Radial Basis Functions

João Neto

November 2013

- Radial basis function network

A radial basis function, RBF, $\phi(x)$ is a function with respect to the origin or a certain point c, ie, $\phi(x) = f(||x-c||)$ where the norm is usually the Euclidean norm but can be other type of measure.

The RBF learning model assumes that the dataset $\mathcal{D} = (x_n, y_n), n = 1 \ldots N$ influences the hypothesis set $h(x)$, for a new observation x, in the following way:

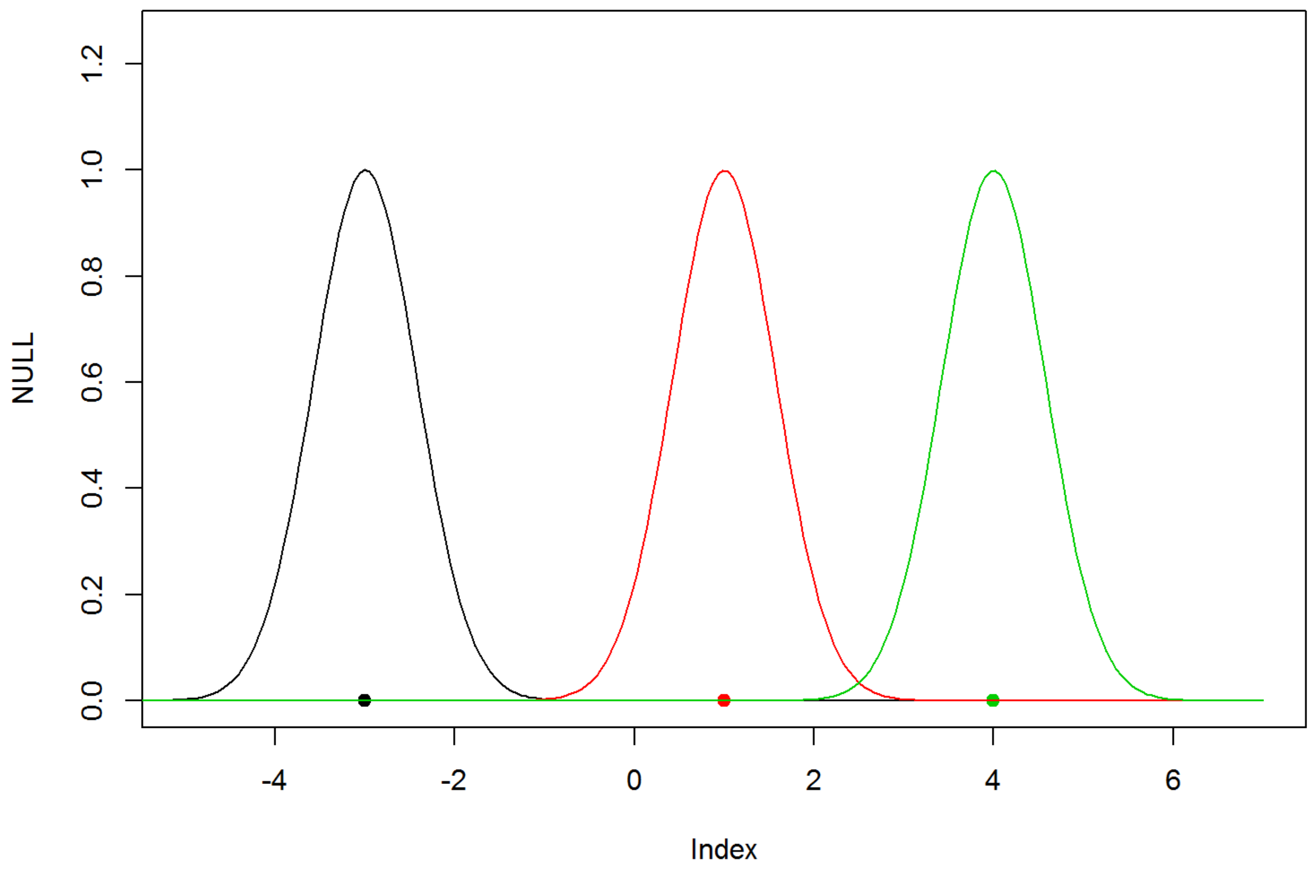
$$h(x) = \sum_{n=1}^{N} w_n \times exp(-\gamma ||x - x_n||^2)$$

which means that each x_i of the dataset influences the observation in a gaussian shape. Of course, if a datapoint is far away from the observation its influence is residual (the exponential decay of the tails of the gaussian make it so). It is an example of a localized function ($x \to \infty \implies \phi(x) \to 0$). Notice that other type of radial functions can be used. Some egs:

- Multi-quadratic: $\phi(x) = \sqrt{x^2 + \gamma^2}$
- Thin plate spline: $\phi(x) = x^2 ln(x)$

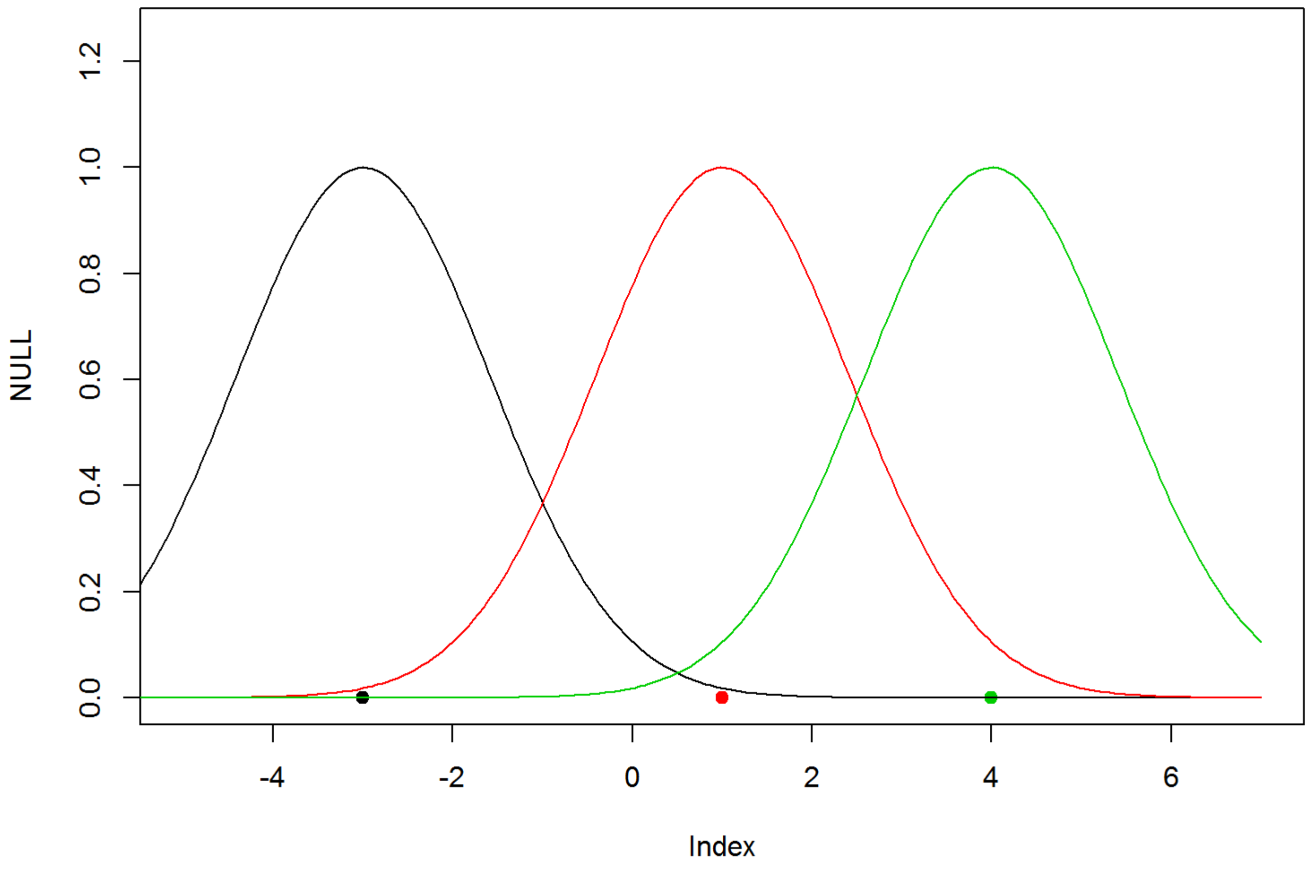
Let's use a one-dimensional dataset as an ilustration of the gaussian influence:

```r
rbf.gauss <- function(gamma=1.0) {

  function(x) {
    exp(-gamma * norm(as.matrix(x),"F")^2 )
  }
}

D <- matrix(c(-3,1,4), ncol=1) # 3 datapoints
N <- length(D[,1])

xlim  <- c(-5,7)
plot(NULL,xlim=xlim,ylim=c(0,1.25),type="n")
points(D,rep(0,length(D)),col=1:N,pch=19)
x.coord = seq(-7,7,length=250)
gamma <- 1.5
for (i in 1:N) {
  points(x.coord, lapply(x.coord - D[i,], rbf.gauss(gamma)), type="l", col=i)
}
```



The value of gamma controls how far or how little the influnce of each datapoint is felt:

```r
plot(NULL,xlim=xlim,ylim=c(0,1.25),type="n")
points(D,rep(0,length(D)),col=1:N,pch=19)
x.coord = seq(-7,7,length=250)
gamma <- 0.25
for (i in 1:N) {
  points(x.coord, lapply(x.coord - D[i,], rbf.gauss(gamma)), type="l", col=i)
}
```



We will need a learning algorithm to find the appropriate $\boldsymbol{w} = (w_1, w_2, \dots, w_n)$.

The choice of $\mathbf{w}$ should follow the goal of minimizing the in-sample error of the dataset $\mathcal{D}$, $E_{in}(\mathcal{D})$, or simply E_{in}.

This means $\mathbf{w}$ should satisfy:

$$\sum_{m=1}^{N} w_m \times exp(-\gamma \|x_n - x_m\|^2) = y_n$$

for each datapoint $x_n \in \mathcal{D}$.

There are N equations for N unknowns. In matrix form:

$$\underbrace{\begin{bmatrix} \exp(-\gamma\|x_1 - x_1\|^2) & \cdots & \exp(-\gamma\|x_1 - x_N\|^2) \\ \exp(-\gamma\|x_2 - x_1\|^2) & \cdots & \exp(-\gamma\|x_2 - x_N\|^2) \\ \vdots & \ddots & \vdots \\ \exp(-\gamma\|x_N - x_1\|^2) & \cdots & \exp(-\gamma\|x_N - x_N\|^2) \end{bmatrix}}_{\Phi} \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}}_{\boldsymbol{w}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_{\boldsymbol{y}}$$

or simply: $\mathbf{w} = \Phi^{-1} y$ which is an exact interpolation of the dataset.

Sometimes there is an inclusion of a bias parameter, b:

$$h(x) = b + \sum_{n=1}^{N} w_n \times exp(-\gamma \|x - x_n\|^2)$$

This produces the following change in the matrix form above (b becomes w_0):

$$\underbrace{\begin{bmatrix} 1 & \exp(-\gamma\|x_1 - x_1\|^2) & \cdots & \exp(-\gamma\|x_1 - x_N\|^2) \\ 1 & \exp(-\gamma\|x_2 - x_1\|^2) & \cdots & \exp(-\gamma\|x_2 - x_N\|^2) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \exp(-\gamma\|x_N - x_1\|^2) & \cdots & \exp(-\gamma\|x_N - x_N\|^2) \end{bmatrix}}_{\Phi} \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}}_{\boldsymbol{w}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_{\boldsymbol{y}}$$

Notice now that Φ is not a square matrix, so we need to compute the pseudo-inverse:

$$w = (\Phi^T\Phi)^{-1}\Phi^T y$$

This is a regression method. To adapt it for classification tasks (i.e., $y_n \in \{-1, 1\}$), we compute:

$$h(x) = \text{sign}(b + \sum_{n=1}^{N} w_n \times exp(-\gamma\|x - x_n\|^2))$$

where $\text{sign}()$ is the typical sign function.

However, if N is large, the need to use all in-sample datapoints is exagerated and might overfit. A variant is to choose K representatives ($K << N$) that can do the interpolation. Choose the best K centers among the possible $x_i \in \mathcal{D}$ is NP-hard, so a possible option is to select K centers using a clustering algorithm, namely a K-means clustering. Let's name those centers $\mu_1, \ldots, \mu_K$. These μ_k define the set of *basis functions*.

Thus, the matrix form reduce the matrix Φ from a $N \times (N+1)$ matrix to a smaller $N \times (K+1)$ matrix:

$$\underbrace{\begin{bmatrix} 1 & exp(-\gamma\|x_1 - \mu_1\|^2) & \cdots & exp(-\gamma\|x_1 - \mu_K\|^2) \\ 1 & exp(-\gamma\|x_2 - \mu_1\|^2) & \cdots & exp(-\gamma\|x_2 - \mu_K\|^2) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & exp(-\gamma\|x_N - \mu_1\|^2) & \cdots & exp(-\gamma\|x_N - \mu_K\|^2) \end{bmatrix}}_{\Phi} \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}}_{\boldsymbol{w}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_{\boldsymbol{y}}$$

And again:

$$w = (\Phi^T\Phi)^{-1}\Phi^T y$$

Let's implement this in R

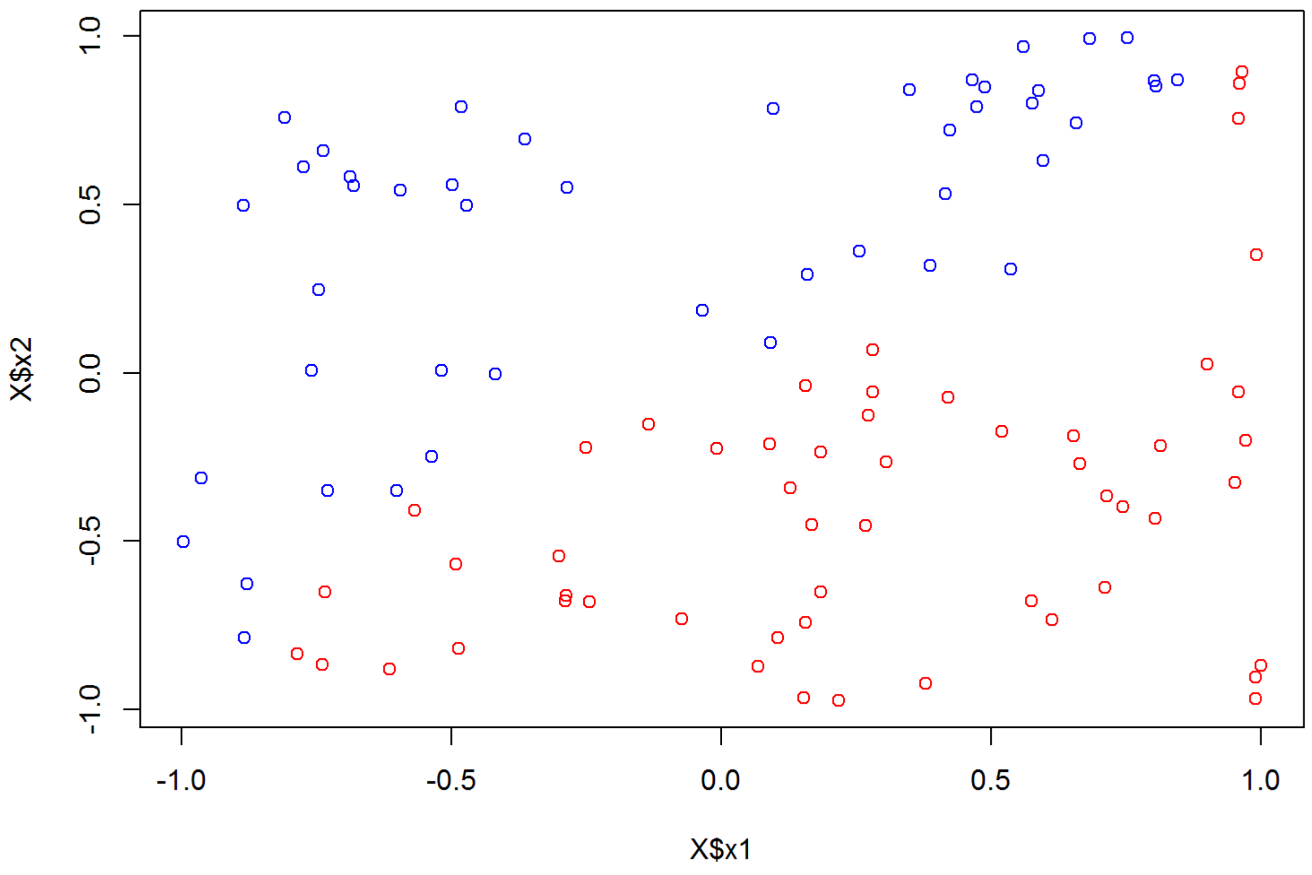
```r
# returns a rbf model given the:
# * observations x1, xN of dataset D
# * output value for each observation
# * number of centers
# * gamma value

rbf <- function(X, Y, K=10, gamma=1.0) {
  N     <- dim(X)[1] # number of observations
  ncols <- dim(X)[2] # number of variables

  repeat {
    km <- kmeans(X, K)  # let's cluster K centers out of the dataset
    if (min(km$size)>0) # only accept if there are no empty clusters
      break
  }

  mus <- km$centers # the clusters points

  Phi <- matrix(rep(NA,(K+1)*N), ncol=K+1)
  for (lin in 1:N) {
    Phi[lin,1] <- 1    # bias column
    for (col in 1:K) {
      Phi[lin,col+1] <- exp( -gamma * norm(as.matrix(X[lin,]-mus[col,]),"F")^2 )
    }
  }

  w <- pseudoinverse(t(Phi) %*% Phi) %*% t(Phi) %*% Y  # find RBF weights

  list(weights=w, centers=mus, gamma=gamma)  # return the rbf model
}
```

And also an implementation for the prediction function:

```
rbf.predict <- function(model, X, classification=FALSE) {

  gamma   <- model$gamma
  centers <- model$centers
  w       <- model$weights
  N       <- dim(X)[1]    # number of observations

  pred <- rep(w[1],N)  # we need to init to a value, so let's start with the bias

  for (j in 1:N) {
    # find prediction for point xj
    for (k in 1:length(centers[,1])) {
      # the weight for center[k] is given by w[k+1] (because w[1] is the bias)
      pred[j] <- pred[j] + w[k+1] * exp( -gamma * norm(as.matrix(X[j,]-centers[k,]),"F")^2 )
    }
  }

  if (classification) {
    pred <- unlist(lapply(pred, sign))
  }
  pred
}
```

Let’s see an example:

```
target <- function(x1, x2) {
  2*(x2 - x1 + .25*sin(pi*x1) >= 0)-1
}

N <- 100
X <- data.frame(x1=runif(N, min=-1, max=1),
                x2=runif(N, min=-1, max=1))
Y <- target(X$x1, X$x2)
plot(X$x1, X$x2, col=Y+3)
```



Now let’s learn the dataset using the RBFs:

```
rbf.model <- rbf(X, Y) # using default values for K and gamma
rbf.model
```

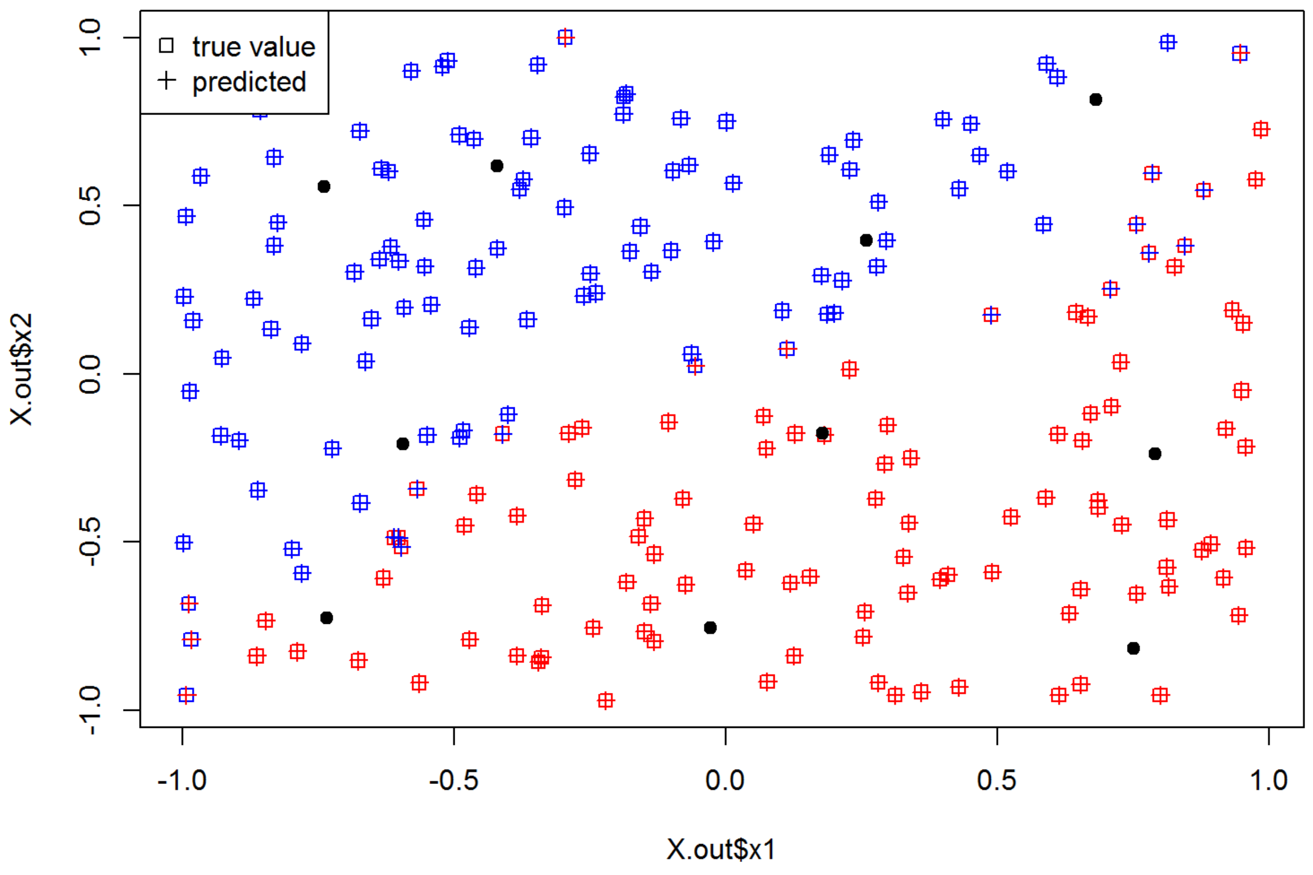
```
## $weights
##             [,1]
##  [1,] -4.1869647
##  [2,] -2.0964171
##  [3,]  2.1608708
##  [4,]  8.3543678
##  [5,] -4.8642154
##  [6,]  0.6223674
##  [7,] -0.6106960
##  [8,]  3.5330173
##  [9,] -6.3038222
## [10,]  5.8279082
## [11,]  7.5053400
##
## $centers
##             x1         x2
## 1   0.79089169 -0.2369843
## 2   0.68147040  0.8154599
## 3  -0.73936896  0.5569151
## 4   0.17883923 -0.1770207
## 5  -0.59411279 -0.2094878
## 6  -0.02798912 -0.7536152
## 7  -0.73454084 -0.7268801
## 8  -0.42022077  0.6182727
## 9   0.75134910 -0.8170974
## 10  0.25960235  0.3975370
##
## $gamma
## [1] 1
```

And make a prediction over a new test set:

```
N.test <- 200
X.out <- data.frame(x1=runif(N.test, min=-1, max=1),
                    x2=runif(N.test, min=-1, max=1))
Y.out <- target(X.out$x1, X.out$x2)

rbf.pred <- rbf.predict(rbf.model, X.out, classification=TRUE)
binary.error <- sum(rbf.pred != Y.out)/N.test
binary.error
```

```
## [1] 0.1
```

```
plot(X.out$x1, X.out$x2, col=Y.out+3, pch=0)
points(X.out$x1, X.out$x2, col=rbf.pred+3, pch=3)
points(rbf.model$centers, col="black", pch=19) # draw the model centers
legend("topleft",c("true value","predicted"),pch=c(0,3),bg="white")
```



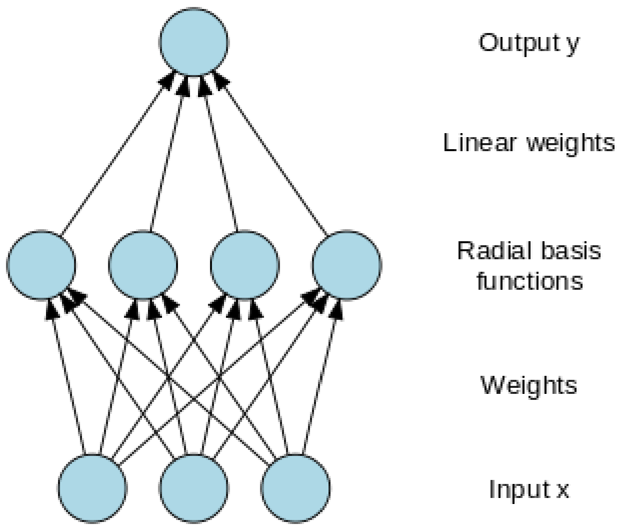
It is also possible to generalize this method to assign different values of γ for each center, i.e., instead of having one γ we would have $\gamma_1, \ldots, \gamma_K$. In this scenario the model will contain a mixture of gaussians.

One way to solve this problem is to iterate using a EM algorithm in the following way:

1. Set init values for γ_k
2. Solve RBF to find w
3. With fixed w minimize error with respect to γ_k
4. Repeat 1&2 until a convergence condition is satisfied

Radial basis function network

This radial basis function can be organized into the hidden layer of a neural network, and this type of network is called RBF Networks. The output of the network is a linear combination of RBFs of the inputs and neuron parameters.



Given the an observation x, and using K neurons (like before, there correspond to centers computed by clustering), the output is

$$y = \phi(x) = \sum_{i=1}^{K} w_i exp(-\gamma \|x - \mu_i\|^2)$$

where w_i is the weigth of the i^{th} neuron and μ_i is the center vector of that same neuron.

Given the gaussian decays, changing parameters of one neuron has only a small effect for input values that are far away from the center of that neuron. RBF network are universal approximators, so with enough hidden neurons they approximate any continuous function with arbitrary precision.

Again a learning algorithm chooses w_i, μ_i and γ to optimize the fit between $\phi(x)$ and the training dataset $\mathcal{D}$.

Let’s try an example using the RSNNS R package:

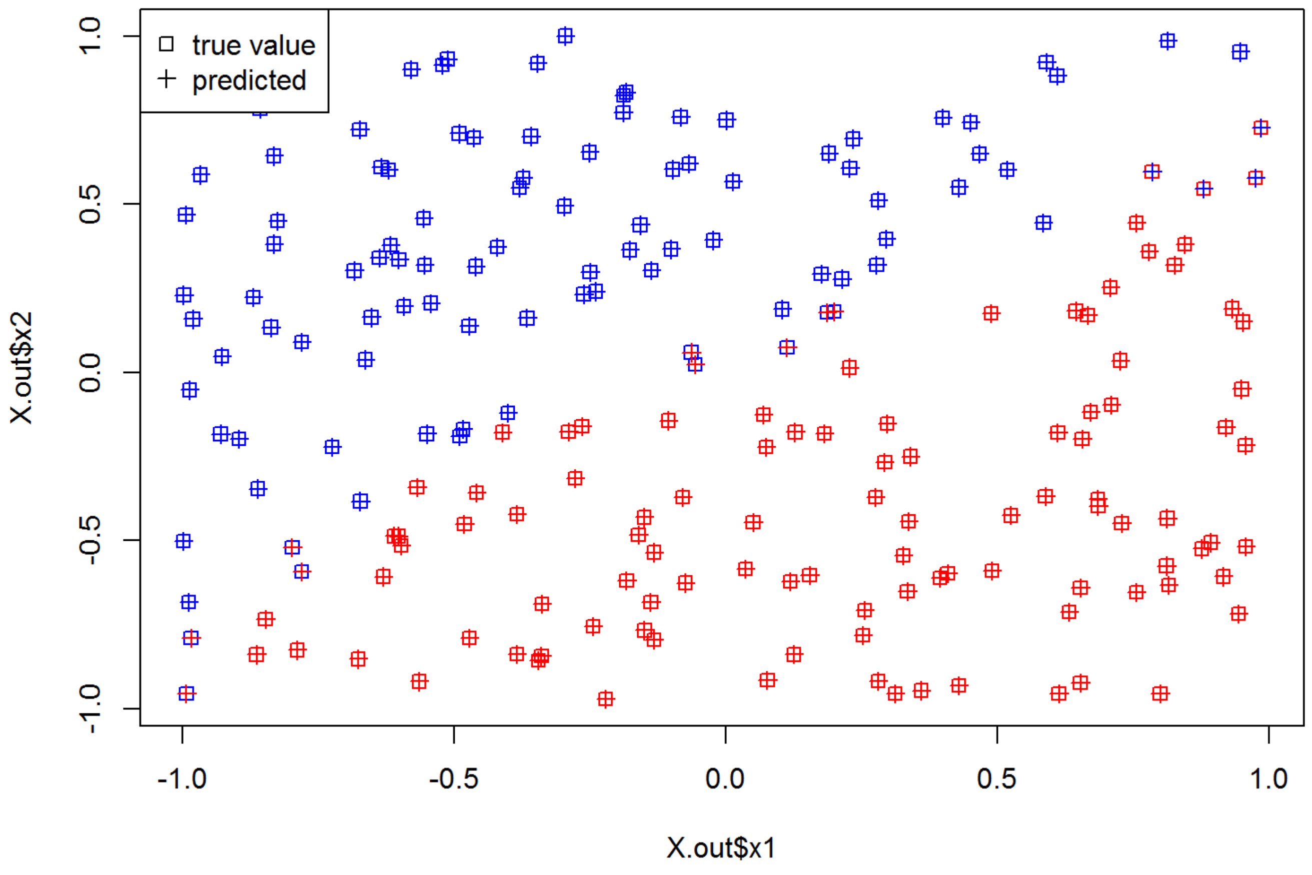
```
library("RSNNS")
```

```
## Loading required package: Rcpp
##
## Attaching package: 'RSNNS'
##
## The following object is masked _by_ '.GlobalEnv':
##
##     rbf
```

```
rbfn.model <- RSNNS::rbf(as.matrix(X,ncol=2),
                         as.matrix(Y),
                         size=30,    # number of centers, ie, number of neurons in hidden lay
er
                         maxit=1000, # max number of iterations to learn
                         linOut=TRUE) # linear activation function (otherwise logistic)

rbf.network.pred <- sign( predict(rbfn.model, X.out) ) # apply sign, since this is a classifi
cation eg
binary.error <- sum(rbf.network.pred != Y.out)/N.test
binary.error
```

```
## [1] 0.065
```

```
plot(X.out$x1, X.out$x2, col=Y.out+3, pch=0)
points(X.out$x1, X.out$x2, col=rbf.network.pred+3, pch=3)
legend("topleft",c("true value","predicted"),pch=c(0,3),bg="white")
```


□ true value
+ predicted
X.out$x2
X.out$x1