Machine Learning for BCI

The whirlwind tour

CSE599e: Brain-Computer Interfaces

Liberally inspired by

http://autonlab.org/tutorials

(Andrew W. Moore's slides on all things data-mining)

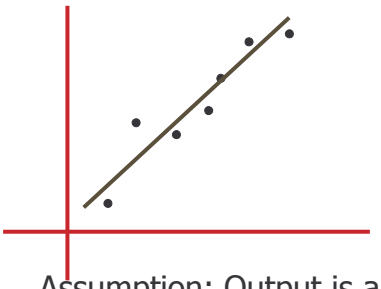
Outline

- Supervised Learning: Regression
  - Linear, polynomial.
  - RBFs, perceptrons, multilayer networks.
- Supervised Learning: Classification
  - Linear classifiers, max margin, kernels.
  - NN-based classifiers.
- Cross-validation
  - Model selection, preventing overfitting.

Why?

- Dominant paradigms in BCI literature
  - Correlate instantaneous hand position with neural firings $\Rightarrow$ *regression.* (Invasive BCIs)
  - Guess whether you're thinking tomatoes or oranges $\Rightarrow$ *classification.* (EEG BCIs)
- In each case, we have *example data.*
  - Supervised learning.

Linear Regression



x	y
1	3.1
2	6.4
3.1	8.7
0.7	2

Assumption: Output is a linear function of input, i.e.,

$$y_i = wx_i + noise_i$$

where noise is independent, gaussian, unknown fixed variance.

Linear Regression

Thus, y_i are drawn from $N(wx_i, \sigma^2)$.

Likelihood of data (y_i, x_i) for a given w,

$\Pi_i\ p(y_i \mid w, x_i)$ *i.e.,*

$\Pi_i\ \exp(-0.5\ (y_i - wx_i))^2/\sigma^2$

Maximize the likelihood of data given w.

i.e., maximize: $\Sigma_i\ -0.5(y_i - wx_i)^2/\sigma^2$

i.e., minimize: $\Sigma_i\ (y_i - wx_i)^2$

Easy to show that $w = \Sigma\ x_i\ y_i\ /\ \Sigma\ (x_i)^2$

Multivariate regression

Suppose inputs are vectors. We assume:

$$y = \mathbf{w}^T\mathbf{x} + \text{noise}.$$

We can write the data points as:

$$X = \begin{bmatrix} x_{11}\ x_{12}\ \dots \\ \dots \\ x_{i1}\ x_{i2}\dots \\ \dots \end{bmatrix} \qquad Y = \begin{bmatrix} y_1 \\ \dots \\ y_i \\ \dots \end{bmatrix}$$

Then, maximum likelihood **w** is

$$\mathbf{w} = (X^TX)^{-1}(X^TY)$$

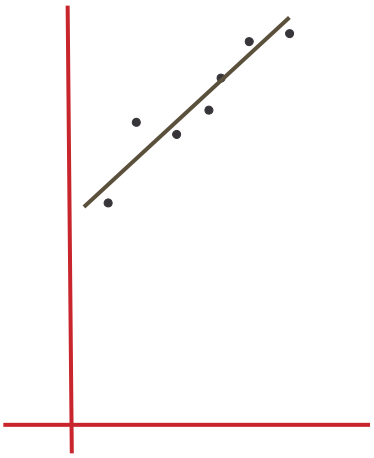
Linear regression: constants

What if data does not go through origin?

Simple hack for adding constant term:

$$y = wx + c$$

Any guesses?



Linear Regression: constants

- Add a dummy input fixed at 1.

x	y
1	8.1
2	11.4
3.1	13.7
0.7	7

→

z_1	z_2	y
1	1	8.1
1	2	11.4
1	3.1	13.7
1	0.7	7

Learn the regression function $y = w^T z + \text{noise}$.

Regression with Polynomials

- Quadratic, polynomial functions:
  - Same trick: replace input by extended-input

x_1	x_2	y
…	…	…
x_{i1}	x_{i2}	y_i
…	…	…

→

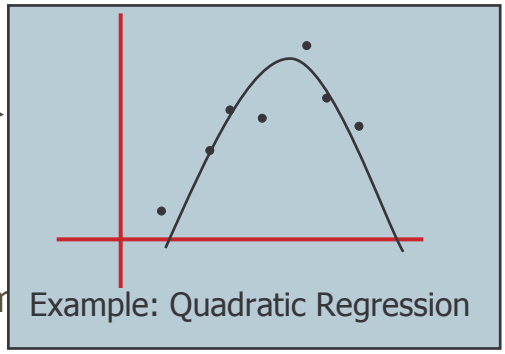
z_1	z_2	z_3	…	y
…	…	…	…	…
x_{i1}	x_{i2}	$(x_{i1})^2$	$x_{i1}x_{i2}$	y_i
…	…	…	…	…

- Again, learn the model $y = w^T z + \text{noise}$.

Regression with Polynomials

- Quadratic, polynomial functions:
  - Same trick: replace input by extended-input

x_1	x_2	y
…	…	…
x_{i1}	x_{i2}	y_i
…	…	…



Example: Quadratic Regression

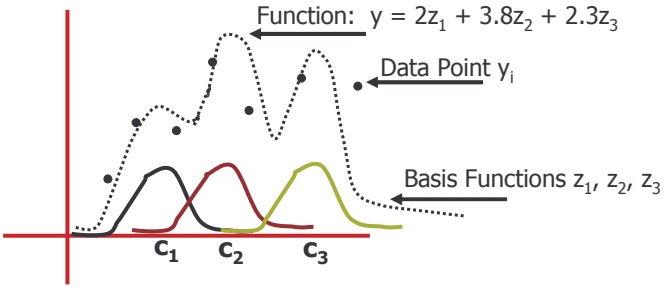
- Again, learn the m

Radial Basis Functions

- Create features that are some function of the entire input vector, e.g.,

  z_i = KernelFunction($|x_i - c_i|/\gamma_i$)
- c_i s and γ_i s are constant (will revisit).
- Idea: the kernel functions have reasonable overlap, and cover the interesting regions of input space.

RBFs: an example



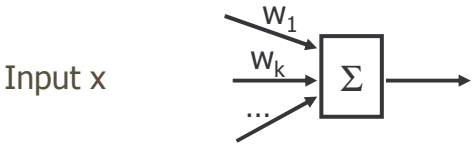
With c_i, γ_i known, this is standard linear regression.

How to select c_i and γ_i?

Answer: *Gradient Descent.*

Perceptrons

Another view of linear regression:



What if output E is not the expected value (y)?

Learning rule: $w_k \leftarrow w_k - \eta\ (O - y)\ x_{ik}$

(where η is the *learning rate*)

Intuition: Use error to appropriately correct w.

Guarantee: Will converge to ML estimate of w.

Perceptrons: Learning rule(s)

- Essentially performing *gradient descent:*

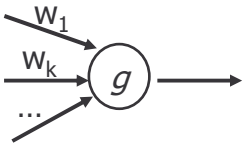
$$w_k \leftarrow w_k - \eta\ \delta(\text{error})/\delta w_k$$

- Possibly useful for incremental update.
- Idea generalizes to larger networks.
- Useful analogy to brain's learning mechanisms.

Sigmoid Perceptrons

- Add a sigmoid to the output:

Input x

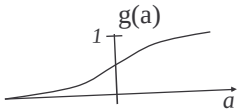


Out(x) = $g(w^T x)$

Suitable for *classification*

- Output is (almost) binary.
- Gradient descent guaranteed to converge, even with sigmoid.
- Performs well in practice.

$$g(a) = \frac{1}{1+e^{-\beta a}}$$

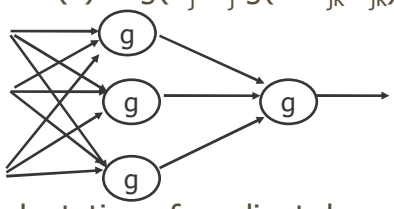


Multilayer Networks

Perceptron can only express limited class of functions:
$out(x) = g(w^T x)$

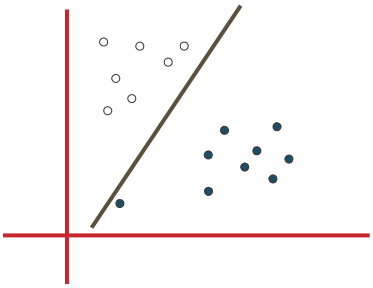
Solution: Chain individual nodes together:

$$Out(x) = g(\Sigma_j w_j g(\Sigma w_{jk} x_{jk}))$$



Learning is by adaptation of gradient descent—
back-propagate errors down the network.

Linear classifiers



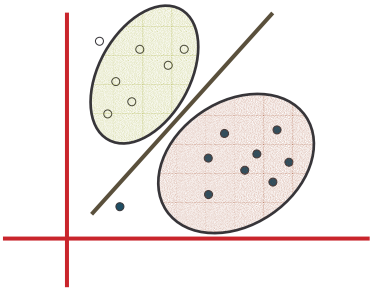
Data x_i are points in n-dimensional space, with labels +1/-1

Choose a plane (w, b) separating them.

Then, for any point x, $label(x) = sign(w^T x + b)$

Note: can use *magnitude* of output too, as a measure of plus-ousity.

Linear classifiers: LDA



Assume each class is a *gaussian cloud*

$N(\mu_k, \Sigma_k)$, $k = 1,2$

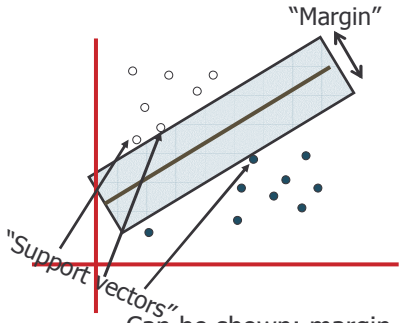
Define w as follows:

$w = \Sigma^{-1}(\mu_2 - \mu_1)$

Easy to compute.

Works if data is sufficiently low-dimensional.

Support Vector Machines



Choose "thickest hyperplane" for w.

Intuitively seems like a good pick, avoids outlier problems.

Can be shown: margin = $2/ w^T w$

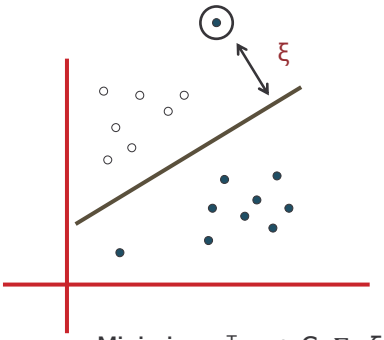
Maximize margin subject to the following:

$w^T x_i + b > +1$ for class1 points,

$w^T x_i + b < -1$ for class2 points.

Solved with Quadratic Programming.

SVMs: Soft Margin



What about outliers?

Allow *errors* ξ_i

How to control errors?

Trade off margin with errors.

Minimize $w^T w + C \sum_i \xi_i$ *subject to:*

$w^T x_i + b + \xi_i > +1$ for class1 points,

$w^T x_i + b - \xi_i < -1$ for class2 points,

and $\xi_i > 0$

How to choose C?

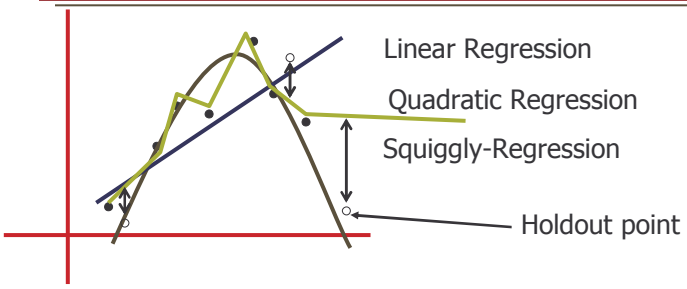
SVMs: Using Kernels

- Remember trick used for polynomial regression?
  - Replace inputs with functions of inputs:
    $x \leftarrow \Phi(x)$
- Compute a large-margin *linear* classifier in the "higher-dimensional space" of Φ.
- **Kernel trick:** we only ever need to look at *scalar products of points*: $\Phi(x_1) \cdot \Phi(x_2)$
- **Kernel trick:** define a *kernel function* k(x,y), e.g., $\Phi(x) \cdot \Phi(y) = k(x,y) = \exp(-(x-y)^2/2\sigma^2)$
- Can use (almost) arbitrary kfs, where the equivalent Φ space cannot be represented.

Multiclass Classifiers

- Guesses?
- Train M classifiers (for M-class problem)
  - class1 vs not-class1, and so on.
  - combine classifier outputs.
- Train M(M-1)/2 classifiers:
  - class1 vs class2 for every pair.
  - combine classifier outputs.

Cross-validation



Question: Which line? The squiggly *overfits* the data.

Answer: *Hold out* some of the points, evaluate performance on the held-out points. (This is what we want it to do in practice)

What if points are (un)lucky?

Cross-validation

- Rinse-repeat-cycle: for each choice of holdout set, do:
  - Do regression on remaining points.
  - Evaluate error on held-out points.
  - add to total error score.
- Choose model with least *generalization error.*
- Leave-one-out: *each point* is used as a holdout set.
- K-fold: Split data into k blocks, *each block* is used as holdout set.

CV for classification

- Choose model with least *generalization error.*
  - Can also use to choose *parameter* of the model (e.g., remember the cost tradeoffs in SVM classifier?)
  - Try a range of values (usually exponentially increasing).
  - Pick parameter value with least CV error.

Fin

References:
http://autonlab.org/tutorials
(Andrew W. Moore)