Machine Intelligence 1

Exercise Sheet 03

Gruppe:
MeSi
Autoren:
Jens Meiners
Arne Siebenmorgen

Classification

Training Data Create a sample of $P = 120$ training patterns $\{\boldsymbol{x}_\alpha, t_\alpha\},\ \alpha = 1, \ldots, P$. The input values $\boldsymbol{x}_\alpha \in \mathbb{R}^2$ should be drawn from a mixture of Gaussians with centers in an XOR-configuration according to the following scheme:

- Generate 60 samples from each of the two conditional distributions:

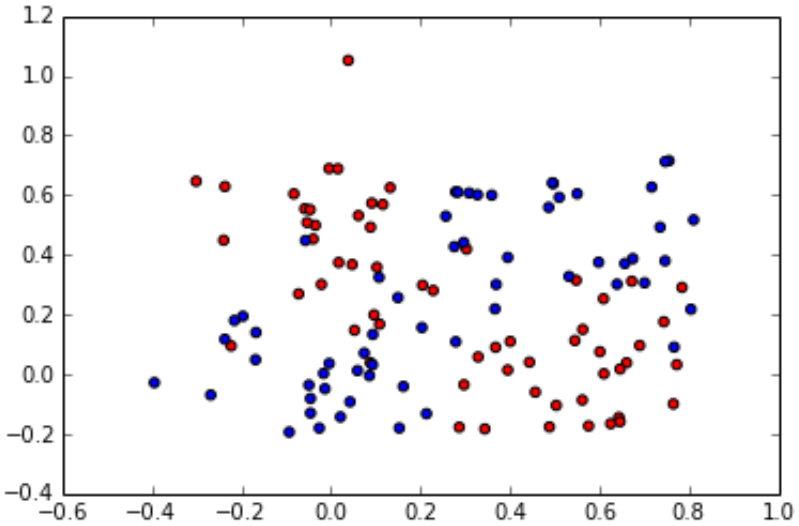
$$\begin{aligned} p_1 &:= p(\boldsymbol{x}|\mathcal{C}_1) = 0.5[\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_1, \sigma) + \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_2, \sigma)] \\ p_2 &:= p(\boldsymbol{x}|\mathcal{C}_2) = 0.5[\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_3, \sigma) + \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_4, \sigma)] \end{aligned}$$

  with $\boldsymbol{\mu}_1 = (0,1)^\top, \boldsymbol{\mu}_2 = (1,0)^\top, \boldsymbol{\mu}_3 = (0,0)^\top, \boldsymbol{\mu}_4 = (1,1)^\top$ and a variance of $\sigma^2 = 0.1$

- The corresponding target values $t_\alpha \in \{-1, 1\}$ describe the assignment to the two classes $\mathcal{C}_1, \mathcal{C}_2$ and indicate from which distribution (p_1 vs. p_2) the data point was drawn.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

#training data
cov = [[0.1, 0], [0, 0.1]]

mean1 = [0. , 1.]
mean2 = [1. , 0.]
mean3 = [0. , 0.]
mean4 = [1. , 1.]

p1 = 0.5*(np.vstack((np.random.multivariate_normal(mean1, cov, 30), np.ran
dom.multivariate_normal(mean2, cov, 30))))
p2 = 0.5*(np.vstack((np.random.multivariate_normal(mean3, cov, 30), np.ran
dom.multivariate_normal(mean4, cov, 30))))

plt.scatter(p1[:,0], p1[:,1], c='red')
plt.scatter(p2[:,0], p2[:,1], c='blue')
plt.show()


p1 = np.append(p1,np.ones((60,1)), axis=1)
p2 = np.append(p2,np.zeros((60,1)), axis=1)

trndata = np.append(p1,p2, axis =0)
np.random.shuffle(trndata)
```



6.1 k Nearest Neighbors (kNN) (2 points)

(a) Build a kNN classifier that classifies new data (*query points*) by voting of the k nearest neighbors from the training set. Thus the *electoral committee* is selected from the training patterns $\{\boldsymbol{x}_\alpha, t_\alpha\}$, $\alpha = 1, \ldots, P$ according their Euclidean distance to the query point. The predicted class is determined by the target values of the majority of those k nearest patterns.

In [2]:

```
def kNNclassifier(point, trndata, k, verbose=False):
    #find k nearest neighbors
    dist = np.zeros((len(trndata),1))
    point = np.asarray(point)
    for i in range(len(trndata)):
        dist[i] = np.linalg.norm(point-trndata[i,0:2])
    distdata = np.concatenate((trndata,dist), axis = 1)
    distdata = distdata[distdata[:,3].argsort()]
    neighbors = distdata[0:k,:]
    #classify
    cls = 0
    vote = np.sum(neighbors[:,2])/k
    if(vote>=0.5):
        cls = 1
    if(verbose):
        print "neighbors",neighbors
        print "vote",vote
    return cls
```

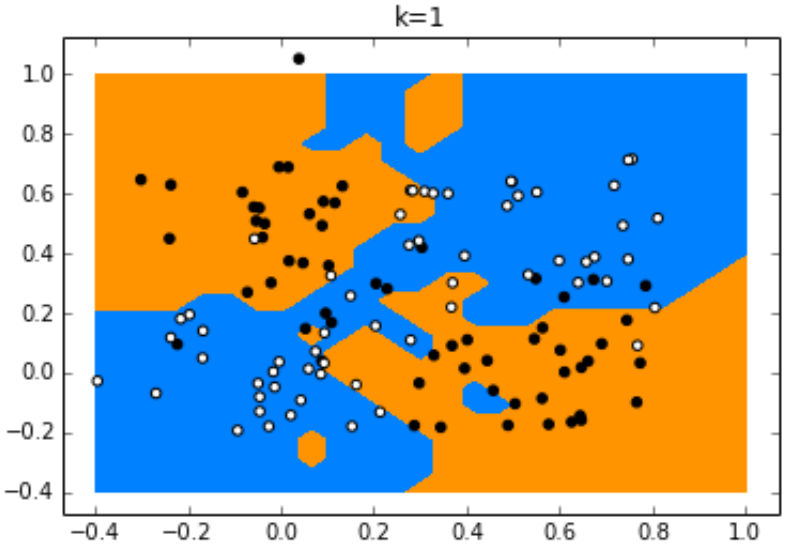
(b) **Visualization:** Plot the training patterns and the decision boundary (e.g. using a contour plot) in input space for $k = 1, 3, 5$.

In [3]:

```
x_range = np.linspace(-0.4,1,25)
y_range = np.linspace(-0.4,1,25)

xx,yy = np.meshgrid(x_range,y_range)
```

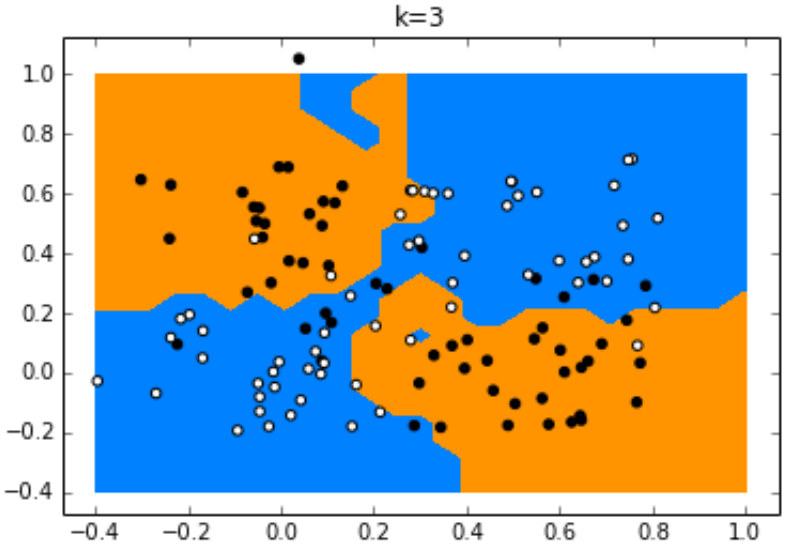
In [4]:

```
#k=1
zz = np.zeros(xx.shape)
for x in range(len(x_range)):
    for y in range(len(y_range)):
        zz[x,y]=  kNNclassifier((xx[0,x],yy[y,0]),trndata,1)

plt.contourf(xx, yy, zz, levels=[0.,0.5,1.])
plt.scatter(p1[:,0], p1[:,1], c='black')
plt.scatter(p2[:,0], p2[:,1], c='white')
plt.title("k=1")
plt.show()
```



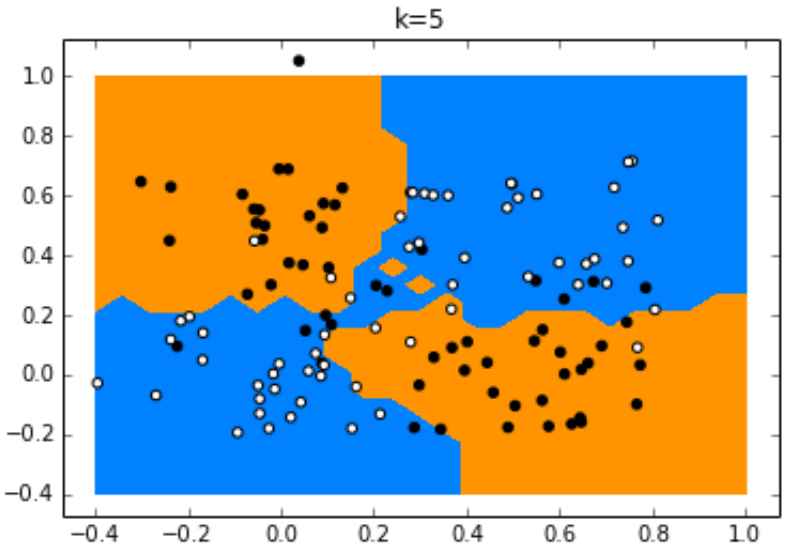
In [5]:

```
#k=3
zz = np.zeros(xx.shape)
for x in range(len(x_range)):
    for y in range(len(y_range)):
        zz[x,y]=  kNNclassifier((xx[0,x],yy[y,0]),trndata,3)

plt.contourf(xx, yy, zz, levels=[0.,0.5,1.])
plt.scatter(p1[:,0], p1[:,1], c='black')
plt.scatter(p2[:,0], p2[:,1], c='white')
plt.title("k=3")
plt.show()
```



In [6]:

```
#k=5
zz = np.zeros(xx.shape)
for x in range(len(x_range)):
    for y in range(len(y_range)):
        zz[x,y]=  kNNclassifier((xx[0,x],yy[y,0]),trndata,5)

plt.contourf(xx, yy, zz, levels=[0.,0.5,1.])
plt.scatter(p1[:,0], p1[:,1], c='black')
plt.scatter(p2[:,0], p2[:,1], c='white')
plt.title("k=5")
plt.show()
```



6.2 "Parzen Window" classifier (2 points)

This classifier implements a *weighted voting scheme*. All training points (not only the k nearest ones) make a vote for the query point but their vote is weighted by a *Parzen window* or kernel function depending on the distance between the training and query points.

(a) Use a Gaussian window function parametrized by the variance σ^2 to determine the classification results for $\sigma^2 = 0.5, 0.1, 0.01$.

In [7]:

```
from scipy.stats import multivariate_normal
```

In [8]:

```
def parzenclassifier(point, trndata, sig, verbose=False):

    dist = np.zeros((len(trndata),1))
    vote0 = 0
    vote1 = 0
    point = np.asarray(point)
    for i in range(len(trndata)):

        g_pdf = multivariate_normal(mean=trndata[i,0:2], cov=[[sig,0],[0,s
ig]])

        if(trndata[i,2]==0):
            vote0 += g_pdf.pdf(point)
        elif(trndata[i,2]==1):
            vote1 += g_pdf.pdf(point)

    cls = 0
    if(vote1>=vote0):
        cls = 1
    if(verbose):
        print "vote",vote
    return cls
```

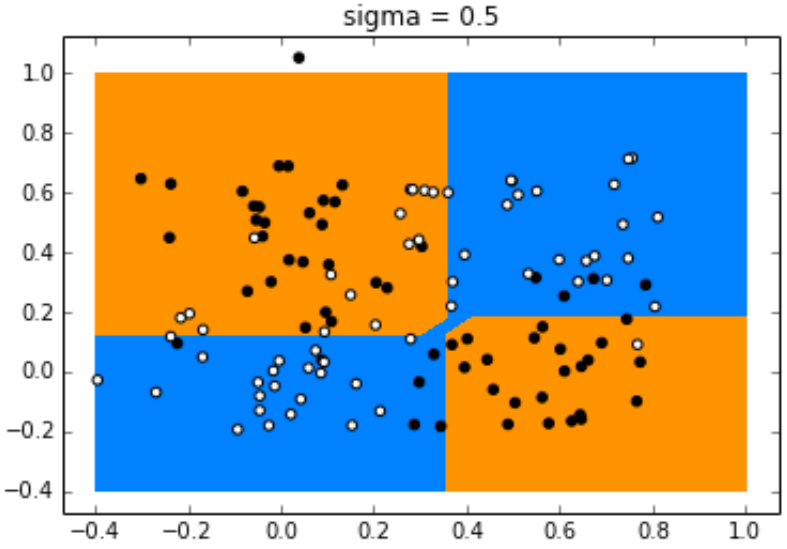
(b) **Visualization:** Plot the training patterns and the decision boundary (e.g. using a contour plot) in input space for each σ^2.

In [9]:

```
x_range = np.linspace(-0.4,1,25)
y_range = np.linspace(-0.4,1,25)

xx,yy = np.meshgrid(x_range,y_range)
```

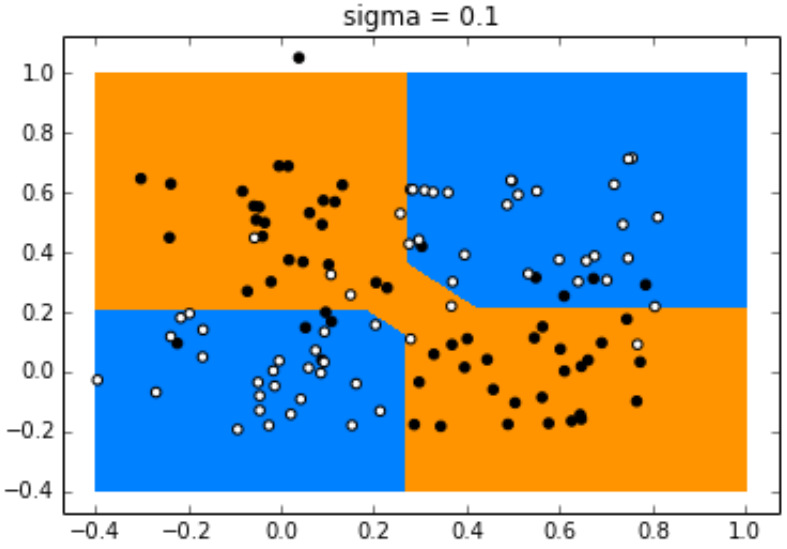
In [10]:

```
zz = np.zeros(xx.shape)
for x in range(len(x_range)):
    for y in range(len(y_range)):
        zz[x,y]=  parzenclassifier((xx[0,x],yy[y,0]),trndata,0.5)

plt.contourf(xx, yy, zz, levels=[-1,0,1.])
plt.scatter(p1[:,0], p1[:,1], c='black')
plt.scatter(p2[:,0], p2[:,1], c='white')
plt.title("sigma = 0.5")
plt.show()
```



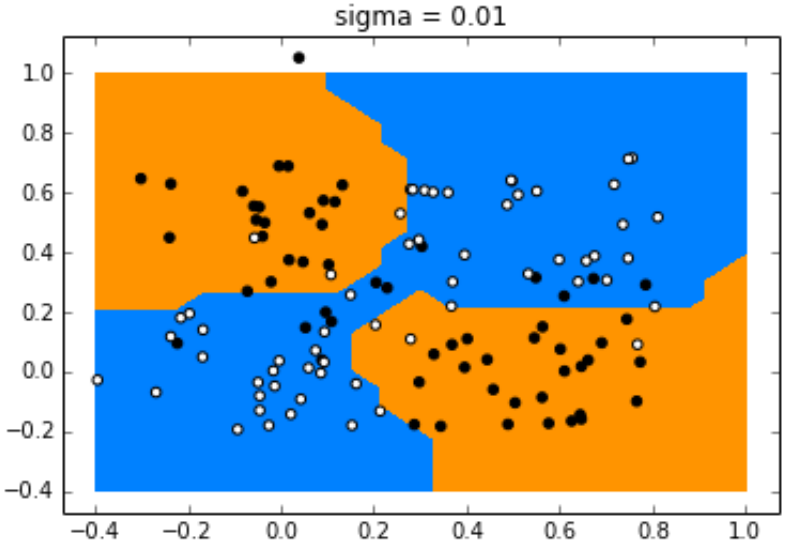
In [11]:

```
zz = np.zeros(xx.shape)
for x in range(len(x_range)):
    for y in range(len(y_range)):
        zz[x,y]=  parzenclassifier((xx[0,x],yy[y,0]),trndata,0.1)

plt.contourf(xx, yy, zz, levels=[0.,0.5,1.])
plt.scatter(p1[:,0], p1[:,1], c='black')
plt.scatter(p2[:,0], p2[:,1], c='white')
plt.title("sigma = 0.1")
plt.show()
```



In [12]:

```
zz = np.zeros(xx.shape)
for x in range(len(x_range)):
    for y in range(len(y_range)):
        zz[x,y]=  parzenclassifier((xx[0,x],yy[y,0]),trndata,0.01)

plt.contourf(xx, yy, zz, levels=[0.,0.5,1.])
plt.scatter(p1[:,0], p1[:,1], c='black')
plt.scatter(p2[:,0], p2[:,1], c='white')
plt.title("sigma = 0.01")
plt.show()
```



(c) How would you modify the two methods if you add 60 more data points from a third class centered on $\mu_3 = (0.5, 0.5)^\top$ with variance $\tilde{\sigma}^2 = 0.05$ and want to classify all data points into 3 groups?

To change the kNN-classifier to classify into three groups the following changes need to be made:

- The classifier would need to use real classes. With two classes a numerical representation is sufficient
- The training set would vote for their classes and the class with the most votes is chosen
- A problem would be, that with 3 classes there is the possibility of ambiguous votes

To change the kNN-classifier to classify into three groups the following changes need to be made:

- This classifier would just need to get a counter and a request for the third class

6.3 Radial Basis Functions (RBF) Network (4 points)

Similar to the Parzen window, RBF networks classify data according to a weighted vote but the voting committee now consists of $k < P$ "representatives", which parametrize the RBFs. These representatives do not have to be previously seen data points and can be "prototypes" derived from the training data via k-means clustering. Construct a RBF-net as follows:

-1- Determine the k representatives via *k-means* clustering (you can implement the online-algorithm described in the lecture notes or use available packages)

In [13]:

```
from sklearn.cluster import KMeans
from scipy import *
from scipy.linalg import norm, pinv
np.random.seed(42)
```

In [14]:

```
def k_repr(X, n):
    model = KMeans(init='k-means++',n_clusters=n)
    model.fit(X)
    return model.cluster_centers_
```

-2- For a given weight vector $\boldsymbol{w}$, the predicted classification for a query point $\boldsymbol{q}$ is:

$$\boldsymbol{y}(\boldsymbol{q}) = \text{sign}(\boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{q})),$$

where $\boldsymbol{\phi}$ is a $(k+1) \times 1$ vector containing the bias and the basis function values $\phi_i(\boldsymbol{q})$.

In [15]:

```
def _phi(x_a, mu_j, sigma):
    return exp(-norm(x_a - mu_j)**2/(2*sigma**2))

def _y(w, q, c, sigma):
    return np.sign(w.T.dot(_phi(q, c, sigma)))
```

-3- Determine the weight vector as: $\boldsymbol{w} = \left(\boldsymbol{\Phi}^\top \boldsymbol{\Phi}\right)^{-1} \boldsymbol{\Phi}^\top \boldsymbol{t} \equiv \boldsymbol{\Phi}^\dagger \boldsymbol{t}$ where $\boldsymbol{t}$ is the vector of target values and $\boldsymbol{\Phi}$ is the $P \times k$ *design matrix* with

$$\Phi_{\alpha j} := \phi_j(\boldsymbol{x}_\alpha) \equiv \exp\left(-\frac{||\boldsymbol{x}_\alpha - \boldsymbol{\mu}_j||^2}{2\sigma_\phi^2}\right), \qquad j = 1, \ldots, k; \quad \alpha = 1, \ldots, P$$

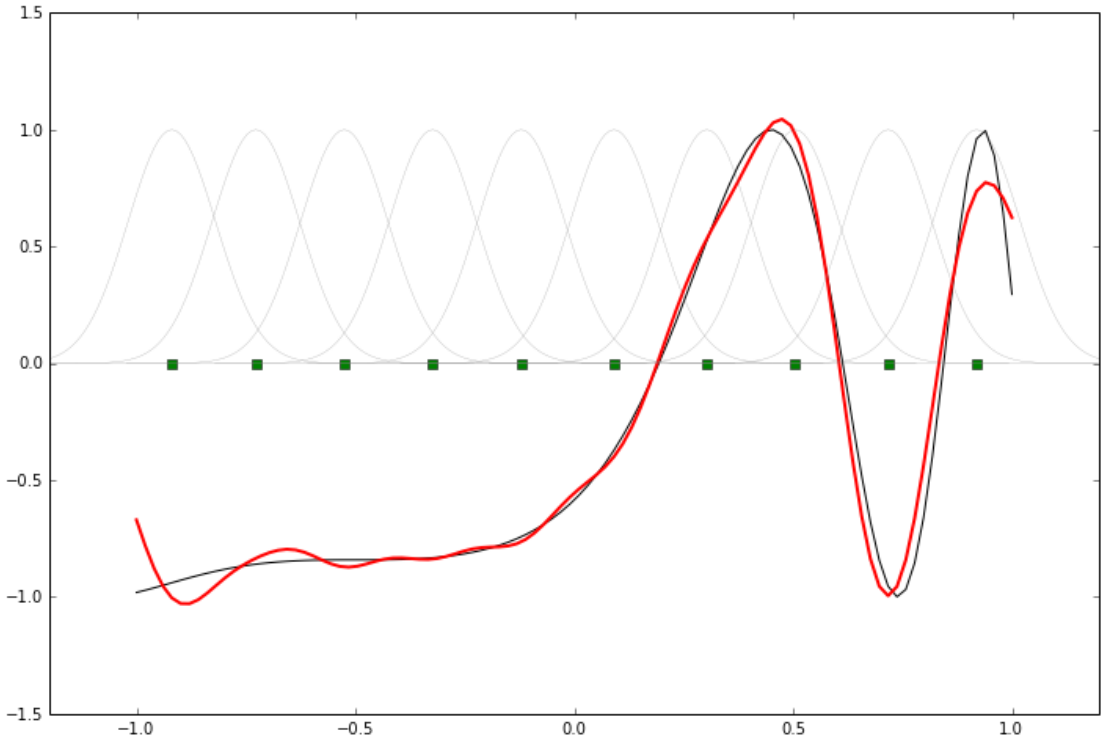
where $||\boldsymbol{v}||$ denotes the length (Euclidean Norm) of the vector $\boldsymbol{v}$. Please use predefined functions (e.g. `linalg.pinv` in Python or `pinv` in Matlab) to calculate the pseudo-inverse $\boldsymbol{\Phi}^\dagger$. To add a bias to the weighted sum, expand the design matrix by a column of ones.

In [16]:

```
class RBFNN:
    """
    construct a Radial Basis Function Network class
    """

    def __init__(self, _in, nc, _out, sigma):
        """
        _in: number of inputs
        nc: number of hidden nodes (centroids)
        _out: number of outputs
        sigma: used for rbf
        """
        self._in = _in
        self._out = _out
        self.nc = nc
        self.sigma = sigma
        self.w = random.random((self.nc, self._out))

    def _function(self, q, c):
        return _phi(q, c, self.sigma)

    def _activate(self, X):
        _Phi = zeros((X.shape[0], self.nc), float)
        for ci, c in enumerate(self.c):
            for xi, x in enumerate(X):
                _Phi[xi,ci] = self._function(x, c)
        return _Phi

    def train(self, X, Y, centers=None):
        """ X: matrix of dimensions n x indim
            y: column vector of dimension n x 1 """

        if centers:
            self.c = centers
        else:
            self.c = k_repr(X, self.nc)

        _Phi = self._activate(X)

        self.w = dot(pinv(_Phi), Y)

    def _y(self, X):
        """ X: matrix of dimensions n x indim """
        _Phi = self._activate(X)
        Y = dot(_Phi, self.w)
        return Y
```

In [17]:

```
######################
# TEST THE NETWORK CLASS
######################

n = 100
x = mgrid[-1:1:complex(0,n)].reshape(n, 1)
# set y and add random noise
y = sin(3*(x+0.5)**3 - 1)
# y += random.normal(0, 0.1, y.shape)

# rbf regression
rbf = RBFNN(1, 10, 1, 0.1)
rbf.train(x, y)
z = rbf._y(x)

# plot original data
plt.figure(figsize=(12, 8))
plt.plot(x, y, 'k-')

# plot learned model
plt.plot(x, z, 'r-', linewidth=2)

# plot rbfs
plt.plot(rbf.c, zeros(rbf.nc), 'gs')

for c in rbf.c:
    # RF prediction lines
    cx = arange(c-0.7, c+0.7, 0.01)
    cy = [rbf._function(array([cx_]), array([c])) for cx_ in cx]
    plt.plot(cx, cy, '-', color='gray', linewidth=0.2)

plt.xlim(-1.2, 1.2)
plt.show()
```



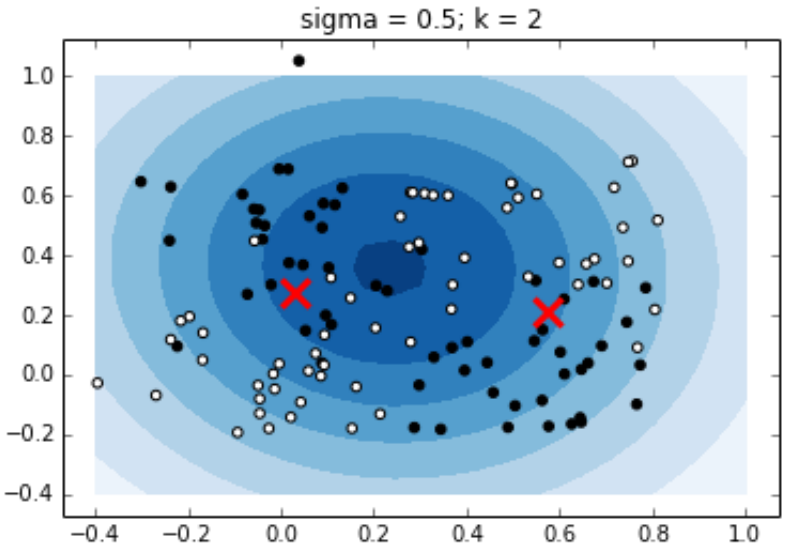
(a) Plot the decision boundaries together with the training patterns and locations of the representatives for $k \in \{2, 4\}$. Do this for two different (reasonable) widths σ_ϕ of the radial basis functions ϕ, yielding a total of four plots.

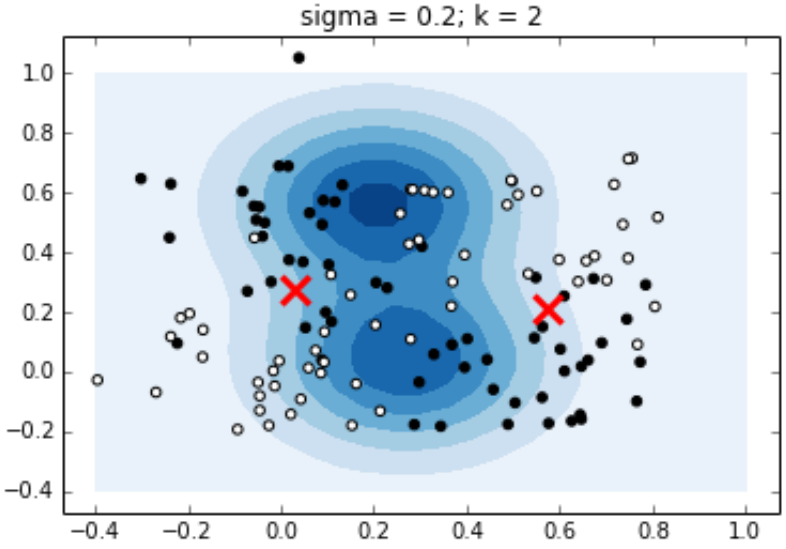
In [18]:

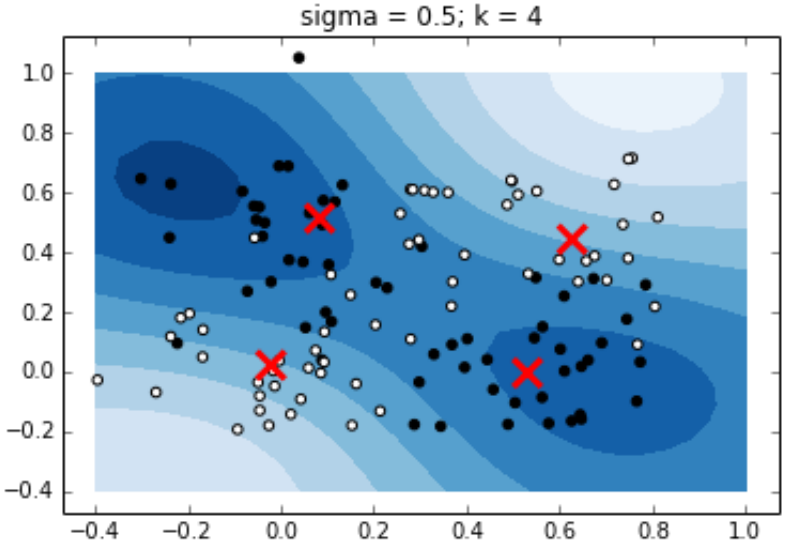
```
x_range = np.linspace(-0.4,1,25)
y_range = np.linspace(-0.4,1,25)

xx,yy = np.meshgrid(x_range,y_range)

def plot_rbf(rbf):
    zz = np.zeros(xx.shape)
    for x in range(len(x_range)):
        for y in range(len(y_range)):
            zz[x,y] = rbf._y(np.array([[xx[0,x],yy[y,0]]]))

    plt.contourf(xx, yy, zz, cmap=plt.cm.Blues)#levels=[0.,0.5,1.])
    plt.scatter(p1[:,0], p1[:,1], c='black')
    plt.scatter(p2[:,0], p2[:,1], c='white')
    plt.scatter(rbf.c[:, 0], rbf.c[:, 1],
            marker='x', s=169, linewidths=3,
            color='r', zorder=10)
    plt.title("sigma = "+str(rbf.sigma)+"; k = "+str(rbf.nc))
    plt.show()
```

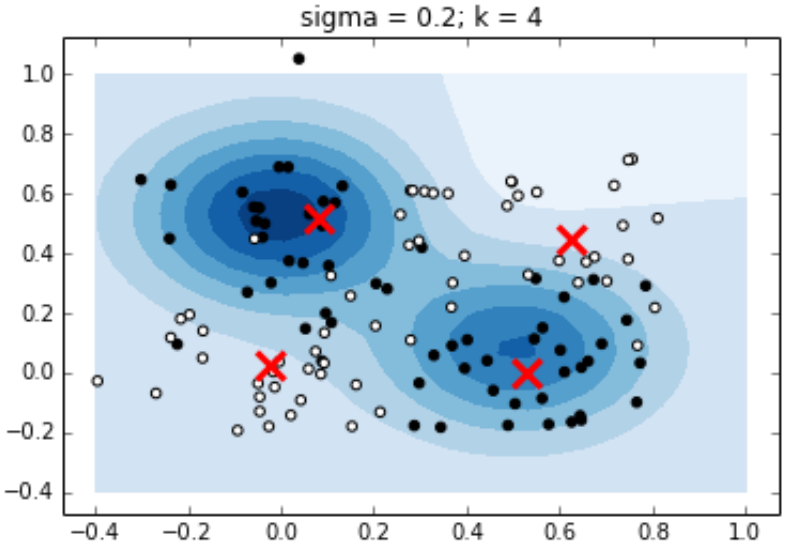
In [19]:

```
ks = [2,4]
sigmas = [0.5,0.2]

for i,k in enumerate(ks):
    for j,s in enumerate(sigmas):
        rbf = RBFNN(2, k, 1, s)
        rbf.train(trndata[:,0:2], trndata[:,2])
        plot_rbf(rbf)
```







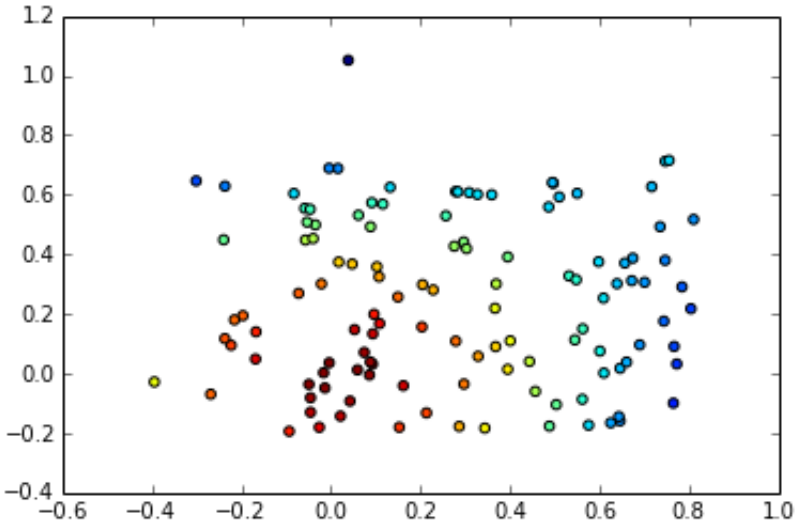


(b) Construct a RBF-network with 2 RBFs and set the centers to $\mu_1 = (0, 0)$ and $\mu_2 = (1, 1)$. For $\sigma_\phi = 0.45$, make a scatter plot of the data in the space of RBF-activations, i.e. for each data point α plot $\phi_1(x_\alpha)$ vs. $\phi_2(x_\alpha)$ and indicate their class-assignment via their color. Feel free to reduce the data-variance σ (e.g. to 0.2) to make the cluster-structure more prominent.

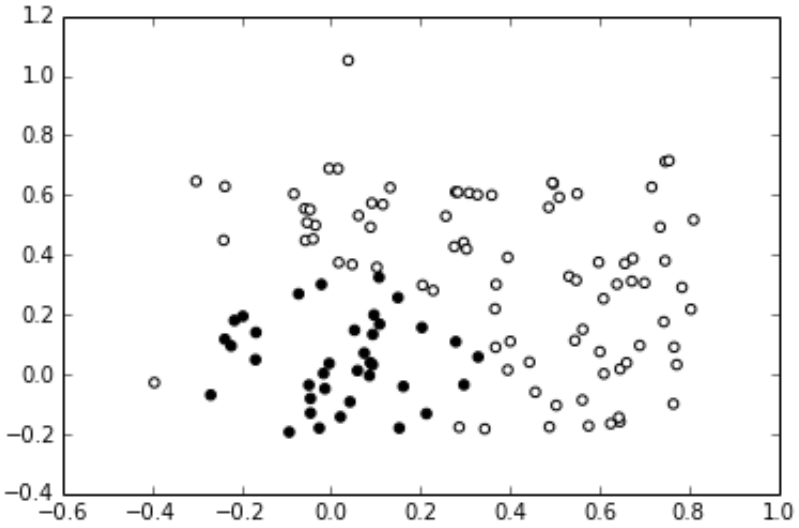
In [20]:

```
centers = [[0,0],[1,1]]
s = 0.45
k = 2

rbf = RBFNN(2, k, 1, s)
rbf.train(trndata[:,0:2], trndata[:,2], centers=centers)

y = []
for p in trndata[:,0:2]:
    y.append(rbf._y(np.atleast_2d(p)))
y = np.array(y)

print 'colour shading determines the propability of belonging to class red
or blue'
plt.scatter(trndata[:,0], trndata[:,1], c=y)
plt.show()

print 'classes via decision boundry'
plt.scatter(trndata[:,0], trndata[:,1], c=y<0.5, cmap='gray')
plt.show()
```

colour shading determines the propability of belonging to cla
ss red or blue



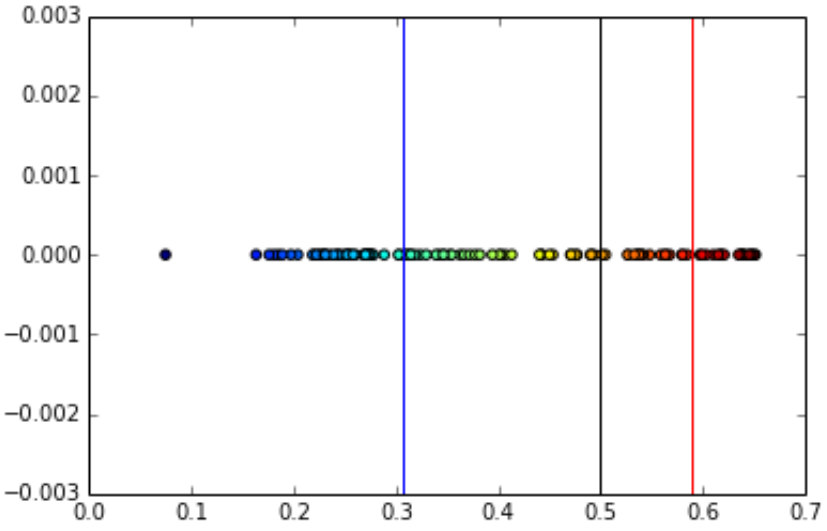
classes via decision boundry



(c) Plot the predictions $\tilde{y} = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x})$ on a line with color coding the class assignment. How would a similar plot look for the k=4 network from task (a) or when trying to predict with a linear classifier from the original features x_1, x_2?

In [35]:

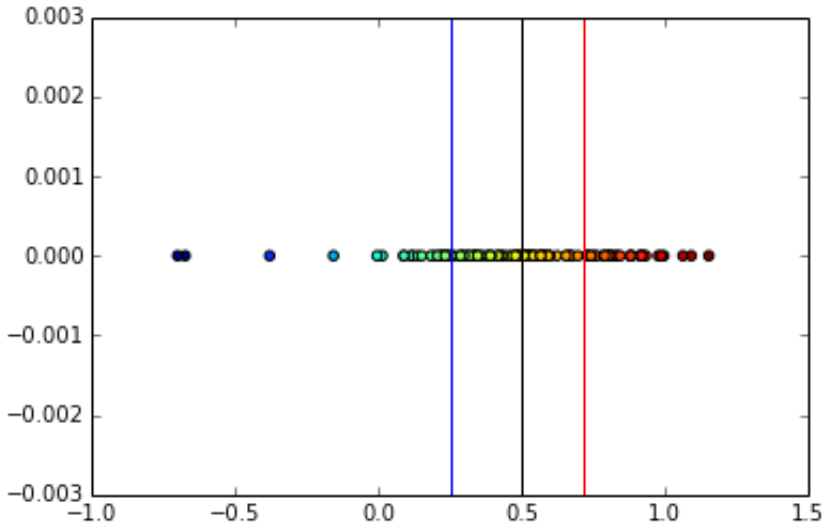
```
centers = [[0,0],[1,1]]
s = 0.45
k = 2

rbf = RBFNN(2, k, 1, s)
rbf.train(trndata[:,0:2], trndata[:,2], centers=centers)

y = []
for p in trndata[:,0:2]:
    y.append(rbf._y(np.atleast_2d(p)))
y = np.array(y)

plt.scatter(y, np.zeros(y.shape), c=y)
plt.axvline(x=0.5, c='black')
plt.axvline(x=y[y<0.5].mean(), c='blue')
plt.axvline(x=y[y>0.5].mean(), c='red')
plt.show()
```



colour shading determines the propability of belonging to class red or blue. the vertical lines (red,blue) show the class means (not centroid positions). the black line shows the decision boundry.

In [34]:

```
centers = [[0,0],[1,1],[1,0],[0,1]]
s = 0.45
k = 4

rbf = RBFNN(2, k, 1, s)
rbf.train(trndata[:,0:2], trndata[:,2], centers=centers)

y = []
for p in trndata[:,0:2]:
    y.append(rbf._y(np.atleast_2d(p)))
y = np.array(y)

plt.scatter(y, np.zeros(y.shape), c=y)
plt.axvline(x=0.5, c='black')
plt.axvline(x=y[y<0.5].mean(), c='blue')
plt.axvline(x=y[y>0.5].mean(), c='red')

plt.show()
```



As above, but using 4 supervised centroids. It can be seen, that the datapoints are not as clearly classified as with the $k = 2$ example. Hence there are more centroids, the distance of a point to its neares centroid is smaller than in the previous plot. Therefor, this result makes perfectly sense. However, note that if the original features are evaluated against these results, the $k = 4$ example will probably lead to more precise results.

If a linear classifier learned on the results of the previous plots and applied to the original results will most probably return bad results and vice versa. this is simply due to the fact that the original features are representing a XOR space and the previous results are not.

6.4 Kullback-Leibler Distance (2 points)

Jensen's inequality states that if f is a convex function and X is a random variable, then

$$\mathbb{E}[f(X)] \geq f(\mathbb{E}[\mathbb{X}]). \tag{1}$$

If f is strictly convex, then equality in eq(1) implies that $X = \mathbb{E}[X]$ with probability 1, i.e. X is a constant.

The *relative entropy* or *Kullback-Leibler distance* between two probability mass functions $p(x)$ and $q(x), x \in \mathcal{X}$ is defined as

$$D_{\mathrm{KL}}(p||q) := \sum_{x\in\mathcal{X}} p(x) \log \frac{p(x)}{q(x)} = \mathbb{E}_p \log \frac{p(X)}{q(X)} \tag{2}$$

Use Jensen's inequality to show that $D(p||q) \geq 0$ with equality if and only if $p(x) = q(x) \forall x$.

with $f(g) = log(g)$ and $g = \frac{p(x)}{q(x)}$ (1)
and $log\frac{a}{b} = -log\frac{b}{a}$ because of $log\frac{a}{b} - log\frac{b}{a} = 0$ (2)

$$-D_{KL}(p||q) \overset{(2)}{=} \sum_{x\in X} p(x) log \frac{q(x)}{p(x)}$$

and therefor

$$-D_{KL}(p||q) = \sum_{x\in X} p(x) log \frac{q(x)}{p(x)} \overset{\text{Jensen with (1)}}{\leq} log \sum_{x\in X} p(x) \frac{q(x)}{p(x)} = log \sum_{x\in X} q(x) = 0$$

So we see that $D_{KL}(p||q) \geq 0$

also

$$\sum_{x\in X} p(x) log \frac{q(x)}{p(x)}$$

with $q(x) = p(x)$ becomes

$$\sum_{x\in X} p(x) log(1) = \sum_{x\in X} p(x) \cdot 0 = 0$$