Statistical Methods for Physicists

Lecture 2: Introduction to Multivariate Methods

www.pp.rhul.ac.uk/~cowan/stat/granada20

UNIVERSIDAD DE GRANADA

University of Granada
4,5 May 2020

Glen Cowan
Physics Department
Royal Holloway, University of London
g.cowan@rhul.ac.uk
www.pp.rhul.ac.uk/~cowan

Outline

Lecture 1: Introduction and review of fundamentals

- Probability, random variables, pdfs
- Parameter estimation, maximum likelihood
- Introduction to statistical tests

→ Lecture 2: More on statistical tests

- Multivariate methods
- Neural networks

Lecture 3: Framework for full analysis

- *p*-values, discovery, limits
- Tests from likelihood ratio

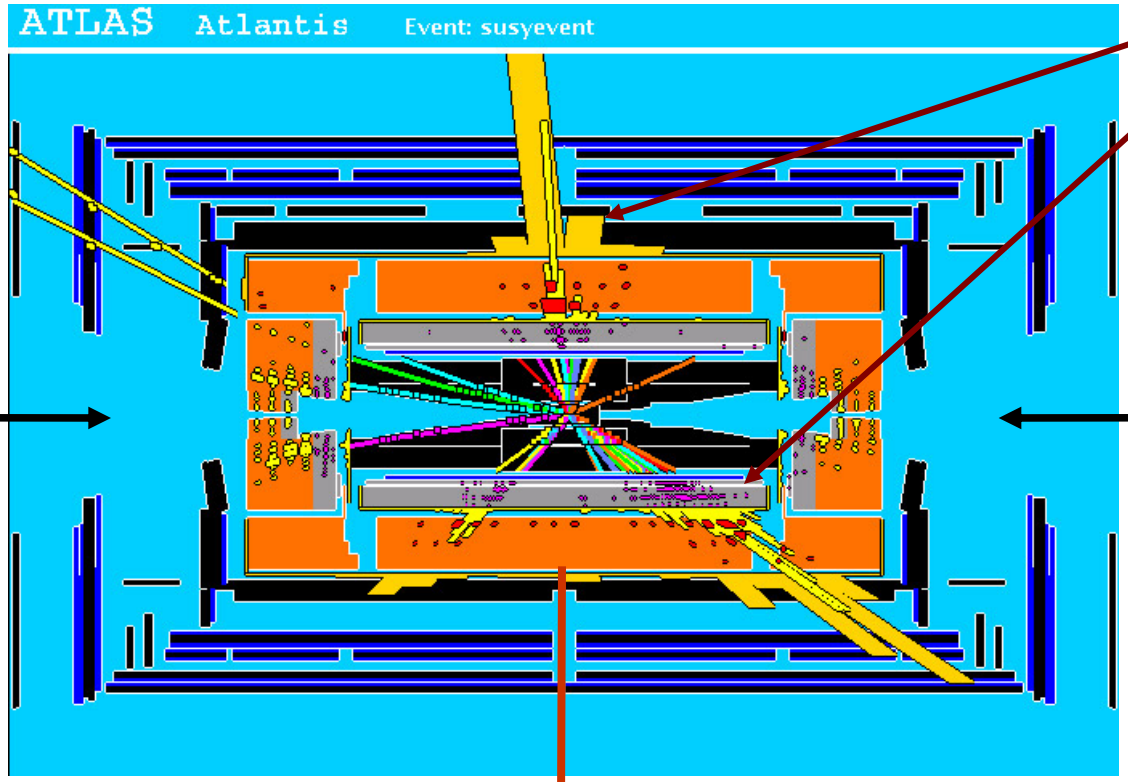
Lecture 4: Further topics

- Nuisance parameters and systematic uncertainties
- More parameter estimation, Bayesian methods
- Experimental sensitivity

Prelude to statistical tests: A simulated SUSY event

high p_T muons

high p_T jets of hadrons

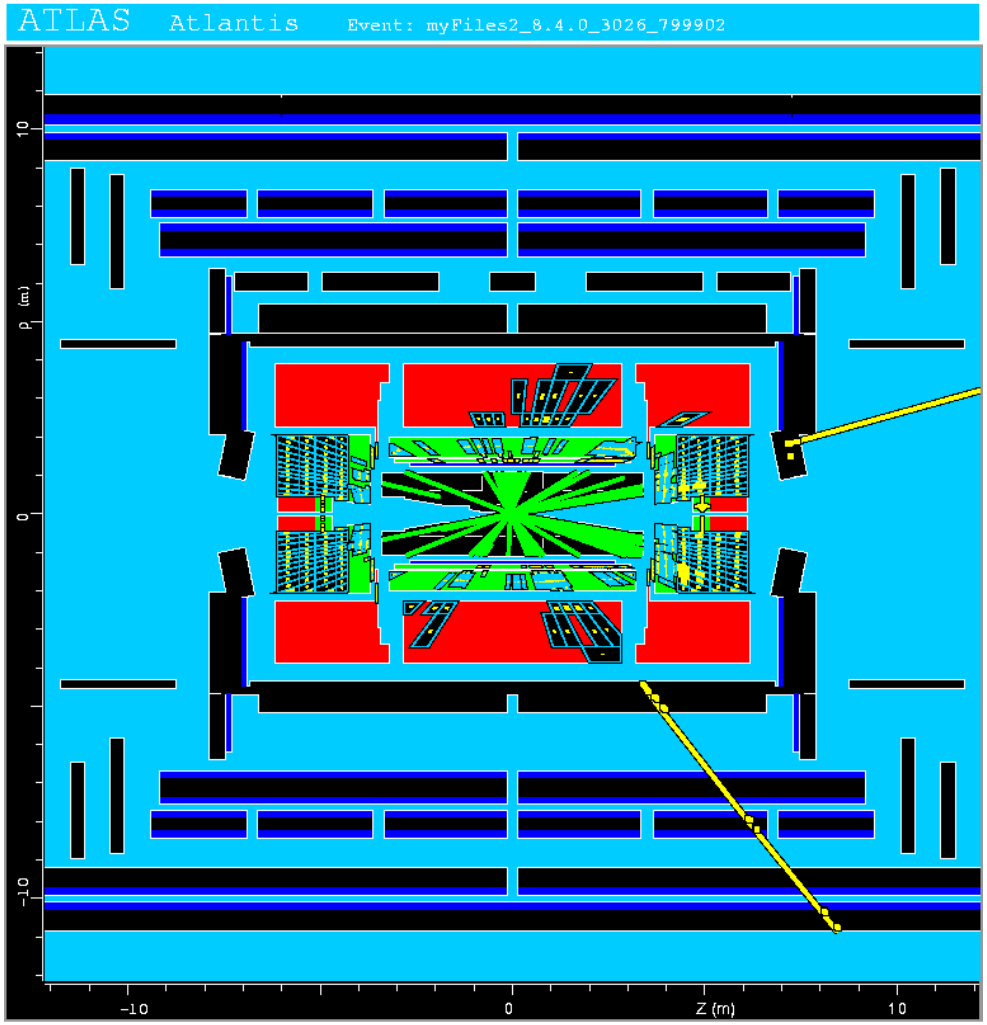


p

p

missing transverse energy

Background events



This event from Standard Model ttbar production also has high p_T jets and muons, and some missing transverse energy.

→ can easily mimic a SUSY event.

Statistical tests for event selection

Suppose the result of a measurement for an individual event is a collection of numbers $\vec{x} = (x_1, \ldots, x_n)$

x_1 = number of muons,

x_2 = mean p_T of jets,

x_3 = missing energy, ...

$\vec{x}$ follows some n-dimensional joint pdf, which depends on the type of event produced, i.e., was it

$$pp \to t\bar{t} \ , \quad pp \to \tilde{g}\tilde{g} \ , \ldots$$

For each reaction we consider we will have a hypothesis for the pdf of $\boldsymbol{x}$, e.g., $p(\boldsymbol{x}|\mathrm{b})$, $p(\boldsymbol{x}|\mathrm{s})$

E.g. here call H_0 the background hypothesis (the event type we want to reject); H_1 is signal hypothesis (the type we want).

Selecting events

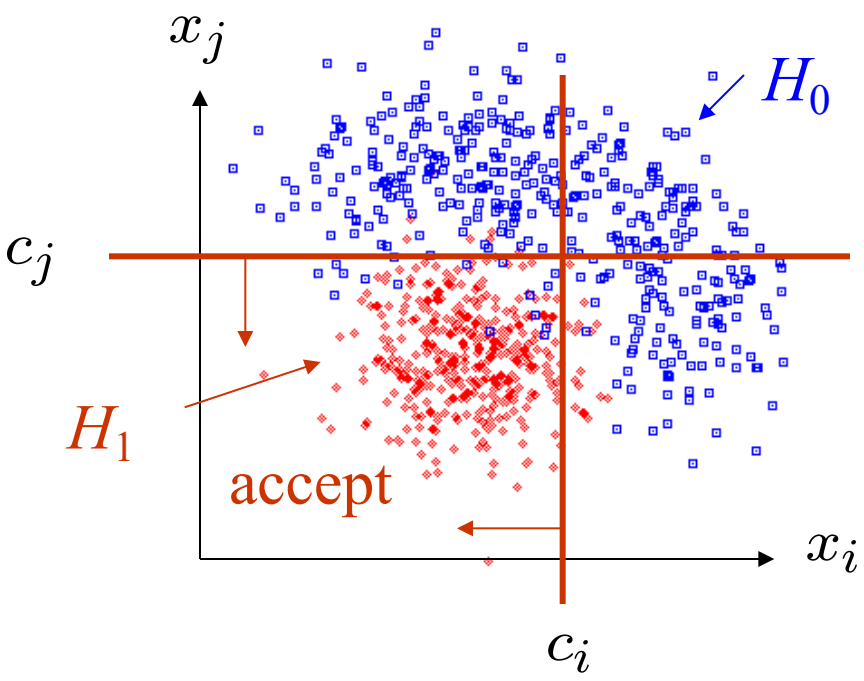
Suppose we have a data sample with two kinds of events, corresponding to hypotheses H_0 and H_1 and we want to select those of type H_1.

Each event is a point in $\vec{x}$ space. What 'decision boundary' should we use to accept/reject events as belonging to event types H_0 or H_1?

Perhaps select events with 'cuts':

$$x_i < c_i$$

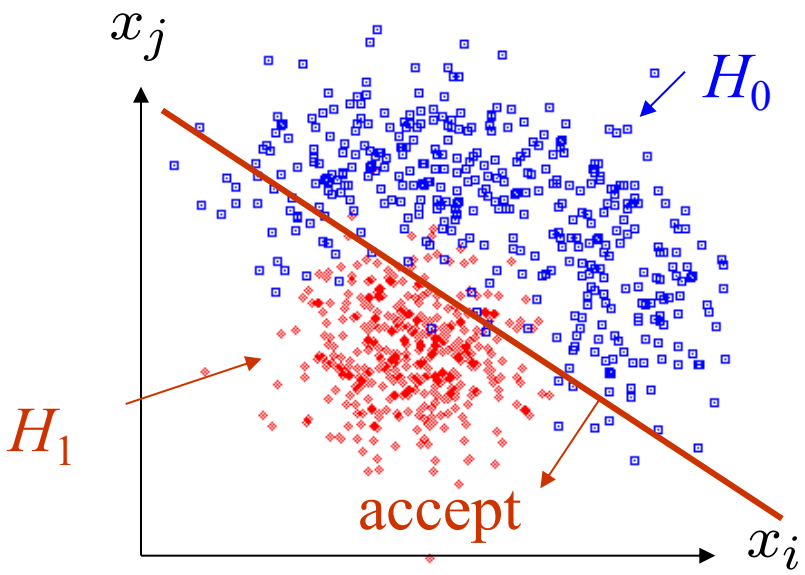
$$x_j < c_j$$



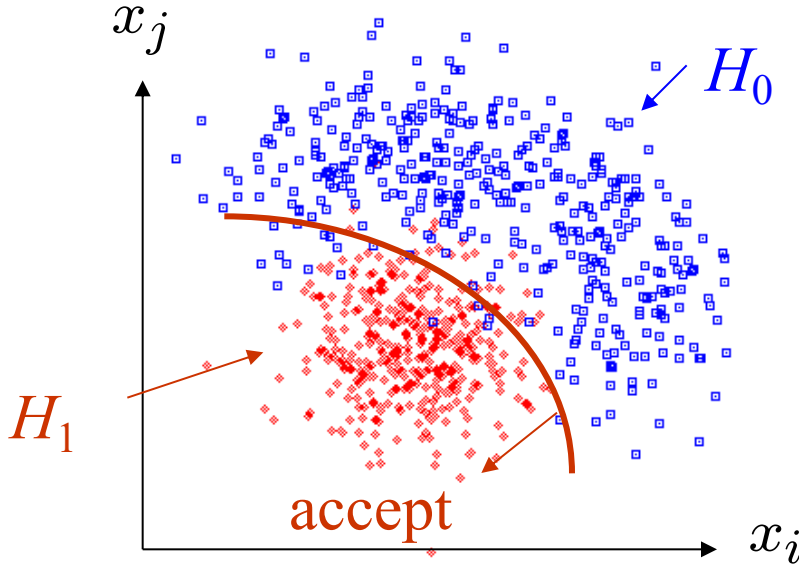
Other ways to select events

Or maybe use some other sort of decision boundary:

linear



or nonlinear



How can we do this in an 'optimal' way?

Test statistics

The boundary of the critical region for an n-dimensional data space $\boldsymbol{x} = (x_1,..., x_n)$ can be defined by an equation of the form

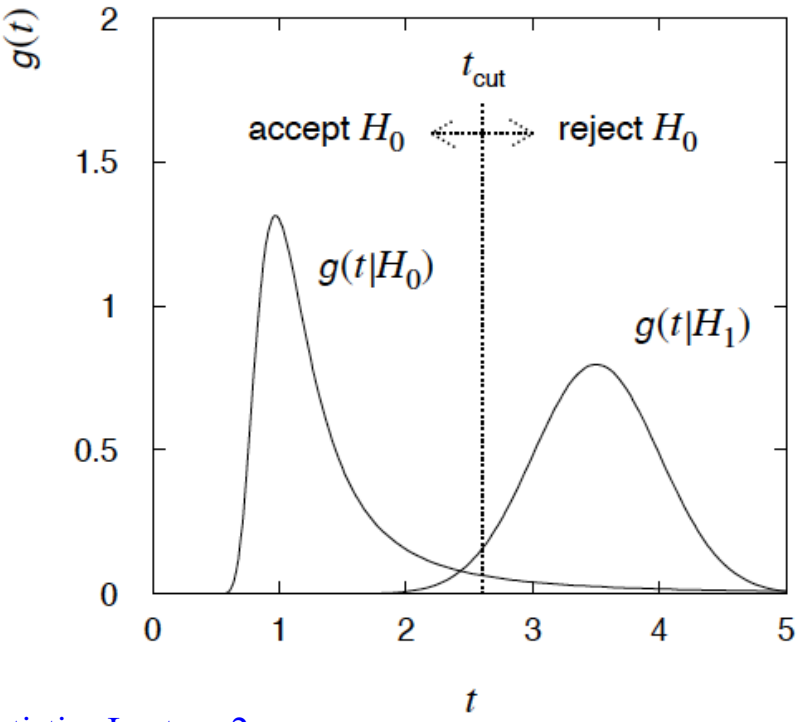
$$t(x_1, \ldots, x_n) = t_{\text{cut}}$$

where $t(x_1,\ldots, x_n)$ is a scalar test statistic.

We can work out the pdfs $g(t|H_0),\ g(t|H_1),\ \ldots$

Decision boundary is now a single 'cut' on t, defining the critical region.

So for an n-dimensional problem we have a corresponding 1-d problem.



Test statistic based on likelihood ratio

How can we choose a test's critical region in an 'optimal way'?

Neyman-Pearson lemma states:

To get the highest power for a given significance level in a test of H_0, (background) versus H_1, (signal) the critical region should have

$$\frac{f(\mathbf{x}|H_1)}{f(\mathbf{x}|H_0)} > c$$

inside the region, and $\leq c$ outside, where c is a constant chosen to give a test of the desired size.

Equivalently, optimal scalar test statistic is

$$t(\mathbf{x}) = \frac{f(\mathbf{x}|H_1)}{f(\mathbf{x}|H_0)}$$

N.B. any monotonic function of this is leads to the same test.

Neyman-Pearson doesn't usually help

We usually don't have explicit formulae for the pdfs $f(\boldsymbol{x}|\mathrm{s})$, $f(\boldsymbol{x}|\mathrm{b})$, so for a given $\boldsymbol{x}$ we can't evaluate the likelihood ratio

$$t(\mathbf{x}) = \frac{f(\mathbf{x}|s)}{f(\mathbf{x}|b)}$$

Instead we may have Monte Carlo models for signal and background processes, so we can produce simulated data:

$$\text{generate } \boldsymbol{x} \sim f(\boldsymbol{x}|\mathrm{s}) \quad \rightarrow \quad \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$$

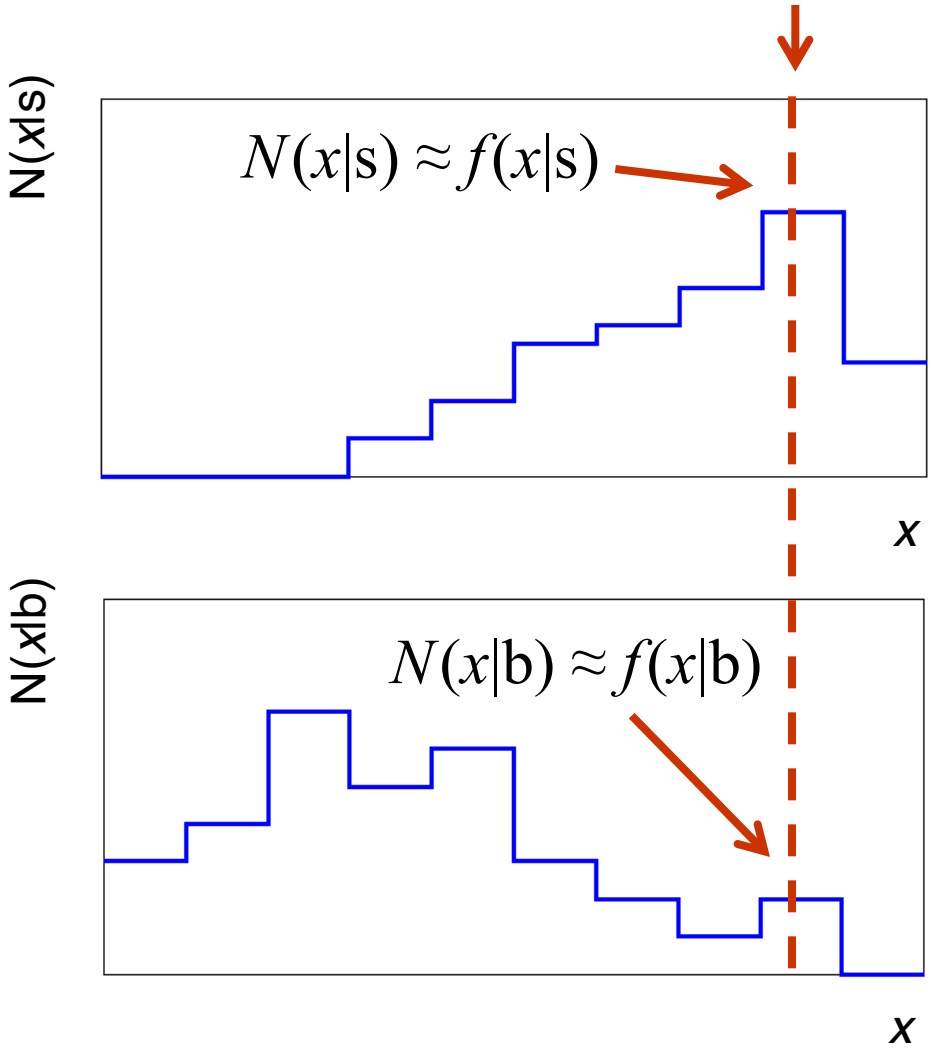
$$\text{generate } \boldsymbol{x} \sim f(\boldsymbol{x}|\mathrm{b}) \quad \rightarrow \quad \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$$

This gives samples of "training data" with events of known type.

Can be expensive (1 fully simulated LHC event ~ 1 CPU minute).

Approximate LR from histograms

Want $t(x) = f(x|\mathrm{s})/f(x|\mathrm{b})$ for x here



One possibility is to generate MC data and construct histograms for both signal and background.

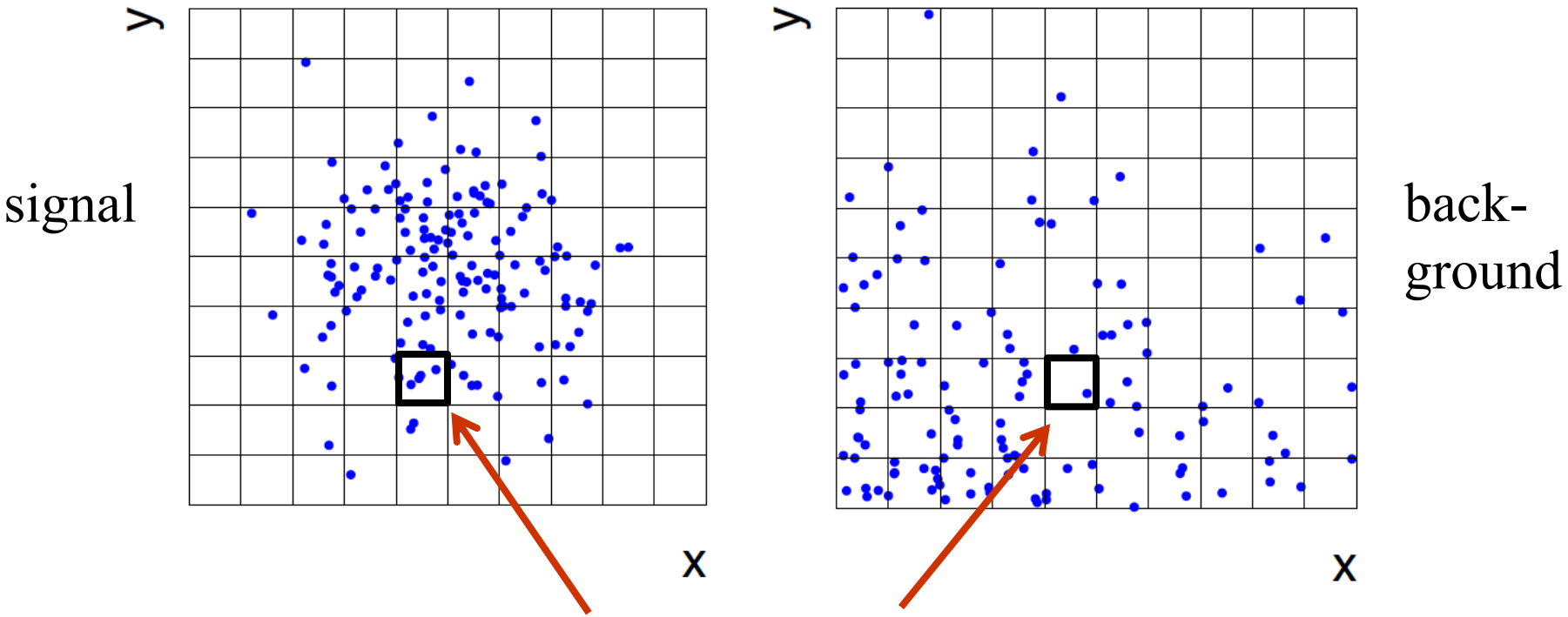
Use (normalized) histogram values to approximate LR:

$$t(x) \approx \frac{N(x|s)}{N(x|b)}$$

Can work well for single variable.

Approximate LR from 2D-histograms

Suppose problem has 2 variables. Try using 2-D histograms:



Approximate pdfs using $N(x,y|\mathrm{s})$, $N(x,y|\mathrm{b})$ in corresponding cells.

But if we want M bins for each variable, then in n-dimensions we have M^n cells; can't generate enough training data to populate.

→ Histogram method usually not usable for $n > 1$ dimension.

Strategies for multivariate analysis

Neyman-Pearson lemma gives optimal answer, but cannot be used directly, because we usually don't have $f(\boldsymbol{x}|\mathrm{s})$, $f(\boldsymbol{x}|\mathrm{b})$.

Histogram method with M bins for n variables requires that we estimate M^n parameters (the values of the pdfs in each cell), so this is rarely practical.

A compromise solution is to assume a certain functional form for the test statistic $t(\boldsymbol{x})$ with fewer parameters; determine them (using MC) to give best separation between signal and background.

Alternatively, try to estimate the probability densities $f(\boldsymbol{x}|\mathrm{s})$ and $f(\boldsymbol{x}|\mathrm{b})$ (with something better than histograms) and use the estimated pdfs to construct an approximate likelihood ratio.

Multivariate methods

Many new (and some old) methods esp. from Machine Learning:

- Linear (Fisher) discriminant
- (Deep) neural networks
- Kernel density methods
- Support Vector Machines
- Decision trees
  - Boosting, Bagging

This is a large topic -- see e.g.

http://www.pp.rhul.ac.uk/~cowan/stat/stat_2.pdf (from around p 38)

and references therein. Below only a brief discussion of linear discriminant and neural networks.

Resources on multivariate methods

C.M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006

T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, 2nd ed., Springer, 2009.

Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani, *An Introduction to Statistical Learning*, Springer, 2013.

R. Duda, P. Hart, D. Stork, Pattern Classification, 2nd ed., Wiley, 2001

A. Webb, Statistical Pattern Recognition, 2nd ed., Wiley, 2002.

Ilya Narsky and Frank C. Porter, *Statistical Analysis Techniques in Particle Physics*, Wiley, 2014.

Software

Rapidly growing area of development – two important resources:

TMVA, Höcker, Stelzer, Tegenfeldt, Voss, Voss, physics/0703039

- From **`tmva.sourceforge.net`**, also distributed with ROOT
- Variety of classifiers
- Good manual, widely used in HEP

scikit-learn

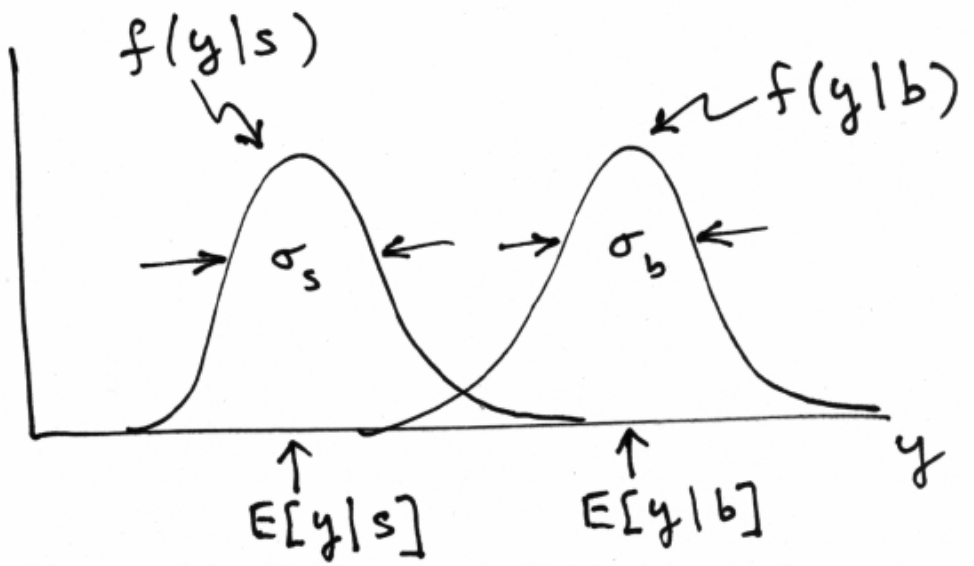
- Python-based tools for Machine Learning
- **`scikit-learn.org`**
- Large user community

Linear test statistic

Suppose there are n input variables: $\boldsymbol{x} = (x_1, \ldots, x_n)$.

Consider a linear function: $$y(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i$$

For a given choice of the coefficients $\boldsymbol{w} = (w_1, \ldots, w_n)$ we will get pdfs $f(y|\mathrm{s})$ and $f(y|\mathrm{b})$:



Linear test statistic

Fisher: to get large difference between means and small widths for $f(y|\mathrm{s})$ and $f(y|\mathrm{b})$, maximize the difference squared of the expectation values divided by the sum of the variances:

$$J(\mathbf{w}) = \frac{(E[y|s] - E[y|b])^2}{V[y|s] + V[y|b]}$$

Setting $\partial J / \partial w_i = 0$ gives:

$$\mathbf{w} \propto W^{-1}(\boldsymbol{\mu}_\mathrm{b} - \boldsymbol{\mu}_\mathrm{s})$$

$$W_{ij} = \mathrm{cov}[x_i, x_j|\mathrm{s}] + \mathrm{cov}[x_i, x_j|\mathrm{b}]$$

$$\mu_{i,\mathrm{s}} = E[x_i|s], \qquad \mu_{i,\mathrm{b}} = E[x_i|b]$$

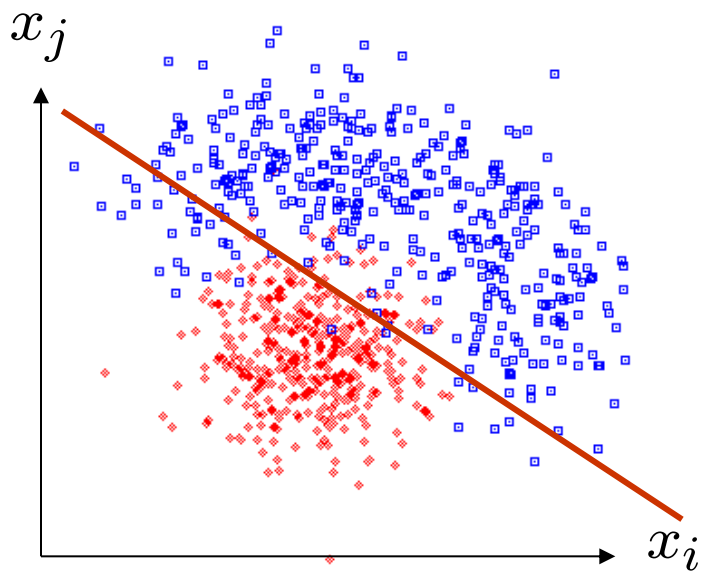
The Fisher discriminant

The resulting coefficients w_i define a Fisher discriminant.

Coefficients defined up to multiplicative constant; can also add arbitrary offset, i.e., usually define test statistic as

$$y(\mathbf{x}) = w_0 + \sum_{i=1}^{n} w_i x_i$$

Boundaries of the test's critical region are surfaces of constant $y(\boldsymbol{x})$, here linear (hyperplanes):



Fisher discriminant for Gaussian data

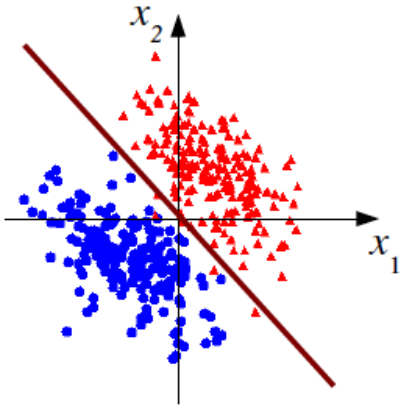
Suppose the pdfs of the input variables, $f(\boldsymbol{x}|\mathrm{s})$ and $f(\boldsymbol{x}|\mathrm{b})$, are both multivariate Gaussians with same covariance but different means:

$$f(\boldsymbol{x}|\mathrm{s}) = \mathrm{Gauss}(\boldsymbol{\mu}_\mathrm{s}, V)$$

$$f(\boldsymbol{x}|\mathrm{b}) = \mathrm{Gauss}(\boldsymbol{\mu}_\mathrm{b}, V)$$

Same covariance

$V_{ij} = \mathrm{cov}[x_i, x_j]$



In this case it can be shown that the Fisher discriminant is

$$y(\mathbf{x}) \sim \ln \frac{f(\mathbf{x}|\mathrm{s})}{f(\mathbf{x}|\mathrm{b})}$$

i.e., it is a monotonic function of the likelihood ratio and thus leads to the same critical region. So in this case the Fisher discriminant provides an optimal statistical test.

Transformation of inputs

If the data are not Gaussian with equal covariance, a linear decision boundary is not optimal. But we can try to subject the data to a transformation

$$\varphi_1(\vec{x}), \ldots, \varphi_m(\vec{x})$$

and then treat the ϕ_i as the new input variables. This is often called "feature space" and the ϕ_i are "basis functions". The basis functions can be fixed or can contain adjustable parameters which we optimize with training data (cf. neural networks).

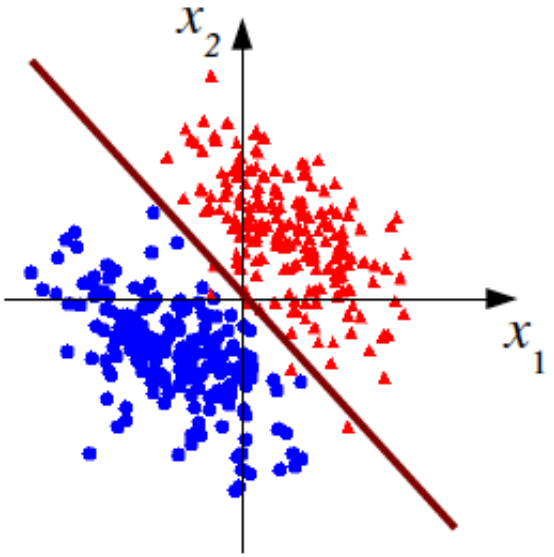
In other cases we will see that the basis functions only enter as dot products

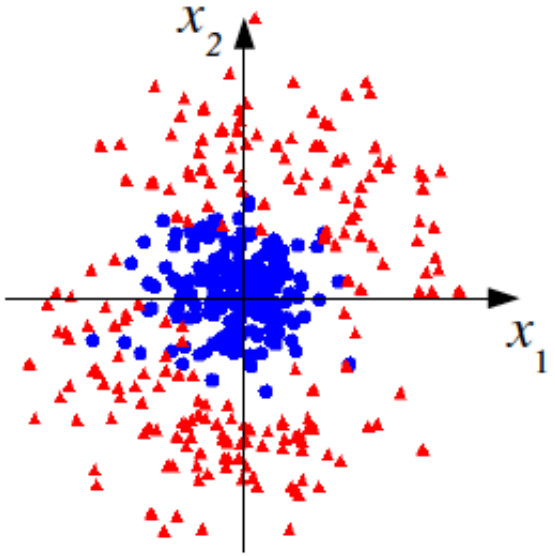
$$\vec{\varphi}(\vec{x}_i) \cdot \vec{\varphi}(\vec{x}_j) = K(\vec{x}_i, \vec{x}_j)$$

and thus we will only need the "kernel function" $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$

Linear decision boundaries

A linear decision boundary is only optimal when both classes follow multivariate Gaussians with equal covariances and different means.





For some other cases a linear boundary is almost useless.

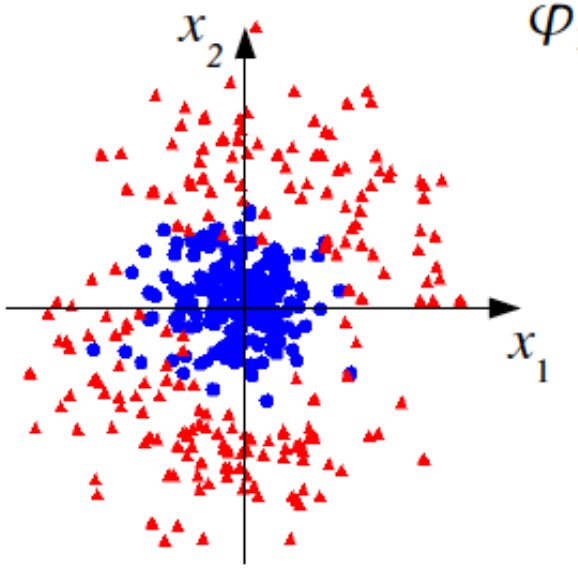
Nonlinear transformation of inputs

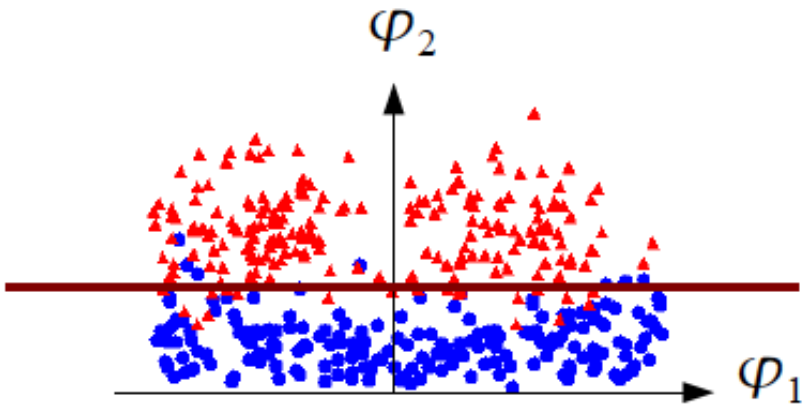
We can try to find a transformation, $x_1, \ldots, x_n \to \varphi_1(\vec{x}), \ldots, \varphi_m(\vec{x})$ so that the transformed "feature space" variables can be separated better by a linear boundary:

$$\varphi_1 = \tan^{-1}(x_2/x_1)$$

$$\varphi_2 = \sqrt{x_1^2 + x_2^2}$$

Here, guess fixed basis functions (no free parameters)





Neural networks

Neural networks originate from attempts to model neural processes (McCulloch and Pitts, 1943; Rosenblatt, 1962).

Widely used in many fields, and for many years the only "advanced" multivariate method popular in HEP.

We can view a neural network as a specific way of parametrizing the basis functions used to define the feature space transformation.

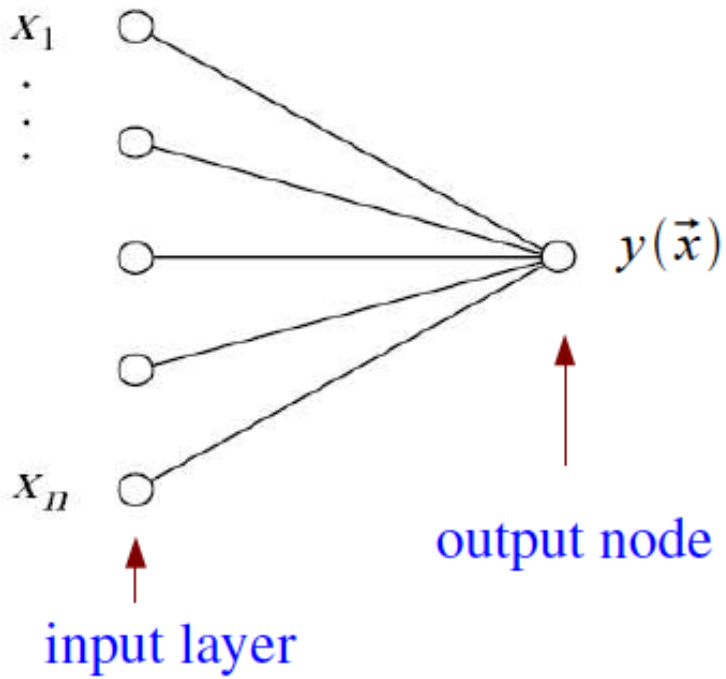
The training data are then used to adjust the parameters so that the resulting discriminant function has the best performance.

The single layer perceptron

Define the discriminant using $y(\vec{x}) = h\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)$

where h is a nonlinear, monotonic activation function; we can use e.g. the logistic sigmoid $h(x) = (1 + e^{-x})^{-1}$.

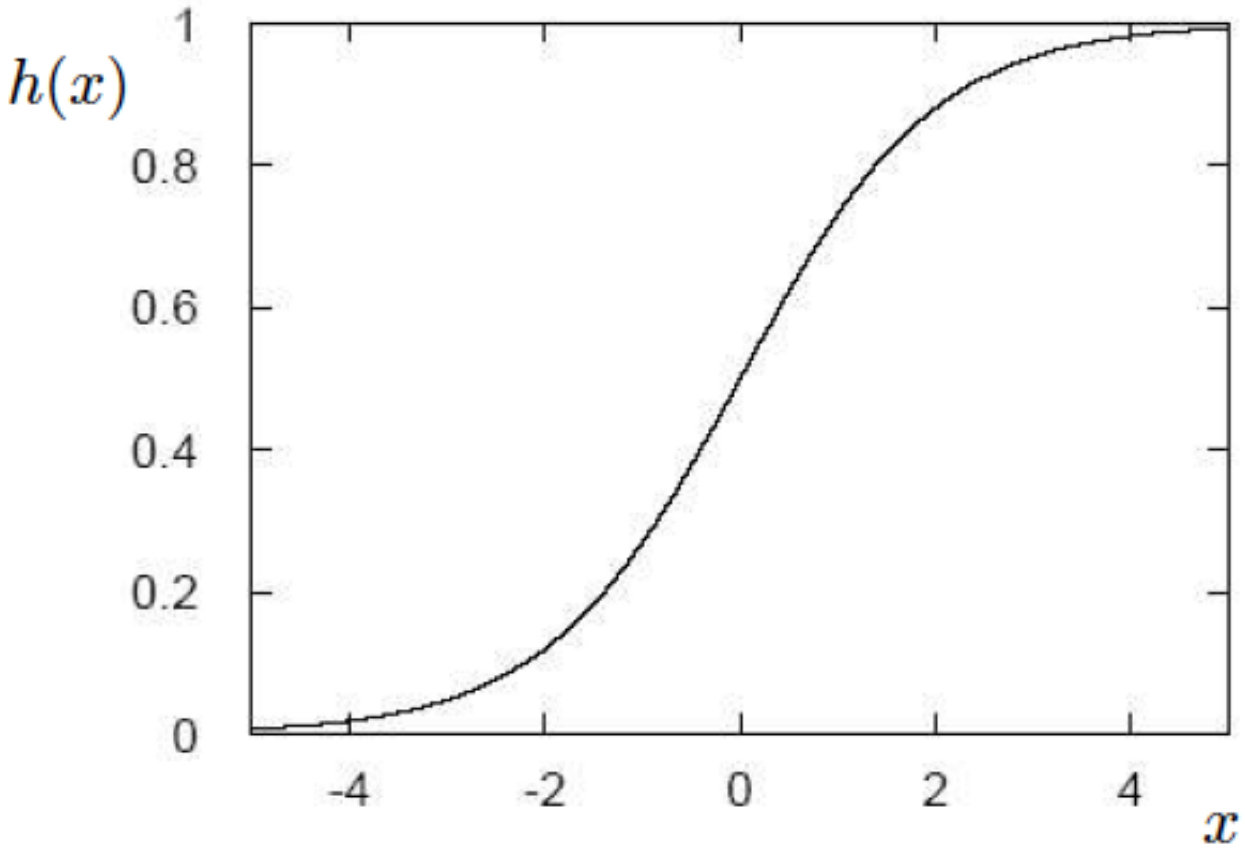
If the activation function is monotonic, the resulting $y(\boldsymbol{x})$ is equivalent to the original linear discriminant. This is an example of a "generalized linear model" called the single layer perceptron.



The activation function

For activation function $h(\cdot)$ often use logistic sigmoid:

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



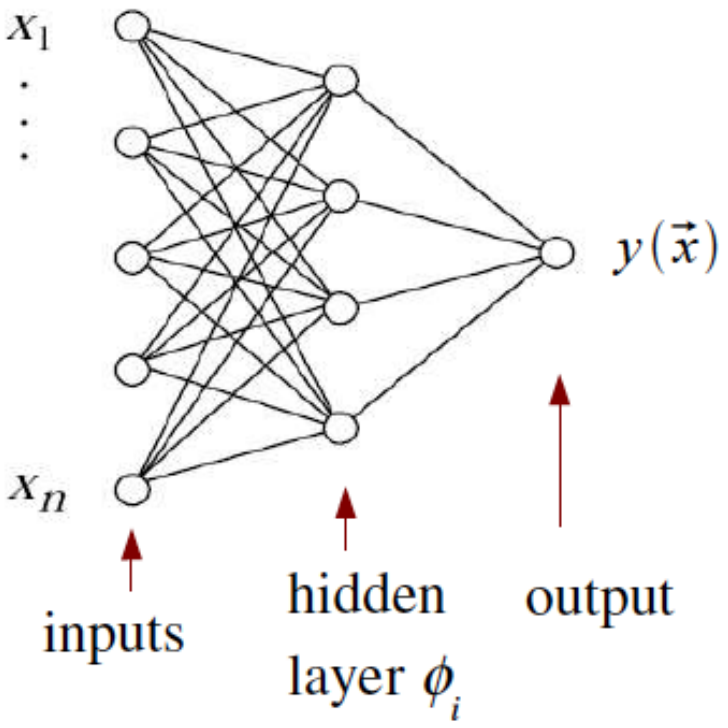
The multilayer perceptron

Now use this idea to define not only the output $y(\boldsymbol{x})$, but also the set of transformed inputs $\varphi_1(\vec{x}), \ldots, \varphi_m(\vec{x})$ that form a "hidden layer":

Superscript for weights indicates layer number

$$\varphi_i(\vec{x}) = h\left(w_{i0}^{(1)} + \sum_{j=1}^{n} w_{ij}^{(1)} x_j\right)$$

$$y(\vec{x}) = h\left(w_{10}^{(2)} + \sum_{j=1}^{n} w_{1j}^{(2)} \varphi_j(\vec{x})\right)$$

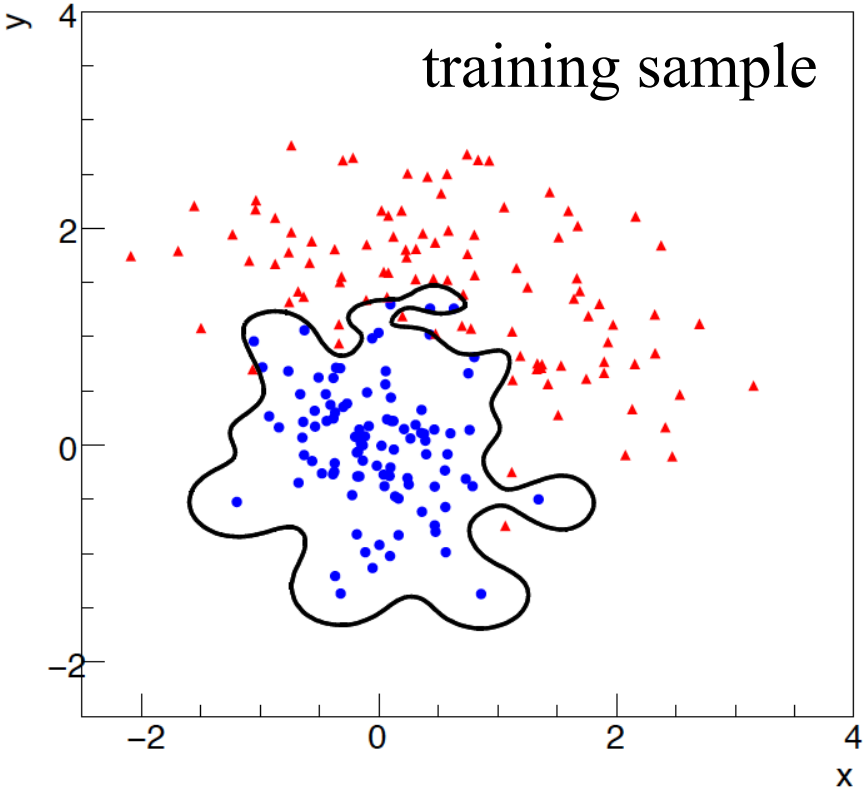


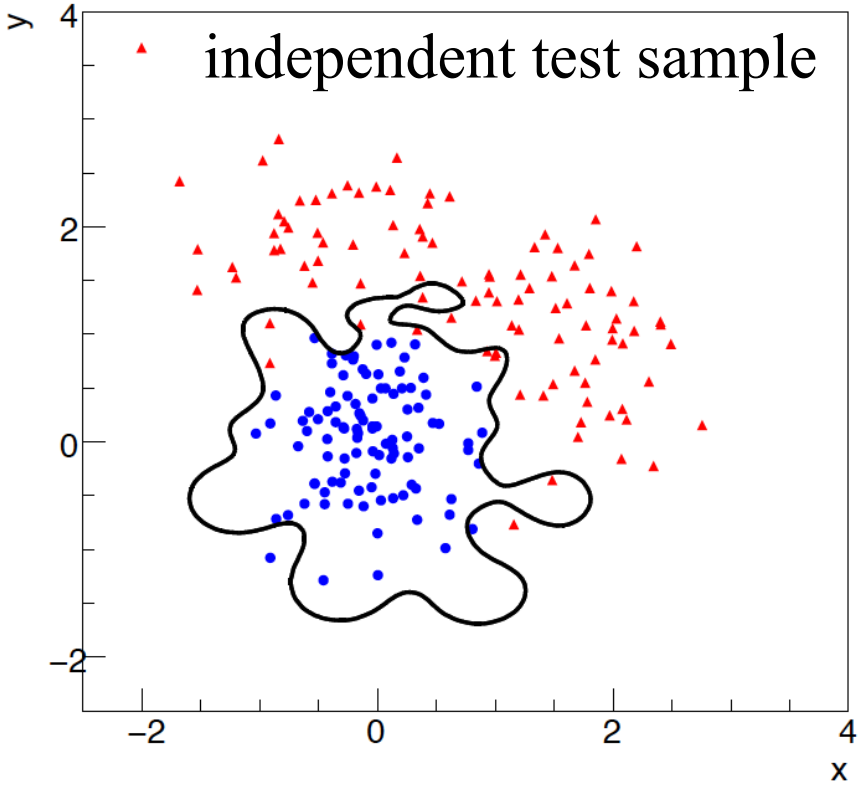
This is the multilayer perceptron, our basic neural network model; straightforward to generalize to multiple hidden layers.

Overtraining

Including more parameters in a classifier makes its decision boundary increasingly flexible, e.g., more nodes/layers for a neural network.

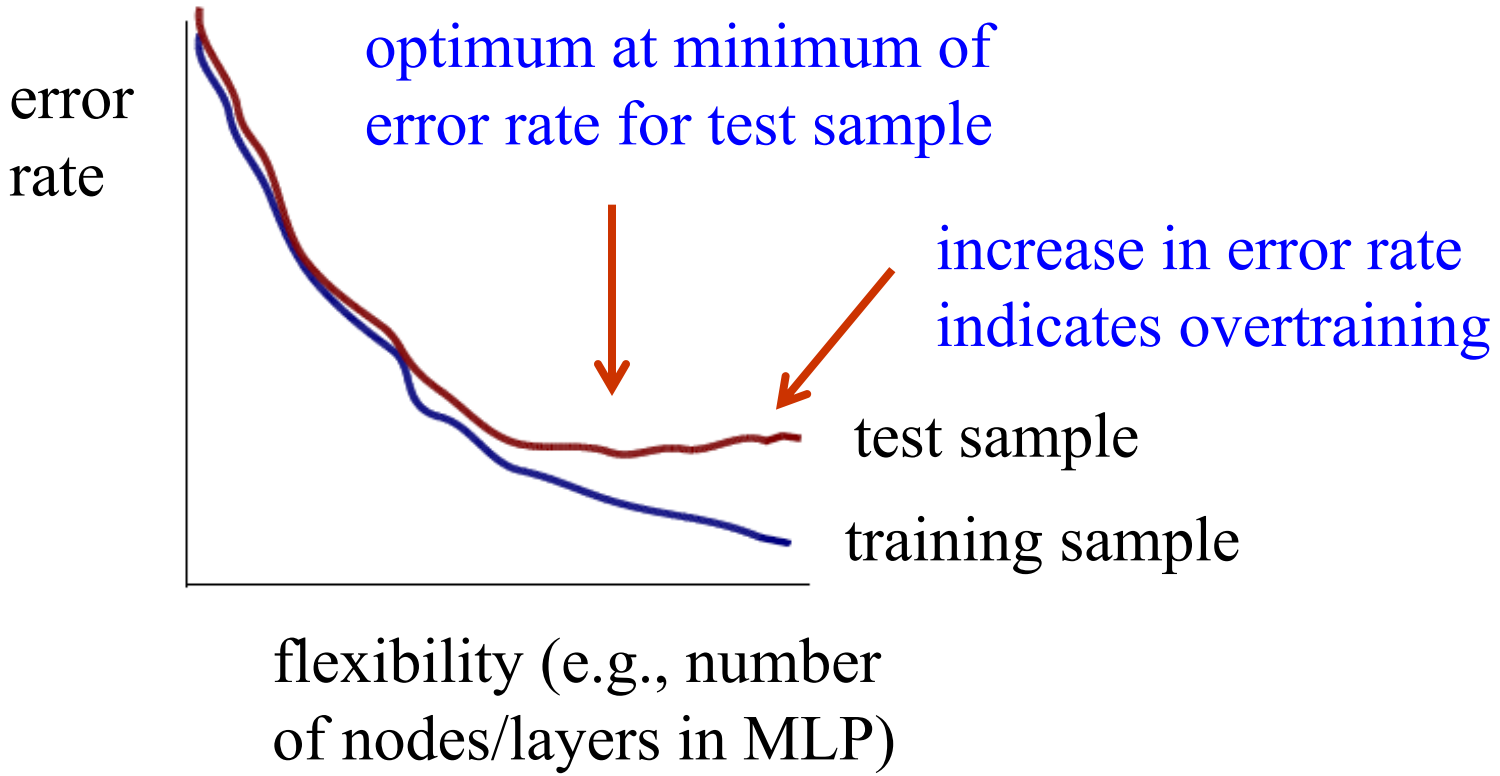
A "flexible" classifier may conform too closely to the training points; the same boundary will not perform well on an independent test data sample (→ "overtraining").





Monitoring overtraining

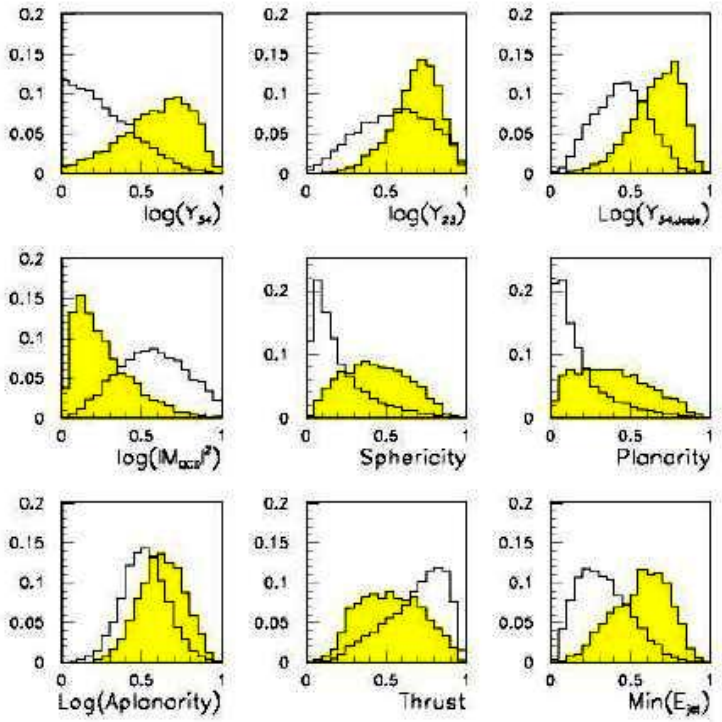
If we monitor the fraction of misclassified events (or similar, e.g., error function $E(\boldsymbol{w})$) for test and training samples, it will usually decrease for both as the boundary is made more flexible:



Neural network example from LEP II

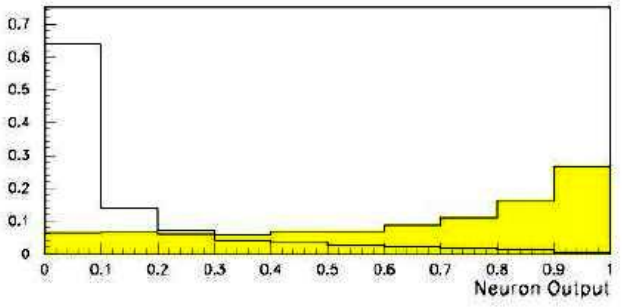
Signal: $e^+e^- \to W^+W^-$ (often 4 well separated hadron jets)

Background: $e^+e^- \to qqgg$ (4 less well separated hadron jets)



← input variables based on jet structure, event shape, ...
none by itself gives much separation.

Neural network output:



(Garrido, Juste and Martinez, ALEPH 96-144)

Extra slides

Network training

The type of each training event is known, i.e., for event a we have:

$\vec{x}_a = (x_1, \ldots, x_n)$ the input variables, and

$t_a = 0, 1$ a numerical label for event type ("target value")

Let $\boldsymbol{w}$ denote the set of all of the weights of the network. We can determine their optimal values by minimizing a sum-of-squares "error function"

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{a=1}^{N} |y(\vec{x}_a, \boldsymbol{w}) - t_a|^2 = \sum_{a=1}^{N} E_a(\boldsymbol{w})$$

Contribution to error function from each event

Numerical minimization of $E(\boldsymbol{w})$

Consider gradient descent method: from an initial guess in weight space $\boldsymbol{w}^{(1)}$ take a small step in the direction of maximum decrease. I.e. for the step τ to $\tau+1$,

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta \nabla E(\boldsymbol{w}^{(\tau)})$$

learning rate ($\eta>0$)

If we do this with the full error function $E(\boldsymbol{w})$, gradient descent does surprisingly poorly; better to use "conjugate gradients".

But gradient descent turns out to be useful with an online (sequential) method, i.e., where we update $\boldsymbol{w}$ for each training event a, (cycle through all training events):

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta \nabla E_a(\boldsymbol{w}^{(\tau)})$$

Error backpropagation

Error backpropagation ("backprop") is an algorithm for finding the derivatives required for gradient descent minimization.

The network output can be written $y(\boldsymbol{x}) = h(u(\boldsymbol{x}))$ where

$$u(\vec{x}) = \sum_{j=0} w_{1j}^{(2)} \varphi_j(\vec{x}), \qquad \varphi_j(\vec{x}) = h\left(\sum_{k=0} w_{jk}^{(1)} x_k \right)$$

where we defined $\phi_0 = x_0 = 1$ and wrote the sums over the nodes in the preceding layers starting from 0 to include the offsets.

So e.g. for event a we have $\dfrac{\partial E_a}{\partial w_{1j}^{(2)}} = (y_a - t_a)\, h'(u(\vec{x}))\, \varphi_j(\vec{x})$

derivative of activation function

Chain rule gives all the needed derivatives.