1 Linear kernel

```
def linear_kernel(x, y):
    return np.dot(x, y) + 1
```

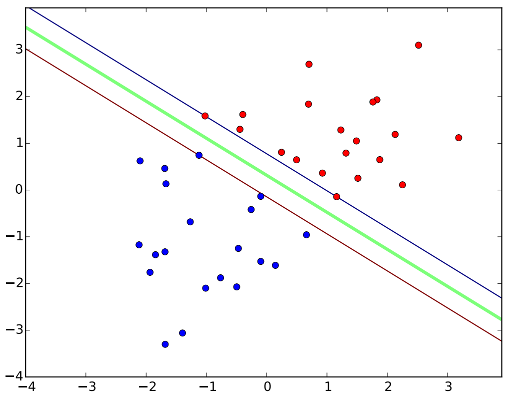

Figure 1: Linear kernel

2 Radial basis kernel

```
def radial_basis_kernel(x, y, sigma=1):
    return np.exp(-np.sum(np.power((x-y), 2))/(2*sigma**2))
```

Larger sigmas seem to make the boundary more “stiff“ and less prone of over fitting.

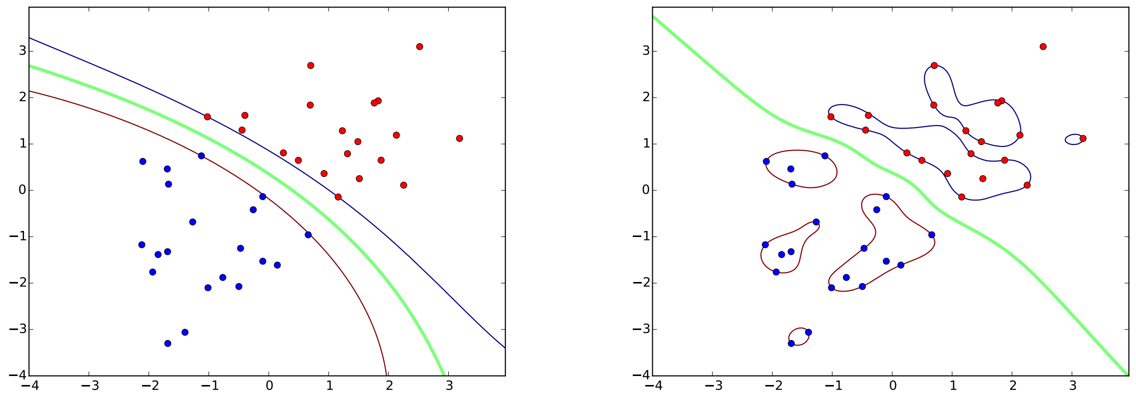

Figure 2: Left: $\sigma = 4$, right $\sigma = 0.5$

An example where the boundary gets really twisted (figure 3).

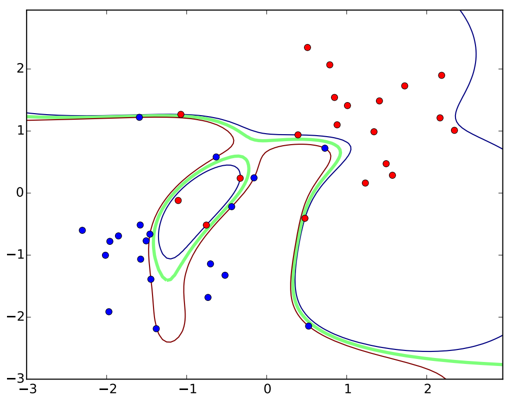

Figure 3: Radial basis with a really twisted boundary

3 Polynomial kernels

```
def polynomial_kernel(x, y, p):
    return np.power(linear_kernel(x, y), p)
```

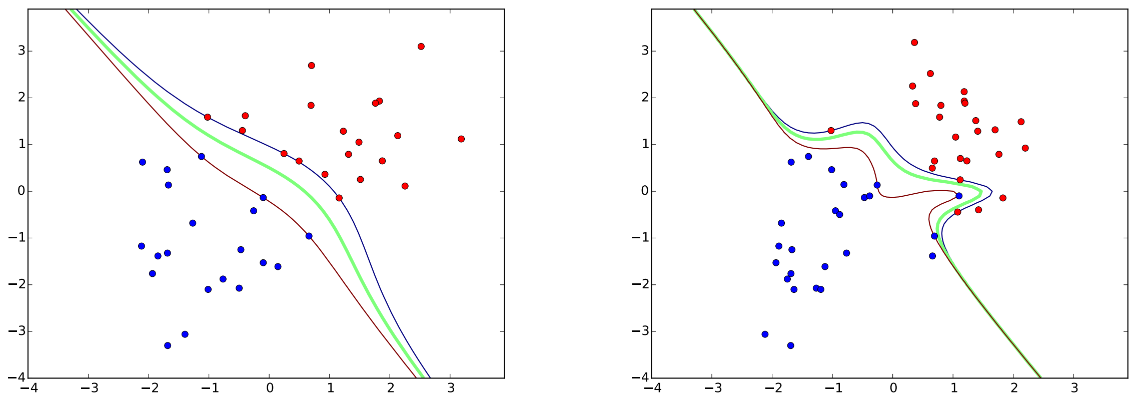

Figure 4: Polynomial kernels of degree 3 and 4

4 Code

```
from cvxopt.solvers import qp
from cvxopt.base import matrix, spmatrix
from itertools import combinations_with_replacement, product
import numpy as np
import pylab as pl
```

```
# Kernels
def linear_kernel(x, y):
    return np.dot(x, y) + 1

def polynomial_kernel(x, y, p):
    return np.power(linear_kernel(x, y), p)

def radial_basis_kernel(x, y, sigma=1):
    return np.exp(-np.sum(np.power((x-y), 2))/(2*sigma**2))

def sigmoid_kernel(x, y, k, delta):
    return np.tanh(k*np.dot(x, y) - delta)

def build_P_matrix(xs, ts, K):
    """
    :param xs: 2d array where each row is a data point [x, y]
    :param ts: labels corresponding to each row of xs
    :param kernel: handle to a kernel function which only takes two vectors as arguments
                   (rewrite as lambda function if necessary)
    :returns: the matrix t_i t_j K(x_i, x_j)
    """
    points = list(zip(xs, ts))
    n = len(points)
    # List comprehension to get all combinations and apply t_i*t_j*K(x_i, x_j)
    P_ls = np.array([ a[1]*b[1]*K(a[0], b[0]) for a, b in product(points, repeat=2)])
    return matrix(P_ls.reshape((n,n)))

def qp_args(P):
    n = np.sqrt(len(P))
    G = matrix(-np.eye(n))
    q = matrix(-np.ones(n))
    h = matrix(np.zeros(n))

    return q, G, h

def indicator(x, xs, labels, alphas, K):
    res = 0
    for i in range(0, len(alphas)):
        #               label
        res += alphas[i]*labels[i]*K(xs[i,:], x)

    return res

# Set up training data
# The total amount of data points
n = 50
np.random.seed(1)
kernel = lambda x,y: sigmoid_kernel(x, y, 1/2, 0)
pts1   = np.array([np.random.normal(-1,1,n//2), np.random.normal(-1,1,n//2)]).T
pts2   = np.array([np.random.normal(1,1,n//2), np.random.normal(1,1,n//2)]).T
pts    = np.vstack((pts1, pts2))
```

```
labels = [1] * (n//2) + [-1] * (n//2)

# Set up arguments for cvxopt
P = build_P_matrix(pts, labels, kernel)
(q, G, h) = qp_args(P)

# Optimize
r = qp(P, q, G, h)
alphas = r['x']
alphas = map(lambda x: (x>1e-5)*x, alphas)

# Calculate decision boundary for plotting
xr = np.arange(-4, 4, 0.1)
yr = np.arange(-4, 4, 0.1)
zvals  = [[indicator([x,y], pts, labels, r['x'], kernel) for y in yr] for x in xr]

# Plot
pl.figure()
pl.plot(pts1[:,1], pts1[:,0], 'bo')
pl.plot(pts2[:,1], pts2[:,0], 'ro')
pl.contour(xr, yr, zvals, (-1.0, 0.0, 1.0), linewidths=(1,3,1))
pl.savefig("polynomial_5.pdf")
pl.show()
```