Regression

Machine Learning I, Week 5

N. Schraudolph
based in part on material by:
Andrew W. Moore
http://www.cs.cmu.edu/~awm/tutorials

Overview

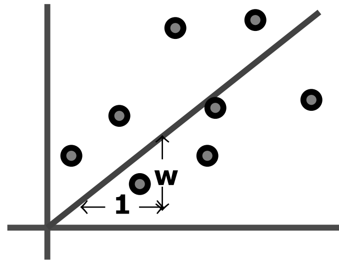
Last week:

- ❑ parametric *vs.* non-parametric models
  semi-parametric models, mixture models
- ❑ density estimation methods
  use of density estimation for classification

Today:

- ❑ least-squares regression: SVD, gradient descent
- ❑ linear *vs.* non-linear models, basis functions
- ❑ use of non-linear regression for classification

Linear Regression



DATA:

inputs	outputs
$x_1 = 1$	$y_1 = 1$
$x_2 = 3$	$y_2 = 2.2$
$x_3 = 2$	$y_3 = 2$
$x_4 = 1.5$	$y_4 = 1.9$
$x_5 = 4$	$y_5 = 3.1$

Linear regression assumes that the expected value of the output given an input, $E[y|x]$, is linear.

Simplest case: $\hat{y} = f(x) = wx$ for some unknown parameter w.

Given some data points (x_i, y_i), we can estimate w.

1-D Linear Regression

Assume that the data is formed by $y_i = wx_i + \varepsilon_i$,
with ε_i » $N(0, \sigma^2)$ (i.i.d. Gaussian noise).

$P(y_i|w, x_i)$ is then normally distributed with mean wx
and variance σ^2: $P(y|w,x)$ » $N(wx, \sigma^2)$.

The likelihood is $\prod_{i=1}^{n} P(y_i|w, x_i) = \prod_{i=1}^{n} \exp(-\frac{1}{2}(\frac{y_i - wx_i}{\sigma})^2)$

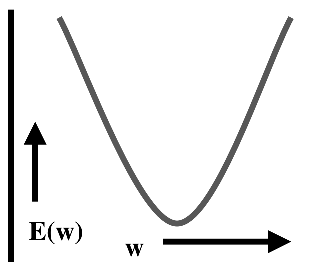
As usual, we minimize the negative log-likelihood

$$E(w) = \sum_{i=1}^{n} \frac{1}{2}\left(\frac{y_i - wx_i}{\sigma}\right)^2 \propto \sum_{i=1}^{n} \left(y_i - wx_i\right)^2$$

Linear Least-Squares Regression

The maximum likelihood w is the one that minimizes the sum-of-squares of residuals y – ŷ, *i.e.* the quadratic function

$$E(w) = \sum_i (y_i - wx_i)^2$$
$$= \sum_i y_i^2 - \left(2\sum x_i y_i\right)w + \left(\sum x_i^2\right)w^2$$



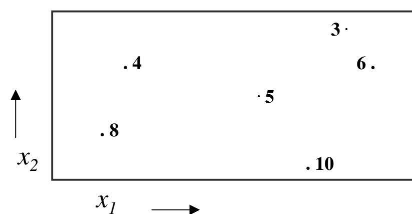
Setting the derivative to zero gives:

$$(\partial/\partial w)\, E(w^*) = 2w^* \sum x_i^2 - 2 \sum x_i y_i \;) \qquad w^* = \frac{\sum x_i y_i}{\sum x_i^2}$$

Multivariate Regression

What if the inputs are vectors?

2-d input example:



For R data points in m dimensions, data set has form

$$\mathbf{X} = \begin{bmatrix} \ldots\ldots \mathbf{x}_1 \ldots\ldots \\ \ldots\ldots \mathbf{x}_2 \ldots\ldots \\ \vdots \\ \ldots\ldots \mathbf{x}_R \ldots\ldots \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1m} \\ x_{21} & x_{22} & \ldots & x_{2m} \\ & & \vdots & \\ x_{R1} & x_{R2} & \ldots & x_{Rm} \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_R \end{bmatrix}$$

Multivariate Regression

The linear regression model assumes a parameter vector $\boldsymbol{w}$

$$\hat{y} \;=\; f(\boldsymbol{x},\boldsymbol{w}) \;=\; \boldsymbol{w}^T\boldsymbol{x} \;=\; w_1x_1 + w_2x_2 + \ldots. + w_mx_m$$

or in matrix notation: $\hat{\mathbf{y}} = \boldsymbol{X}\boldsymbol{w}$. The sum-squared loss is now

$E(\boldsymbol{w}) \;=\; (\hat{\mathbf{y}} - \mathbf{y})^T (\hat{\mathbf{y}} - \mathbf{y})$; setting its derivative to zero gives

$$(\partial/\partial \boldsymbol{w})\, E(\boldsymbol{w}^*) \;=\; 2\boldsymbol{X}^T(\hat{\mathbf{y}} - \mathbf{y}) \;=\; 0 \;\Rightarrow\; \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{y}$$

$$\Rightarrow\; \boldsymbol{w}^* \;=\; (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \;=\; \boldsymbol{X}^+\boldsymbol{y}.$$

$\boldsymbol{X}^+ = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T$ is called the **pseudo-inverse** of $\boldsymbol{X}$; it is best obtained by **singular value decomposition** (SVD).

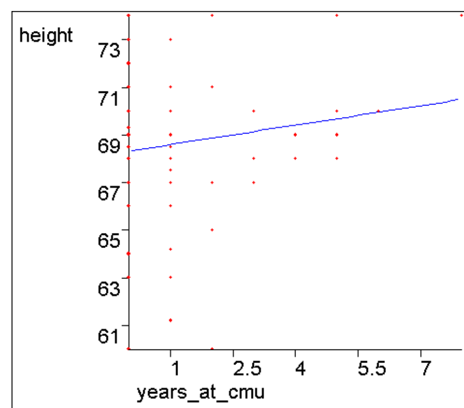
in Matlab: pinv(X)

What about a constant offset?

We may have linear data that does not go through the origin.

To deal with this, there is a simple, universally adopted hack.

Can you guess what it is?



The Bias Input

The trick: create a "fake" **bias input** x_0 that is always equal to 1:

x_1	x_2	y
2	4	16
3	4	17
5	5	20

x_0	x_1	x_2	y
1	2	4	16
1	3	4	17
1	5	5	20

Before:

$y = w_1x_1 + w_2x_2$

…bound to be a lousy fit!

After:

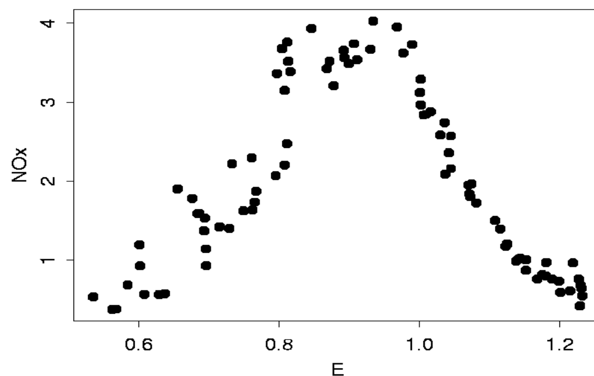
$y = w_0x_0 + w_1x_1 + w_2x_2$

$= w_0 + w_1x_1 + w_2x_2$

…has a fine constant term w_0!

What about non-linear data?

Relative concentration of NO and NO_2 in exhaust fumes as a function of the richness of the ethanol/air mixture burned in a car engine:



No straight line will ever fit this data well!

Generalized Least-Squares Regression

The trick: add extra inputs whose values are some **non-linear** function(s) of the original inputs. Least-squares regression now allows us to find the linear combination of these non-linear **basis functions** that best fits our data.

Example:

quadratic basis

x_0	x_1	x_2	x_1^2	x_2^2	x_1x_2	y
1	2	4	4	16	8	16
1	3	4	9	16	12	17
1	5	5	25	25	25	20

The algorithm (compute pseudo-inverse of ***X*** via SVD) remains the same, except that now ***X*** may be substantially bigger.

Basis Functions

Commonly used basis functions include

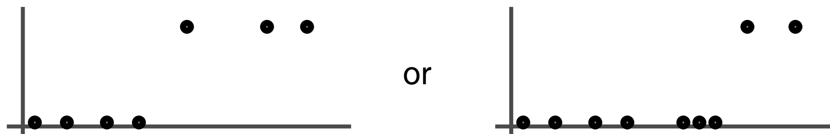
- ❒ polynomial basis: $x_k = x^k$ (not good for extrapolation!)
- ❒ Fourier basis: $x_k = [\sin(kx), \cos(kx)]$ (for periodic signals)
- ❒ Wavelets, in 2-D: Gabor functions, Gaussians, ...

Many of these bases are **orthogonal** (*i.e.*, $E(x_i x_j) = 0$ 8 $i \neq j$) and/or **universal,** *i.e.* any „reasonable“ function can be approximated to any degree given a sufficiently large basis.

Important: the choice of basis determines the character of the regression model; this form of prior is called **inductive bias.**

Classification via Regression

Example: two-class problems



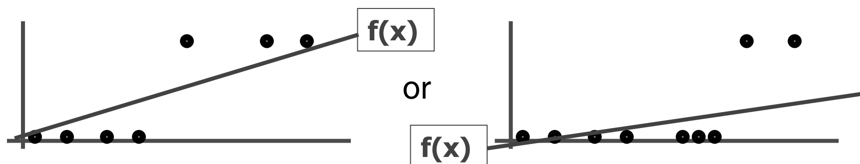
Could do a linear least-squares fit onto a 0/1 indicator function for class membership. Our ML prediction could then be

"class 0 if f($\boldsymbol{x}$)≤1/2, class 1 if f($\boldsymbol{x}$)>1/2"

What's the big problem with this?

Classification via Regression

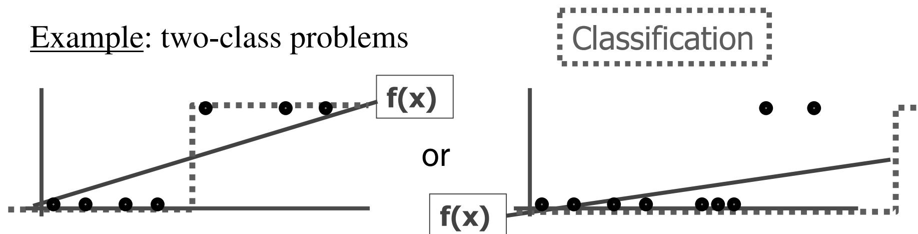
Example: two-class problems



Linear least-squares fit onto 0/1 class indicator function.

Classification via Regression

Example: two-class problems



Linear least-squares fit onto 0/1 class indicator function; thresholded to obtain 0/1 classification. Terrible!

This illustrates that sum-squared loss is **not** an appropriate loss function for classification problems.

Solution I: Classical Perceptron

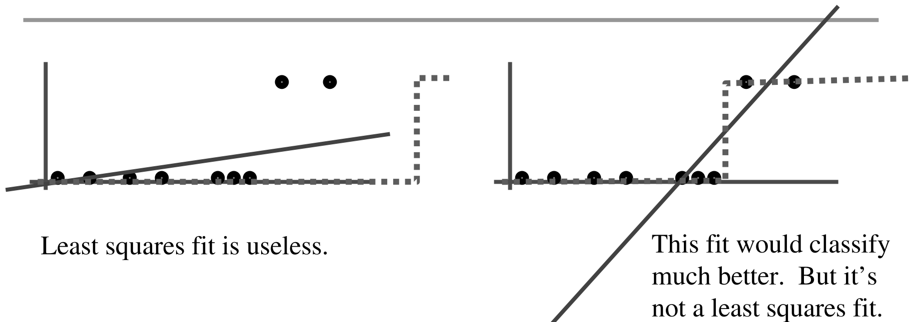
Don't minimize $\sum (y_i - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i)^2$. Minimize number of misclassifications instead:

$\sum (y_i - \mathrm{sign}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i))$ [Assume outputs are –1/+1, not 0/1]

This **zero-one loss** is minimized by the classical **perceptron learning rule** (remember the first week of class?):

- if $(\boldsymbol{x}_i, y_i)$ correctly classed, don't change $\boldsymbol{w}$
- if wrongly predicted as +1: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \boldsymbol{x}_i$
- if wrongly predicted as –1: $\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{x}_i$

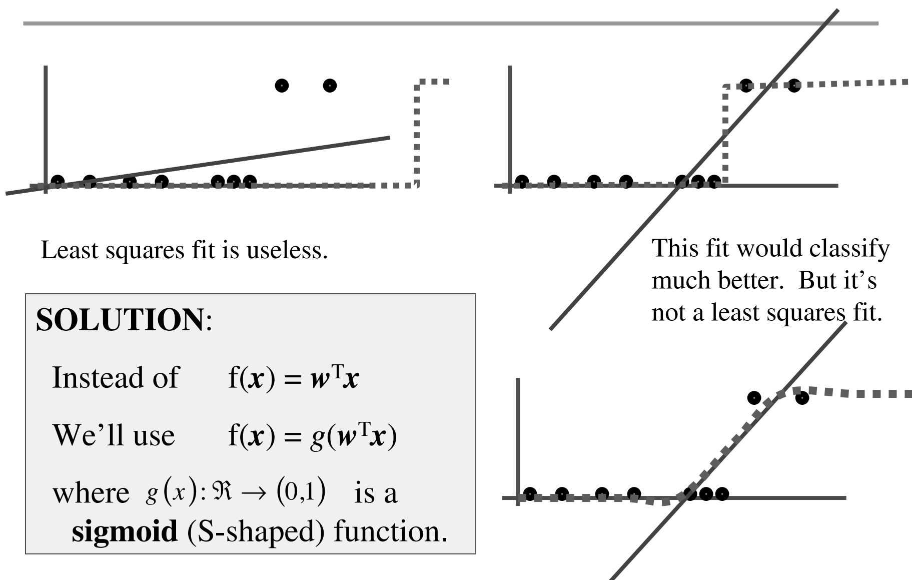
Solution II: Sigmoid Output



Least squares fit is useless.

This fit would classify much better. But it's not a least squares fit.

Solution II: Sigmoid Output



Least squares fit is useless.

This fit would classify much better. But it's not a least squares fit.

SOLUTION:

Instead of $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$

We'll use $f(\boldsymbol{x}) = g(\boldsymbol{w}^T\boldsymbol{x})$

where $g(x): \Re \to (0,1)$ is a **sigmoid** (S-shaped) function.

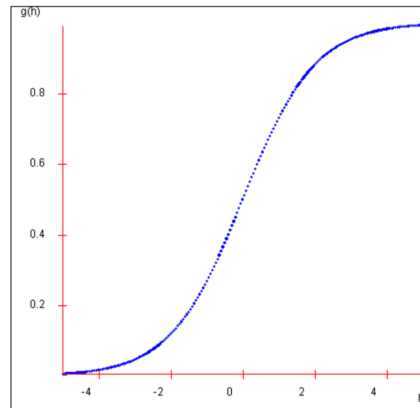
The Logistic Sigmoid

$$g(h) = \frac{1}{1+\exp(-h)}$$

If you rotate this curve through 180° centered on *(0,1/2)* you get back the same curve, *i.e.*

$g(h) = 1 - g(-h)$

Can you prove this?



Classification with Logistic Sigmoid

The non-linearity of g unfortunately means that we cannot directly minimize a loss function that depends on it by computing the pseudo-inverse. Instead, we have to use an iterative method: **gradient descent**.

When we use linear regression, followed by a logistic function, to implement a two-class classifier, the decision boundary will be a straight line in input space. This means that only **linearly separable** classification problems can be solved in this way.

As in regression, this limitation can be overcome by augmenting the input with non-linear basis functions.

Simple Gradient Descent

Given $f(\boldsymbol{w}): \Re^m \to \Re$, the **gradient**

$$\nabla f(\boldsymbol{w}) = \begin{pmatrix} \frac{\partial}{\partial w_1} f(\boldsymbol{w}) \\ \vdots \\ \frac{\partial}{\partial w_m} f(\boldsymbol{w}) \end{pmatrix}$$

points in the direction of **steepest ascent** of f.

Simple gradient descent rule: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla f(\boldsymbol{w})$

for the jth weight: $w_j \leftarrow w_j - \eta \frac{\partial}{\partial w_j} f(\boldsymbol{w})$

where **learning rate** η is a small positive number, *e.g.* η = 0.05.

Gradient Descent for LS Regression

$$E(\boldsymbol{w}) = \sum_{k=1}^{R} (y_k - \boldsymbol{w}^T \boldsymbol{x}_k)^2$$

Gradient descent tells us we should update **w** thusly if we wish to minimize *E:*

$$w_j \leftarrow w_j - \eta \frac{\partial E}{\partial w_j}$$

So what's $\frac{\partial E}{\partial w_j}$?

$$\frac{\partial E}{\partial w_j} = \sum_{k=1}^{R} \frac{\partial}{\partial w_j} (y_k - \mathbf{w}^T \mathbf{x}_k)^2$$

$$= \sum_{k=1}^{R} 2(y_k - \mathbf{w}^T \mathbf{x}_k) \frac{\partial}{\partial w_j} (y_k - \mathbf{w}^T \mathbf{x}_k)$$

$$= -2 \sum_{k=1}^{R} \delta_k \frac{\partial}{\partial w_j} \mathbf{w}^T \mathbf{x}_k$$

where $\delta_k = y_k - \boldsymbol{w}^T \boldsymbol{x}_k$

$$= -2 \sum_{k=1}^{R} \delta_k \frac{\partial}{\partial w_j} \sum_{i=1}^{m} w_i x_{ki}$$

$$= -2 \sum_{k=1}^{R} \delta_k x_{kj}$$

Gradient Descent for LS Regression

$$E(\boldsymbol{w}) = \sum_{k=1}^{R} (y_k - \boldsymbol{w}^T \boldsymbol{x}_k)^2$$

Gradient descent tells us we should update **w** thusly if we wish to minimize *E:*

$$w_j \leftarrow w_j - \eta \frac{\partial E}{\partial w_j}$$

where

$$\frac{\partial E}{\partial w_j} = -2 \sum_{k=1}^{R} \delta_k x_{kj}$$

We frequently neglect the 2 (meaning we halve the learning rate)

$$w_j \leftarrow w_j + 2\eta \sum_{k=1}^{R} \delta_k x_{kj}$$

where $\delta_k = y_k - \mathbf{w}^T \mathbf{x}_k$

Gradient Descent for Classification

The cross-entropy loss for our two-class problem with logistic sigmoid g is

$$E(\boldsymbol{w}) = \sum_k [y_k \log g(\boldsymbol{w}^T \boldsymbol{x}_k) + (1 - y_k) \log (1 - g(\boldsymbol{w}^T \boldsymbol{x}_k))].$$

You'd expect its gradient wrt. $\boldsymbol{w}$ to be terribly complicated. As it turns out though, many terms cancel, and you're left with the **same learning rule** as for least-squares regression! The only change is that obviously the residual is now $\delta_k = y_k - g(\boldsymbol{w}^T \boldsymbol{x}_k)$.

This is not a miraculous accident, but a deep consequence of cross-entropy loss being "right" for classification (in the sense of behaving the same way as least-squares does for regression).