HT2015: SC4
Statistical Data Mining and Machine Learning

Dino Sejdinovic
Department of Statistics
Oxford

http://www.stats.ox.ac.uk/~sejdinov/sdmml.html

Nearest Neighbours

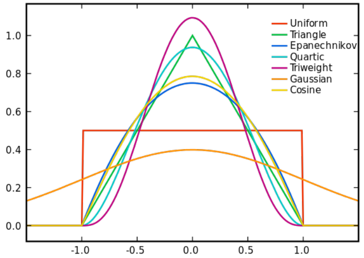
Nonlinear Methods

- Nonlinearity by data transformation: $x \mapsto \varphi(x)$ (explicit or implicit).
- A **global** approach. Decision function and optimal parameters can depend on training examples in the whole domain $\mathcal{X}$.
- Alternative approach: decision function $f(x)$ depends only on instances in the **local neighbourhood** of x.

Smoothing kernels

- Recall the plug-in generative classifier $f(x) = \operatorname{argmax}_{l \in \{1,\ldots,K\}} \hat{\pi}_l \hat{g}_l(x)$
- What if we do not want to assume that the true class-l conditional density $g_l(x)$ takes any particular form (i.e., multivariate normal)?
- Use a **kernel density estimate**

$$\hat{g}_l(x) = \frac{1}{n_l} \sum_{i:\, y_i = l} \kappa(x - x_i)$$



$$\underbrace{\text{smoothing (Parzen) kernel}}_{\text{local similarity}} \neq \underbrace{\text{positive-semidefinite (Mercer) kernel}}_{\text{inner product between features}}$$

Smoothing kernels

- **Kernel density estimate**

$$\hat{g}_l(x) = \frac{1}{n_l} \sum_{i:\, y_i = l} \kappa(x - x_i)$$

- since $\hat{\pi}_l = \frac{n_l}{n}$, discrimination based on total similarity of x to instances in each of the classes:

$$f(x) = \underset{l \in \{1,\ldots,K\}}{\operatorname{argmax}} \sum_{i:\, y_i = l} \kappa(x - x_i)$$

- Posterior class probabilities

$$\hat{\mathbb{P}}\left(Y = l | X = x\right) = \frac{\hat{\pi}_l \hat{g}_l(x)}{\sum_{j=1}^{K} \hat{\pi}_j \hat{g}_j(x)} = \frac{\sum_{i:\, y_i = l} \kappa(x - x_i)}{\sum_{j=1}^{n} \kappa(x - x_j)}$$

---

k-Nearest Neighbours

- Prediction at a data vector x is determined by the set $ne_{\mathrm{k}}(x)$ of k nearest neighbours of x among the training set.
- Classification: **majority vote** of the neighbours:

$$f_{\mathrm{kNN}}(x) = \underset{l}{\operatorname{argmax}} \quad |\{j \in ne_{\mathrm{k}}(x) : y_j = l\}|.$$

- Regression: average among the neighbours:

$$f_{\mathrm{kNN}}(x) = \frac{\sum_{j \in ne_{\mathrm{k}}(x)} y_j}{\mathrm{k}}.$$

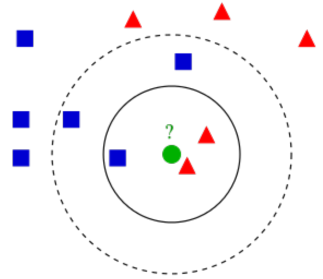
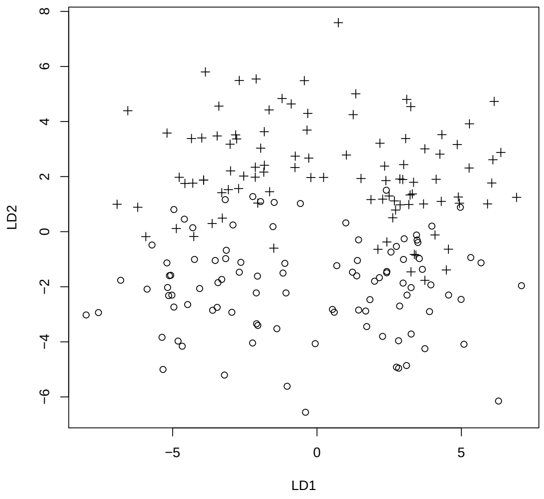


figure by A. Ajanki

---

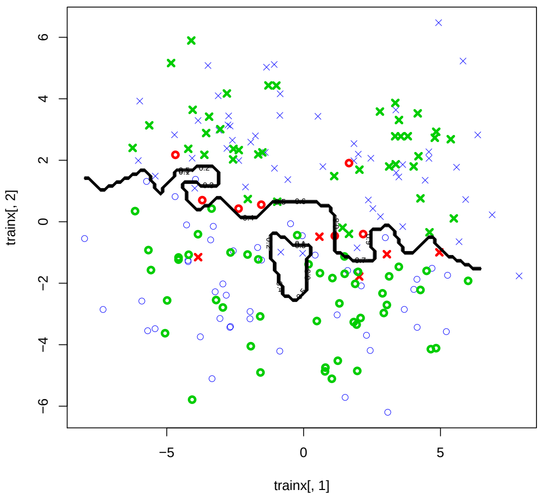
k-Nearest Neighbour Demo



Data

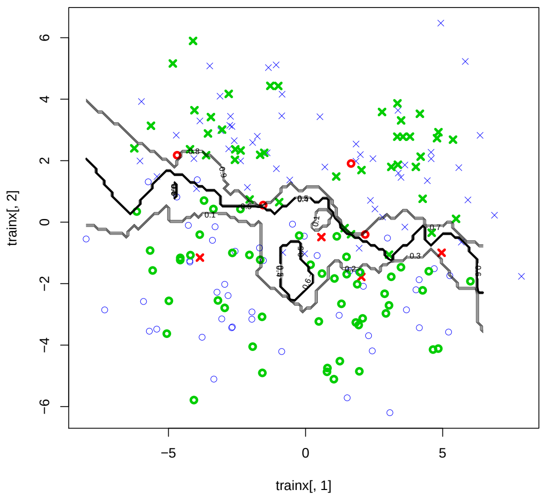
---

k-Nearest Neighbour Demo



Result of 1NN

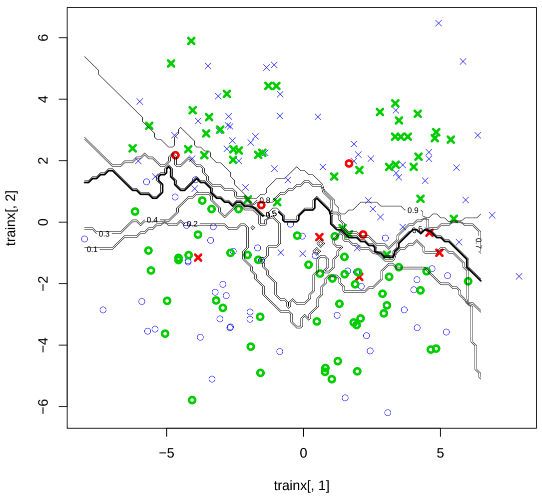
k-Nearest Neighbour Demo



Result of 3NN

---

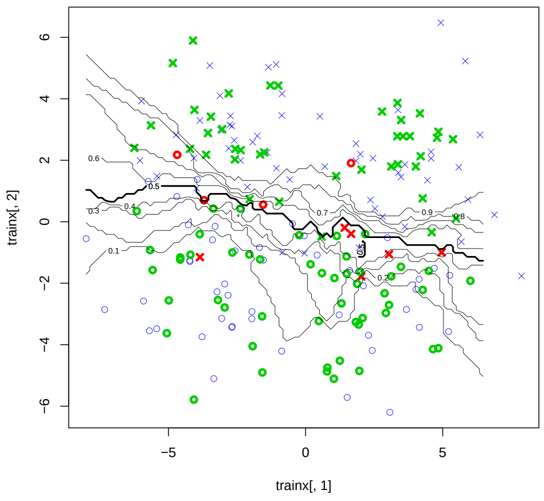
k-Nearest Neighbour Demo



Result of 5NN

---

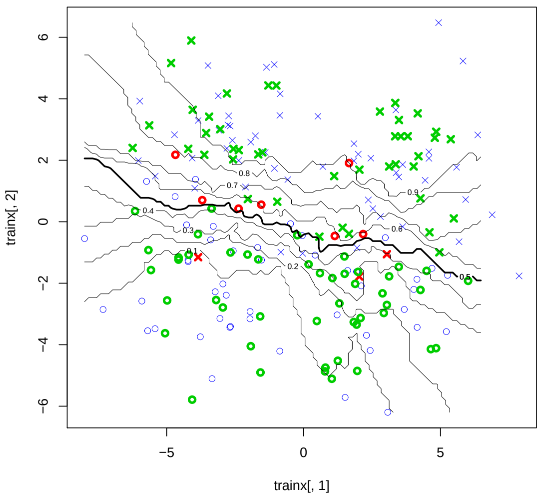
k-Nearest Neighbour Demo



Result of 11NN

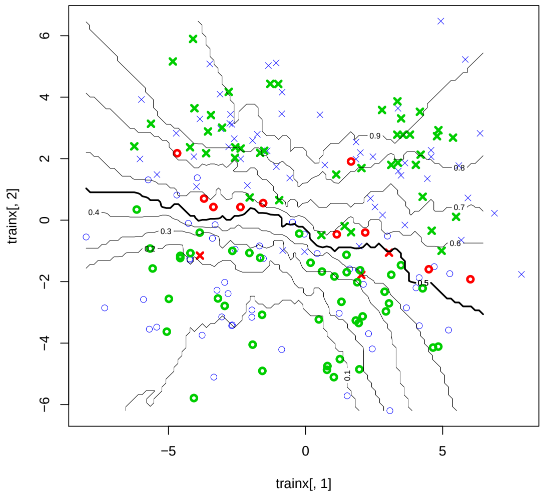
---

k-Nearest Neighbour Demo



Result of 21NN

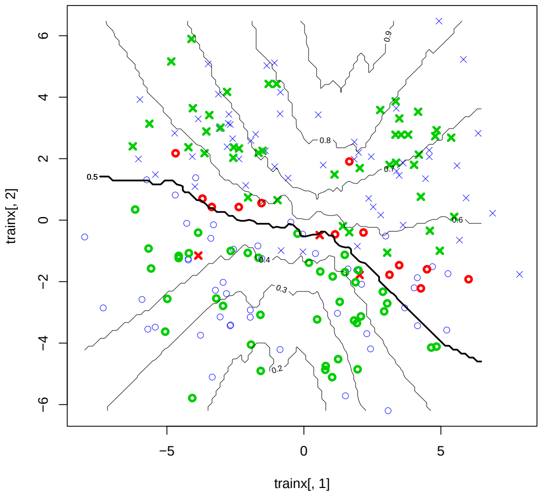
k-Nearest Neighbour Demo



Result of 31NN

---

k-Nearest Neighbour Demo



Result of 51NN

---

k-Nearest Neighbour Demo – R Code I

```
library(MASS)
## load crabs data
data(crabs)
ct <- as.numeric(crabs[,1])-1+2*(as.numeric(crabs[,2])-1)
## project to first two LD
cb.lda <- lda(log(crabs[,4:8]),ct)
cb.ldp <- predict(cb.lda)
x <- as.matrix(cb.ldp$x[,1:2])
y <- as.numeric(crabs[,2])-1
x <- x + rnorm(dim(x)[1]*dim(x)[2])*1.5
eqscplot(x,pch=2*y+1,col=1)
n      <- length(y)

#get training indices
i <- sample(rep(c(TRUE,FALSE),each=n/2),n,replace=FALSE)

kNN <- function(k,x,y,i,gridsize=100) {

  p      <- dim(x)[2]

  train  <- (1:n)[i]
  test   <- (1:n)[!i]
  trainx <- x[train,]
  trainy <- y[train]
  testx  <- x[test,]
  testy  <- y[test]

  trainn <- dim(trainx)[1]
  testn  <- dim(testx)[1]

  gridx1 <- seq(min(x[,1]),max(x[,2]),length=gridsize)
  gridx2 <- seq(min(x[,2]),max(x[,2]),length=gridsize)
  gridx  <- as.matrix(expand.grid(gridx1,gridx2))
  gridn  <- dim(gridx)[1]
```

---

k-Nearest Neighbour Demo – R Code II

```
  # calculate distances
  trainxx <- t((trainx*trainx) %*% matrix(1,p,1))
  testxx  <- (testx*testx)   %*% matrix(1,p,1)
  gridxx  <- (gridx*gridx)   %*% matrix(1,p,1)
  testtraindist <- matrix(1,testn,1) %*% trainxx +
    testxx %*% matrix(1,1,trainn) -
    2*(testx %*% t(trainx))
  gridtraindist <- matrix(1,gridn,1) %*% trainxx +
    gridxx %*% matrix(1,1,trainn) -
    2*(gridx %*% t(trainx))

  # predict
  testp <- numeric(testn)
  gridp <- numeric(gridn)
  for (j in 1:testn) {
    nearestneighbors <- order(testtraindist[j,])[1:k]
    testp[j] <- mean(trainy[nearestneighbors])
  }
  for (j in 1:gridn) {
    nearestneighbors <- order(gridtraindist[j,])[1:k]
    gridp[j] <- mean(trainy[nearestneighbors])
  }
  predy <- as.numeric(testp>.5)

  plot(trainx[,1],trainx[,2],pch=trainy*3+1,col=4,lwd=.5)
  points(testx[,1],testx[,2],pch=testy*3+1,col=2+(predy==testy),lwd=3)
  contour(gridx1,gridx2,matrix(gridp,gridsize,gridsize),
          levels=seq(.1,.9,.1),lwd=.5,add=TRUE)
  contour(gridx1,gridx2,matrix(gridp,gridsize,gridsize),
          levels=c(.5),lwd=2,add=TRUE)
}
```

Asymptotic Performance of 1NN

- Let $(x_i, y_i)_{i=1}^n$ be training data where $x_i \in \mathbb{R}^p$ and $y_i \in \{1, 2, ..., K\}$.
- We define

$$\begin{aligned} f_{\text{Bayes}}(x) &:= \arg\max_{l \in \{1,\dots,K\}} \pi_l g_l(x), \\ f_{\text{1NN}}^{(n)}(x) &:= y_j, \text{ s.t. } x_j \text{ is the nearest neigbour of } x. \end{aligned}$$

- The (optimal) Bayes risk and 1NN risk are:

$$\begin{aligned} R_{\text{Bayes}} &= \mathbb{E}\left[\mathbf{1}\left(Y \neq f_{\text{Bayes}}(X)\right)\right] \\ R_{\text{1NN}}^{(n)} &= \mathbb{E}\left[\mathbf{1}\left(Y \neq f_{\text{1NN}}^{(n)}(X)\right)\right] \end{aligned}$$

- As $n \to \infty$, $R_{\text{1NN}}^{(n)} \to R_{\text{1NN}}$, where

$$\boxed{R_{\text{Bayes}} \leq R_{\text{1NN}} \leq 2R_{\text{Bayes}} - \frac{K}{K-1} R_{\text{Bayes}}^2.}$$

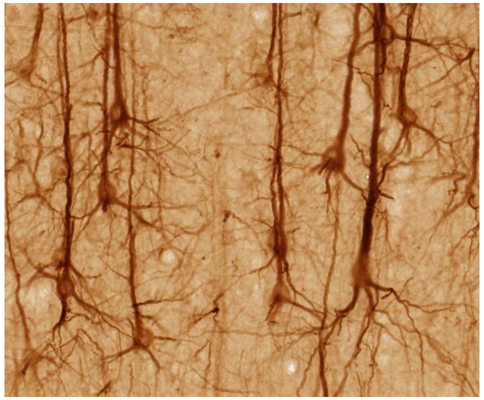
k-Nearest Neighbours – Discussion

- Simple and essentially model-free, i.e., weaker assumptions than LDA, Naïve Bayes and logistic regression.
- Not useful for understanding relationships between attributes and class predictions.
- Sensitive to the choice of distance and to the choice of k
- High computational cost:
  - Need to store **all** training data.
  - Need to compare each test data vector to **all** training data.
  - Need **a lot of data** in high dimensions.
- Mitigation: compute approximate nearest neighbours, using **kd-trees**, **cover trees**, **random forests**.

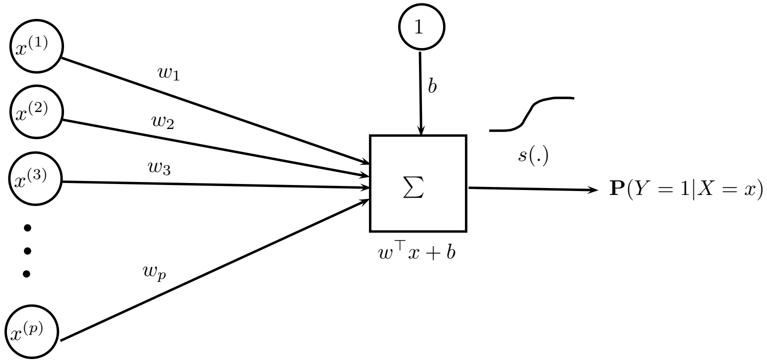
Artificial Neural Networks

Biological inspiration

- Basic computational elements: neurons.
- Receives signals from other neurons via dendrites.
- Sends processed signals via axons.
- Axon-dendrite interactions at synapses.
- $10^{10} - 10^{11}$ neurons.
- $10^{14} - 10^{15}$ synapses.
- Connectionist architecture: the network and its structure govern the computations performed.

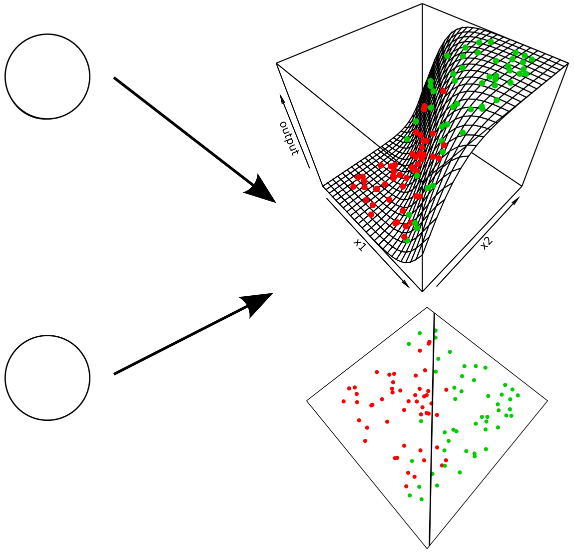


Single Neuron Classifier

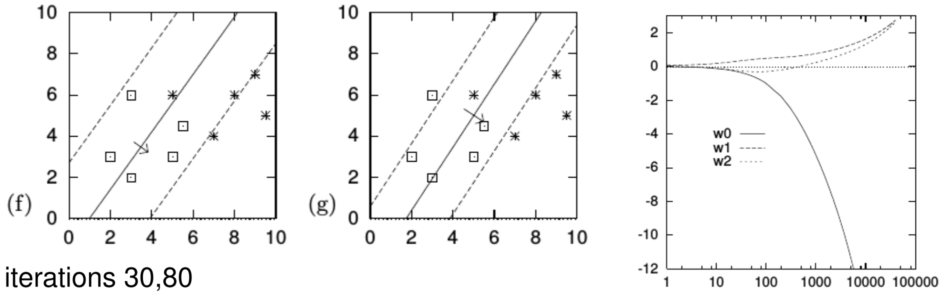


- **activation** $w^\top x + b$ (linear in **inputs** x)
- **activation/transfer function** s gives the **output/activity** (potentially nonlinear in x)
- common nonlinear activation function $s(a) = \frac{1}{1+e^{-a}}$: **logistic regression**
- learn w and b via gradient descent

Single Neuron Classifier



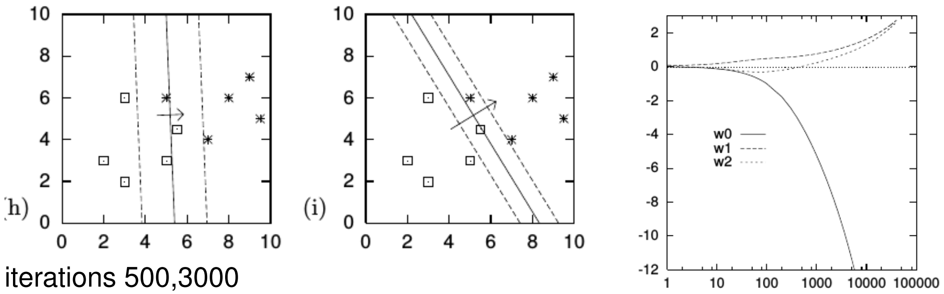
Overfitting



iterations 30,80

Figures from D. MacKay, **Information Theory, Inference and Learning Algorithms**

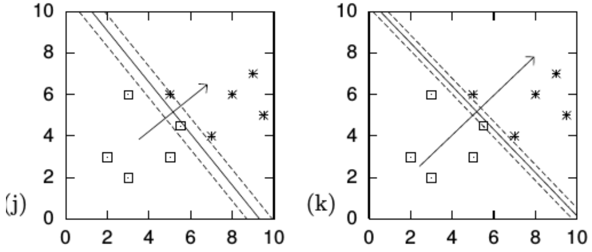
Overfitting

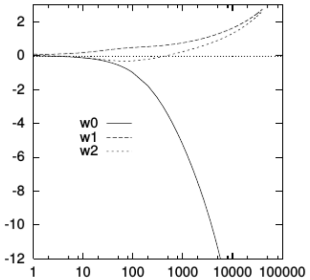


iterations 500,3000

Figures from D. MacKay, **Information Theory, Inference and Learning Algorithms**

Overfitting





iterations 10000,40000

prevent overfitting by:

- **early stopping**: just halt the gradient descent
- regularization: L_2-regularization called **weight decay** in neural networks literature.

Figures from D. MacKay, **Information Theory, Inference and Learning Algorithms**

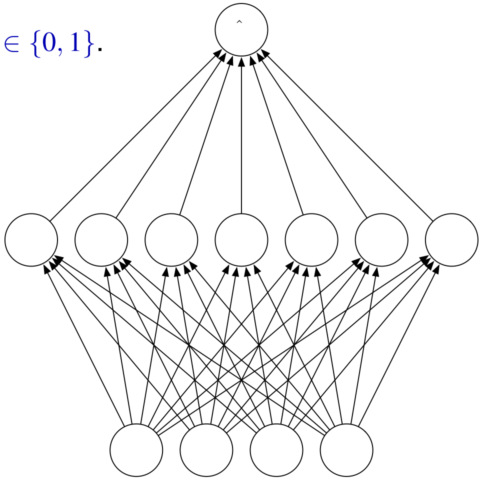
Multilayer Networks

- Data vectors $x_i \in \mathbb{R}^p$, binary labels $y_i \in \{0, 1\}$.
- **inputs** $x_{i1}, \dots, x_{ip}$
- **output** $\hat{y}_i = \mathbb{P}(Y = 1 | X = x_i)$
- **hidden unit activities** $h_{i1}, \dots, h_{im}$
  - Compute **hidden unit activities**:

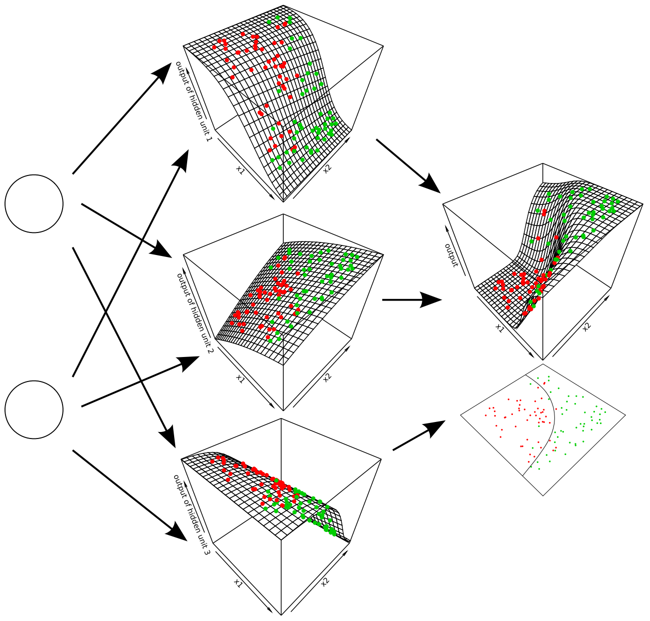
$$h_{il} = s\left(b_l^h + \sum_{j=1}^{p} w_{jl}^h x_{ij}\right)$$

  - Compute **output probability**:

$$\hat{y}_i = s\left(b^o + \sum_{l=1}^{m} w_k^o h_{il}\right)$$



Multilayer Networks



Training a Neural Network

- Objective function: L_2-regularized log-loss

$$J = -\sum_{i=1}^{n} y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) + \frac{\lambda}{2}\left(\sum_{jl} (w_{jl}^h)^2 + \sum_{l} (w_l^o)^2\right)$$

where

$$\hat{y}_i = s\left(b^o + \sum_{l=1}^{m} w_l^o h_{il}\right) \qquad h_{il} = s\left(b_l^h + \sum_{j=1}^{p} w_{jl}^h x_{ij}\right)$$

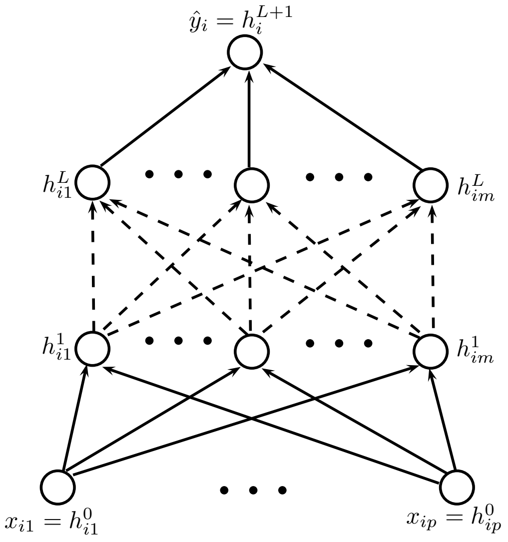
- Optimize parameters $\theta = \{b^h, w^h, b^o, w^o\}$, where $b^h \in \mathbb{R}^m$, $w^h \in \mathbb{R}^{p \times m}$, $b^o \in \mathbb{R}$, $w^o \in \mathbb{R}^m$ with gradient descent.

$$\frac{\partial J}{\partial w_l^o} = \lambda w_l^o + \sum_{i=1}^{n} \frac{\partial J}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial w_l^o} = \lambda w_l^o + \sum_{i=1}^{n} (\hat{y}_i - y_i) h_{il},$$

$$\frac{\partial J}{\partial w_{jl}^h} = \lambda w_{jl}^h + \sum_{i=1}^{n} \frac{\partial J}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial h_{il}} \frac{\partial h_{il}}{\partial w_{jl}^h} = \lambda w_{jl}^h + \sum_{i=1}^{n} (\hat{y}_i - y_i) w_l^o h_{il}(1 - h_{il}) x_{ij}.$$

- L_2-regularization often called **weight decay**.
- Multiple hidden layers: **Backpropagation** algorithm

Multiple hidden layers

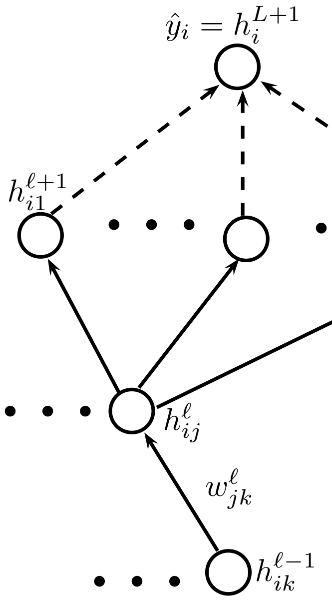


$$h_i^{\ell+1} = \underline{s}\left(W^{\ell+1} h_i^{\ell}\right)$$

- $W^{\ell+1} = \left(w_{jk}^{\ell}\right)_{jk}$: weight matrix at the $(\ell+1)$-th layer, weight w_{jk}^{ℓ} on the edge between $h_{ik}^{\ell-1}$ and h_{ij}^{ℓ}
- $\underline{s}$: entrywise (logistic) transfer function

$$\hat{y}_i = \underline{s}\left(W^{L+1}\underline{s}\left(W^L\left(\cdots\underline{s}\left(W^1 x_i\right)\right)\right)\right)$$

Backpropagation



$$J = -\sum_{i=1}^{n} y_i \log h_i^{L+1} + (1-y_i)\log(1-h_i^{L+1})$$

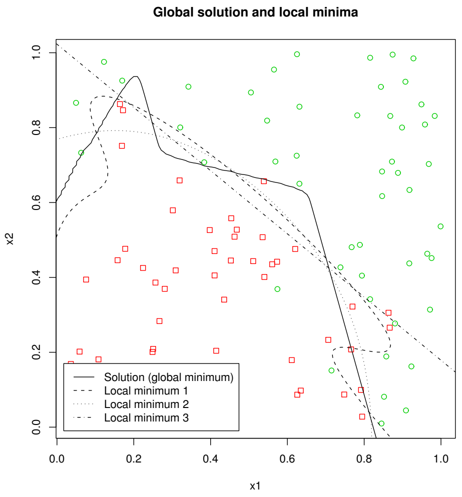
- Gradients wrt h_{ij}^{ℓ} computed by recursive applications of chain rule, and propagated through the network backwards.

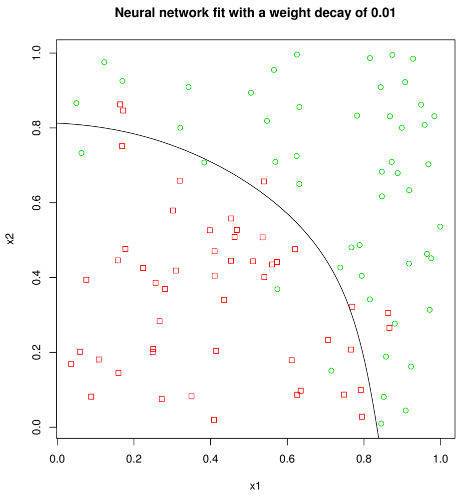
$$\frac{\partial J}{\partial h_i^{L+1}} = -\frac{y_i}{h_i^{L+1}} + \frac{1-y_i}{1-h_i^{L+1}}$$

$$\frac{\partial J}{\partial h_{ij}^{\ell}} = \sum_{r=1}^{m} \frac{\partial J}{\partial h_{ir}^{\ell+1}} \frac{\partial h_{ir}^{\ell+1}}{\partial h_{ij}^{\ell}}$$

$$\frac{\partial J}{\partial w_{jk}^{\ell}} = \sum_{i=1}^{n} \frac{\partial J}{\partial h_{ij}^{\ell}} \frac{\partial h_{ij}^{\ell}}{\partial w_{jk}^{\ell}}$$

Neural Networks





R package implementing neural networks with a single hidden layer: `nnet`.

Neural Networks – Discussion

- Nonlinear hidden units introduce modelling flexibility.
- In contrast to user-introduced nonlinearities, features are global, and can be learned to maximize predictive performance.
- Neural networks with a single hidden layer and sufficiently many hidden units can model arbitrarily complex functions.
- Optimization problem is **not convex**, and objective function can have many local optima, plateaus and ridges.
- On large scale problems, often use **stochastic gradient descent**, along with a whole host of techniques for optimization, regularization, and initialization.
- Recent developments, especially by Geoffrey Hinton, Yann LeCun, Yoshua Bengio, Andrew Ng and others. See also `http://deeplearning.net/`.