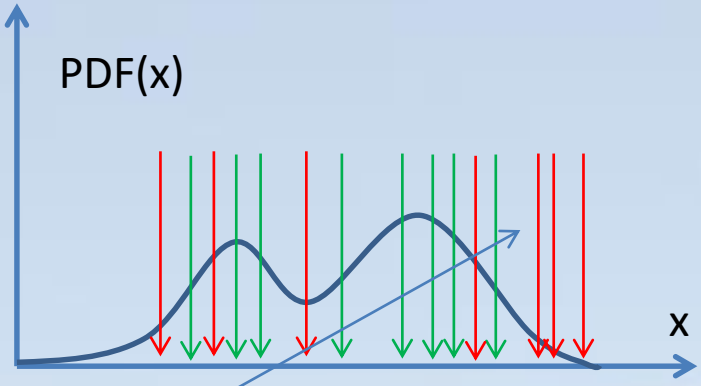
Take a model of the data (a PDF).

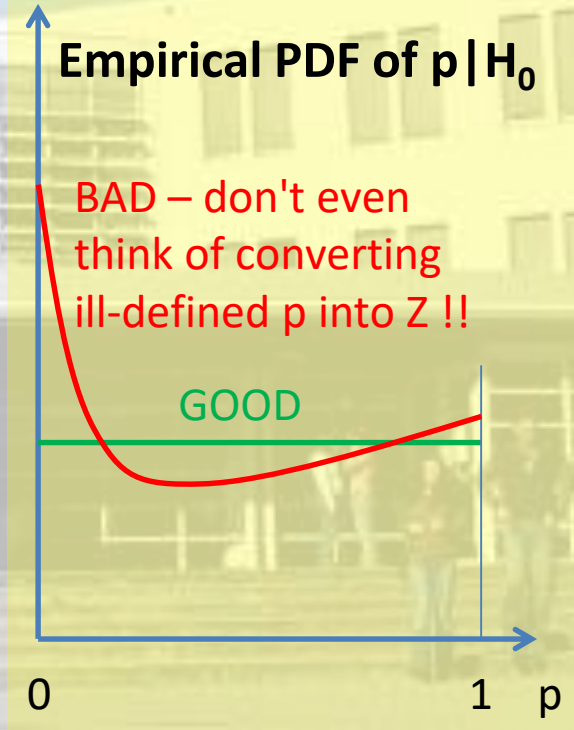
Then consider observations allegedly sampled from that model

If the model is not correct, the data will populate certain regions differently from what the model predicts

If for each data point you do a test of hypothesis, asking how probable it is that it does come from the supposed PDF (as a tail probability!), you can plot the resulting p-values

→ the incorrect model is spotted by a non-flat distribution of p





The Neyman-Pearson Lemma

In a 1932 paper by J. Neyman and E. S. Pearson, "**On the Problem of the most Efficient Tests of Statistical Hypotheses**", the following notable result is demonstrated:

Given two simple hypotheses, parametrized by densities $p(x|\theta_1)$, $p(x|\theta_0)$ *that depend on specific values of a parameter, the likelihood ratio*

$$r(x|\theta_0,\theta_1) = p(x|\theta_1) / p(x|\theta_0)$$

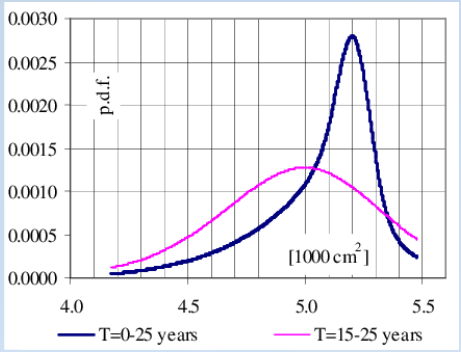
is the most powerful test statistic to discriminate between them.

The importance of this lemma cannot be overstated – it is the ultimate weapon for solving inference problems.

How to Build a Classifier

The function

$$\mathbf{r}(\boldsymbol{x}) = \frac{\boldsymbol{PDF}(\boldsymbol{x}|\boldsymbol{S})}{\boldsymbol{PDF}(\boldsymbol{x}|\boldsymbol{B})}$$



is therefore the best possible classifier of S versus B → **PROBLEM SOLVED! ?**

...**NO**:

- p(x|S), p(x|B) are usually not perfectly known; in typical cases of interest in HEP and astro-HEP they may be only estimated by forward simulation, affected by stochastic phenomena

- hypotheses are not usually "simple vs simple" – nuisance parameters affect their determination

In general: One knows examples of S and B, but not their precise density

A relevant theorem

The Neyman-Pearson lemma can be used effectively in association with a

Theorem:

The likelihood ratio r(x) is invariant under a change of variable y = f(x), provided f(x) is monotonic with r(x):

$$r(x|\theta_0, \theta_1) = \frac{p(x|\theta_0)}{p(x|\theta_1)} = \frac{p(f(x)|\theta_0)}{p(f(x)|\theta_1)} = r(y|\theta_0, \theta_1)$$

Note:

- This is a strict equality, not an inequality.
- The transformation does not squander information on the ratio
- But information about x may be lost by f()

How to construct f(x)

With supervised learning we can get f(x) automagically!

- we start with a binary classifier f* trained to distinguish x sampled from $p(x|\theta_1)$ from x sampled from $p(x|\theta_0)$: a Bernoulli process, modelled with Binomial pdf

$$p(y_i|x_i) = (p_i)^{y_i}(1-p_i)^{1-y_i} \qquad y_i = \{0,1\}$$

the training may be achieved by minimization of **cross-entropy**, e.g.

$$L(\hat{f}) = -\boldsymbol{E}_{p(x|\theta)\pi(\theta)}[\mathbf{1}(\theta=\theta_0)log\hat{f}(x) + \mathbf{1}(\theta=\theta_1)\log\left(1-\hat{f}(x)\right)]$$

The solution at minimum loss approximates the optimal classifier

$$\hat{f}(x) \sim f^*(x) = \frac{p(x|\theta_1)}{p(x|\theta_0)+p(x|\theta_1)}$$

which is monotonic with the likelihood ratio r, so

$$r(x|\theta_0,\theta_1) \sim \hat{r}(x|\theta_0,\theta_1) = \frac{1-\hat{f}(x)}{\hat{f}(x)}$$

This proves that **supervised classification is equivalent to likelihood ratio estimation**, and can be effectively used for inference or discrimination

Other metrics for binary classification

We discussed the ROC curve, and mentioned that the **AUC** (area under curve) can be a measure of how well a classifier is doing

But there are of course other metrics, that fit different tasks

E.g.: Sensitivity and Specificity. Suppose we have to gauge the effectiveness of a signal selection cut.

Sensitivity: TP/(TP+FN)

think of it as "signal efficiency"

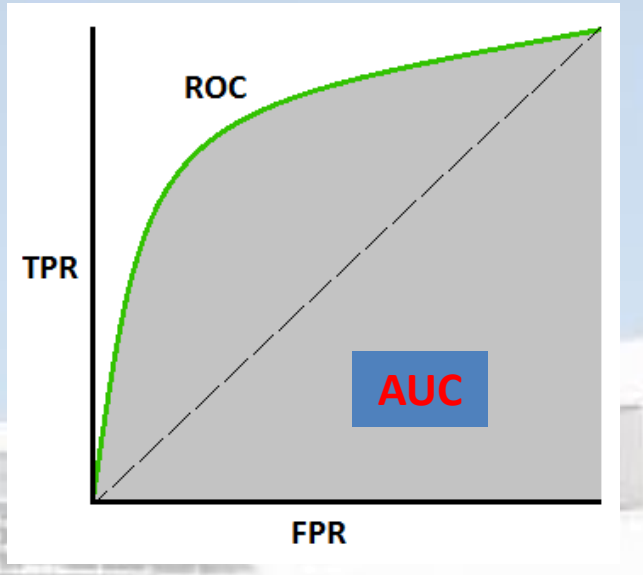
Specificity: TN/(TN+FP)

"background rejected by selection"

Also of interest:

Purity: TP / (TP+FP) (a relative measure)

Can be useful as a measure of the goodness of the selection, but cannot be the only measure, as it strongly depends on initial fractions



	Is signal	Is background
pass cut (positive)	True positives **TP**	False positives **FP**
fail cut (negative)	False negatives **FN**	True negatives **TN**

The loss function

The **mean squared error** is a sound measure of the model accuracy, but it is not necessarily the metric most appropriate for our problem

E.g., in physics analyses we are concerned with the maximum sensitivity to a small signal, and we often use as a figure of merit the ratio s/sqrt(b+s) → **don't do that** (see digression next slide)

In general, the quality of a predictive model can be quantified by constructing a function l(y,f(x)), a measure of the distance between the true class label y and the predicted response f(x).

One may then define the expected distance

$$L(X,Y) = E_{X,Y}\, l\big(Y, f(X)\big) = \sum_{y \in Y} \int_X l\big(y, f(x)\big) P(x,y)\, dx$$

over the full space of X and Y. This can be computed empirically, using labelled data drawn from the pdf p(x,y), by averaging l(y,f(x)):

$$\hat{L} = \frac{1}{N} \sum_{n=1}^{N} l(y_n, f(x_n))$$

Note: the empirical loss depends on the prior distribution of the two classes, P(y). This is not always what the analyzer wants (see note above).

Optimization?

- We all-too-often see analyses blindly optimizing on S/sqrt(B) or S/sqrt(B+S) even in cases when the signal region is going to contain a small number of entries

- One real-life example (recently seen): **a great cut keeps 20% bgr, 60% signal**
  - at preselection, expect 8 signal, 1 background: **S/sqrt(B)=8; S/sqrt(B+S)=0.89**
  - after selection, expect 4.8 signal, 0.2 background: **S/sqrt(B)=10.7, S/sqrt(B+S)=0.96**
  - **Is it a good idea ?**
  - Median of B-only p-value distribution for observing N=8+1=9 in the first case is $p_m=1.1*10^{-6}$ **, twice smaller** than median p-value for observing N = 4.8+0.2 = 5 ($p_m=2.6*10^{-6}$) → **we worsened the expected p-value by a factor of 2 !!!**

- If you really need a quick-and-dirty answer please use: **$Q=2[(S+B)^{0.5}-B^{0.5}]$** which has better properties (case above: **$Q_{presel}=4$; $Q_{sel}=3.58$**)
- In general "optimization" is a word used recklessly. **A full optimization is seldom seen in HEP analyses**. This however should not discourage you from trying!

→When possible, **optimize on final result**, not on "intermediate step" (systematics may wash out your gain if you disregard them while optimizing)

0-1 loss and Bayes error

The predicted response of a classifier can be a **hard label** (e.g. y*=0,1), or a **soft score** f(x). The latter provides a "level of confidence" in the prediction

In the case of a hard label y*, the loss can only be formulated as a "0-1 loss", the classification error

$l(y,y^*) = 0$ if $y=y^*$

$l(y,y^*) = 1$ if $y!=y^*$

One then minimizes the expected loss L(X,Y) by choosing y* such that it maximizes the probability of the corresponding class:

$y^*(x) = \arg\max_y P(y|x)$. ← pick y which maximizes p!

The 0-1 loss equates to the probability of observing one of the less probable classes,

$$\epsilon^* = 1 - \int_X P(y^*|x)P(x)dx$$

The minimal classification error thus obtained is also known as the **Bayes error**.

Finding the best parameters

We can treat a supervised learning problem by using the general strategy for parameter estimation: **cast the problem as a likelihood maximization**.

Using data X, write L as

$$L(w) = p(y|X;w) = \Pi_i\, p(y_i|x_i,w)$$

and select the parameters w that make the observed data the most likely,

$$L(w) = \arg\min_w (-\log L(w)) =$$
$$= \arg\min_w [-\mathrm{Sum}_i \log p(y_i|x_i,w)]$$

E.g. 1: if the targets have a Normal distribution of mean $\mu=f(x)$ and width σ, given data X, compute

$$\log p(y|x) = \log [N(y|\mu,\sigma)] \sim [y-f(x)]^2.$$

E.g. 2: if probability of data X being signal is p, one has a Bernoulli trials distribution of p(x)

$$\log p(y|x) = \log (B(y;p=f(x)) = \log [p^y (1-p)^{1-y}] =$$
$$= y \log f(x) + (1-y) \log (1-f(x))$$

Parameter estimates for a linear model

Let us assume that

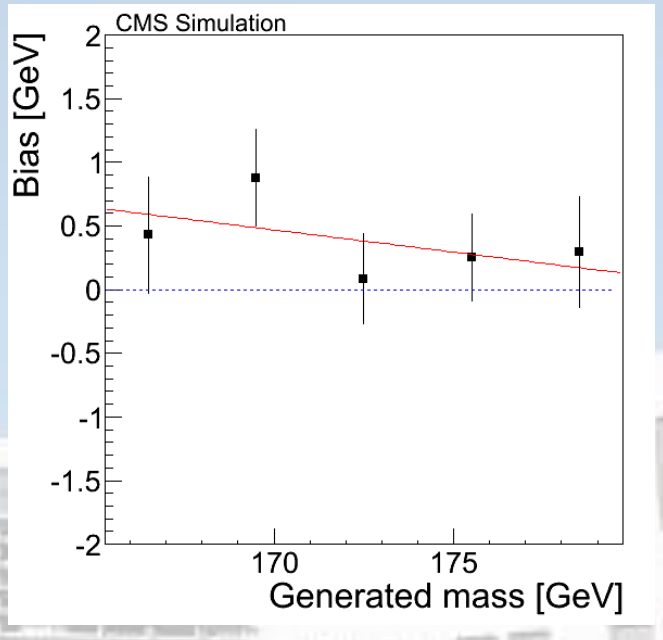
$y_i = f(x_i)+e_i$

with a Gaussian random error

$p(e_i) = \exp(-e_i^2/2\sigma^2)$

The model for y_i is then

$p(y_i|x_i,w) = \exp((wx_i-y_i)^2/2\sigma^2)$



The likelihood can then be written:

$$L(w) = p(y|x,w) = \prod_{i=1}^{n} p(y_i|x_i,w)$$

→ $$-\log L(w) = \sum_{i=1}^{n}(y_i - w_i x_i)^2$$

Thus the negative log-likelihood is equivalent to the least-square loss.

This yields a probabilistic interpretation of the regression operated by a ML tools using the LS loss.

Regularizing the loss

The parameters of a model, learned through loss minimization, can sometimes diverge / be unstable

Often one wishes to regularize the loss by adding some constraining term, e.g.

$$\arg\min_{\mathbf{w}} L(\mathbf{w}, \mathbf{x}) = \arg\min_{\mathbf{w}} \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(\mathbf{x}_i, \mathbf{w})) + \lambda \Omega(\mathbf{w})$$

The function Ω, moderated by a strength parameter λ, penalize some values of the parameters w

Ridge and Lasso

The regularization of the loss is often done with a L2 or a L1 norm on the parameters. The behavior one obtains is different.

A L2 norm

$$\Omega(\mathrm{w}) = \sum {w_i}^2$$

is equivalent to having a Gaussian prior in the PDF of the parameters

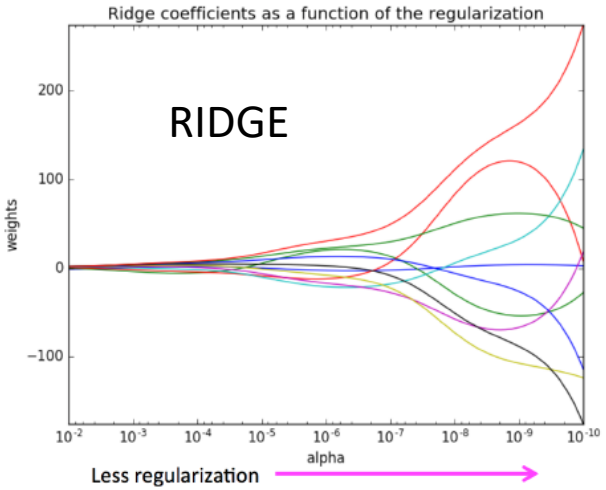
A L1 norm

$$\Omega(\mathrm{w}) = \sum |w_i|$$

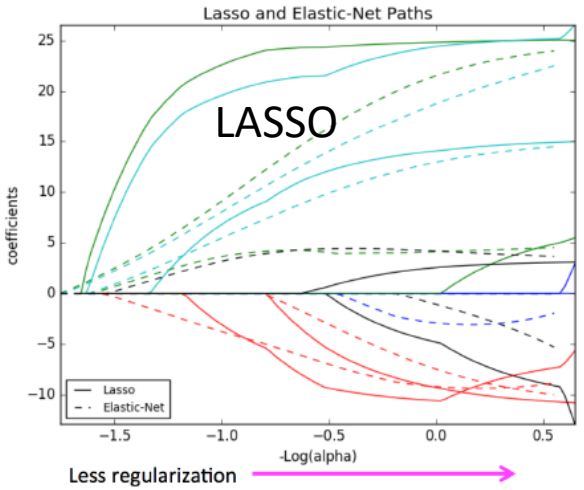
corresponds instead to a prior having a Laplace distribution

$$L(\mathbf{w}) = \frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^2 + \alpha\Omega(\mathbf{w})$$

$$L2: \quad \Omega(\mathbf{w}) = ||\mathbf{w}||^2$$

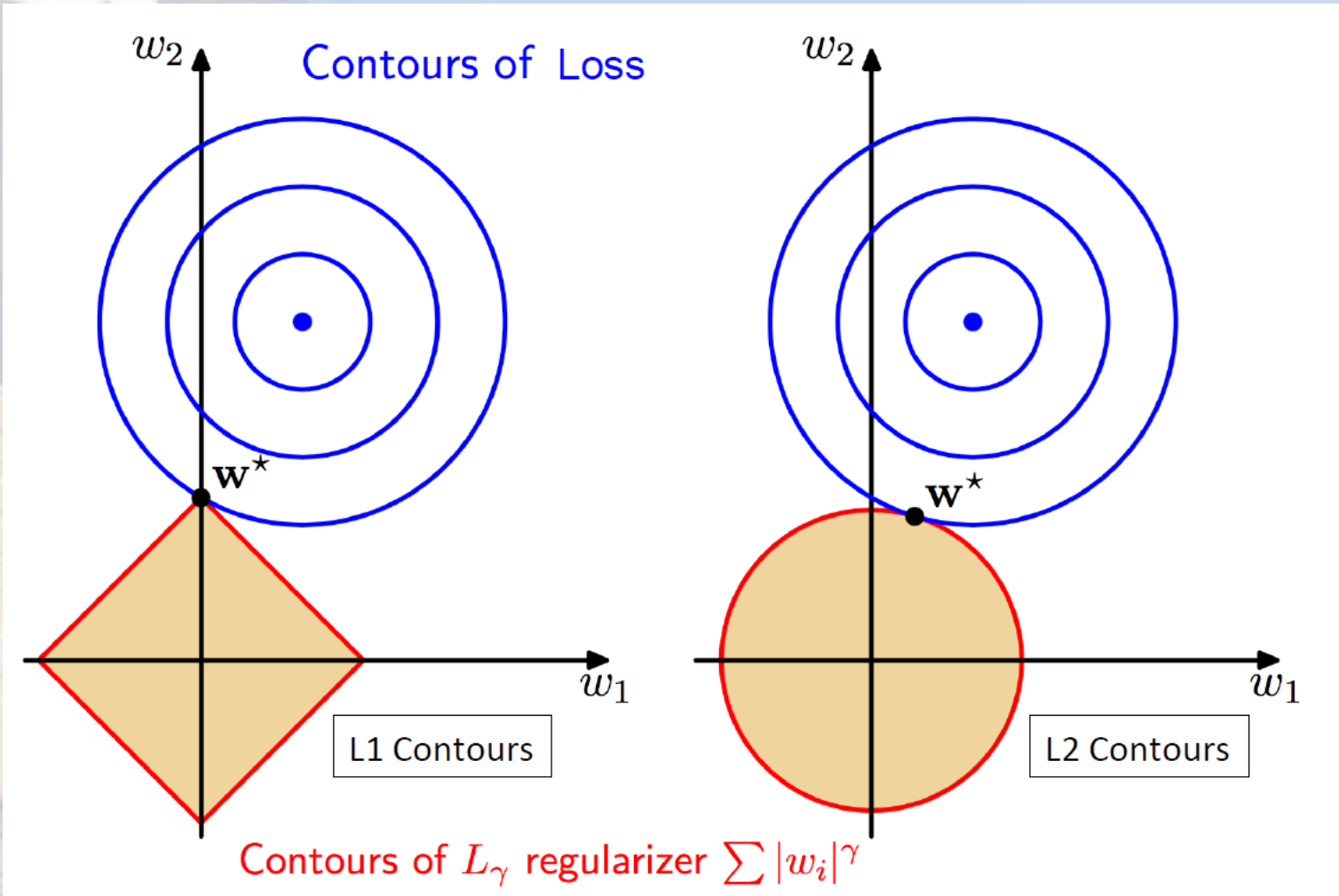


$$L1: \quad \Omega(\mathbf{w}) = ||\mathbf{w}||$$



- L2 keeps weights small, L1 keeps weights sparse!

What it means



Binary Cross-Entropy

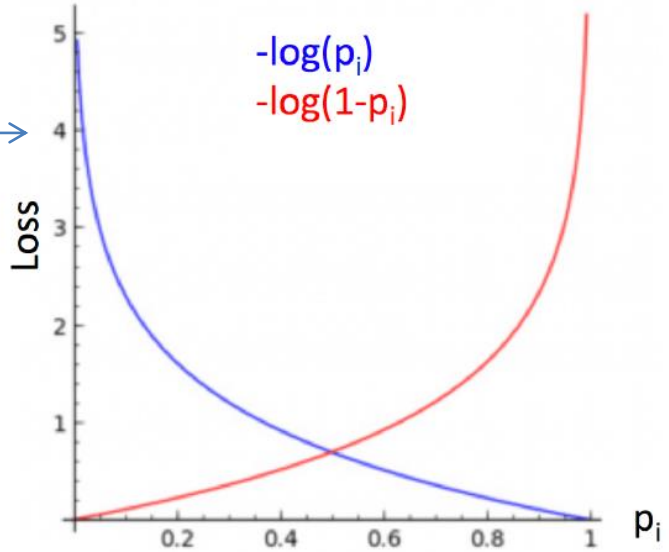
For binary classification, class assignment is modeled by a Bernoulli trial probability p following the Binomial distribution

$$p(y_i|x_i) = (p_i)^{y_i}(1-p_i)^{1-y_i} \qquad y_i = \{0,1\}$$

A commonly used loss function is correspondingly defined as

$$L(\mathbf{w}) = -\sum_i (y_i \ln p_i + (1-y_i)\ln(1-p_i))$$

High loss for events with low probability of being in predicted class



A comparison of loss functions

Square Error Loss:

$$L(h(\mathbf{x};\mathbf{w}),y) = \big(h(\mathbf{x};\mathbf{w}) - y\big)^2$$

- Often used in regression

Cross entropy:

$$L(h(\mathbf{x};\mathbf{w}),y) = -\,y \log h(\mathbf{x};\mathbf{w}) - (1-y)\log(1-h(\mathbf{x};\mathbf{w}))$$

- With y $\in \{0,1\}$
- Often used in classification

Hinge Loss:

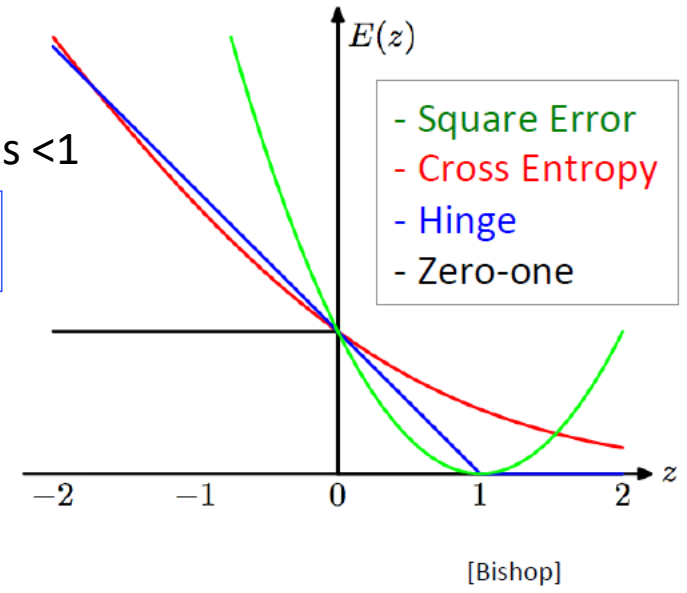
- With y $\in \{-1,1\}$ Only penalizes values <1

$$L(h(\mathbf{x};\mathbf{w}),y) = \max(0, 1 - yh(\mathbf{x};\mathbf{w}))$$

Zero-One loss

- With h(x; w) predicting label

$$L(h(\mathbf{x};\mathbf{w}),y) = 1_{y \neq h(\mathbf{x};\mathbf{w})}$$

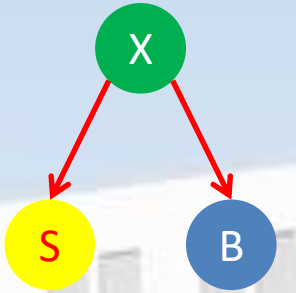


[Bishop]

Classification vs Regression

The two "standard" problems usually addressed by supervised learning methods are apparently quite different

- In classification tasks, the target is an **indicator function** assigning elements to one of two or more classes
- In regression tasks, the target is **continuous**



Consider a binary tree (see later) used for classification of events of type S and B: a node will have relative frequencies p, q and a *measure of impurity* is the associated **Gini index**, for s and b events in a node:

$\phi(p,q) = 2pq = 2sb/(s+b)^2$

= 0 → pure of one class
=1/2 → completely mixed

If taken as a regression problem, the target is y_n=+1 for S, y_n= -1 for B. One may choose the measure to be minimized as a mean-square error:

$MSE = Sum_{n=1,\ldots,N} (y_n - \langle y \rangle)^2 / N$

Classification == regression ?

We may write the mean response in a node with s and b events as

$$\langle y \rangle = s/(s+b)*[+1] + b/(s+b)*[-1] = (s-b)/(s+b)$$

Hence, since N = s+b, the MSE results in

$$MSE = \text{Sum}_{n=1,\ldots,N} (y_n - \langle y \rangle)^2/N = \text{Sum}_{n=1,\ldots,N} (y_n - \langle y \rangle)^2/(s+b) =$$
$$(s[1-\langle y \rangle]^2 + b[-1-\langle y \rangle]^2)/ (s+b) =$$
$$[s(1-(s-b)/(s+b))^2 + b(-1-(s-b)/(s+b))^2] / (s+b) =$$
$$2sb / (s+b) = \phi(p,q)$$

→ **MSE = ϕ(p,q)** !

NB the Gini index is discussed in more detail below, in the context of decision trees

So **regression** based on minimizing the MSE **is equivalent to classification** using the Gini index as criterion for the best split!

This is not a general result (change the minimization recipe and the result does not hold) but shows the tight connection of the two problems.

Diversion: Choose The Model

Suppose you are given some data (x, y+-σ_y) and the task of finding a suitable model y=f(x). The σ are standard error estimates. What model do you choose: a constant, a linear, or a quadratic one?

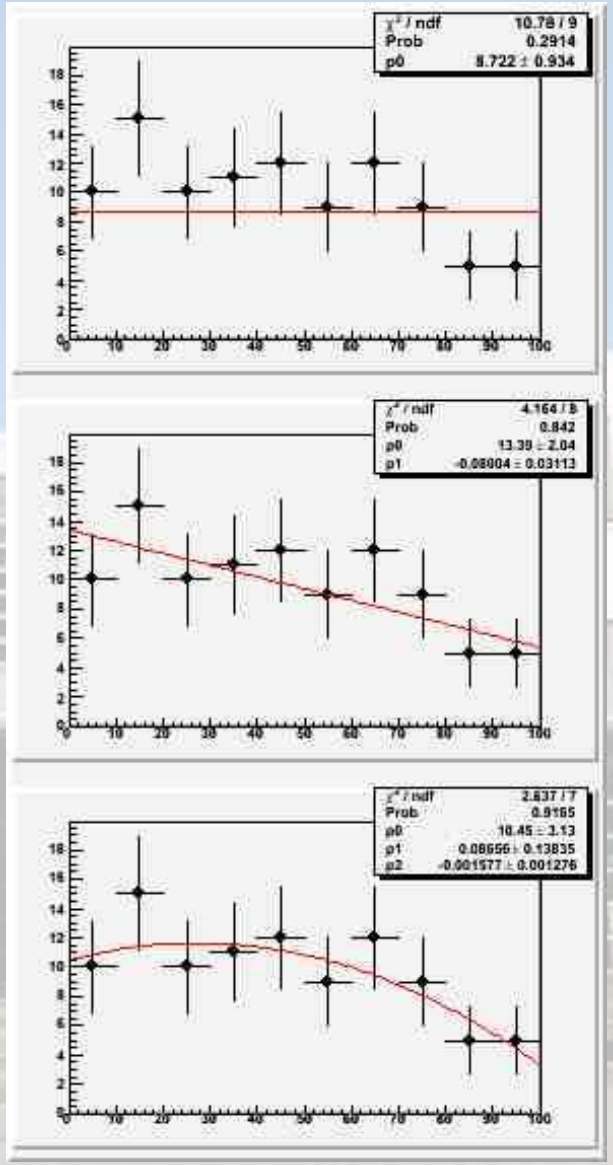
A first note to make is that this question involves a set of two tests of hypotheses (P0 vs P1 or P1 vs P2), and as such it is ill-posed until one specifies a type-1 error rate.

The other thing one might realize is that there is a thing solving this problem for you – it is called F-test.

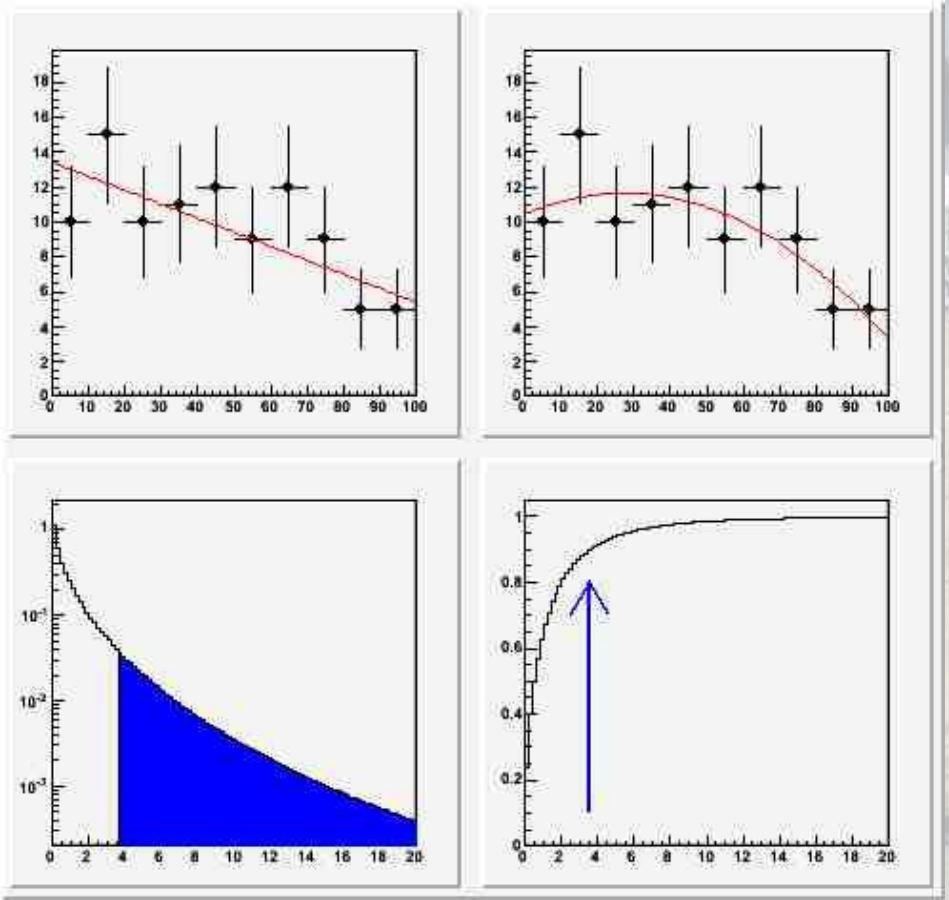
$$F = \frac{\frac{\sum_i (y_i - f_1(x_i))^2 - \sum_i (y_i - f_2(x_i))^2}{p_2 - p_1}}{\frac{\sum_i (y_i - f_2(x_i))^2}{n - p_2}}$$

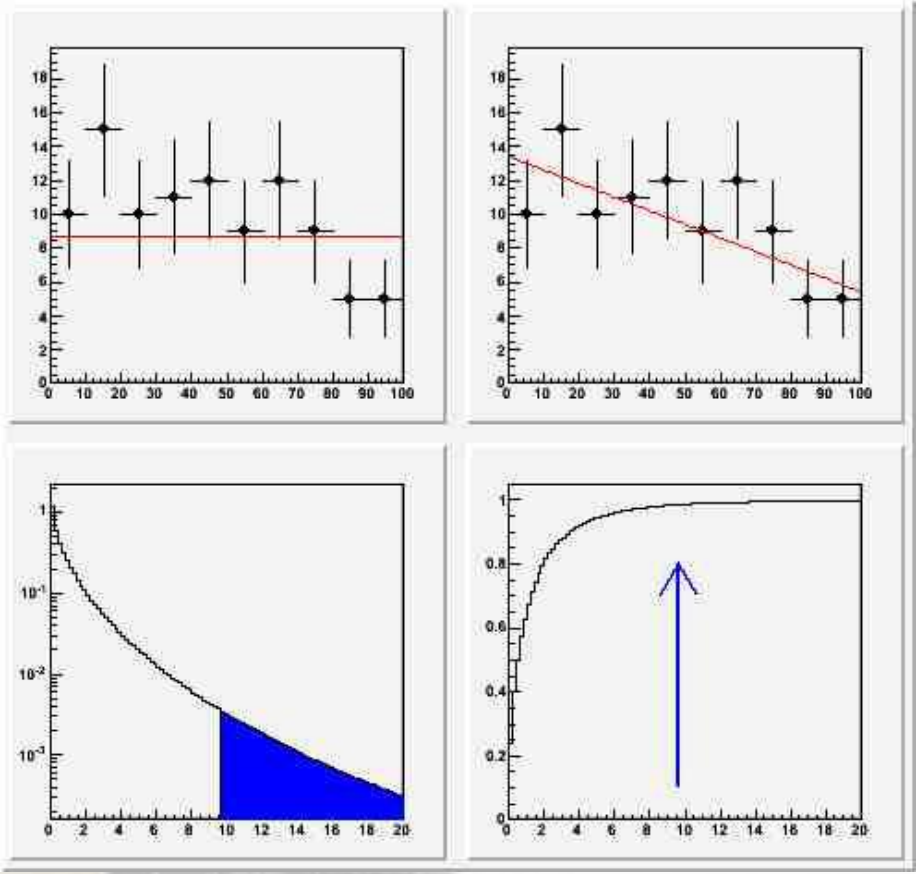
Without diverging, the solution for the data shown (for α=0.1) is that you should pick the linear model.

If you chose the quadratic one you were too greedy: the model overfits the data (it goes through every uncertainty bar)!



The test between constant and line yields **p=0.0146**: there is *evidence* (according to our choice of α) against the null hypothesis (that the additional parameter is useless), so we reject the constant pdf and take the linear fit





The test between linear and quadratic fit yields **p=0.1020**: there is no evidence against the null hypothesis (that the additional parameter is useless). We therefore keep the linear model.

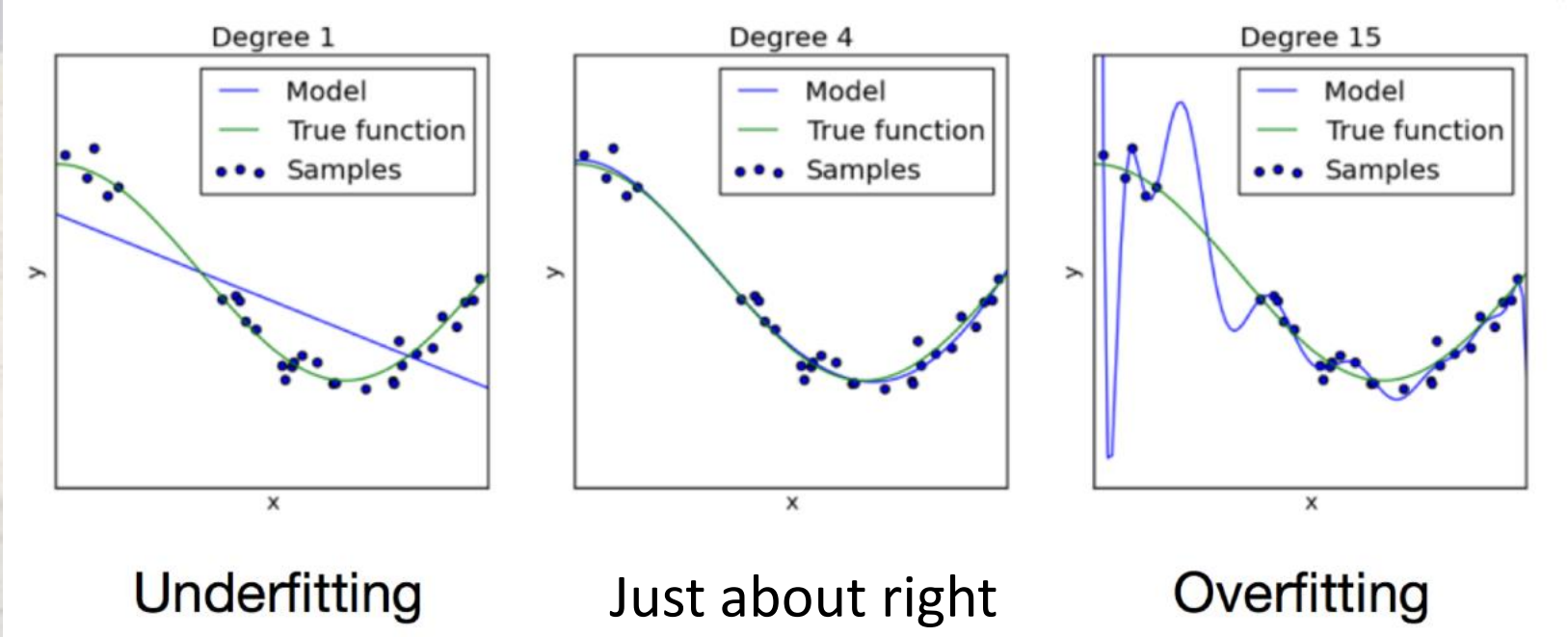
Underfitting and overfitting

What we saw is a general issue: don't overfit!

We will discuss this issue further in the following.

As the complexity of your model increases, you manage to capture more and more of the features of the data. However, one needs to stop somewhere... Your task is to **generalize**, not to interpret just the data you were given.

Generalization fights against systematic uncertainties from the imperfections of the model (**bias**) and against the sensitivity of the prediction (**variance**)



The pains of generalization

We can try to be a bit more formal, to investigate what is going on quantitatively. Suppose you have a model **f(x)** trained on data **x**, that approximates a random variable **t**. So this is a regression task.

In terms of expectation values you can write

$$E[t] = \hat{t}$$

$$E[f(x)] = \hat{f}(x)$$

One can study the generalization error on **t** at any value **x** by expanding

$$E[(f(x) - t)^2] =$$

$$E[(t - \hat{t})^2] +$$

$$\left(\hat{t} - \hat{f}(x)\right)^2 +$$

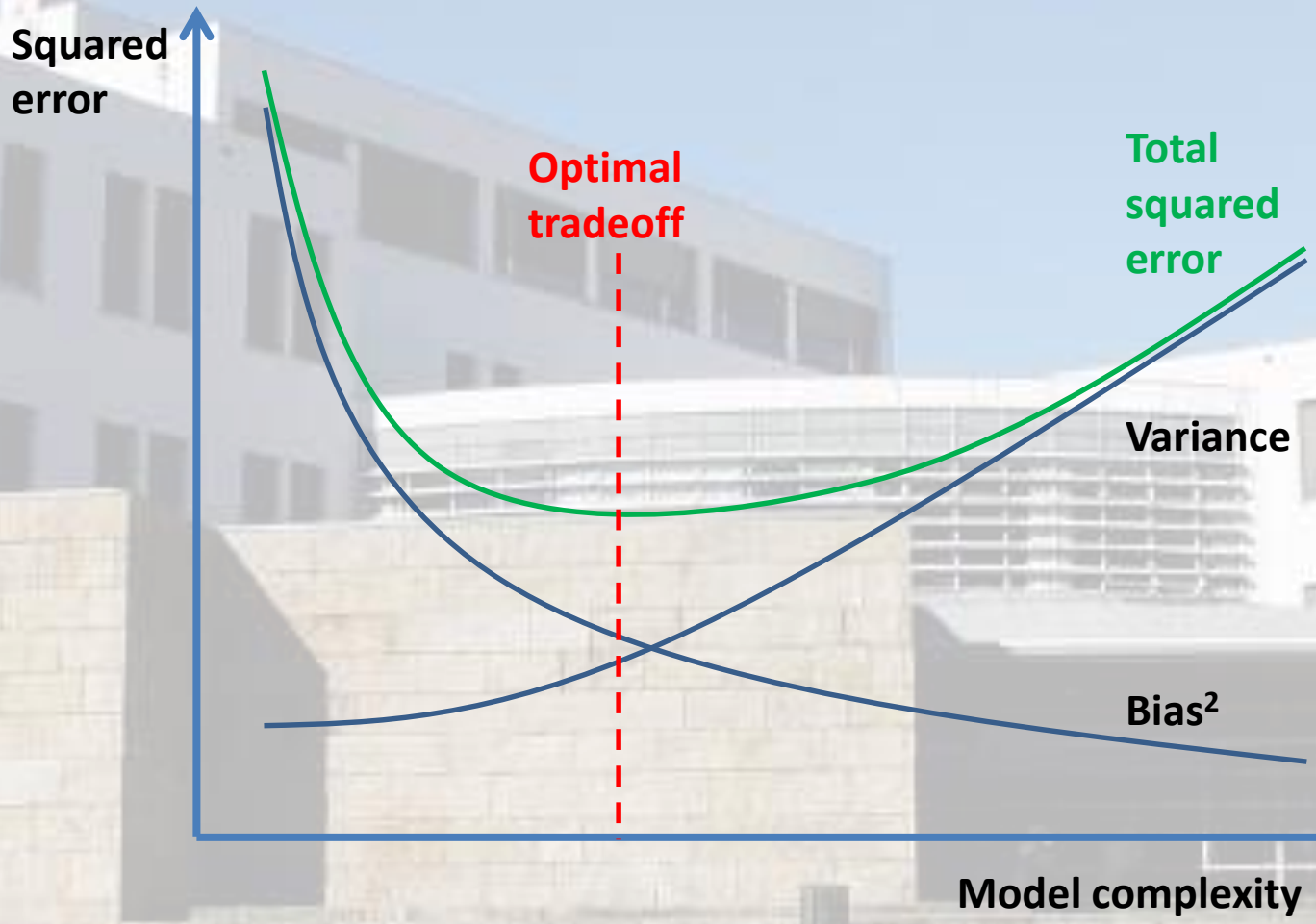
$$E\left[\left(f(x) - \hat{f}(x)\right)^2\right]$$

Squared bias: this can be reduced with a more complex model

Noise term – cannot be improved with modeling

Variance term: the more complex the model, the higher this term, as it tries to capture erratic behavior

Bias-Variance Tradeoff

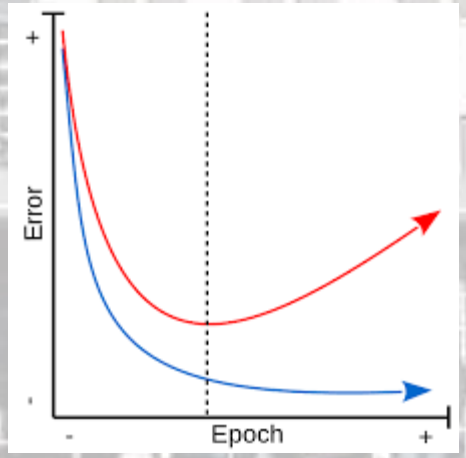


With larger amounts of data the model can become more complex (smaller bias), because the variance diminishes.

Training and testing

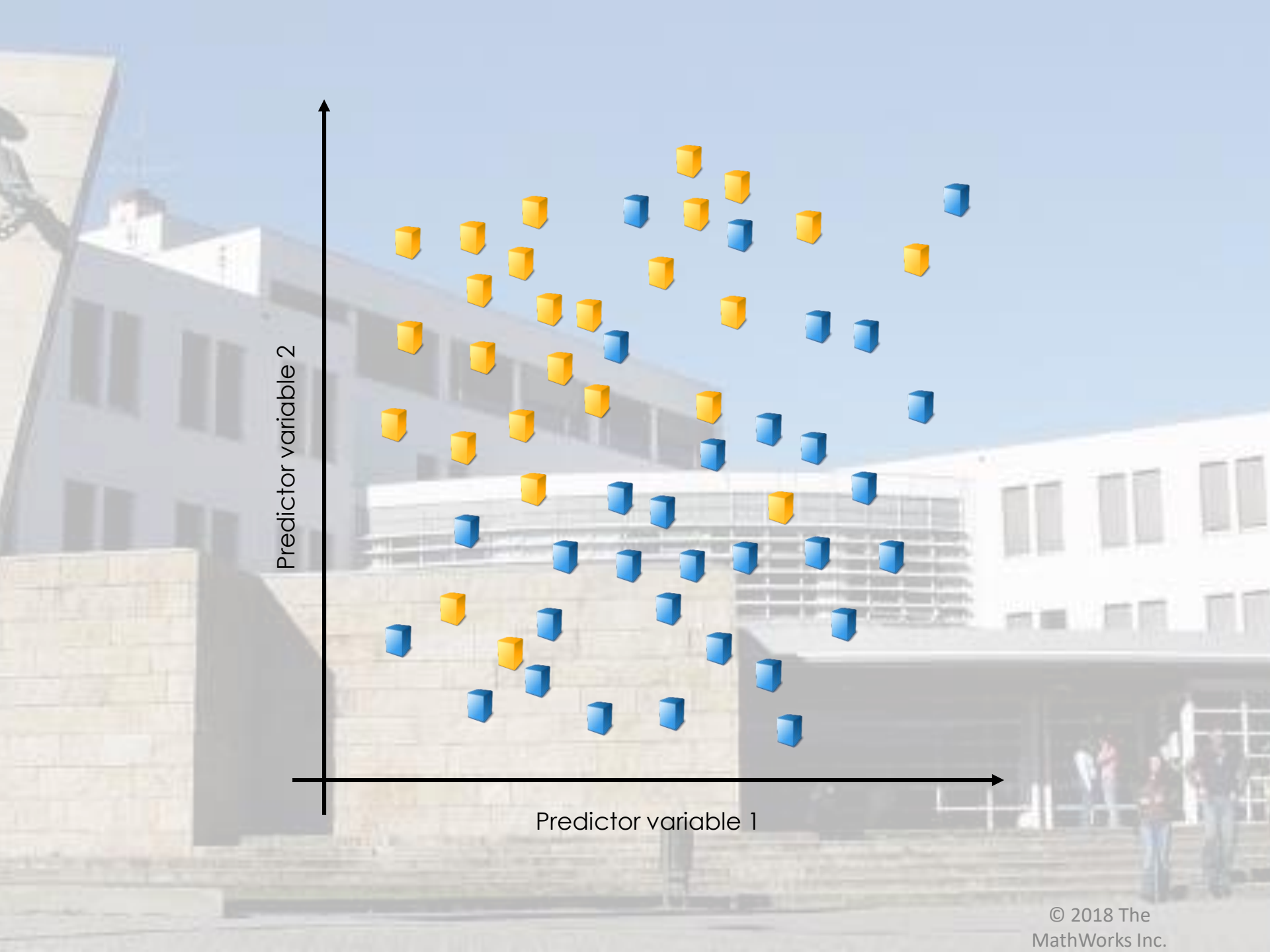
In order to construct and study a classifier built with a supervised algorithm – e.g. a decision tree, one needs labelled data from the two classes. The more data, the better!

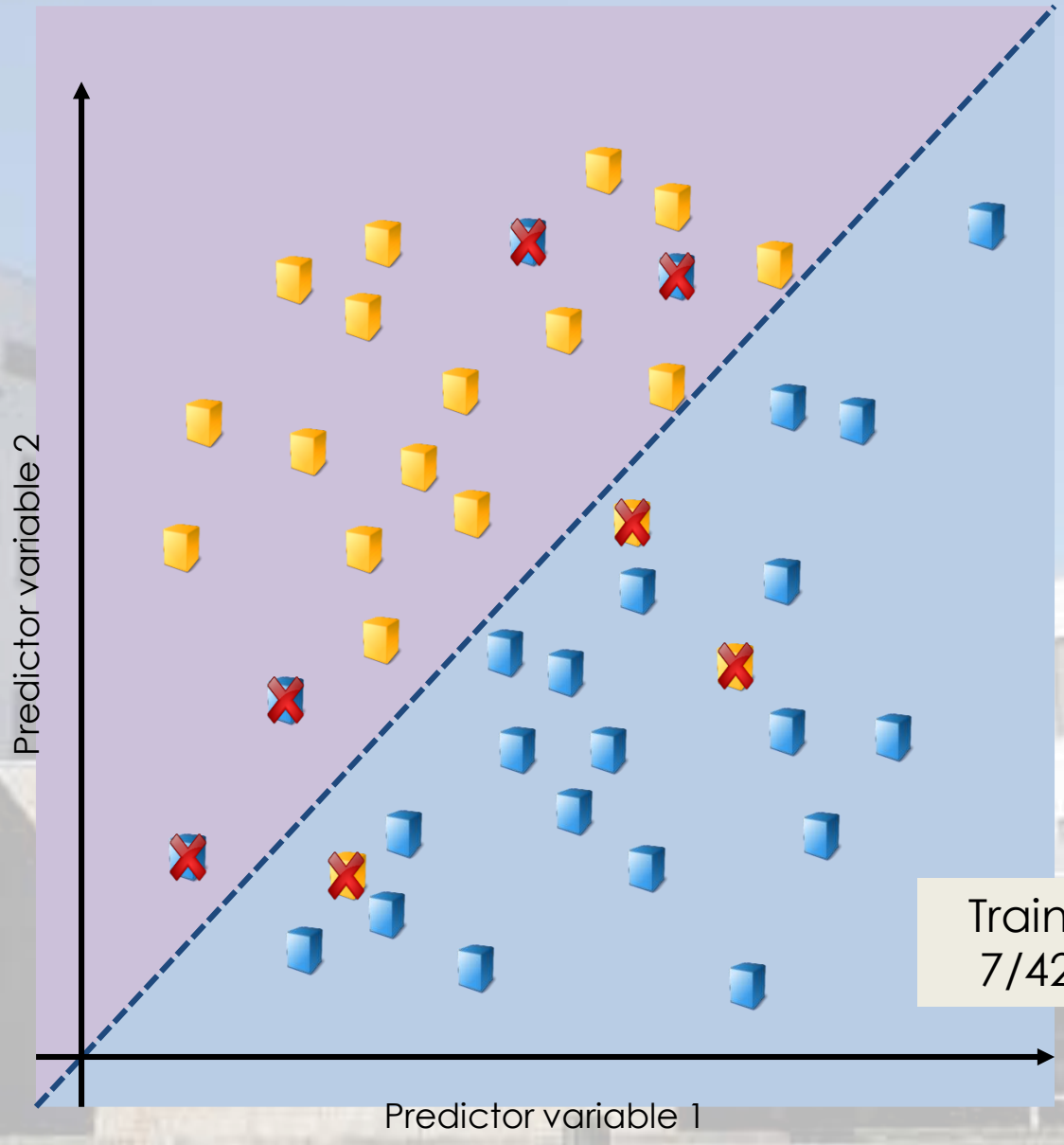
One must decide how to use the available labeled data in the construction of the classifier or regressor. Can we use the same data for both training and testing?
Answer: No – the error estimated in the training phase ("resubstitution error") is optimistic: this is the phenomenon called **over-training**



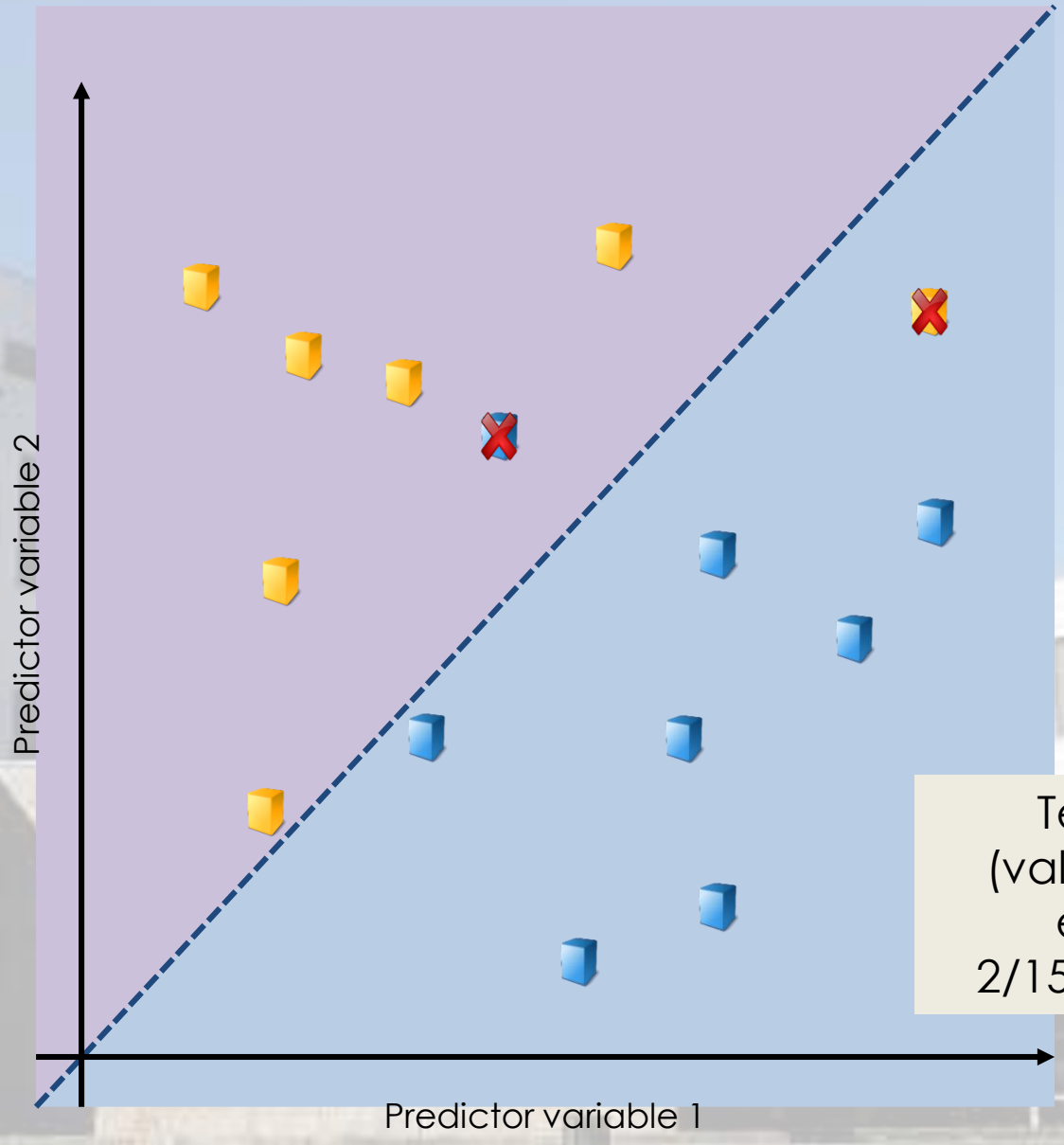
During training, the model attempts to learn the features of the joint distribution p(x,y) of data x and labels (or values, in regression) y

But training data carries **noise** with it – and this component will be learned by a flexible model too, deteriorating the accuracy on data not seen in the training phase

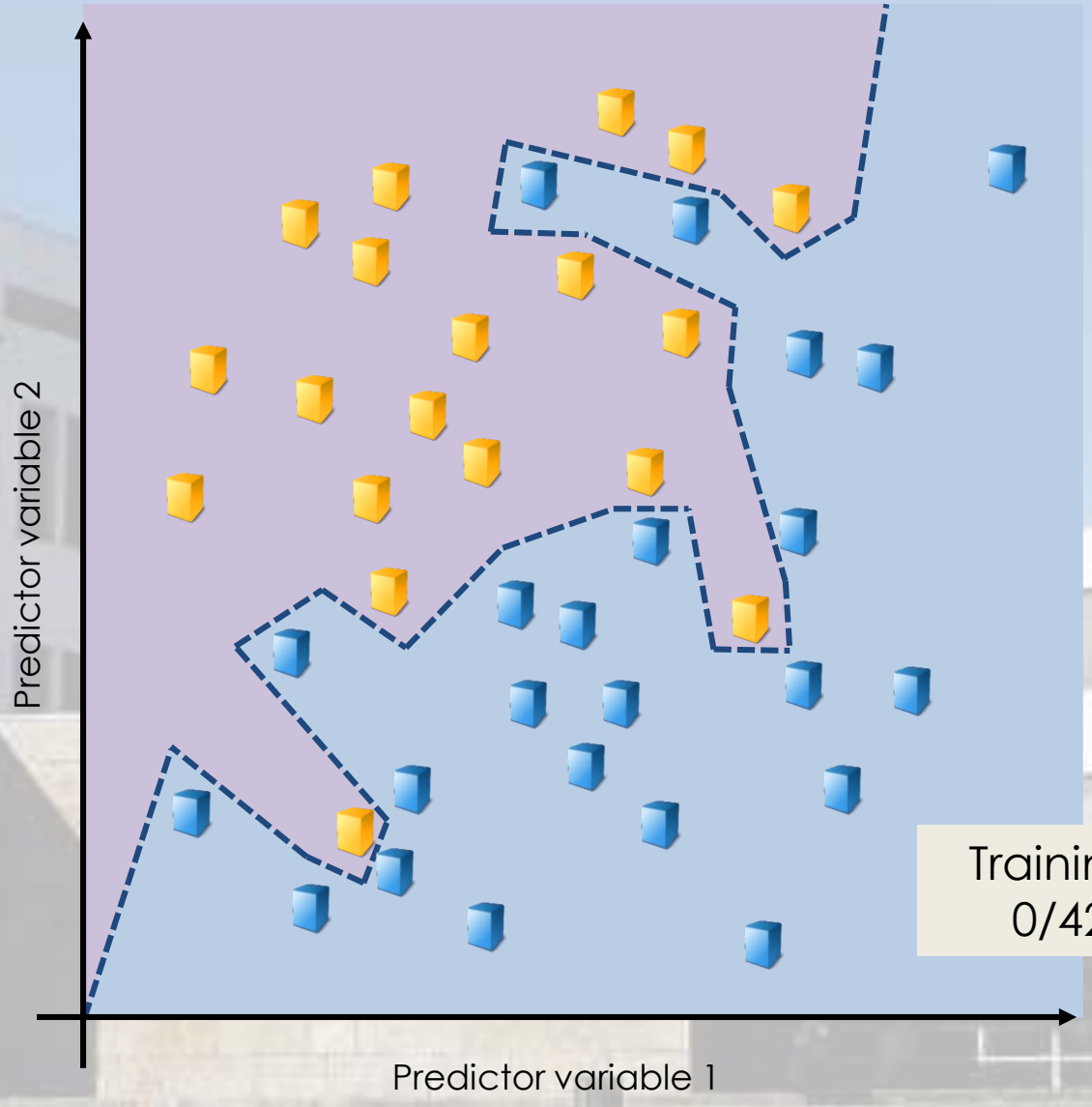


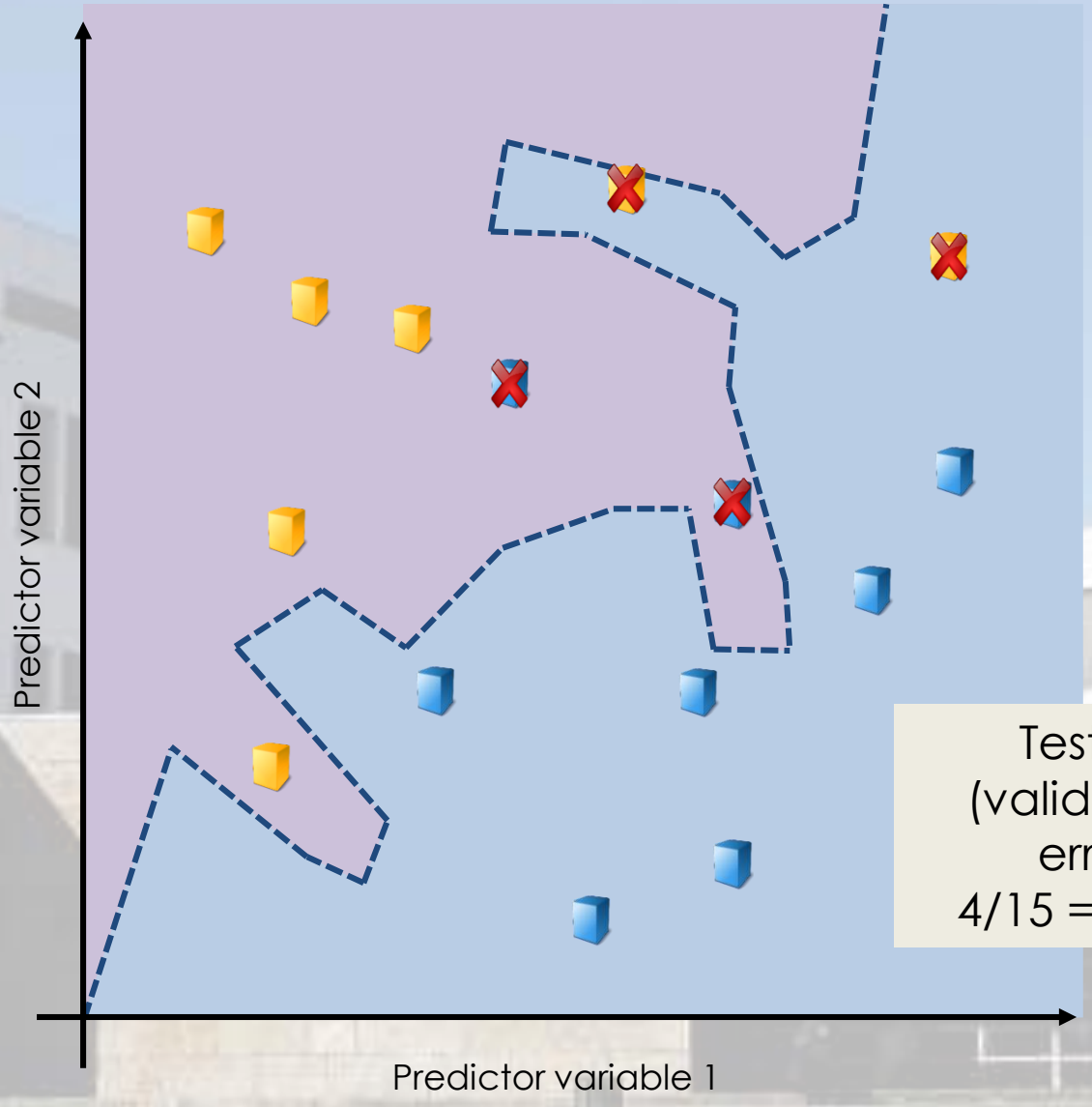


Training error:
7/42 = 16.7%



Testing (validation) error: 2/15 = 13.3%

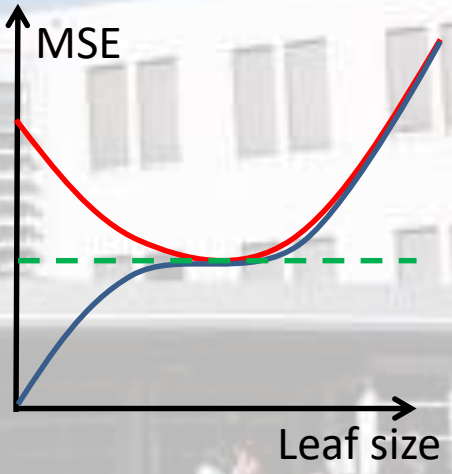




Training vs Testing

So we split the labelled data into training and testing subsets.

Let us consider a decision tree where we vary the leaf size from 1 (perfect classification at training stage) to N (training size, no tree). As the size of leaves increases we go from a flexible, high-variance tree to a robust, but high-bias one.

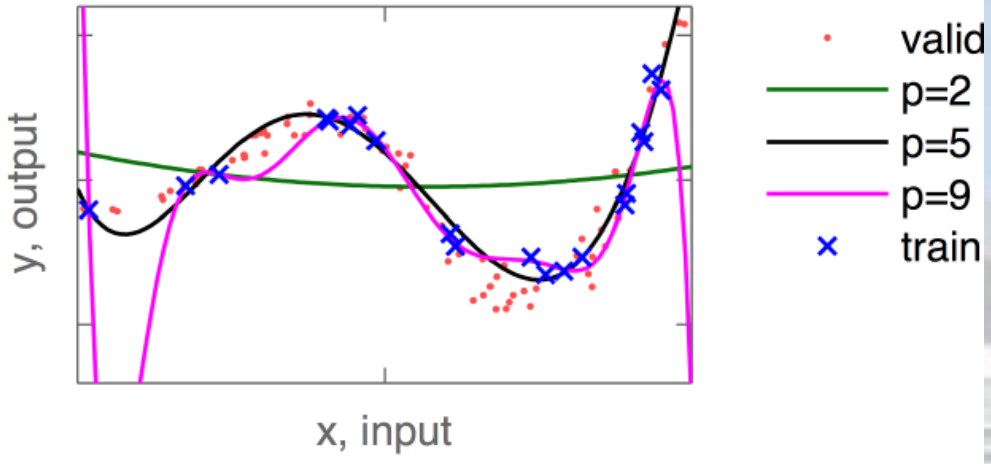


We observe that the training error increases with leaf size.
This is a general property: the training error decreases with the model complexity

The test error instead decreases with leaf size until an optimal value is reached, which is close to the Bayes error.

(The above behaviour is not universal; for e.g. ensemble learners the test error also increases monotonically with leaf size)

One fitting example



Here we have training data (blue crosses) and a few polynomial models that try to capture their variation (y) by minimizing the mean square error

Validation data (red points, call it test data here for now) also shown

The problem cannot be "solved" by training data alone, as a more complex model would always seem to work better than a simpler one

A graph of the MSE vs model complexity is all that is needed to see what model complexity is appropriate.
But **we use test data for that**!

How to compromise?

There is a detail to fix. We want to learn the most accurate possible model given our data, but we also want to **estimate its predictive power** as accurately as possible.

- the predictive power improves with training data size → max that
- the uncertainty on the predictive power decreases with test data size → or max that ?

Data is costly → we need to optimize the partition into training and testing.

We can in principle solve this problem by studying the statistical properties of the misclassification error (see next slide). However, as a spoiler, this will not be sufficient...

Misclassification error

Consider a binary classifier y* = f(x) which yields a misclassification error ε. In statistical jargon we are discussing a Bernoulli trial, modelled with a Binomial distribution

$$P(m, N, \varepsilon) = \binom{N}{m} \varepsilon^m (1-\varepsilon)^{N-m}$$

The test set has size N, and m is the number of failures. The job of the test set is to minimize the length of a confidence interval on ε. The job of the training set is to minimize ε itself.

One can kill both birds with the same stone by minimizing the upper limit on ε. At 95% CL, and under a Normal approx., we may write, for an ensemble of train/test sets:

$$E\varepsilon_{UL} = \varepsilon(N_{train}) + 1.96\sqrt{\frac{\varepsilon(N_{train})(1-\varepsilon(N_{train}))}{N_{test}}}$$

Barring the caveats (Wald approximation, Gaussian approx for UL) is the problem solved?

Yes – the calculation of the appropriate split (which minimizes the expectation value for the UL on the misclassification error) gives a good rule, but...

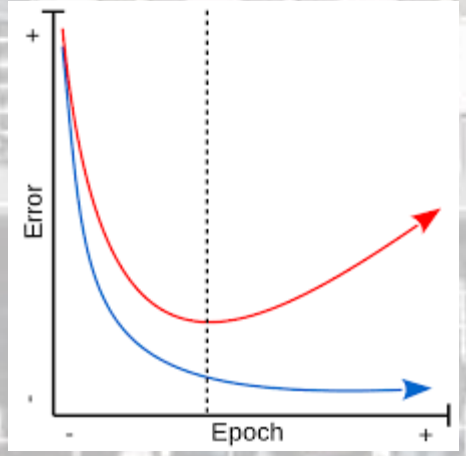
No – we have omitted an important step in the procedure: the optimization of the classifier parameters. **We need an independent sample to do that! → Validation**

Training, Validation, Testing

In order to construct and study a classifier built with a supervised algorithm – e.g. a BDT (see below), one needs labelled data from the two classes. The more data, the better!

Be given a sample of N_S signal events and N_B background events, one usually separates these sets in three parts:

Training set: events used to build the classifier. The algorithm employs them to estimate the prior densities of S and B, or directly the likelihood ratio or a monotonous function of it
Validation set: this is used to understand whether the training was too aggressive (overfitting), and to tune the algorithm parameters for best results
Test set: this sample is totally independent from the former two, and it is used to obtain a unbiased estimate of the final performance of the model, previously learned, validated and optimized.



The quality of the generalization can be further studied by more advanced partitions and resampling techniques. A common one is k-fold cross validation (see below).

Leave-one-out Cross Validation

Data is costly! So, **often it is impractical to keep large holdout samples** for validation.

Imagine you want to optimize a hyperparameter **s** affecting the precision of your model. How to proceed?

You can come up with the idea of removing just one event from the set. Train the learner on N-1 events, and test it on the N^{th} one:

$$MSE_N = (y_N - y^*)^2$$

That is a very rough estimate! But we can iterate on all N and take the mean:

$$\langle MSE \rangle = 1/N \, Sum_i \, (MSE_i)$$

You are using almost all data for training, so the returned answer is stable and has low variance; plus, the method is deterministic.

LOOCV is good, but can be **very CPU consuming** for large samples → use it only for very small data sizes.

k-Fold Cross Validation

Leave-one-out cross validation can be generalized by leaving out $1/k^{th}$ of the data

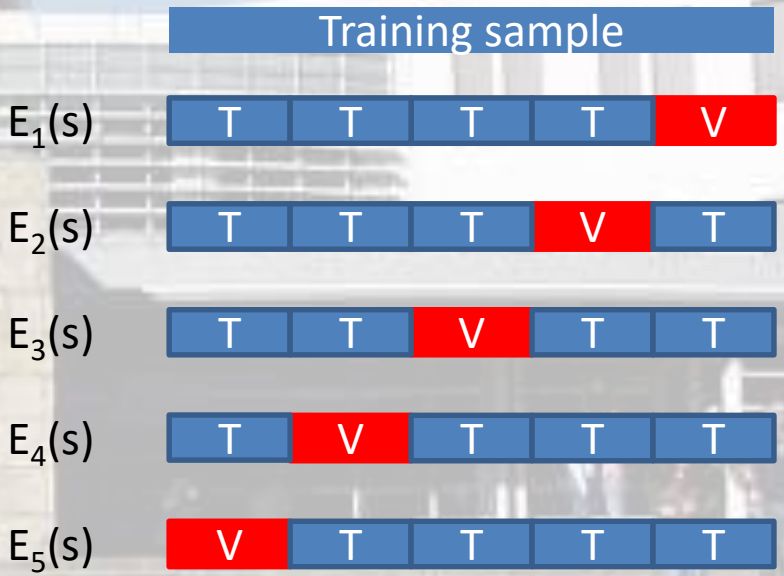
If k is >4, the training accuracy is not affected significantly, and we still get a reasonable estimate of the uncertainty.

Recipe:

- Divide training data into N equally sized subsets (N=5-10 is typical)
- For each k=1...N, train a classifier, or fit data with k-th sample as hold-out; apply to k-th subset and obtain error (or loss) $E_k(s)$
- Obtain CV error by averaging:

  $CVE(s) = 1/N \; Sum(E_k(s))$

- Pick **s** such that CVE is minimum



Narsky advises N=10 for most problems.

LOSS MINIMIZATION

Behind the scene of the picture

In the pictured situation, there are two possible outcomes:

- the dealer has 21 (p=4/13) → you break even
- the dealer does not (p=9/13) → you win 1.5 times your bet

E(win) = 1.5*9/13 = **1.038** (in initial bet units)

It's looking good! However, you might end up not winning anything... max loss = - min win = 0.

If you however place an insurance bet of X units (on the dealer having blackjack),

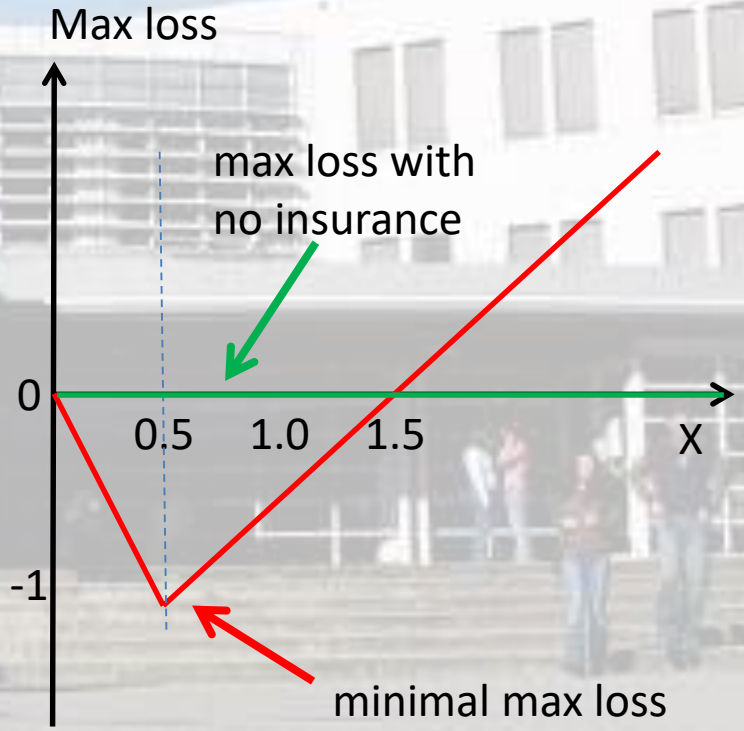
E(win) = (1.5-X)*9/13 +2X*4/13 = **1.038 - X/13**;

So the insurance isn't doing any good to your expectation... But OTOH, now

max loss = -min [1.5-X,2*X]

→ by taking an insurance bet you decrease your expected win, but with X=0.5 you minimize your max loss (i.e. maximize your minimum win)

(**at Casinos, max X allowed is 0.5 anyway**)



Gradient Descent

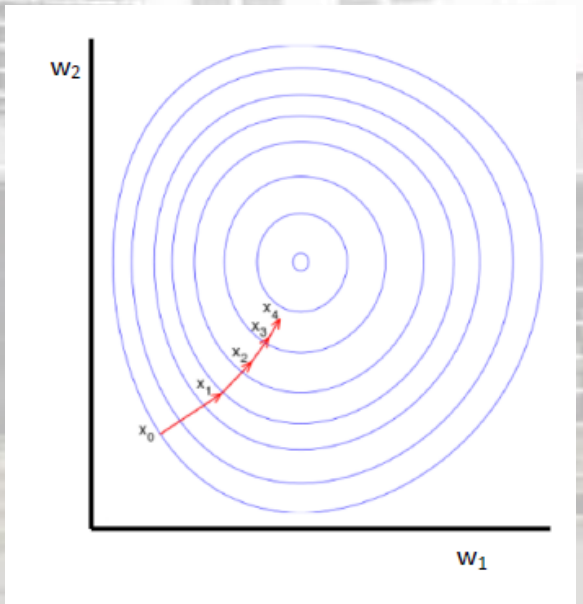
To minimize –log L and find optimal value of model parameters, in the absence of an analytical description, we "descend" toward the minimum by approximating the shortest route with local information:

1) find gradient of L w.r.t. parameters w: $\frac{\partial L(w)}{\partial w}$
2) update parameters:

$$w' \leftarrow w - \eta \frac{\partial L(w)}{\partial w}$$

and iterate.

Success depends on how fast you descend, moduled by "learning rate" η.



Stochastic Gradient Descent

Computing the gradient over the whole training set at each step is sub-optimal:

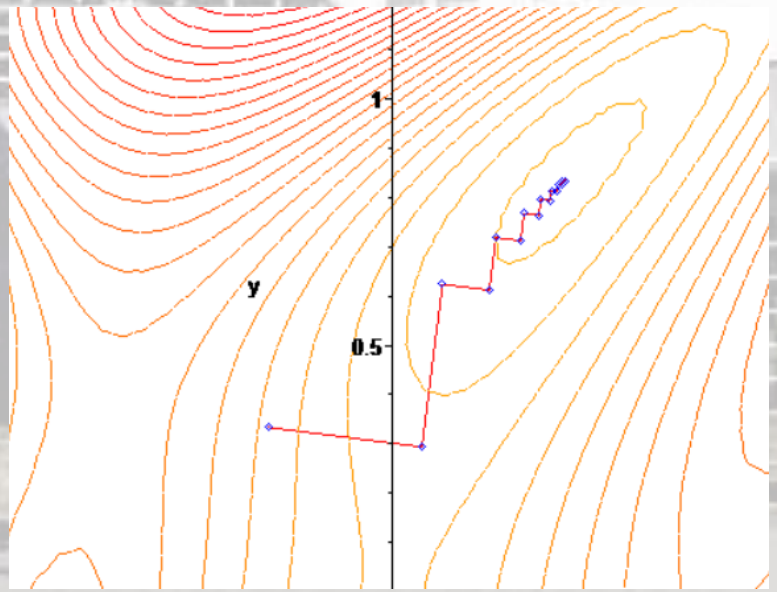
- CPU-intensive (must pass all dataset)
- large memory use, intractable if too large datasets
- does not allow updates on-the-fly (adding data online)

Also, it can become ineffective, as risk of getting stuck in local minima is large in multi-D

Most modern deep ML methods employ "stochastic" techniques to find the optimal working point / parameter values

This relies on the possibility to decompose the loss function into the sum of per-example losses.

SGD updates parameters on a per-event basis → objective function becomes noisy, but this has merits (can jump out of local minima)



Mini-batch SGD

One may improve on per-event SGD updating by computing the gradient over small batches of training data

→ get the best of both worlds:

- fast
- no memory issues
- scales well with data size
- still can jump out of local minima
- noise averages out a bit

Recipe:

$$w \leftarrow w - \eta \cdot \nabla_w \mathrm{L}(\mathrm{w}; x^{i:i+M}, y^{i:i+M})$$

Advanced descent strategies

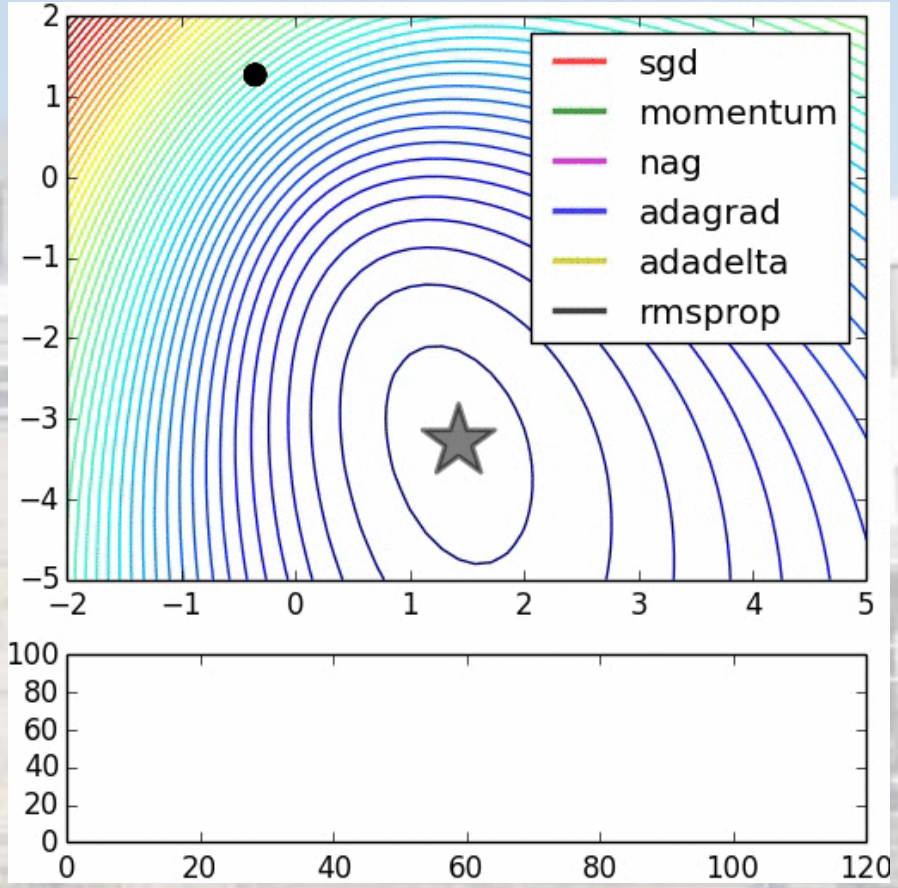
Finding the real, absolute minimum of a function with many parameters in a multi-D space can be *very* tricky, and take a lot of time

A number of variants of mini-batch SGD exist on the market:

- momentum descent
- ADA gradient
- ADAM
- Nesterov accelerated gradient (NAG)
- and others

Their performance depends on the specific problem, the data size, sparsity, etcetera

Take-home bit: don't just pick the first off the shelf. Experiment with different methods, try to understand what is best for your case



http://danielnouri.org/notes/category/deep-learning/#changing-learning-rate-and-momentum-over-time

Escaping saddles

It has been shown that the most challenging issue of SDG is that it tends to get trapped in saddle points, rather than local minima

Also, saddles are way more common!

- Easy to understand: as number of dimensions grow, probability that a point which has zero local gradient is a minimum in all directions goes to zero

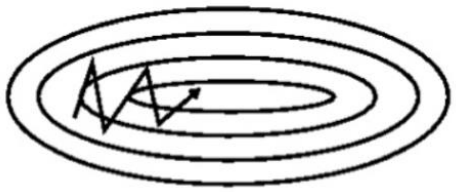
Variants of SGD try to address this in creative ways

- decrease η according to some schedule
- emulate physical rolling down
- etcetera

E.g. Momentum: leverages "memory" of which gradient component consistently show decrease in loss to increase shift in relative direction

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

(a) SGD without momentum



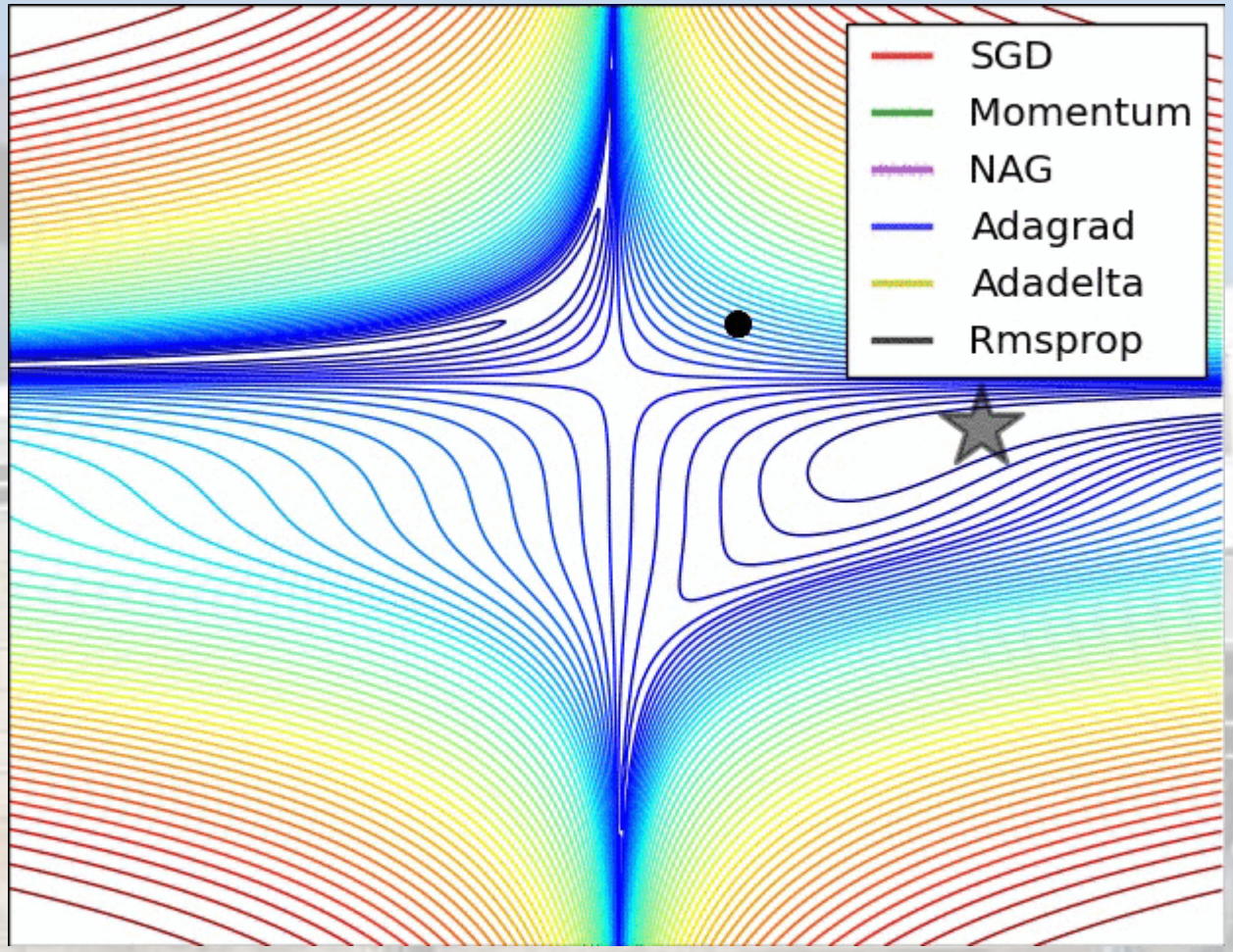
(b) SGD with momentum

Escaping the saddle/2

Another nice animation shows how different SGD strategies perform on a complex surface

Methods that adapt the learning rate to the situation at hand, and explore the gradient components (ADA, RMSprop, ADAM) are faster in escaping saddles and progressing toward the real minimum

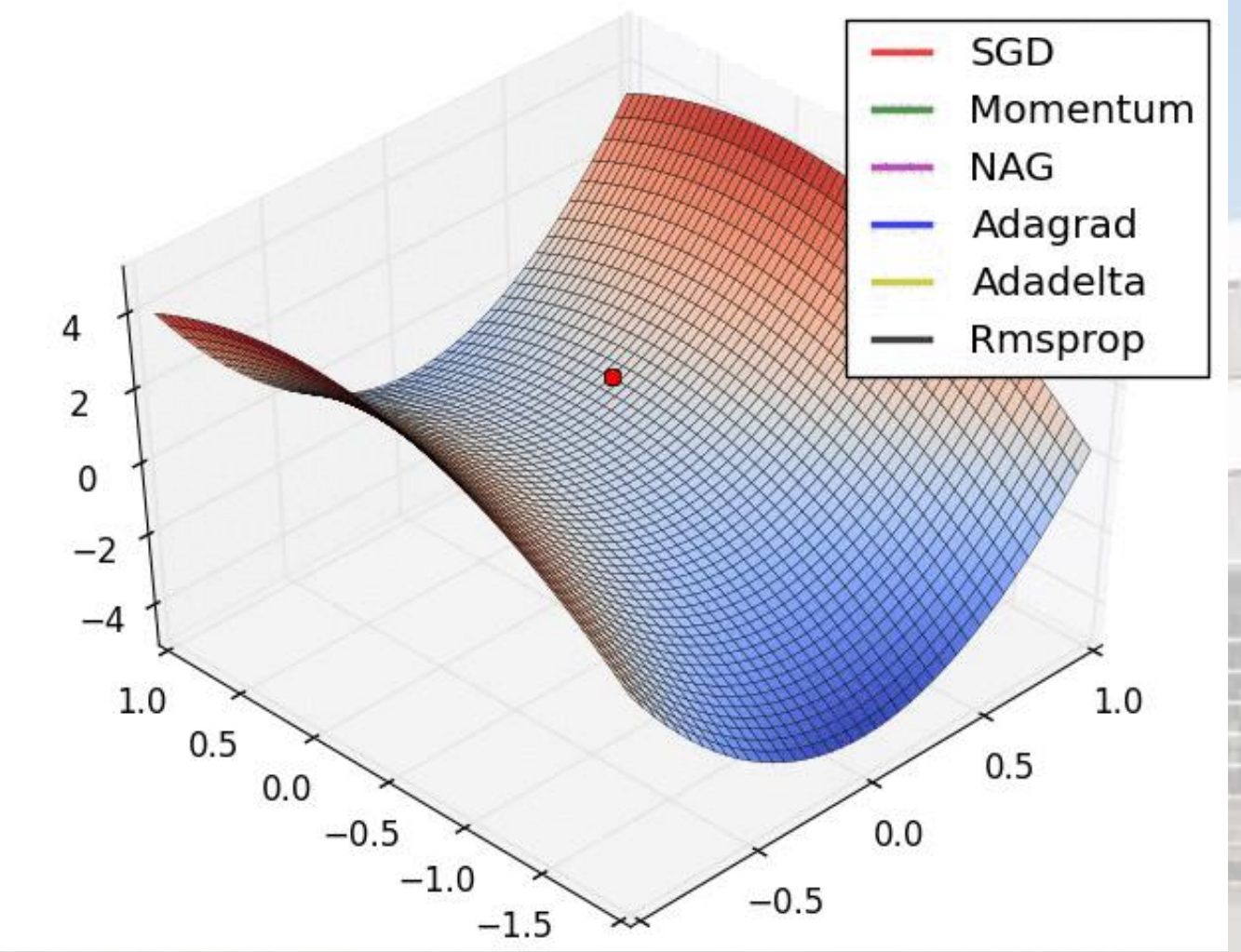
Funny to see how their trajectories look longer and less meaningful – their strength is in the speed here



http://ruder.io/optimizing-gradient-descent/

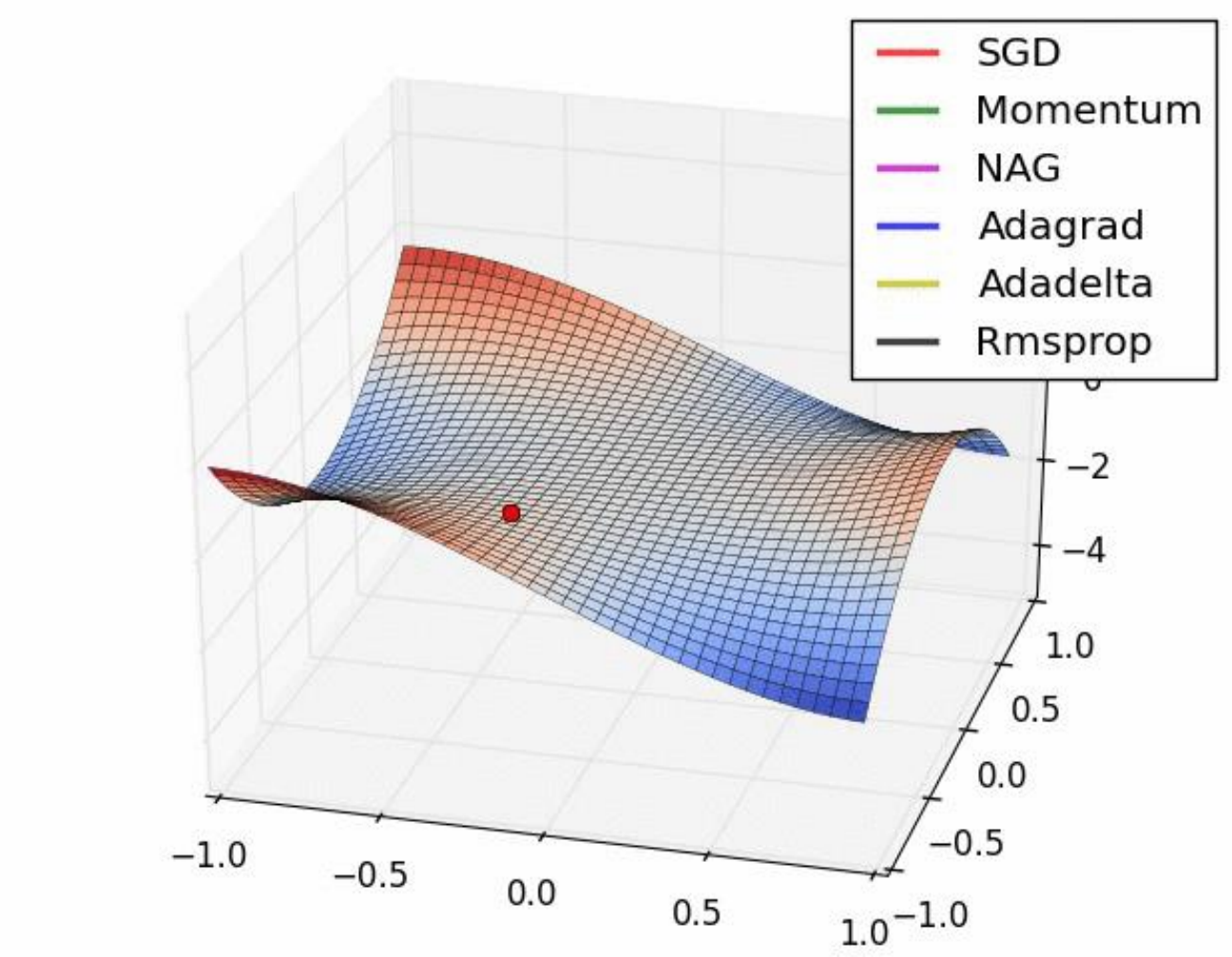
Escaping the saddle / 3

In this example one may more clearly see the risk SGD runs in saddle points



Escaping the saddle / 4

Here the oscillation of the SGD solution is not perfectly aligned with the direction of zero gradient, so it also ultimately escapes, but computationally the shortcomings are evident with respect to more advanced methods



A COUPLE OF METHODS

MVA methods

Life is short, and there are heaps of good ideas to construct smart methods for supervised learning tasks...

In the following we give a quick look at linear discriminants and Support Vector Machines, and then focus on decision trees /BDTs and neural networks

One thing to mention at the start: it _is_ important to learn the easy methods – in many applications they are all that's needed, if not better than complex tools like DNNs with all their bells and whistles

→ Trevor Hastie: kNN is as good or better than anything else in 2/3 of all problems !

I think that's a bit of a stretch (also depends on what prior distribution of problems you are considering), and yet we have to stop and think...

Fisher Linear discriminant

The Fisher discriminant constructs a **linear boundary** between two classes by finding a projection of the data that maximizes their separation

The idea is similar to linear discriminant analysis (not discussed here), however there is no assumption of normality or equal variance of the classes

There are two measures of relevance: the separation of the means along the chosen projection, and the spread of each distribution

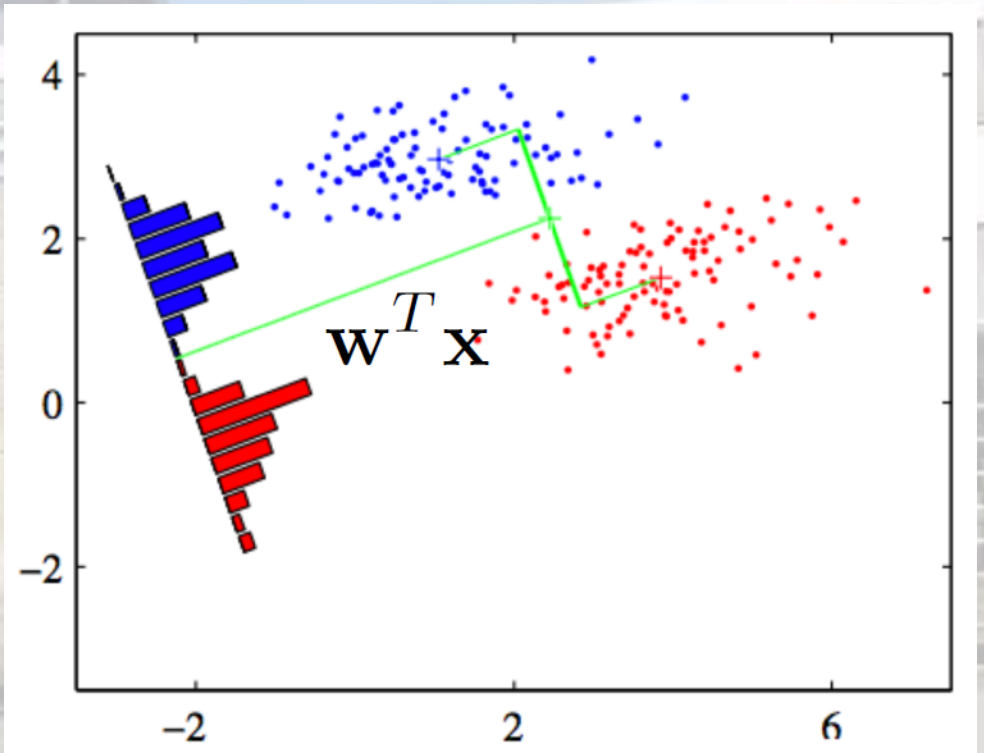
If two classes have means μ_1, μ_2, and covariances C_1, C_2 we can define:

The between-class variation

$$S_B = (\mu_2 - \mu_1)^T(\mu_2 - \mu_1)$$

The within-class variation

$$S_W = \sum_{j=1,2} \sum_{i \in cl\ j} (x_i - \mu_j)^T (x_i - \mu_j)$$



Fisher linear discriminant /2

Fisher criterion tries to maximize SB and minimize SW, so it is in fact written as a ratio of the two. One finds the vector w (orthogonal to the separating plane) which maximizes the function

$$S(w) = \frac{(w \cdot (\mu_2 - \mu_1))^2}{(w^T (C_1 + C_2) w)}$$

The solution is found for w proportional to

$$w \propto (C_1 + C_2)^{-1} (\mu_2 - \mu_1)$$

This is easy to compute.

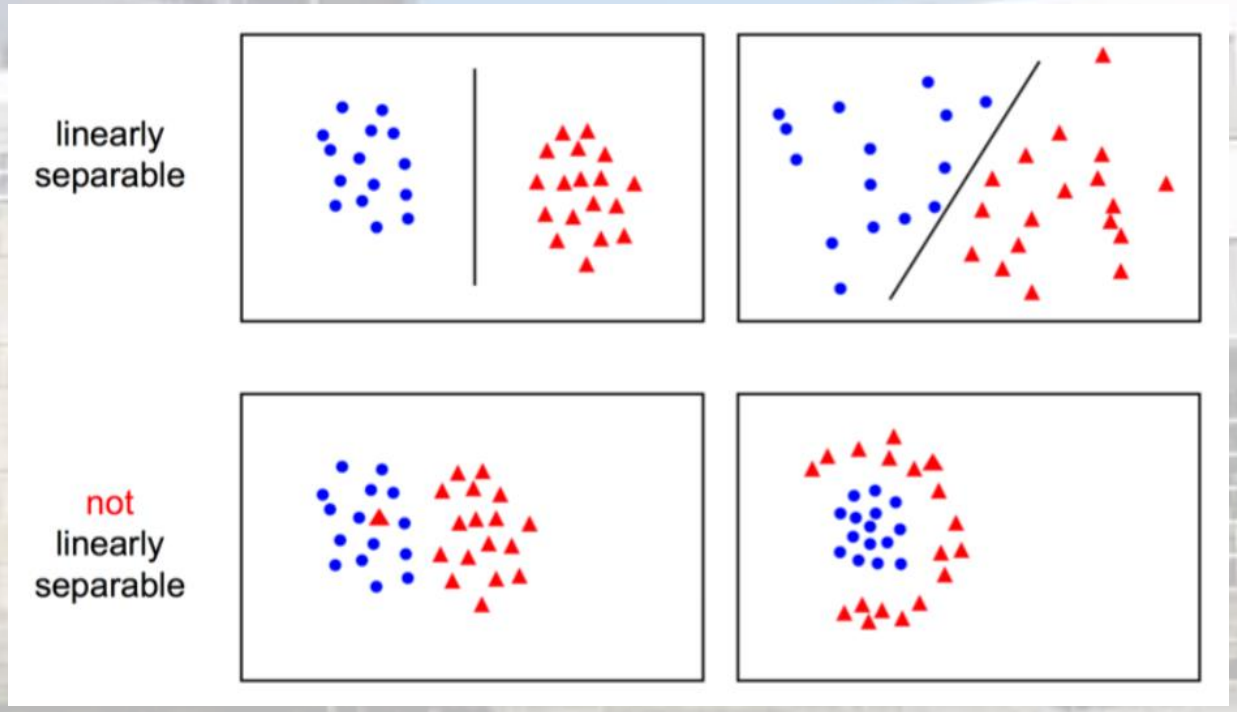
The Fisher discriminant is optimal for normal MV distributed data of different means and same covariance; performance deteriorates with complexity

Support vector machines

A SVM is a binary linear classifier that tries to find the best separation between two classes of data

The basic concept can be again explained with linear separation between the classes (a hyperplane in the feature space)

Non-linear separation is possible by extending the technique to additional dimensions → we do not make the discrimination surface more complex, but we make the data more complex (see below)

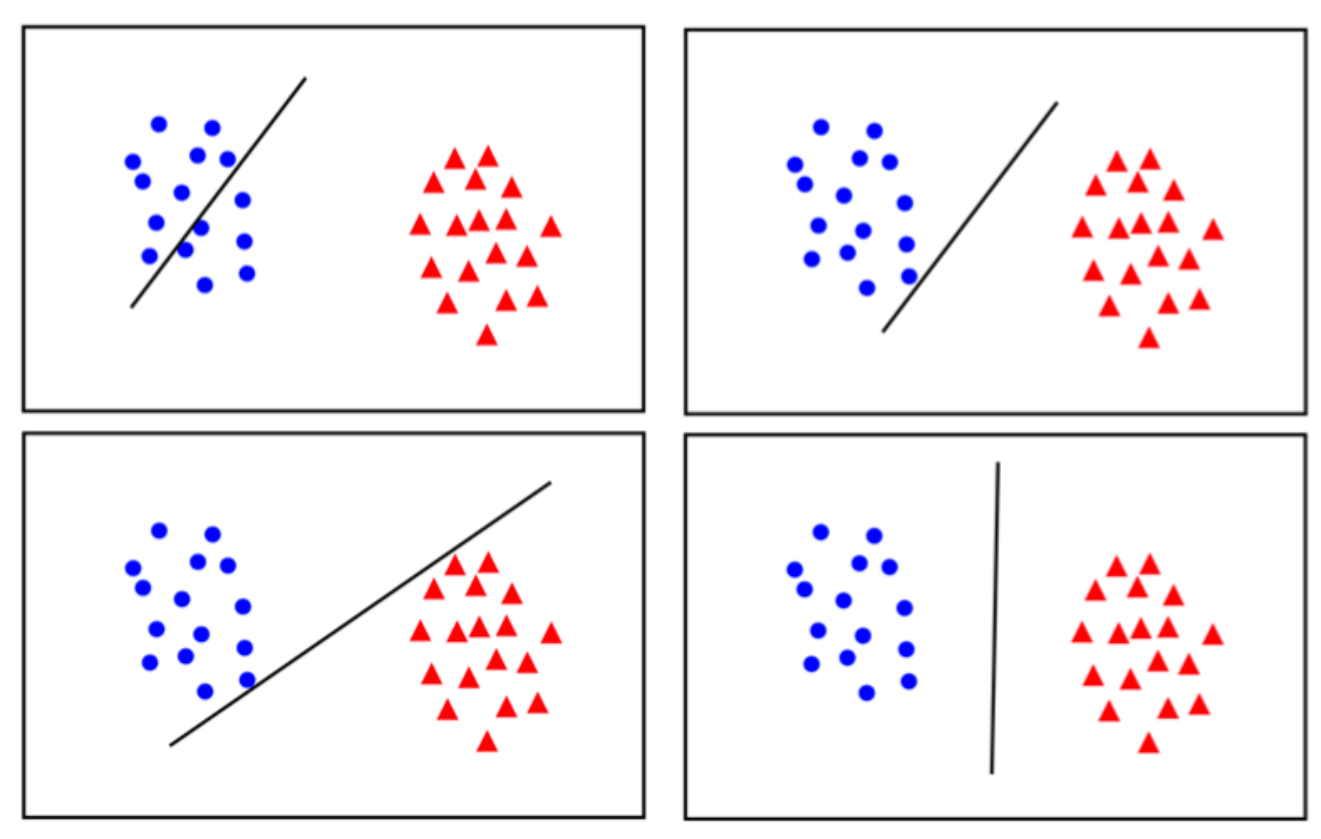


Which hyperplane?

Besides the obviously bad solutions, there can be an infinity of good solutions that work perfectly with linearly-separable training data

However, there is only one "best" solution

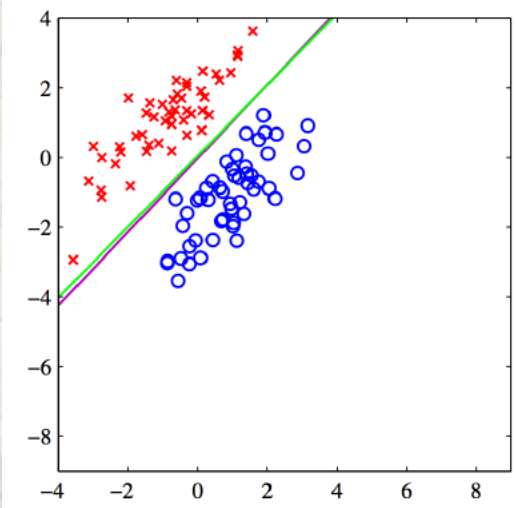
→ need to discuss a proper recipe to find it

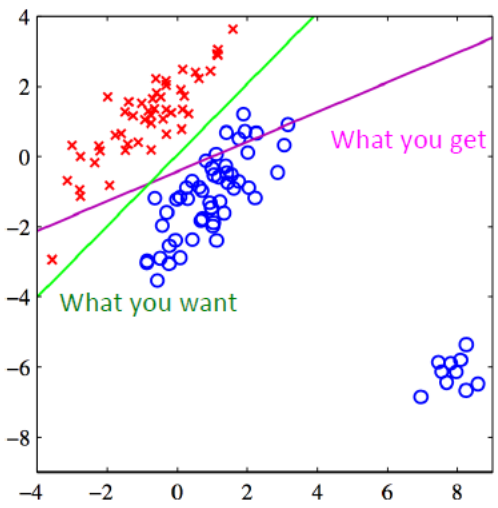


A bad idea

Can we learn a decision boundary between two classes by using a least square loss ?

→not a good idea. Too sensitive to outliers...





$$L(\mathbf{w}) = \frac{1}{2}\sum_i (y_i - \mathbf{w}^T \mathbf{x}_i)^2$$

[Bishop]

Linear SVM

Take two classes of i.i.d. training data $\{(x_i, c_i)\}$ where c is the class label, either +1 or -1, and x is D-dimensional. For now let us consider linear separable classes.

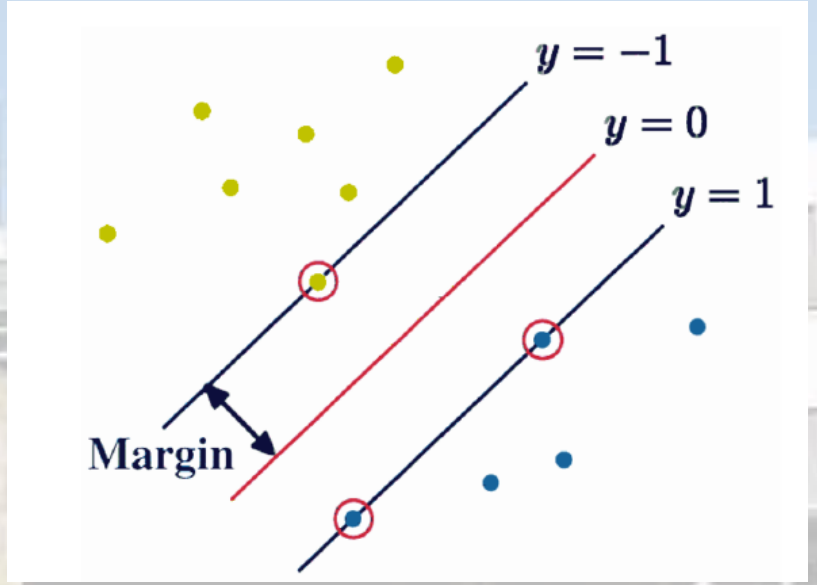
SVM find the hyperplane that maximizes the distance of each class from the plane separating them. The decision score can be set as the value of

$$f(x) = \beta^T x + \beta_0$$

and we can define the separating hyperplane as the one for which f(x)=0; examples with f(x)>0 are then classified in class c=1, and those with f(x)<0 are in class c=-1.

One can always rescale the coefficients such that f(x)=+1 at the plane that intercepts the edge of the positive class, and f(x)=-1 at the plane supported by the edge points of the negative one:

$$c(\beta^T x + \beta_0) = \pm 1$$



The classification margin is cf(x), and the loss function appropriate for training a SVM is called **hinge loss**:

$$\sum_{i=1}^{N} [1 - c_n f(x_n)]_+$$

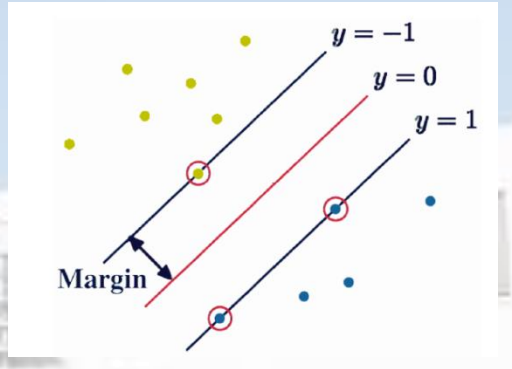
The loss is zero in a linearly separable training sample.

Finding the optimal hyperplane

We write the linear classifier as $h(x;\beta) = \beta^T x + \beta_0$

To find the distance of each data point to the decision boundary we calculate, with c_i=+-1,

$$\frac{c_i(\beta^T x_i + \beta_0)}{\sqrt{\beta^T \beta}}$$



The best boundary will have β such that

$$\arg max_\beta \left\{ \frac{1}{\sqrt{\beta^T \beta}} min_i [c_i(\beta^T x + \beta_0)] \right\}$$

We can equivalently write our optimization problem as the one of

- minimizing $\mathrm{L} = \sqrt{\beta^T \beta}$
- with the condition that $c_i(\beta^T x_i + \beta_0) \geq 1 \quad \forall i$

This corresponds to finding the separating hyperplane that offers the **widest margin** between the points of the two classes

If distributions are not separable

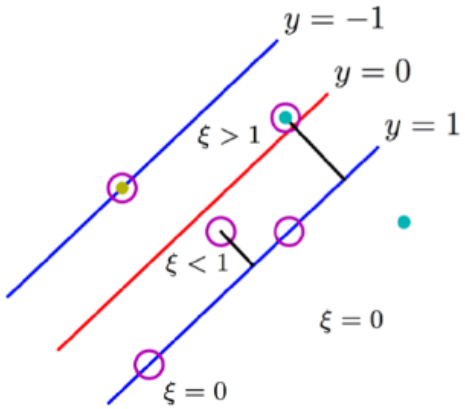
$$\arg\min_{\mathbf{w},w_0} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_i \xi_i$$

$$s.\ t.\ \ y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1 - \xi_i \text{ for all } i$$

$$\text{and } \ \xi_i \geq 0$$

Add a smearing to the margin, $\xi \geq 0$

- If $\xi = 0$, example correctly classifier
- If $0 < \xi < 1$, example correctly classified, but in margin
- If $\xi > 1$, example incorrectly classified

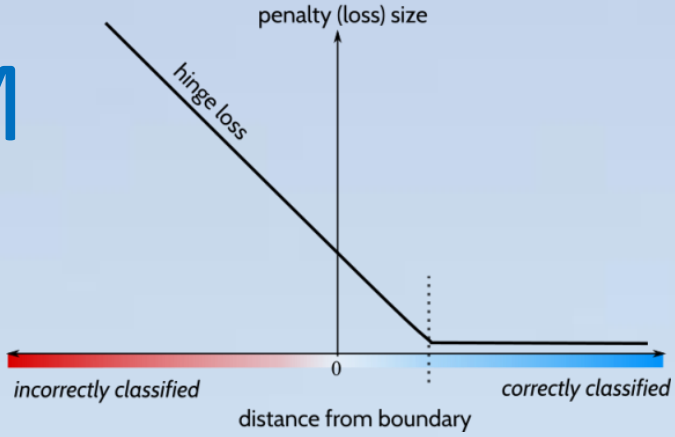


Add regularizer to problem to constrain ξ_i not too large

- C is the regularization hyperparameter that controls how much "softening" of the boundary is allowed, thus how big is margin

[M. Kagan]

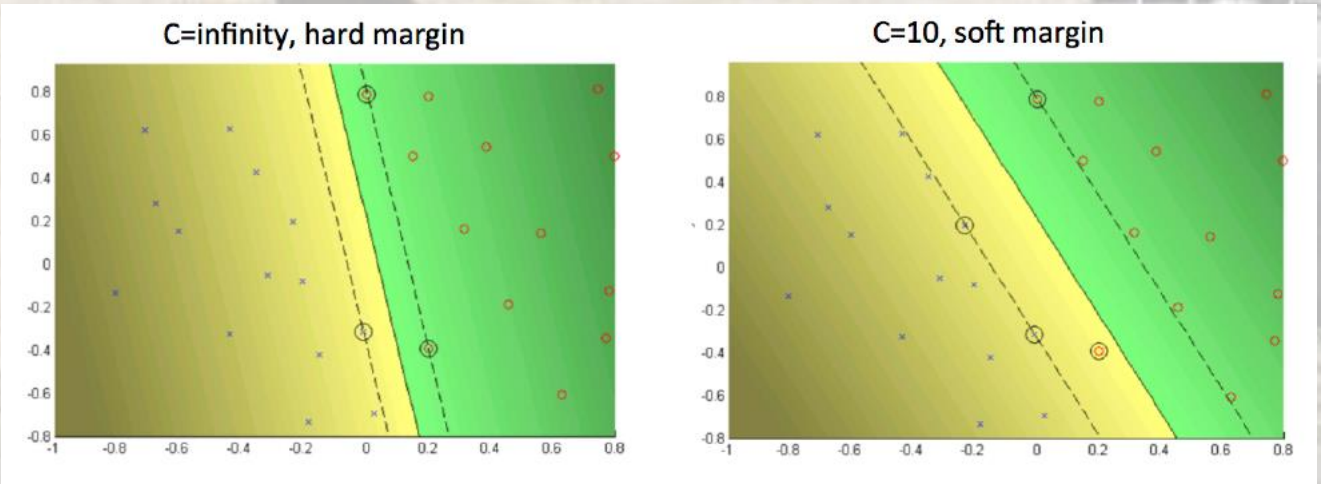
Loss for SVM



The loss function for a SVM can be defined as

$$L(\beta) = c\sum_i \max[0,1 - c_i(\beta^T x + \beta_0)] + \frac{1}{2}\sum_i {\beta_i}^2$$

c is a hyperparameter that controls the regularization: how "hard" is the boundary.



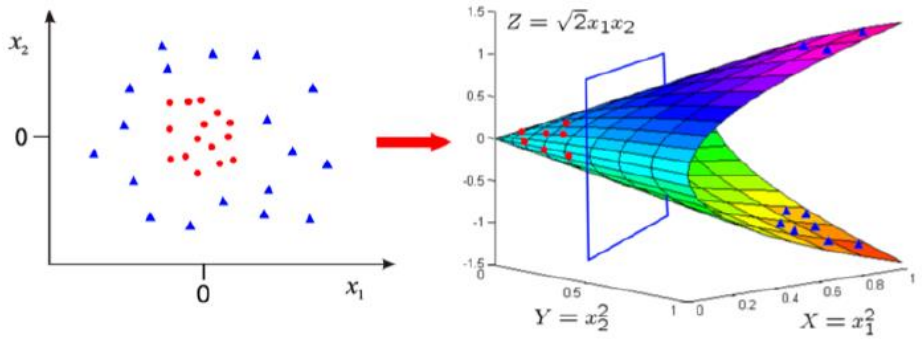
Non-separability: increase dimensions

What if we want a non-linear decision boundary?

– Choose basis functions, e.g: $\phi(x) \sim \{x^2, \sin(x), \log(x), \ldots\}$

$$p(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T \phi(\mathbf{x})}}$$

$$\Phi : \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \begin{pmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \end{pmatrix} \quad \mathbb{R}^2 \rightarrow \mathbb{R}^3$$



[M. Kagan]

The map increases the data dimensionality, $R^m \rightarrow R^k$

$$h(x; a, \beta) = \sum_i a_i y_i \varphi(x)^T \varphi(x) + \beta_0$$

If k large, CPU problems... but we can rely to the Kernel trick (see next slide)

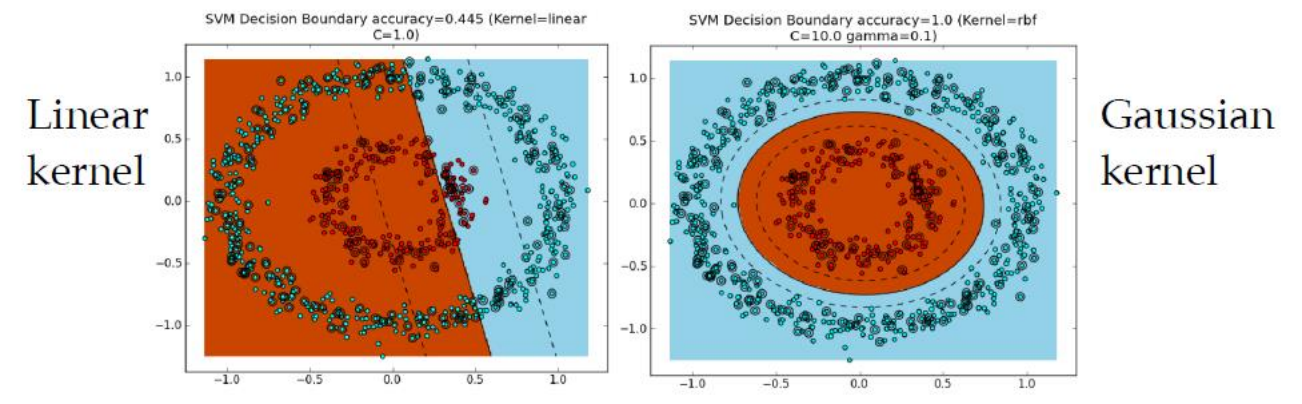
Kernel trick

By transforming the data with a map ϕ(x), we may find a hyperplane in the larger dimensions space where the classes are linearly separable

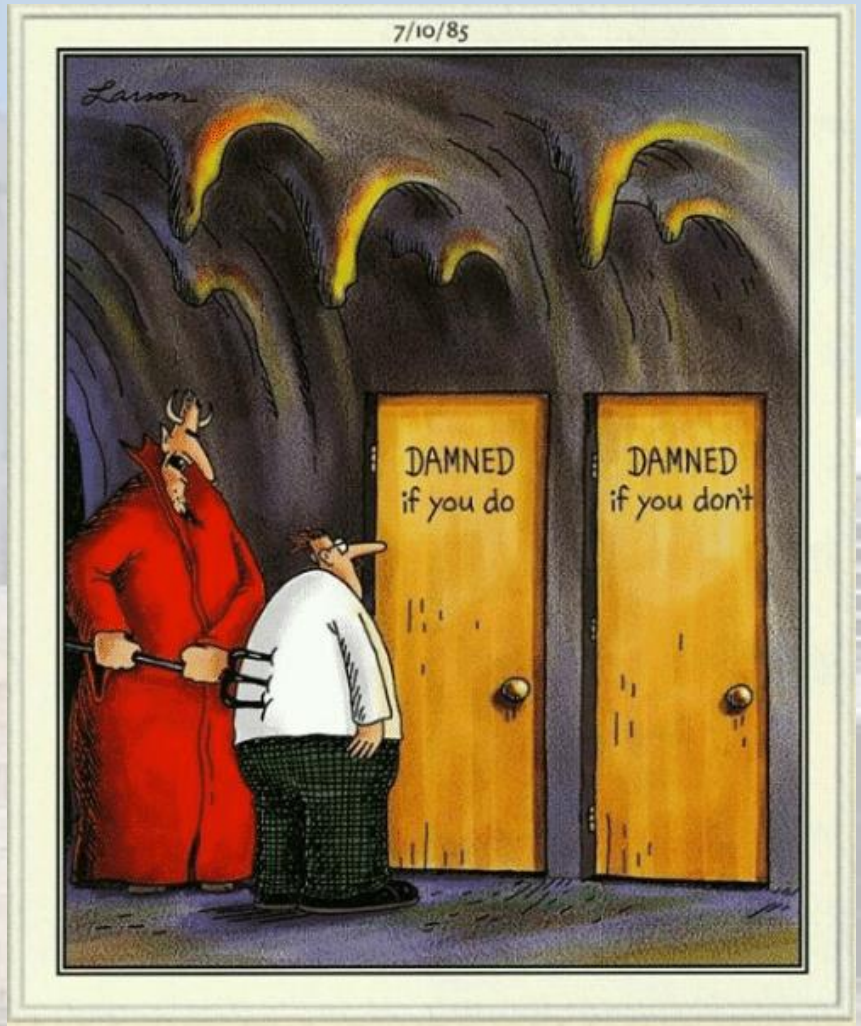
We do not need to compute the full transformation of the data: we just need to compute the "kernel" K(x,x') = ϕ(x)ϕ(x')

Some valid (semi-positive-definite) Kernels:

- Gaussian: $K(x, x') = \exp(-\frac{(x-x')^2}{2\sigma^2})$
- Polynomial: $K(x, x') = [1 + x^T x']^q$
- Inner product (linear kernel): $K(x, x') = x^T x'$



DECISION TREES



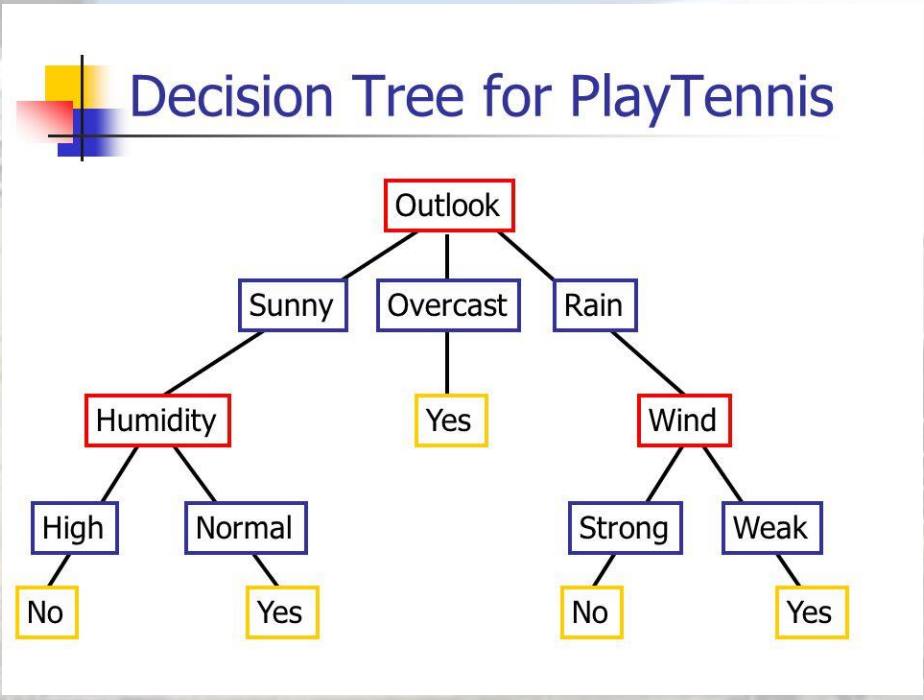
Decision Trees

A "decision tree" is a tree constructed by "leaves" that are rules to split the data in the different classes, based on the data features.

If each event to be classified has variables x_1, x_2, x_3, x_4 ...., I may create a tree by posing conditions on each variable, in a chain

A decision tree is not generally restricted to two possible decisions, but the most simple problems lend themselves to this form

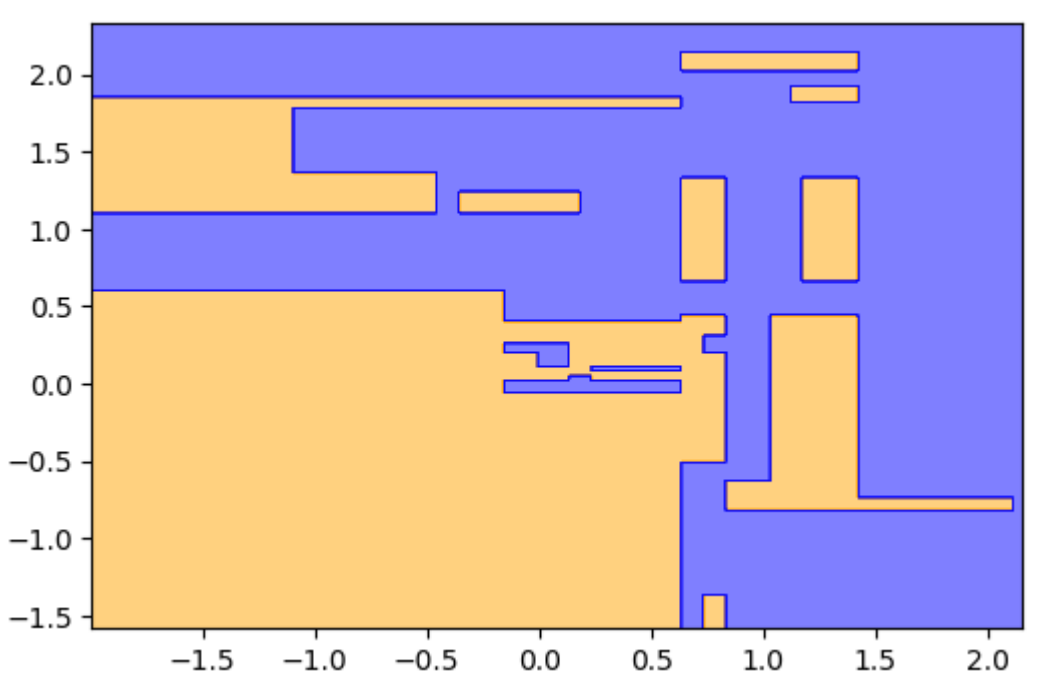
What the tree does for you is to partition the multi-D space describing the possible states of an observation (right: atmospheric weather used to decide whether to play Tennis) in hypercubes where the decision optimizes some quantity of our interest (in this case the satisfaction of playing)



Two-Variable Example

The graph shows how the decision space of two variables could be partitioned by a decision tree with a large number of "splits".

The tree operates "linear cuts" (such as x<x* or similar). Yet, due to the branching nature of the structure, very complex decision boundaries may be created



Due to their simplicity, and the absence of variable transformations,
DTs were among the first MVA algorithms adopted for HEP S/B discrimination problems.

Training a tree

Splits are obtained by conditions on a single feature (j) of the data at a time,

$x^{(j)} < > t^*$

The probability of a class at each split is evaluated by the number of examples of that class in the partitioned set N_m from training data,

$p(c_i) = N_{im}/N_m$

We take the N_m events of the parent node m and for each split criterion $\theta = (x^{(j)}, t^*_m)$ we get two subset Q_{left}, Q_{right} of size n_{left}, n_{right}. We then compute the function

$$G(Q,\theta) = \frac{n_{\text{left}}}{N_m} H(Q_{\text{left}}(\theta)) + \frac{n_{\text{right}}}{N_m} H(Q_{\text{right}}(\theta))$$

H() is an impurity measure (see below). We may now compute the best split:

$$\theta^* = \arg\min_{\theta} G(Q,\theta)$$

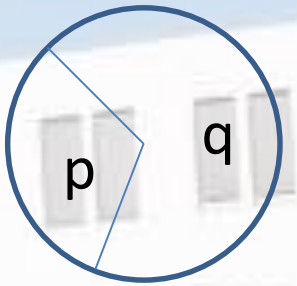
(elsewhere one equivalently finds this described as "maximizing the impurity gain", defining $\Delta I = I_0 - I_L - I_R$, and choosing max ΔI)

Three measures of node impurity

To grow effective trees, we need nodes to be as pure as possible. On the other hand we also need nodes to not have too small probability, or we will be overfitting the training data.

The two criteria are conflicting for classes that overlap. So we need to first of all **define a measure of impurity** of each node.

It is common to define this as a symmetrical function $I(t)=\phi(p,q)$, when p, q are the relative frequencies of the two classes in node t, and with $\phi(0,1) = \phi(1,0) = 0$, and $\phi(½,½)= ½$



If we just look at the classification error for one class as a measure of impurity, then we have

$$\phi(p,q) = 1 - \max(p,q).$$

This obliges the above definitions, but being linear it does not lend itself as an attractive way to optimize the tree construction.

A better (and non-linear) rule is the cross-entropy, defined as

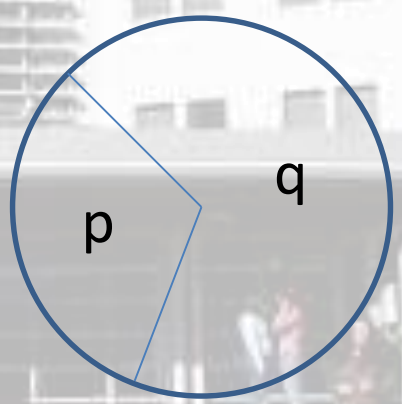
$$\phi(p,q) = -p \log_2 p - q \log_2 q$$

The third impurity measure is the Gini impurity index (see next slide).

The Gini diversity index

The Gini index or Gini coefficient is a statistical measure of distribution developed by the Italian statistician Corrado Gini in 1912. It is very well known as a gauge of economic inequality, but it has much broader applications.

If a population includes two classes of elements, with relative frequencies p and q (i.e. defined so that p+q=1), the Gini index is the symmetric function $\phi(p,q) = 1-p^2-q^2$

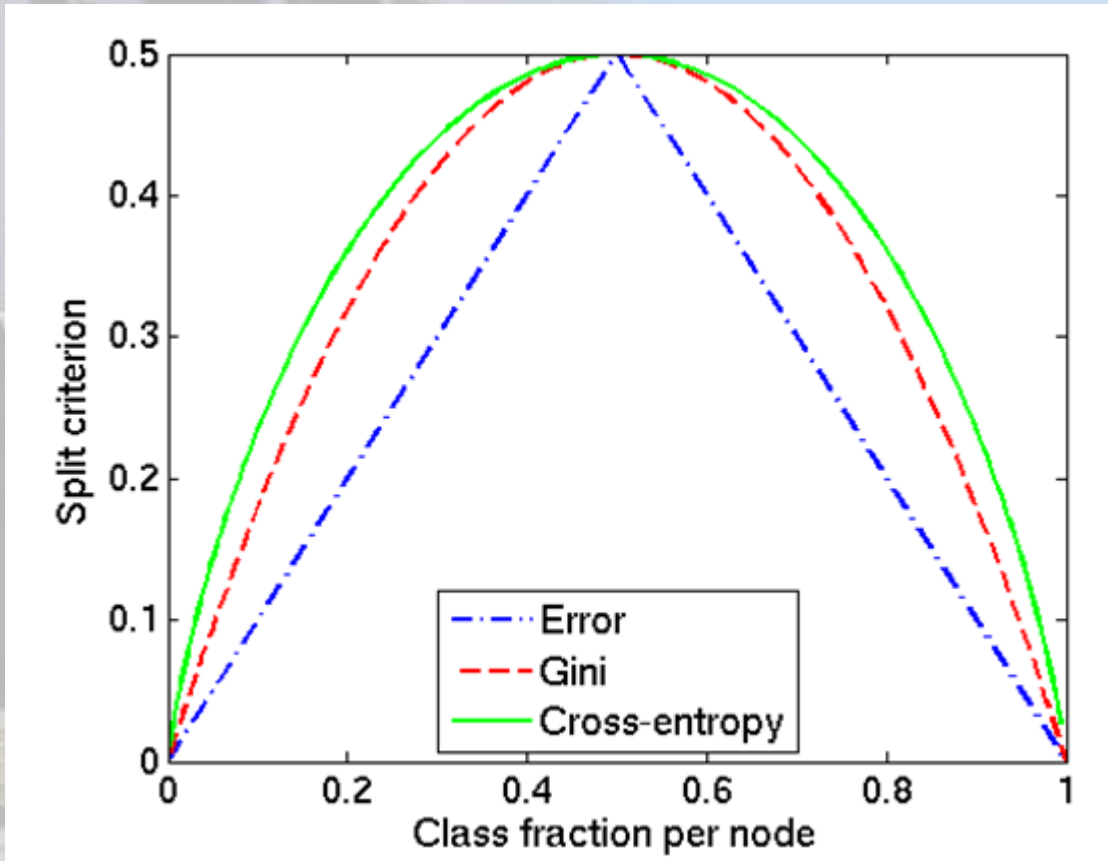


As before, $\phi(0,1) = \phi(1,0) = 0$ indicates that the population is pure (only contains elements of one type), while $\phi(½,½)=½$ indicates maximum entropy.

Clearly an equivalent definition is $\phi(p,q) = 2pq$

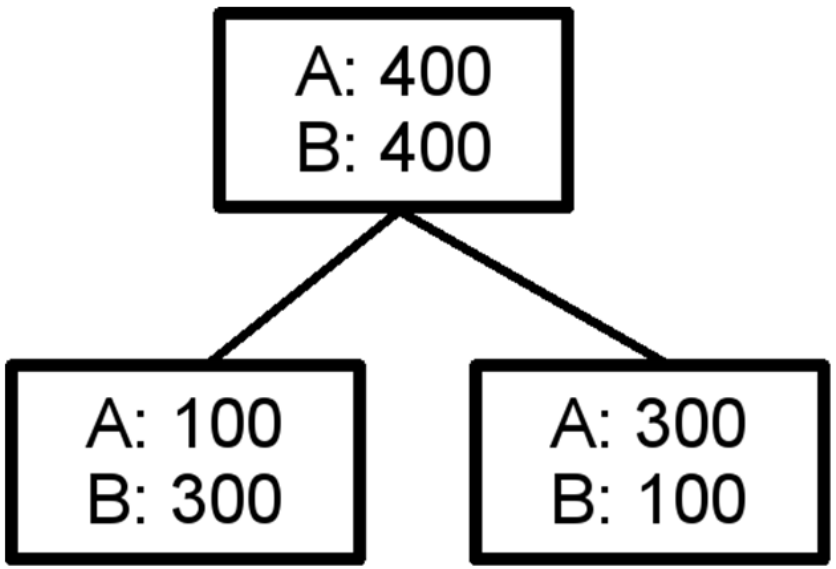
Comparison

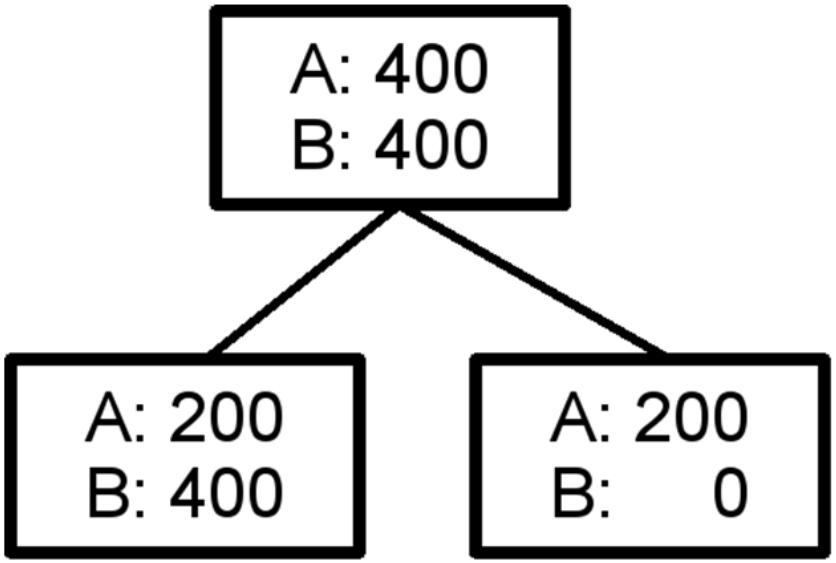
The Gini index and the cross-entropy both provide the necessary non-linearity to find optimal compromises between the two requests (high purity, low variance) for effective tree growth.



One example

Let us consider the following proposed splits for a binary tree (example from I.Narsky). Do you see the problem?





The Gini index decides for us…

MathWorks

Gini index

Maximize $\Delta I = I(t_0) - I(t_L) - I(t_R)$

$i(t_L) = \frac{3}{8}$ $\quad i(t_R) = \frac{3}{8}$

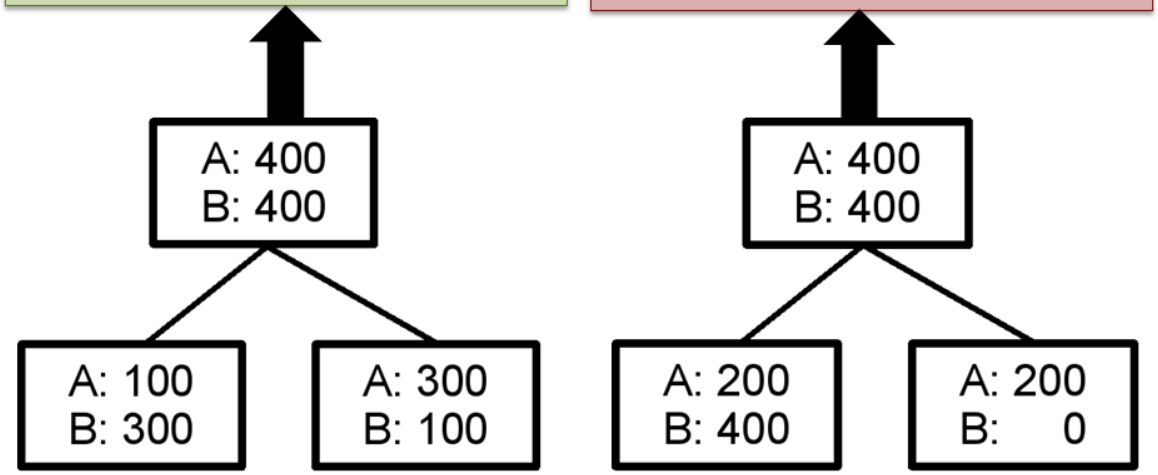
$P(t_L) = \frac{1}{2}$ $\quad P(t_R) = \frac{1}{2}$

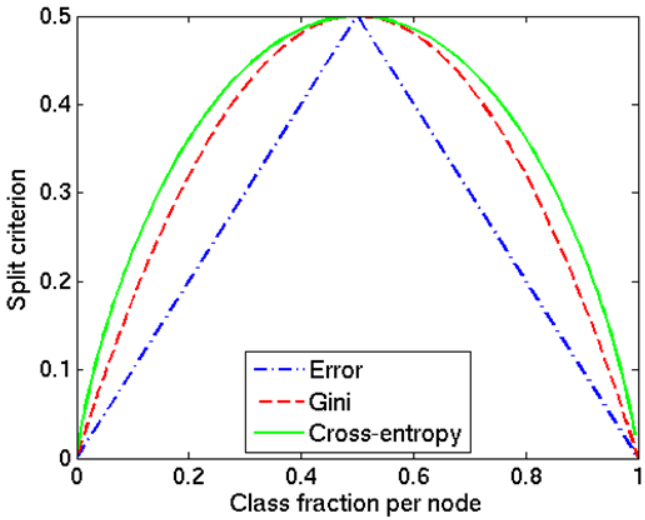
$$I(t_L) + I(t_R) = \frac{3}{8}$$

$i(t_L) = \frac{4}{9}$ $\quad i(t_R) = 0$

$P(t_L) = \frac{3}{4}$ $\quad P(t_R) = \frac{1}{4}$

$$I(t_L) + I(t_R) = \frac{1}{3}$$



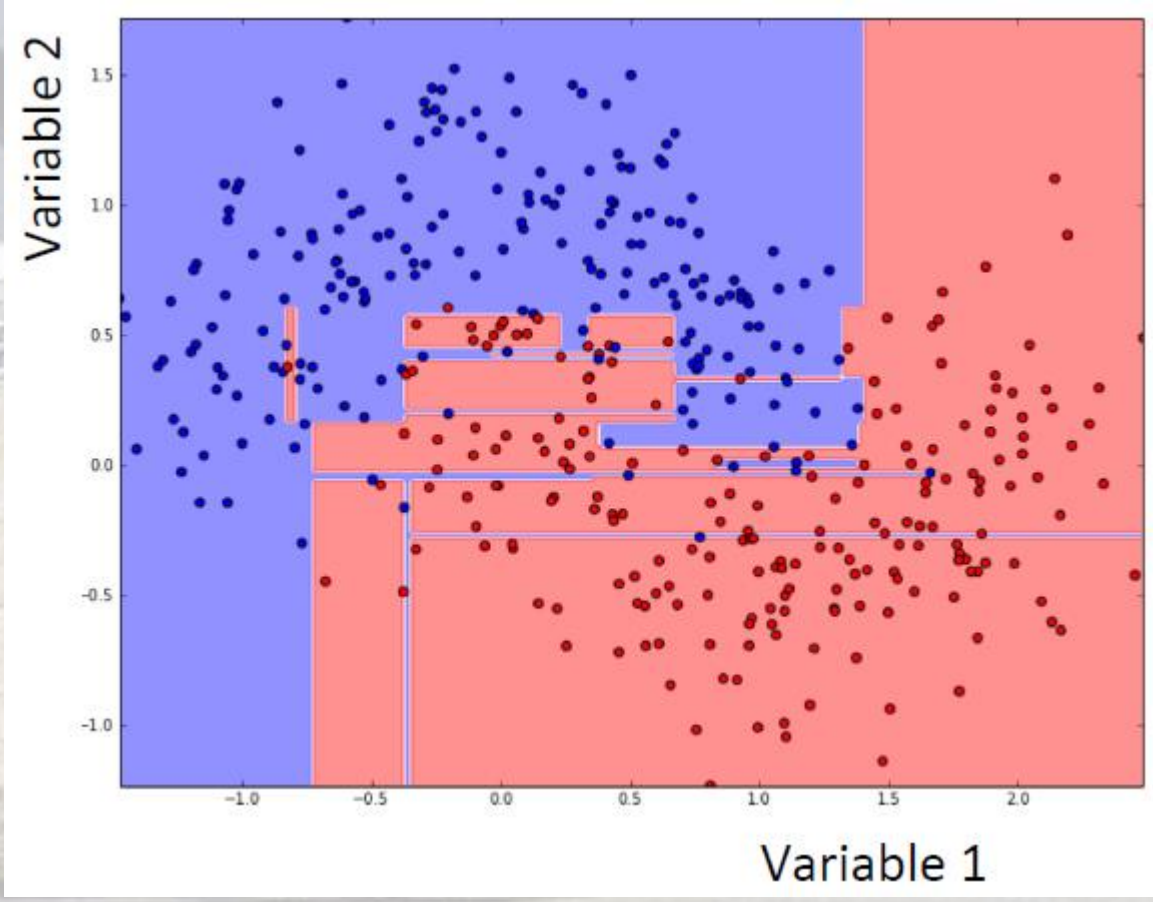


Nonlinear concavity is essential

DT: the perfect classifier?

If you allow your tree to grow indefinitely, it will continue to split impure nodes, ending up classifying events in "pure" leaves

Of course this has huge variance, and in fact it is a blatant example of overfitting the training data



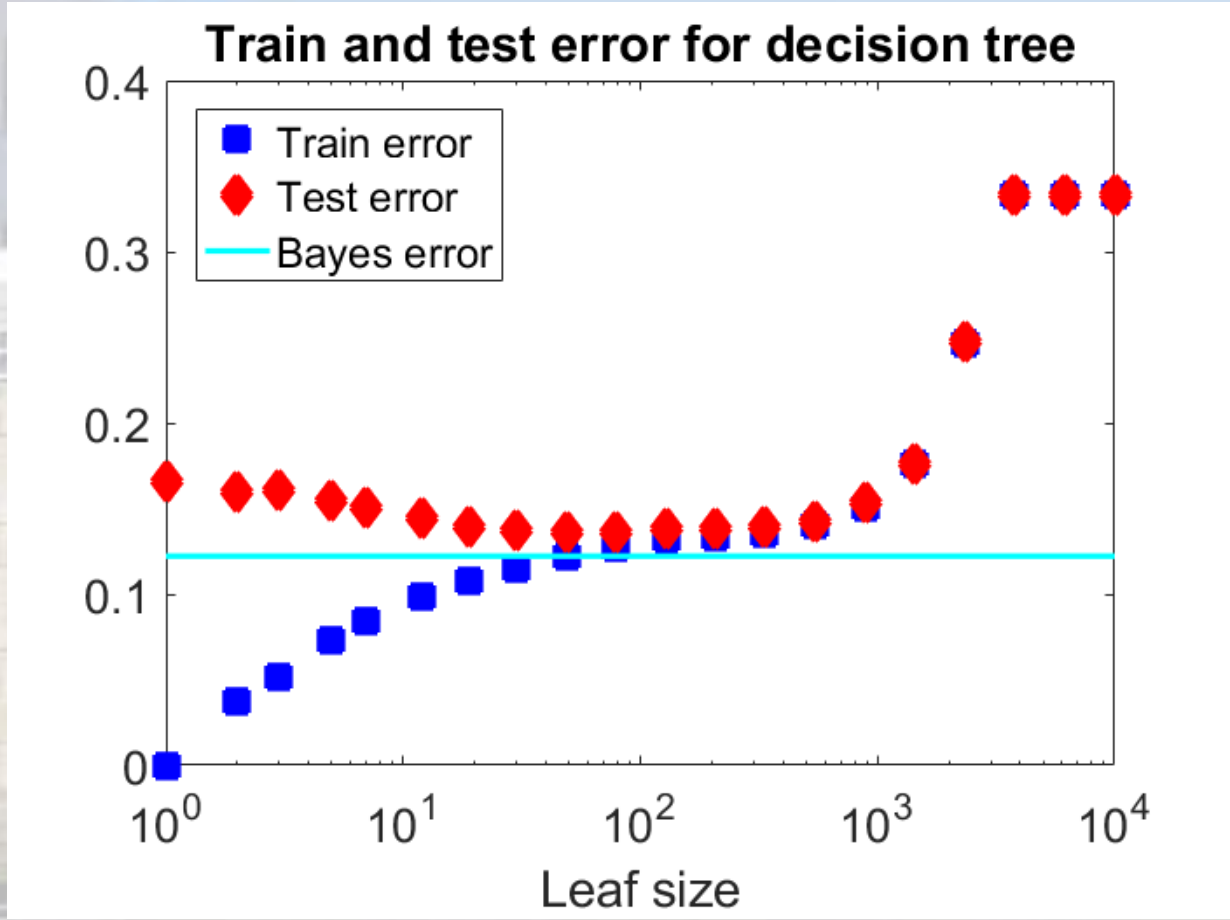
When to stop?

As the leaf size decreases, leaves increasingly contain only events of one class

But this is true only for training data!

Validation is crucial to decide where to stop

The graph on the right, showing classification error versus leaf size, is quite typical of decision trees



Pros and cons

- Decision trees are an attractive choice for applications where performance is not the culprit:
  - they are extremely simple to interpret (in low D!)
  - they demand no preprocessing (scaling, standardization) and handle categorical data easily
  - they work well regardless of number of dimensions of feature space

- But DTs have shortcomings, too:
  - overfitting is under no control → a careful validation is required
  - they are not stable WRT perturbations of the input data: change the training by a little, and the tree can grow very differently
  - The local optimization of tree splits does not guarantee reaching a global optimum (a NP complete problem)

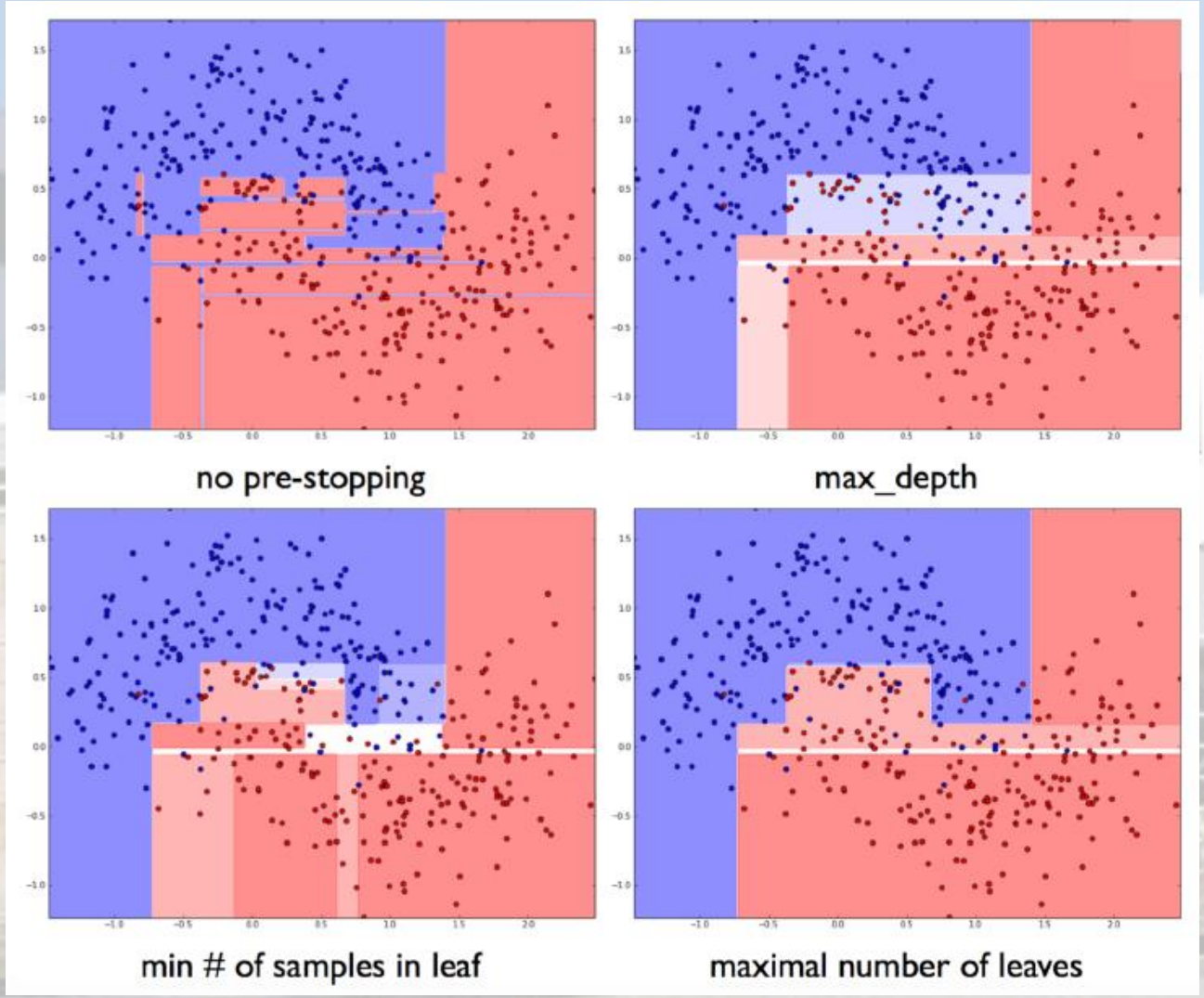
There are ways to overcome the above shortcomings. They rely on **early stopping**, and **ensemble methods**.

Early stopping criteria

To mitigate the overfitting, one can adopt a fixed rule, such as:

- set a minimum number of events per leaf
- set a max number of leaves
- set a max tree depth

These recipes all work, although it is not easy to decide what is best for each case



[Rogozhnikov]

Pruning

Pruning means what it means: cut branches, replacing each with the node at their root. It is **a form of regularization**: we reduce the model complexity, hoping that the variance gain is not offset by a bias loss.

One may treat pruning by defining a risk function for each node t as the classification error of the node weighted by its probability,

$$r(t) = p(t)\varepsilon(t)$$

and for a tree as the sum over all leaves of the tree:

$$r(T) = \sum_{t \in L(t)} r(t)$$

When using training data, r(T) will tend to zero; one can however try to penalize the overtraining by adding to r(T) a term proportional to the number of leaves of the tree.

Ensemble methods

Here we take the case of decision trees to illustrate the power of ensemble methods, which are however applicable to any supervised learning tool

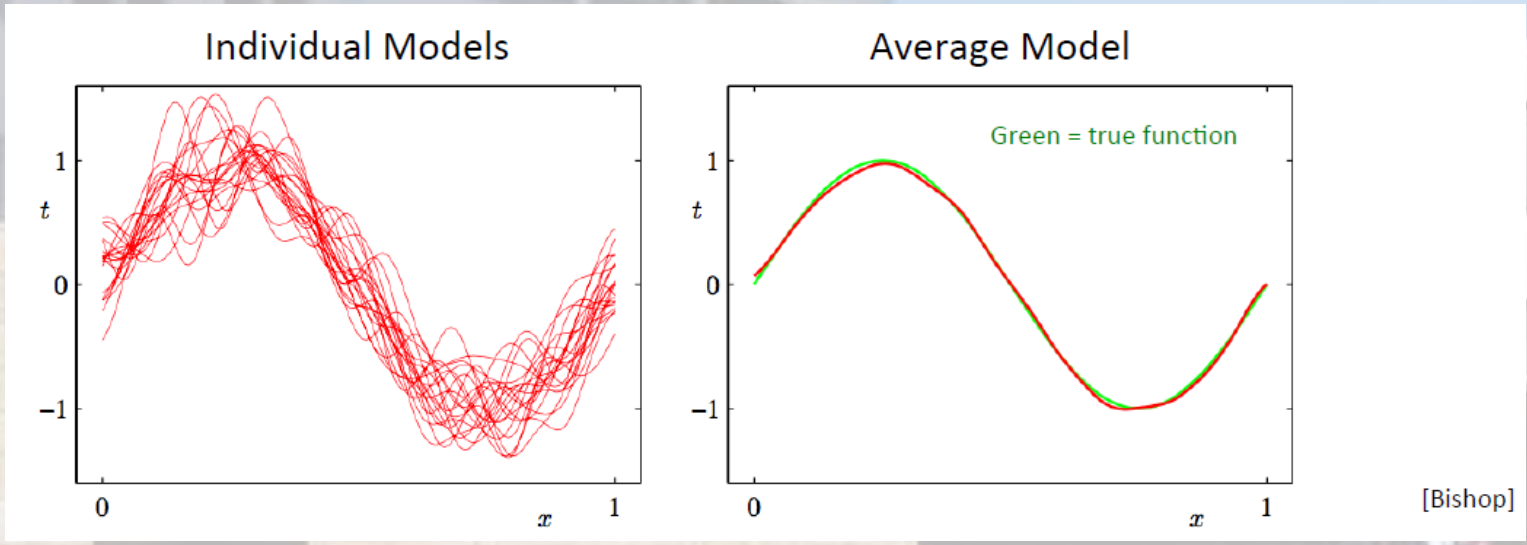
The idea is that we can reduce the variance of a prediction without increasing the bias, a "holy grail" in statistics. How can we do that?

By training slightly different models and **taking a majority vote** (for classification; for regression one would average the scores).

- The bias does not increase because the result behaves similarly to any one of the inputs: the average ensemble performance is equal to the average performance of its members.
- The variance does decrease because fluctuations and noisy predictions are averaged out: a spurious pattern picked up by one model will be damped in the pool

Combining weak learners

A combination of the prediction of several weak learners (small correlation with target value) with high variance can be a powerful model!



For decision trees the benefit is also the control of overfitting, as the "perfect classification" issue of growing pure leaves is solved automatically

Bagging and Boosting

Bagging ("Bootstrap aggregating") is a very effective ensemble technique employing bootstrap to leverage the replica of training sets

The training dataset is sampled with replacement, and every time a new classifier is trained with the resulting set

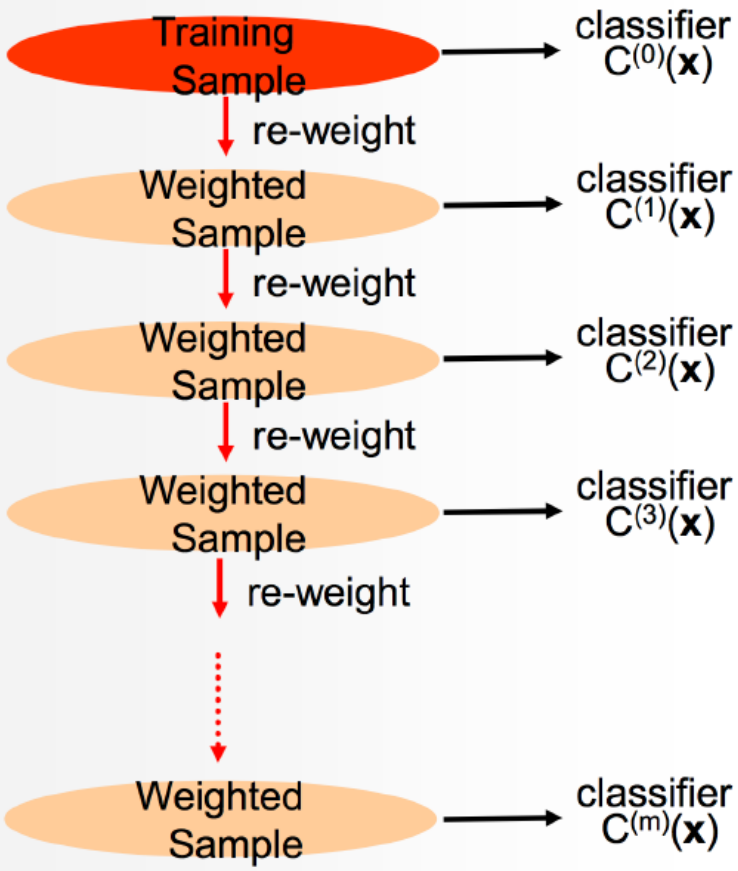
The prediction is obtained by a majority vote of the pool of classifiers, or by an average (in case of regression)

- The **Random Forest algorithm** uses bagging to stabilize the response (see below)

The idea of **Boosting** is instead to train a sequence of models, each of which gives more weight to events not classified correctly by the previous ones.

- At the heart of boosted decision trees

The ADABoost algorithm



AdaBoost re-weights events misclassified by previous classifier by:

$$\frac{1-f_{err}}{f_{err}} \quad \text{with :}$$

$$f_{err} = \frac{\text{misclassified events}}{\text{all events}}$$

AdaBoost weights the classifiers also using the error rate of the individual classifier according to:

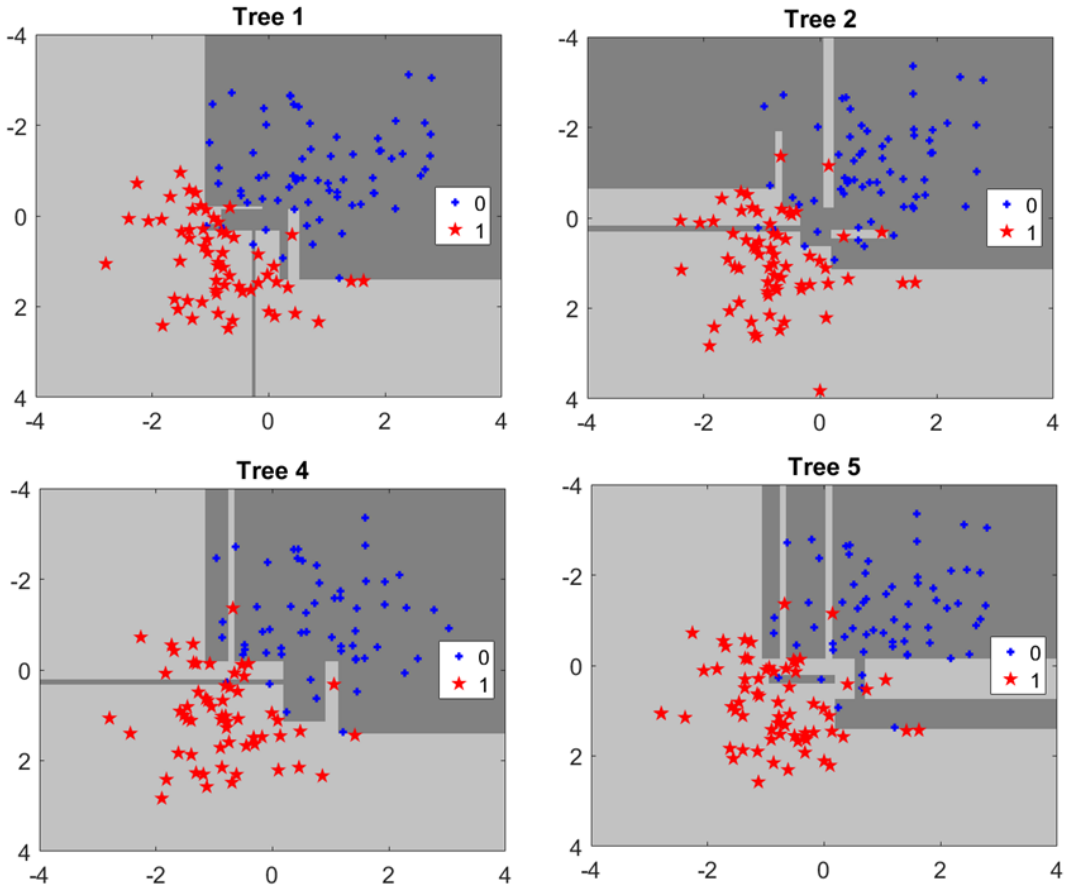
$$y(x) = \sum_{i}^{N_{Classifier}} \log\left(\frac{1-f_{err}^{(i)}}{f_{err}^{(i)}}\right) C^{(i)}(x)$$

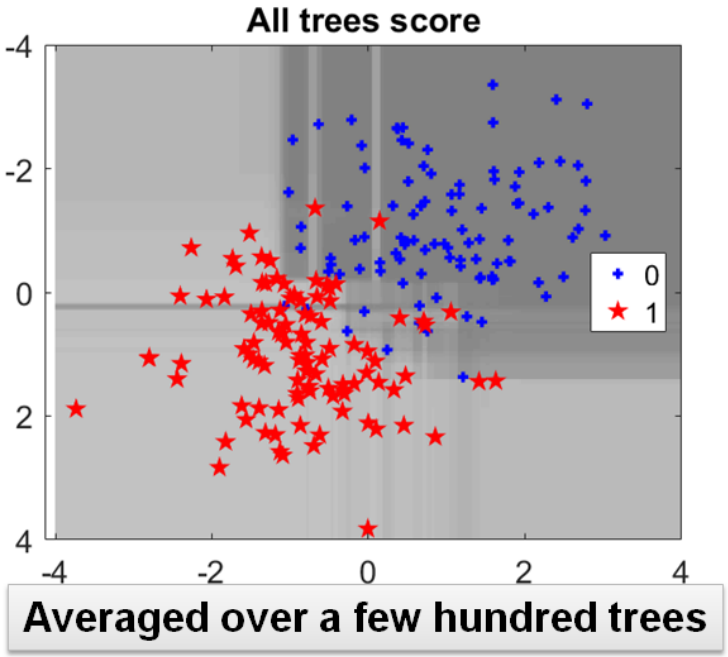
[E. v. Toerne]

Similar schemes are used by other algorithms (Gradient Boosting, XGBoost, ...)

Example: averaging trees

Trees grown on bootstrapped training data learn different decision boundaries; the averaging does better than any of the inputs





Averaged over a few hundred trees

Random Forests

The Random Forest algorithm was first proposed by Breiman (2001), but is based on a 1995 idea of Tim Kan Ho.

RF employs two ensemble techniques. The first is bagging of the training sample, to grow a forest of different trees based on different training data. The second is the subsampling of the feature space.

If I **choose a subset of the variables (e.g. x_1, x_3, x_7)** to create a split in a node of a decision tree, and another subset (x_2, x_4, , x_5, x_7) to create a different one, there will be events that get classified in a different way by the two nodes.

Often there is a dominant variables that is used to decide the split, offsetting the power of the subdominant ones. RF avoids that problem by reducing the correlation of different tres.

RF grows trees where at each node a subset (typically of size $D^{0.5}$, where D is the dimensionality of the feature space) of the features is used to find the best split.

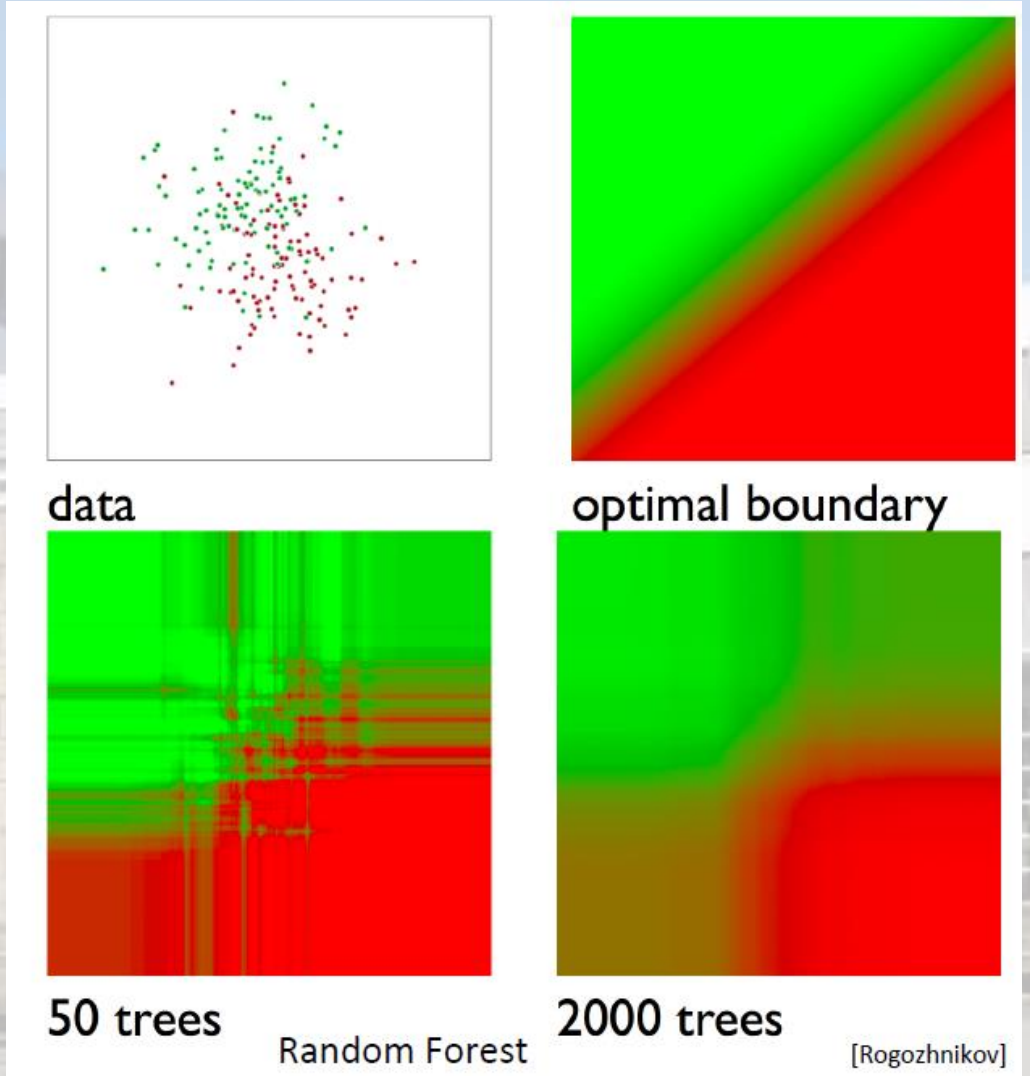
Ensembles of trees: RF

Tree ensembles (like the Random Forest algorithm) have a number of attractive properties

- they usually do not overfit
- they are powerful learners

In addition they retain the advantages of DTs:

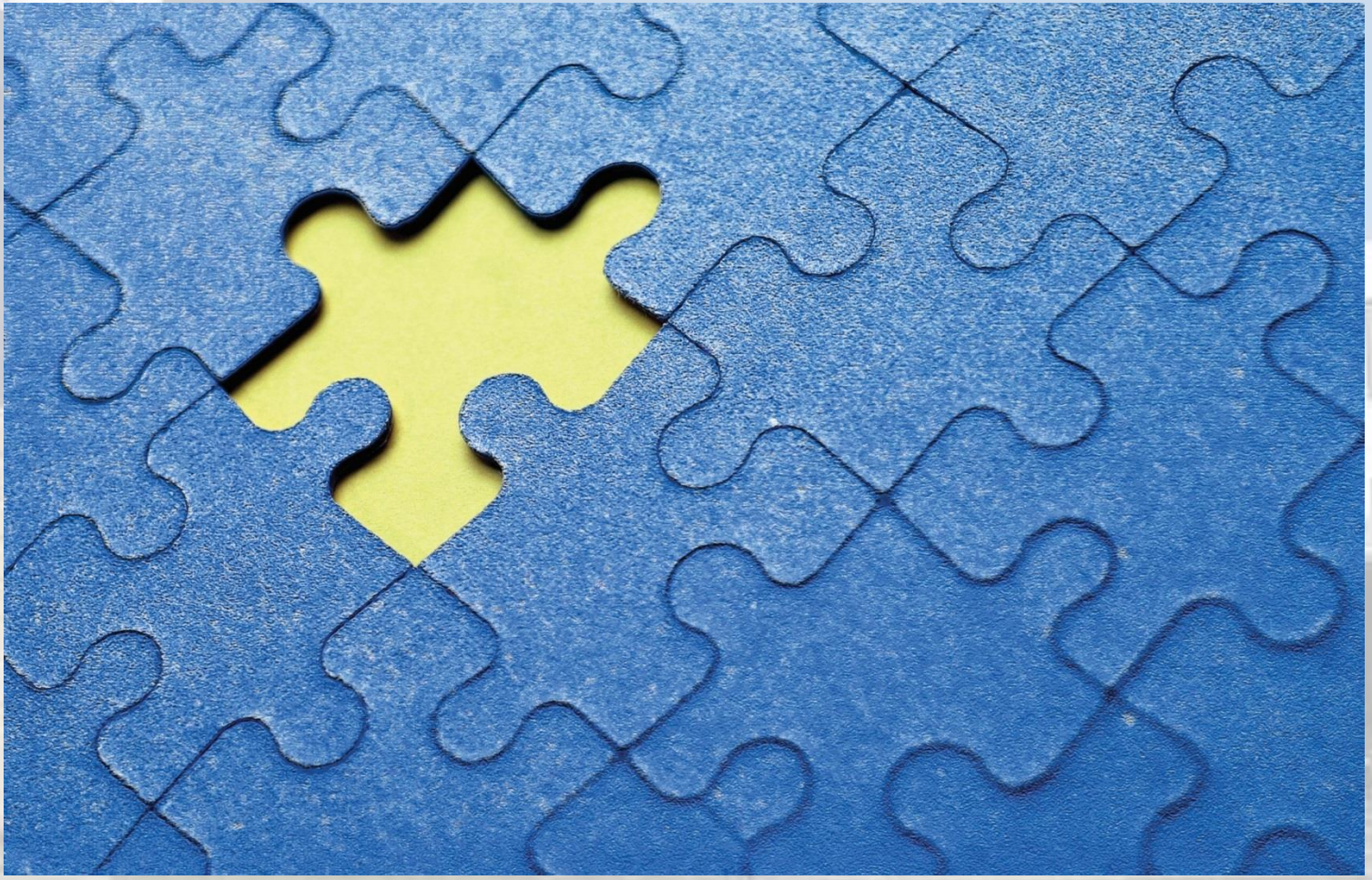
- they are simple to understand and interpret
- easy to train
- They work equally well with continuous as well as categorical data types
- no need to pre-process the data (e.g. invariant to standardization)



Fun with Gradient Boosting

- See http://arogozhnikov.github.io/2016/07/05/gradient_boosting_playground.html

LECTURE 2 CONCLUSIONS



Conclusions for lecture 2

- Classification is a rich subject, and solutions depend on the specific needs of the problem
  - metrics for optimality vary a lot
- The loss function contains the recipe to give you the answer you want – pay attention to put it together (adding regularization where useful)
- Overfitting / overtraining must be avoided by careful testing; optimization handled with independent dataset. Apply cross validation when data is scarce, k=5-10 typically good
- Random Forests / boosted trees are a very flexible, interpretable, powerful learner. Often hard to beat