Support Vector Machines, part 2

Ludwig Krippahl

SVM-2

Summary

- Soft Margins
- Kernel Trick
- Regularization in SVM
- (Assignment 1)

Soft Margins

Soft Margins

- A linear SVM works if the classes are linearly separable:

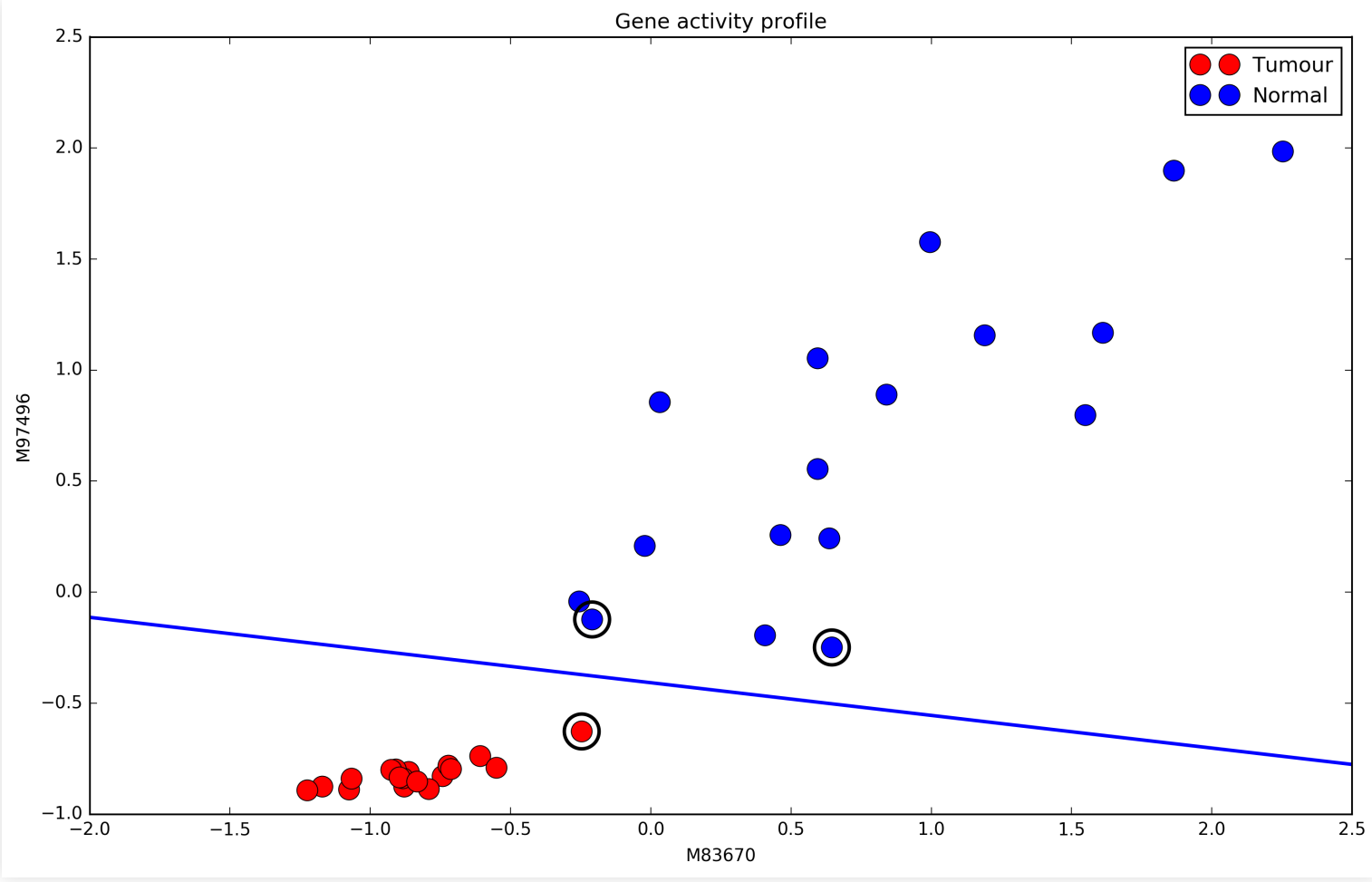
$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \vec{x}_n^T \vec{x}_m$$

  - subject to

$$\sum_{n=1}^{N} \alpha_n y_n = 0 \qquad \alpha_n \geq 0$$

Soft Margins

- A linear SVM works if the classes are linearly separable



Soft Margins

- With this constraint, will not work if the classes are not linearly separable

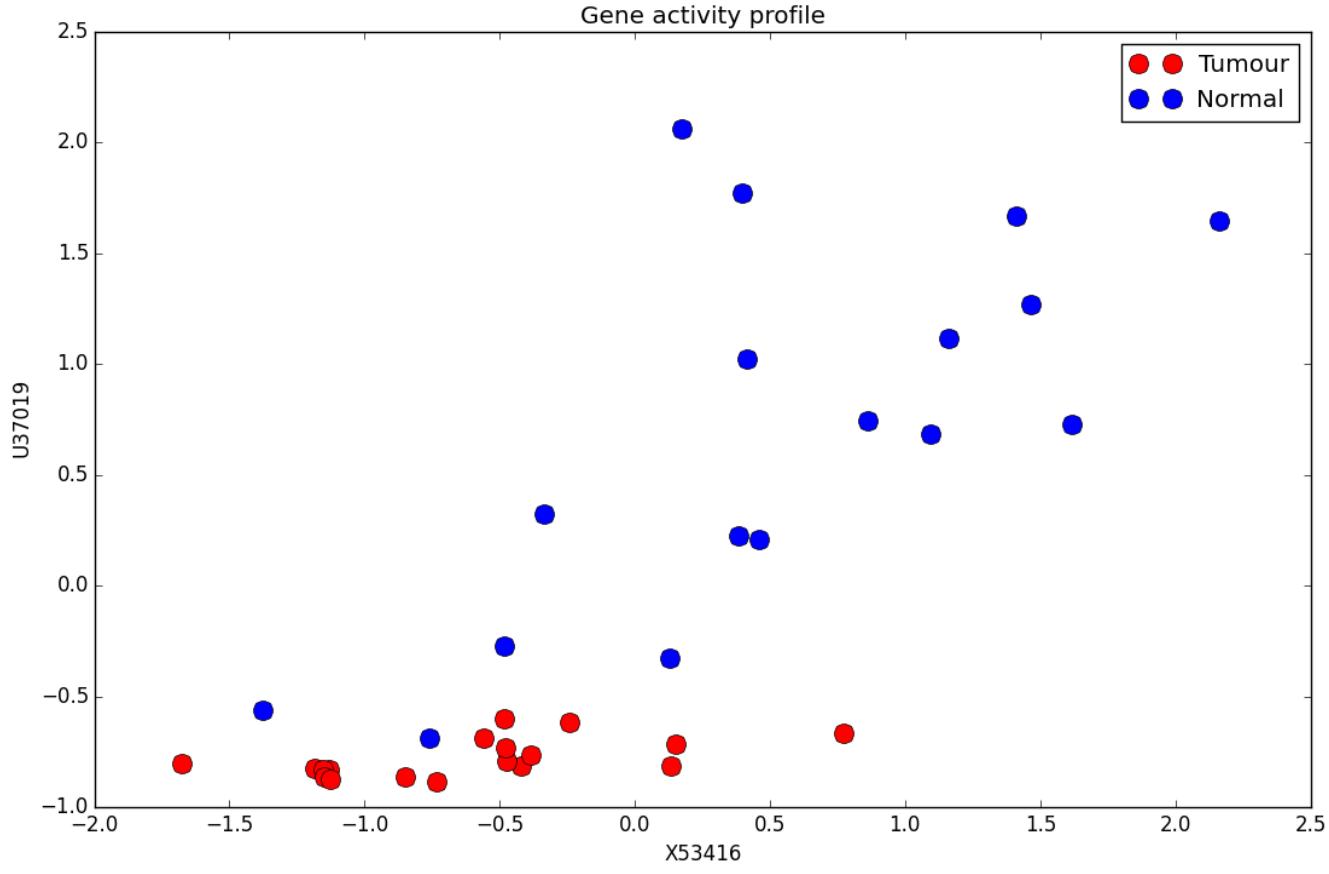
$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \vec{x}_n^T \vec{x}_m$$

$$\sum_{n=1}^{N} \alpha_n y_n = 0 \qquad \alpha_n \geq 0$$

- If vectors are surrounded by vectors of the other class, the α values can increase to infinity and there is no maximum for the function.

Soft Margins

- With this constraint, will not work if the classes are not linearly separable



Soft Margins

- Instead of forcing margin to be at least 1

$$y_n(\vec{w}^T\vec{x}_n + w_0) \geq 1, \forall n \in N$$

- Allow a slack ξ_n for each vector $\vec{x}_n$

$$y_n(\vec{w}^T\vec{x}_n + wo) \geq 1 - \xi_n, \forall n \in N$$

- With $\xi \geq 0$, as it is a distance to the margin
- if $0 < \xi < 1$, then point $\vec{x}$ is inside the margin
- if $\xi > 1$, then point $\vec{x}$ is misclassified
- We want to maximize the margin, minimizing $||\vec{w}||^2$, but also penalizing deviations from the margin

$$C\sum_{n=1}^{N}\xi_n + \frac{1}{2}||\vec{w}||^2$$

Soft Margins

- New Lagrangian:

$$\mathcal{L}(\vec{w}, b, \vec{\alpha}, \vec{\mu}, \vec{\xi}) =$$

$$\frac{1}{2}\|\vec{w}\|^2 + C\sum_{n=1}^{N}\xi_n - C\sum_{n=1}^{N}\alpha_n\left(y_n(\vec{w}^T\vec{x}+b) - 1 + \xi_n\right) - \sum_{n=1}^{N}\mu_n\xi_n$$

- Setting the derivatives to 0, same dual problem:

$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}\alpha_n\alpha_m y_n y_m \vec{x}_n^T\vec{x}_m$$

- But different constraints

$$\sum_{n=1}^{N}\alpha_n y_n = 0$$

$$\frac{\delta\mathcal{L}}{\delta\xi_n} = 0 \Leftrightarrow C - \alpha_n - \mu_n = 0 \Leftrightarrow 0 \leq \alpha_n \leq C$$

Soft Margins

$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \vec{x}_n^T \vec{x}_m$$

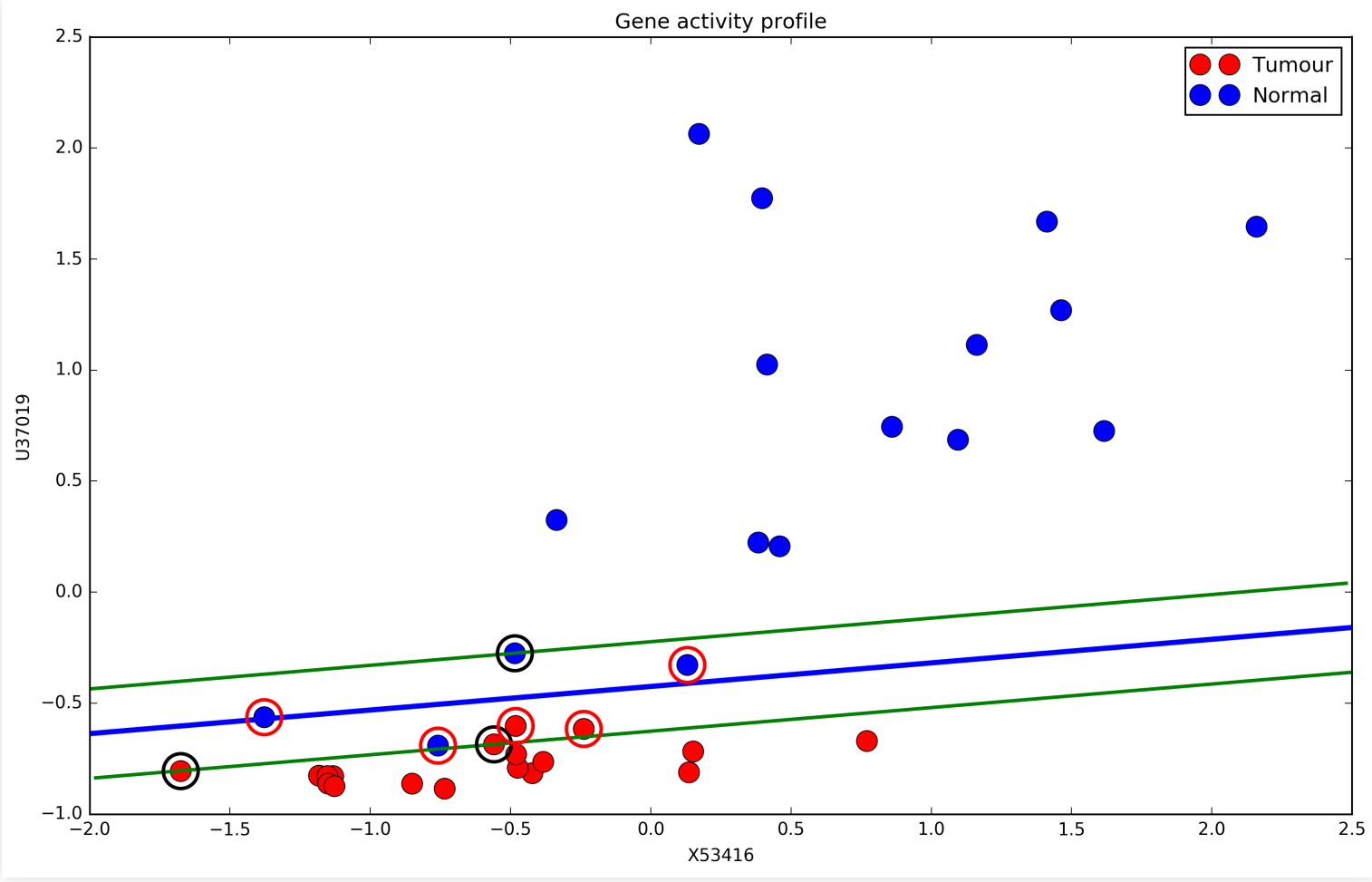
$$\sum_{n=1}^{N} \alpha_n y_n = 0 \qquad 0 \leq \alpha_n \leq C$$

- Intuitively: upper bound on penalization for margin

```
H = H_matrix(Xs,Ys)
A = Ys[:,0] # sum of alphas is zero
cons = {'type':'eq',
        'fun':lambda alphas: np.dot(A,alphas),
        'jac':lambda alphas: A}
bounds = [(0,C)]*Xs.shape[0] #alpha>=0
x0 = np.random.rand(Xs.shape[0])
sol =  minimize(loss, x0, jac=jac, constraints=cons,
                method='SLSQP', bounds = bounds)
```

Soft Margins

- Allow some margin violations or errors (C=10)



Soft Margins

- Allow some margin violations or errors (C=10)
- To compute $\vec{w}$ use all support vectors ($\alpha_n > 0$)
- For w_0, use only margin vectors ($0 < \alpha_n < C$)

- Constraint $y_n(\vec{w}^T\vec{x}_n + w_0) = 1$ not valid for vectors with $\alpha_n = C$

- So average only for $0 < \alpha_n < C$:

$$w_0 = y_n - \vec{w}^T\vec{x}_n$$

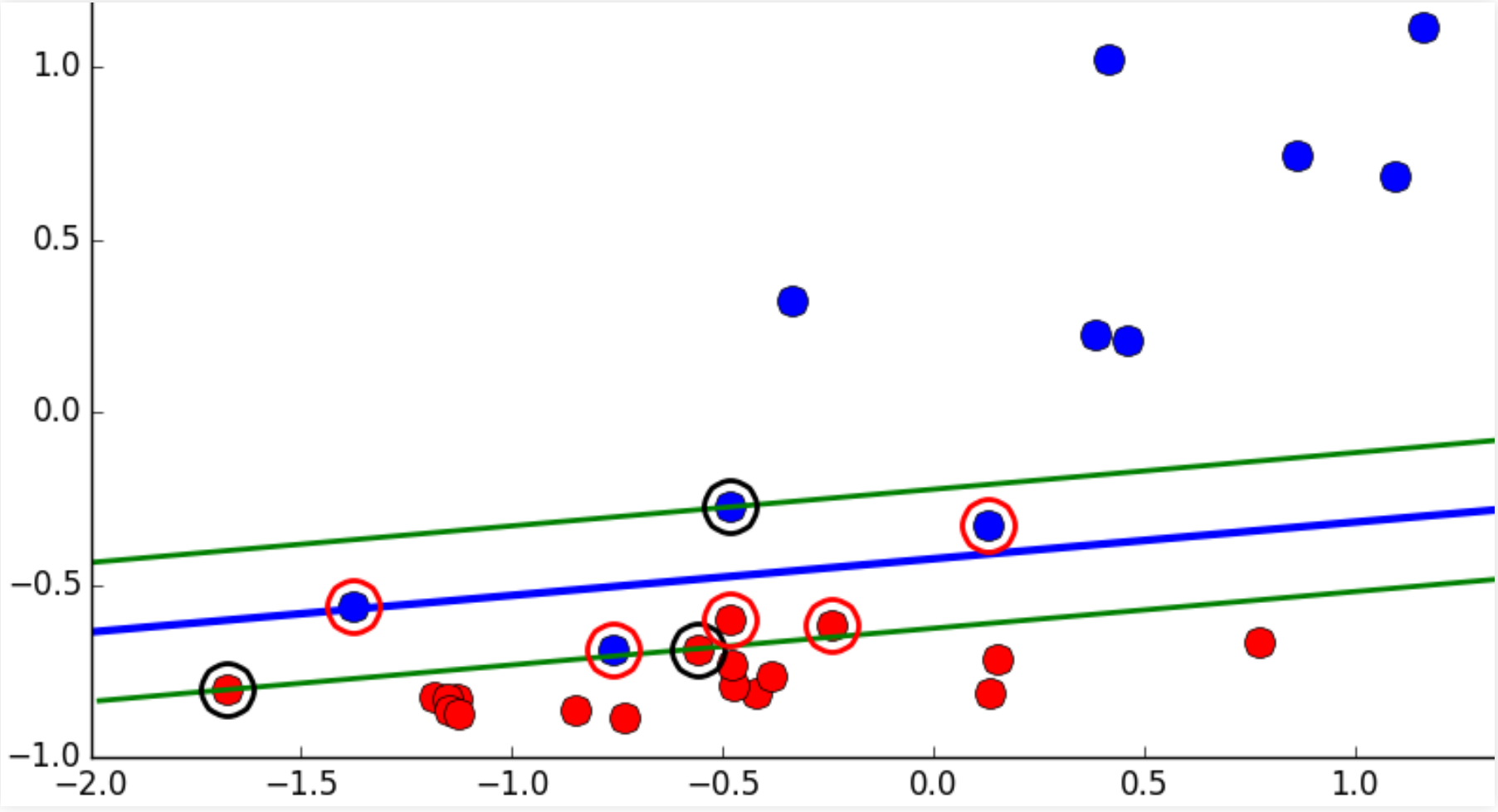
Soft Margins

- Inside margins, $\alpha_n = C$

```
[  1.00000000e+01  -2.89129433e-14   1.62820017e-14   1.17348602e-12
   2.02077358e-15   6.68162351e+00   1.00000000e+01   1.06304794e-12
   1.17694556e-12   7.83296084e-13  -2.99848509e-14   1.02238339e-12
   1.07337714e-14   5.85150761e+00   7.23253298e-13   5.11229227e-13
   9.29336289e-14  -1.36784400e-13  -1.45436364e-14   1.07235005e-13
   3.97298470e-15   1.00000000e+01  -2.91633573e-14   1.00000000e+01
  -6.93493037e-14   1.00000000e+01   1.66339014e-13  -1.23618374e-13
  -3.70160694e-16   4.45905935e-14  -1.87089595e-14   2.70387604e-13
  -2.75057753e-14   9.43679075e-14   2.53313112e+00  -1.08786818e-13]
```

Soft Margins

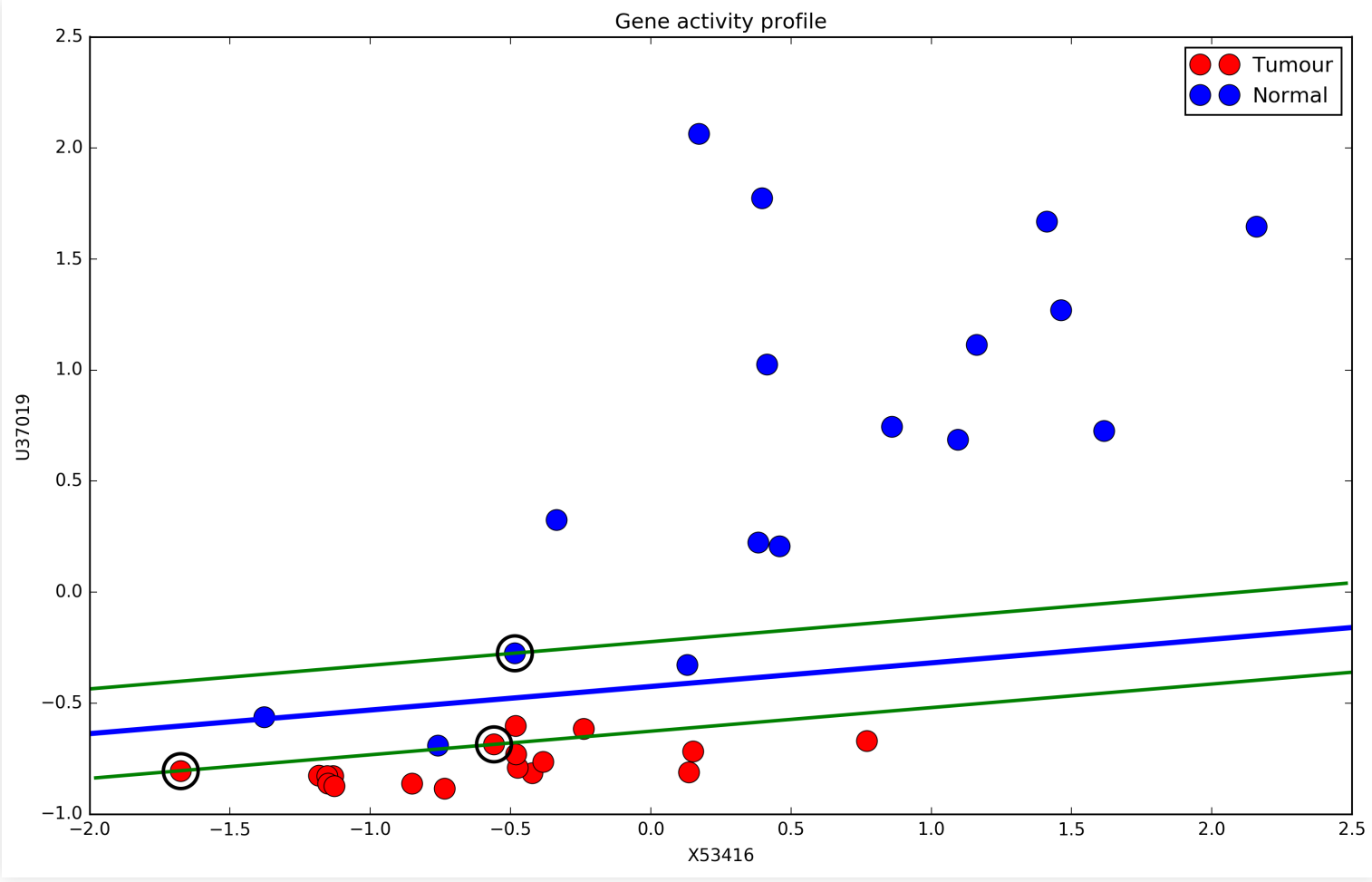
- Support vectors, including vectors inside margins (red)



Non-linear separation

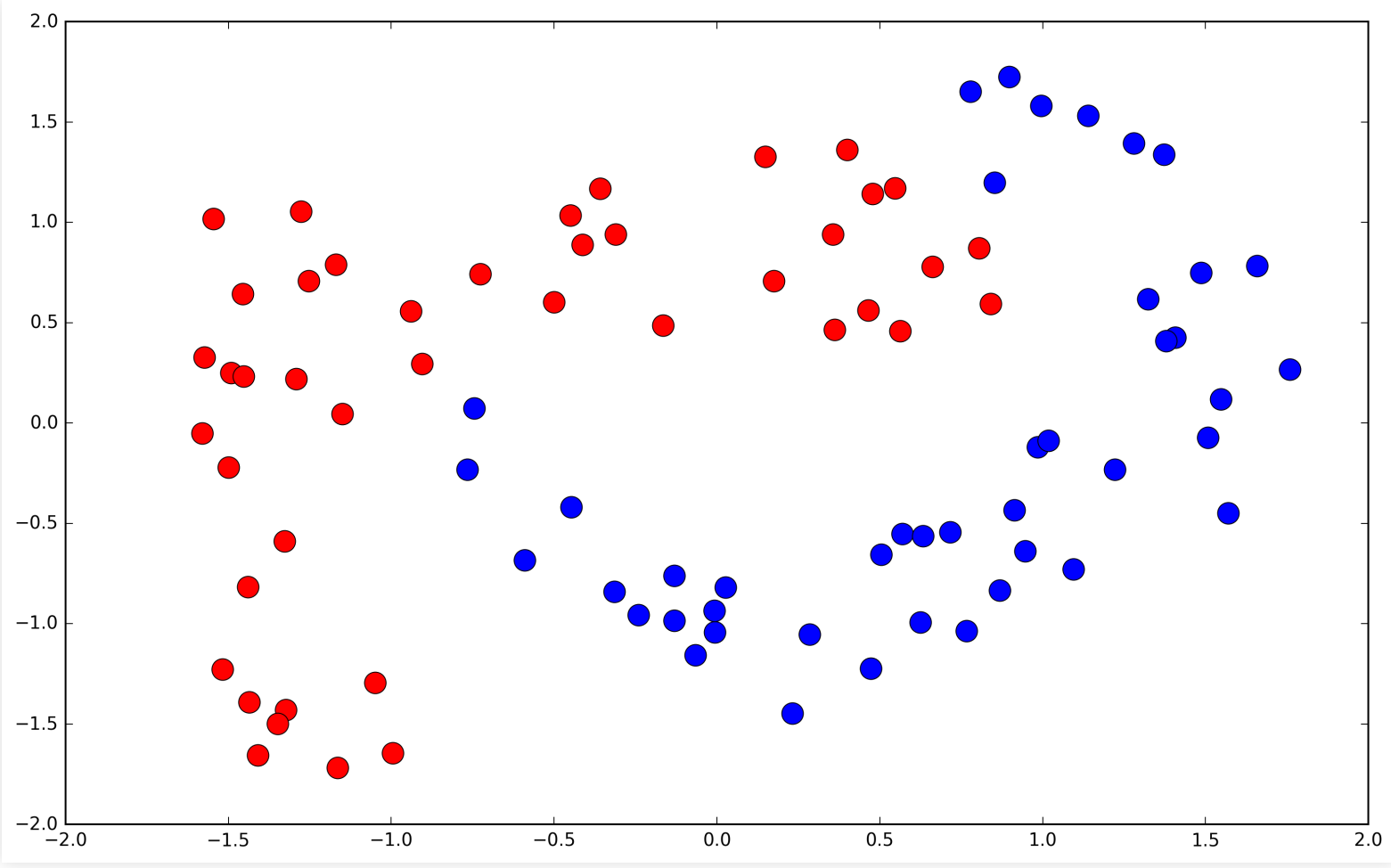
Non-linear separation

- Soft margins ok for slight overlap



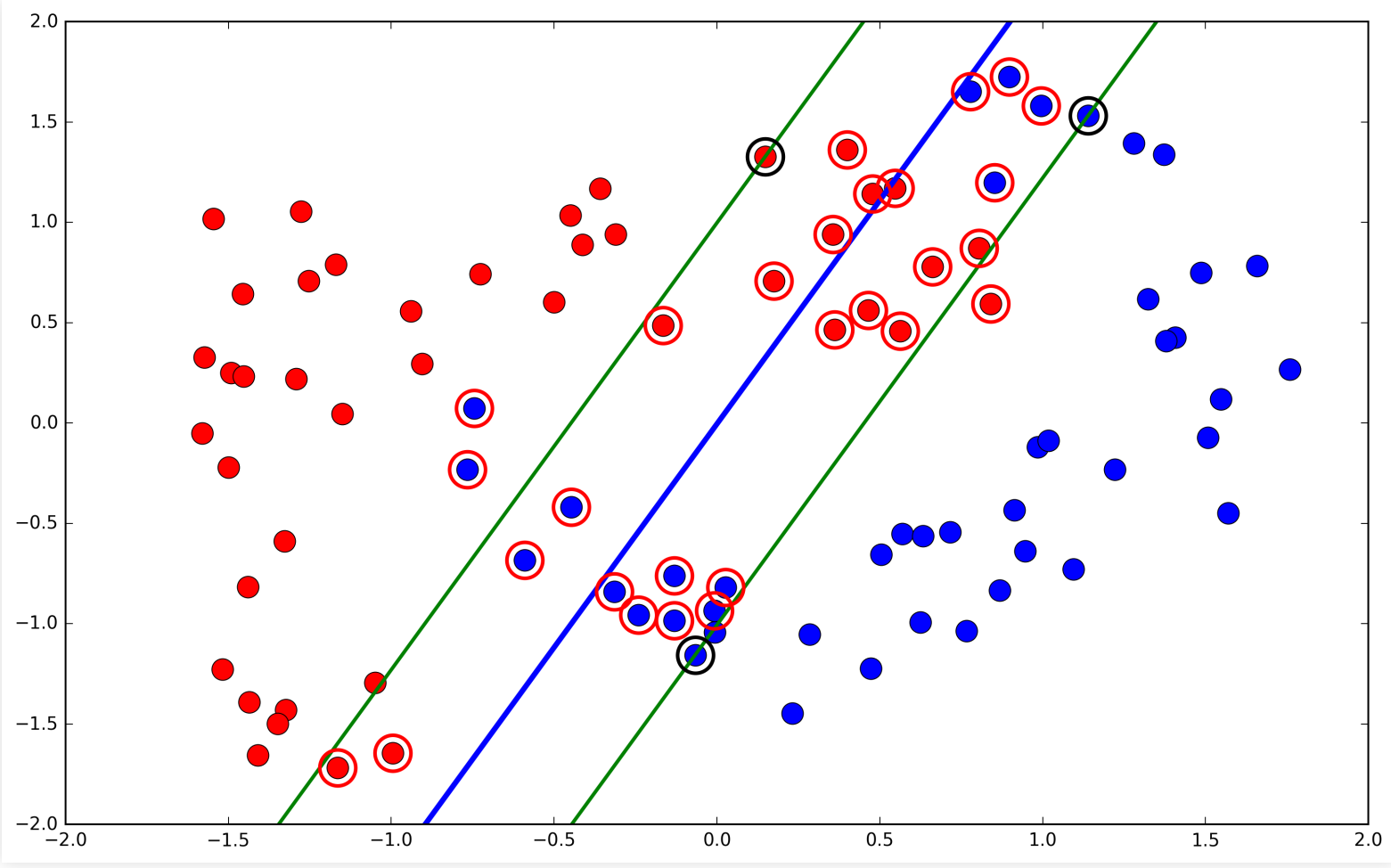
Non-linear separation

- But insufficient if frontier is too far from linear



Non-linear separation

- But insufficient if frontier is too far from linear



Non-linear separation

- As always, the trick is to do linear separation after a nonlinear transformation.
  - For example, expand the features polynomially
- In SVM this is easy to do with the Kernel Trick

Kernel Trick

Kernel trick

- With an SVM we only need the inner products of the examples:

$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \boxed{\vec{x}_n^T \vec{x}_m}$$

- (as we did with the H matrix).

The kernel trick

- A kernel function is a function of two vectors whose result is the inner product of some function of each vector:

$$K(\vec{u}, \vec{v}) = \phi(\vec{u})^T \phi(\vec{v})$$

- Useful if $K(\vec{u}, \vec{v})$ is easier to compute than $\phi(\vec{u})^T \phi(\vec{v})$
- For example $\phi^2(\vec{x}) = [1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1 x_2, x_1^2, x_2^2]^T$

$$\phi^2(\vec{u})^T \phi^2(\vec{v}) = (\vec{u}^T \vec{v} + 1)^2$$

- More generally, $K_{\phi^n}(\vec{u}, \vec{v}) = (\vec{u}^T \vec{v} + c)^n$

Kernel trick

- So now we solve:

$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m K(\vec{x}_n, \vec{x}_m)$$

- Where $K(\vec{x}_n, \vec{x}_m)$ is the kernel function for some non-linear expansion ϕ of our original data
- To classify new points, we do not compute $\vec{w}$ or w_0, because the hyperplane is in the space of $\phi(\vec{x})$
- Instead, we compute the class of $\overrightarrow{x_t}$ from the support vectors:

$$\vec{w}^T \phi(\overrightarrow{x_t}) + w_0 = \sum_{n=1}^{N} \alpha_n y_n K(\overrightarrow{x_n}, \overrightarrow{x_t})$$

Kernel trick

- Example, using a polynomial kernel of degree d:

$$K_{\phi(\vec{x}^d)}(\vec{x_1}, \vec{x_2}) = (\vec{x_1}^T \vec{x_2} + 1)^d$$

```
def H_poly(X,Y,d):
    H = np.zeros((X.shape[0],X.shape[0]))
    for row in range(X.shape[0]):
        for col in range(X.shape[0]):
            k = (np.dot(X[row,:],X[col,:])+1)**d
            H[row,col] = k*Y[row]*Y[col]
    return H
```

- Evaluating a point:

$$\vec{w}^T \phi(\vec{x_t}) + w_0 = \sum_{n=1}^{N} \alpha_n y_n K_{\phi(\vec{x}^d)}(\vec{x_n}, \vec{x_t})$$

```
def poly_k_class(X,alphas,Y,xt,n):
    s = 0
    for ix in range(len(alphas)):
        s = s + (np.dot(X[ix,:],xt)+1)**n*Y[ix]*alphas[ix]
    return s
```

Kernel trick

- Example, using a polynomial kernel:

$$K_{\phi(\vec{x}^n)} = (\vec{x}^T\vec{z} + 1)^n$$

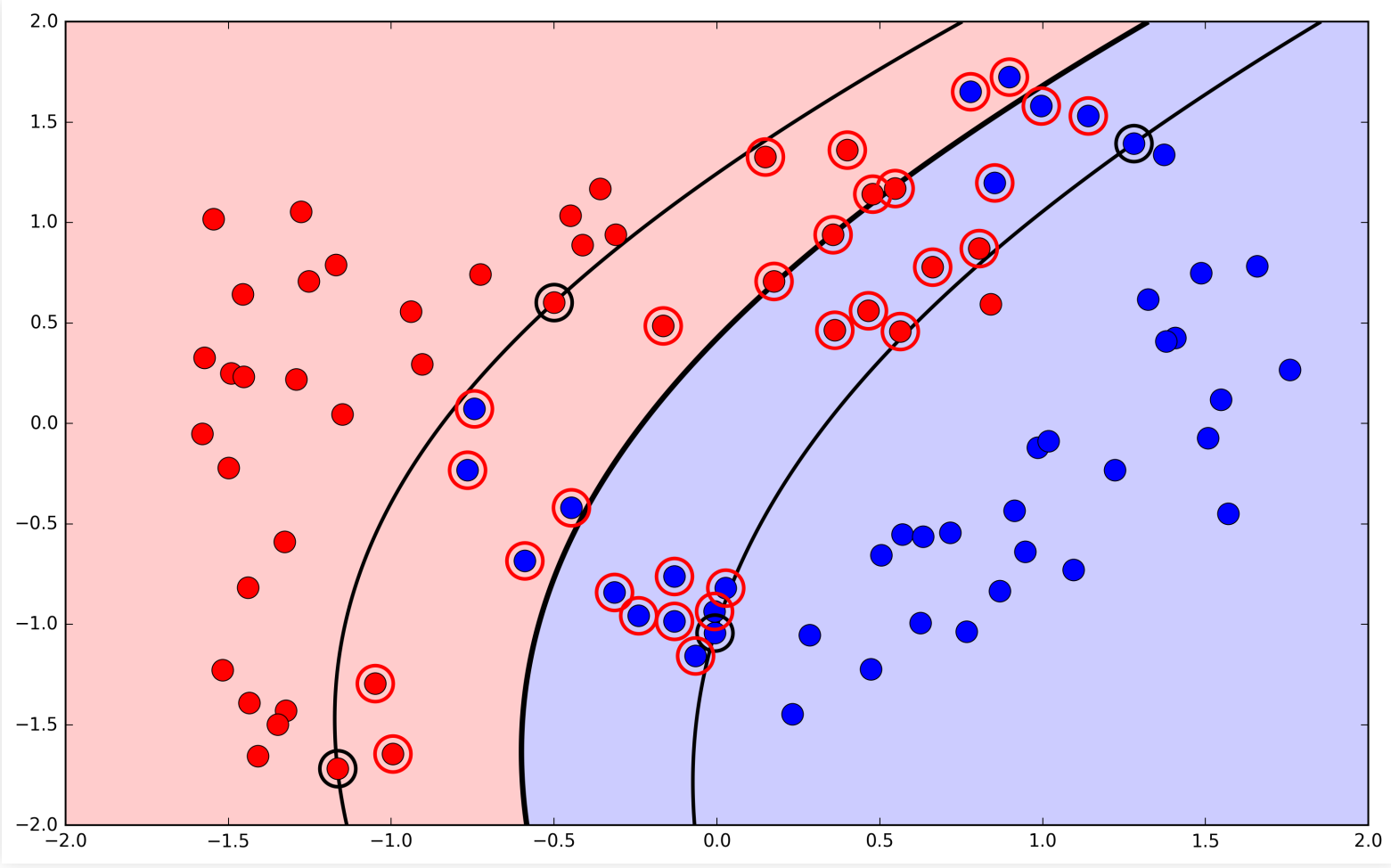
- We will not implement SVM; best to use the SciKit-learn SVC class

```
from sklearn import svm
#load and standardize data
sv = svm.SVC(C=C,kernel = 'poly', degree = poly, coef0 = 1)
sv.fit(Xs,Ys[:,0])
```

- Note: Implementation examples here are only to help explain the algorithm. It is best to use optimized implementations.

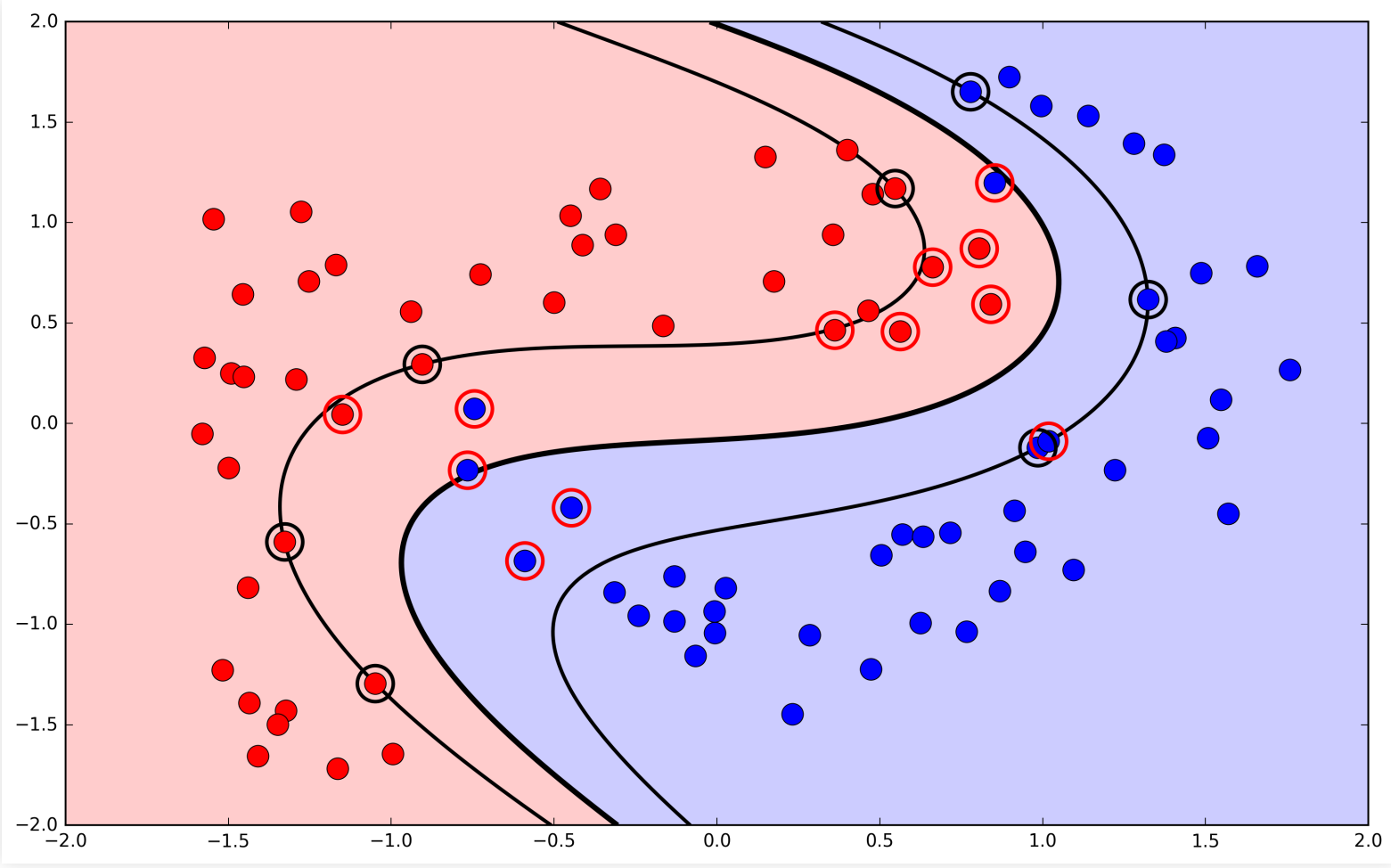
Kernel trick

- Polynomial of Degree 2, C = 1:



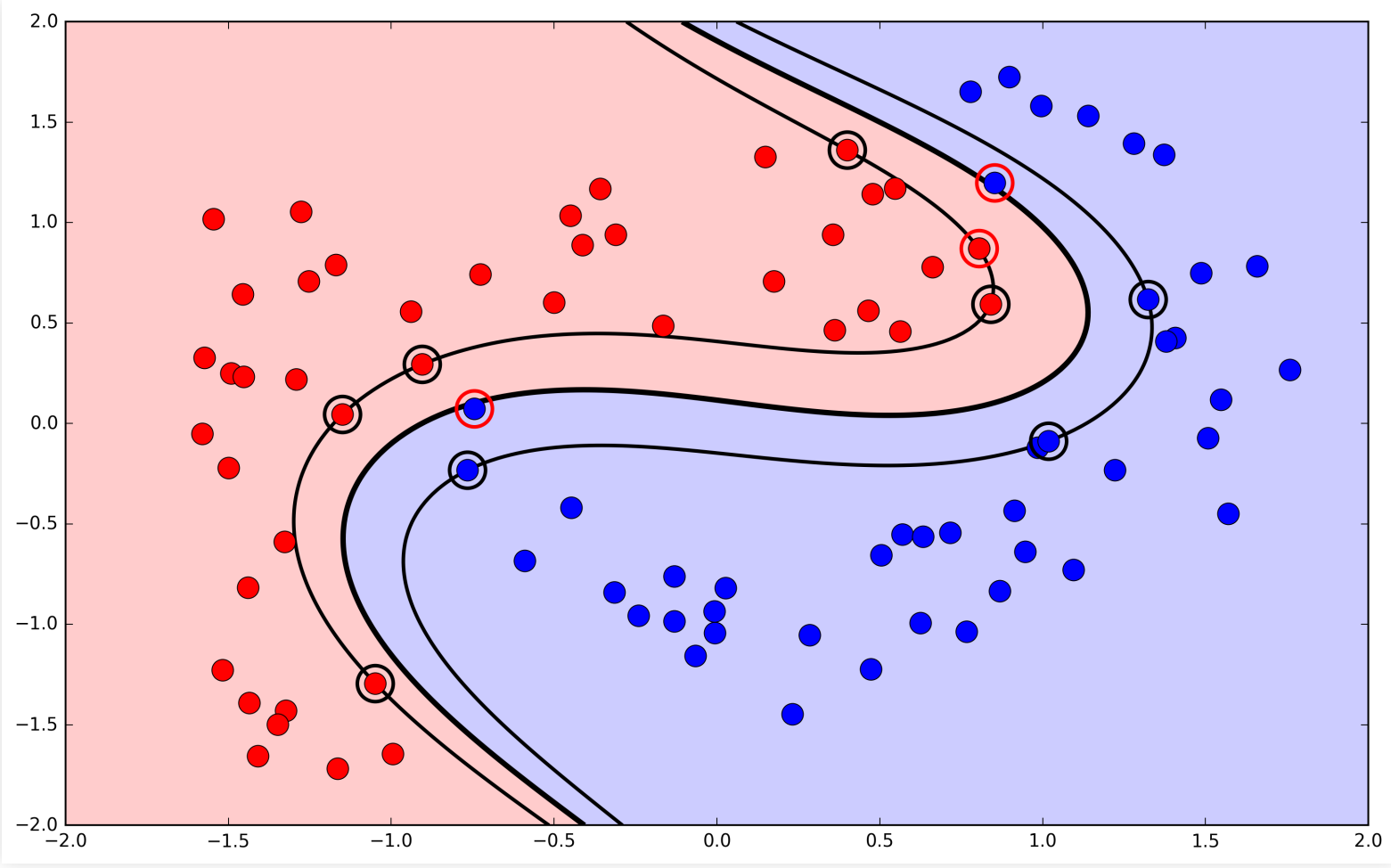
Kernel trick

- Polynomial of Degree 3, C = 1:



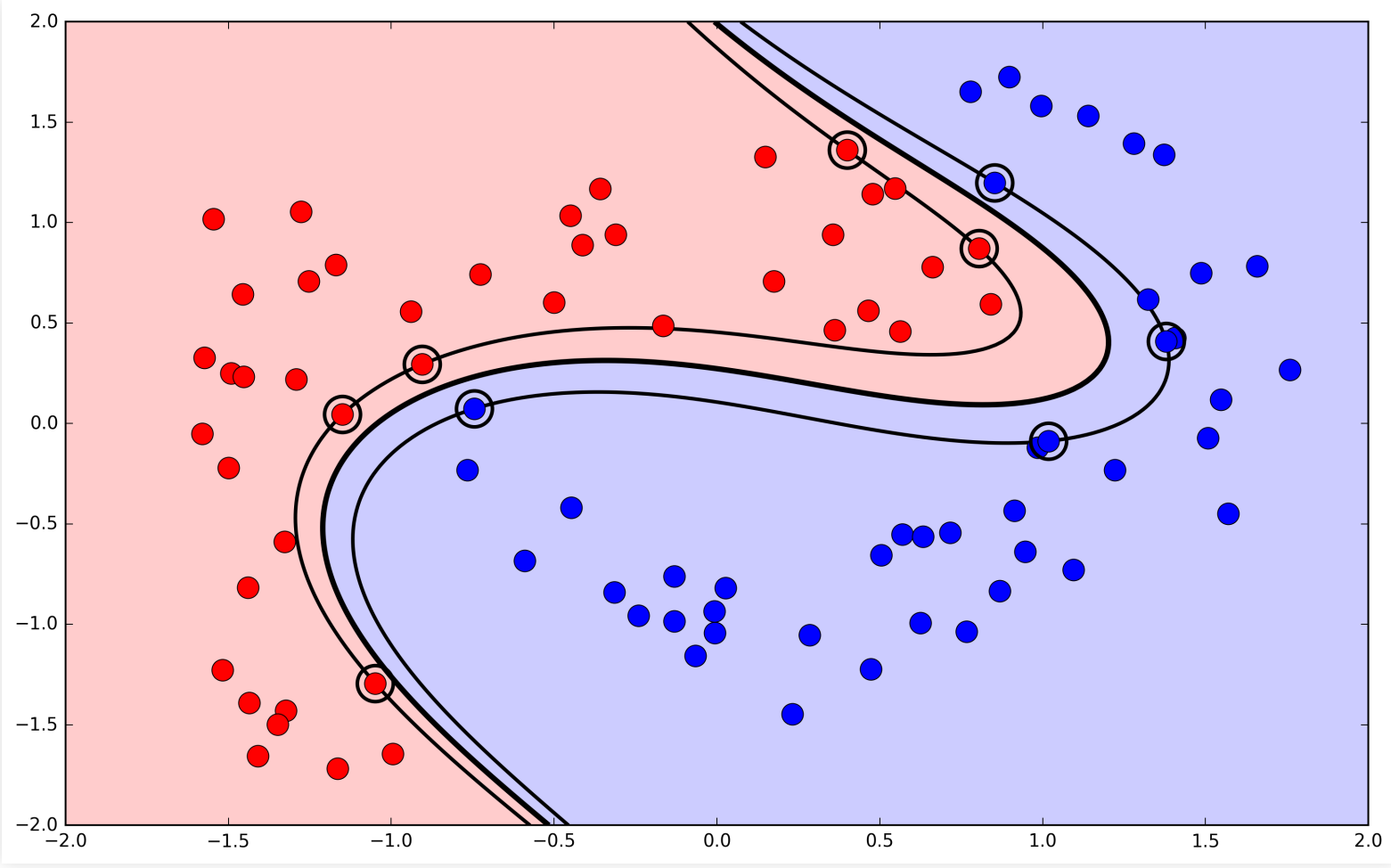
Kernel trick

- Polynomial of Degree 3, C = 10:



Kernel trick

- Polynomial of Degree 3, C = 1000 (higher C, less regularization):



Kernel trick

- Gaussian Radial Basis Function kernel:

$$K(\vec{x_1}, \vec{x_2}) = e^{\frac{-\|\vec{x_1} - \vec{x_2}\|^2}{2\sigma^2}}$$

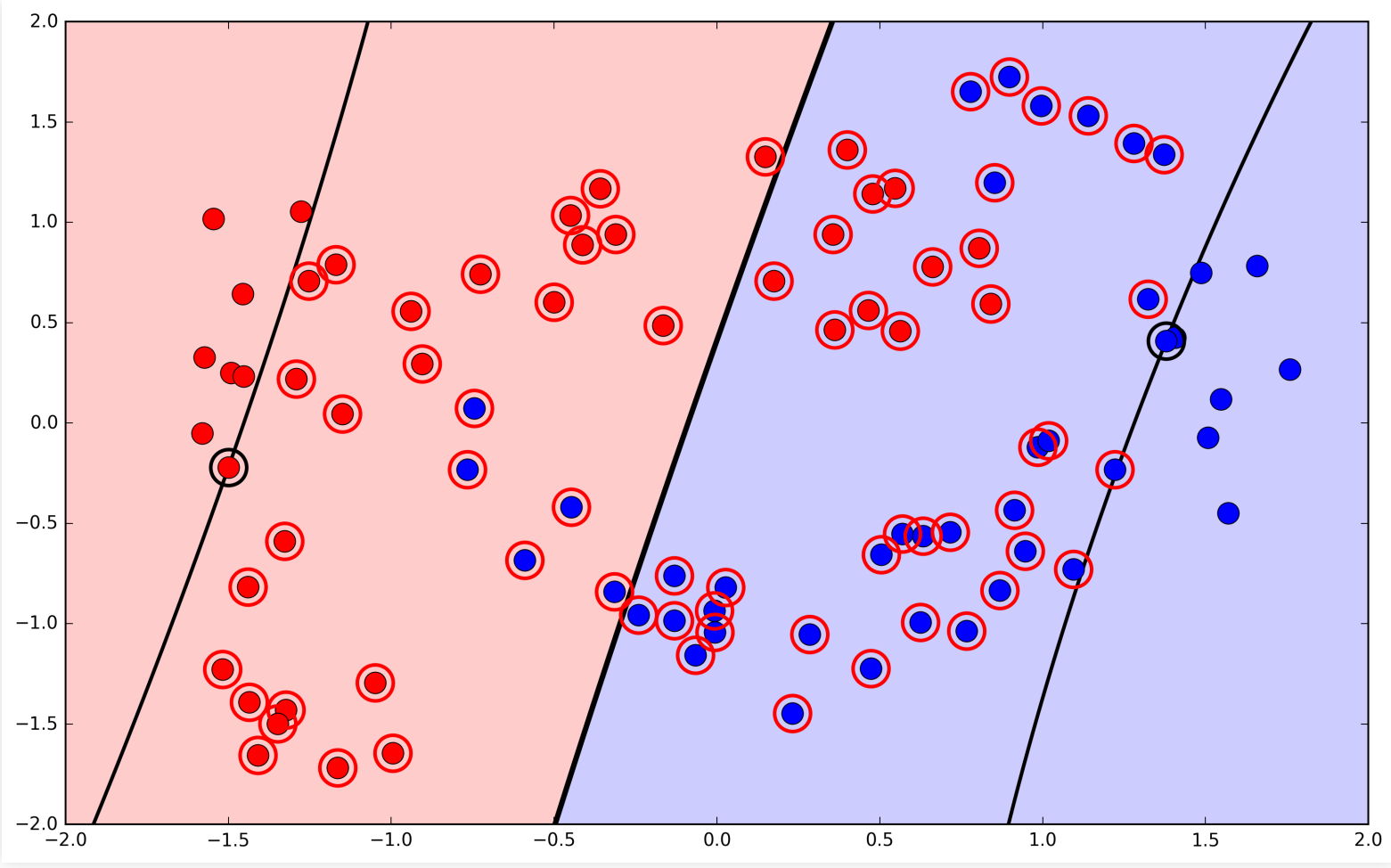
- In Scikit-Learn:

$$K(\vec{x_1}, \vec{x_2}) = e^{-\gamma\|\vec{x_1} - \vec{x_2}\|^2}$$

```
from sklearn import svm
#load and standardize
sv = svm.SVC(C=C,kernel = 'rbf', gamma=gamma)
sv.fit(Xs,Ys[:,0])
```

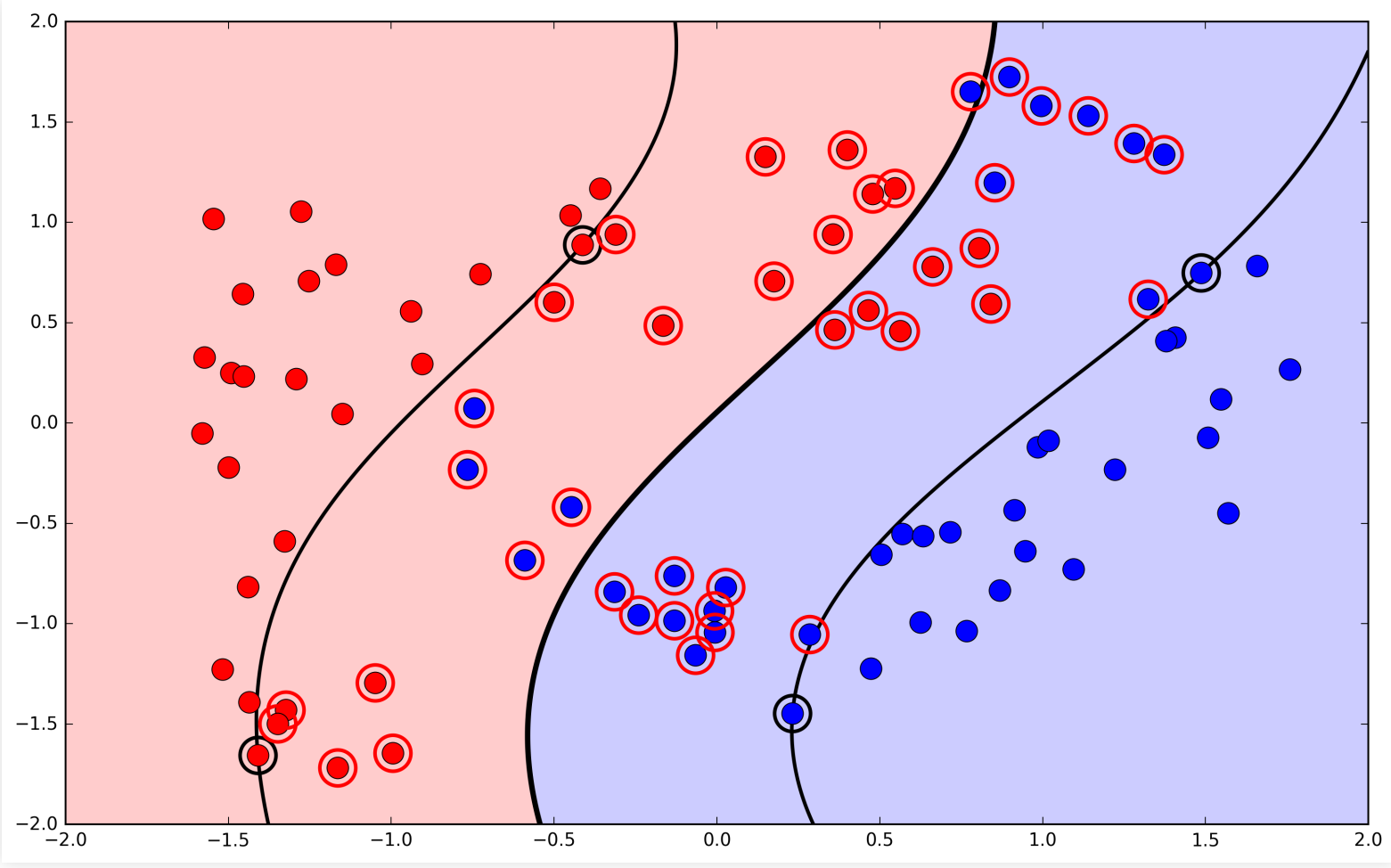
Kernel trick

- Gaussian RBF Kernel, $\gamma = 0.01$, C = 1



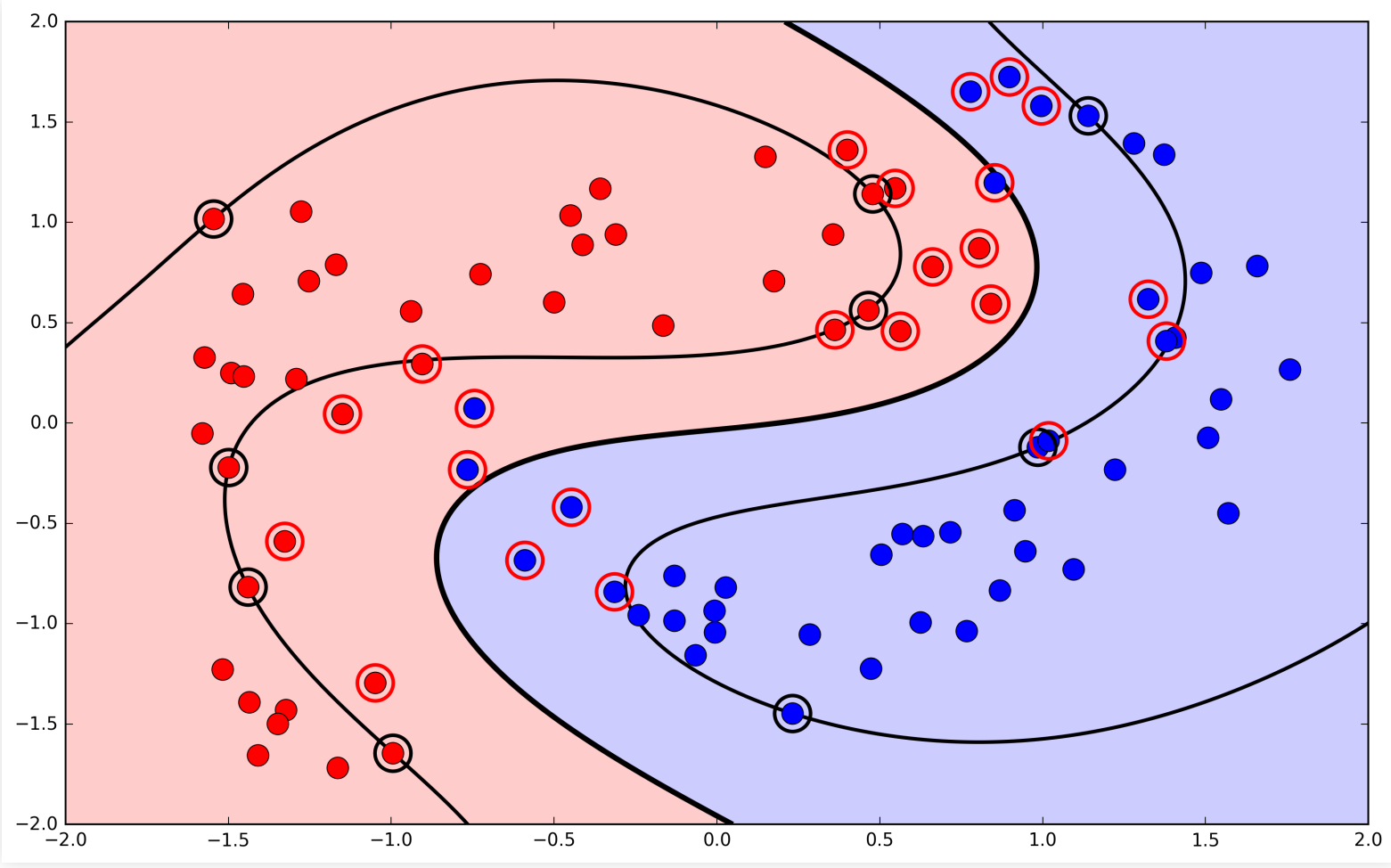
Kernel trick

- Gaussian RBF Kernel, $\gamma = 0.1$, C = 1



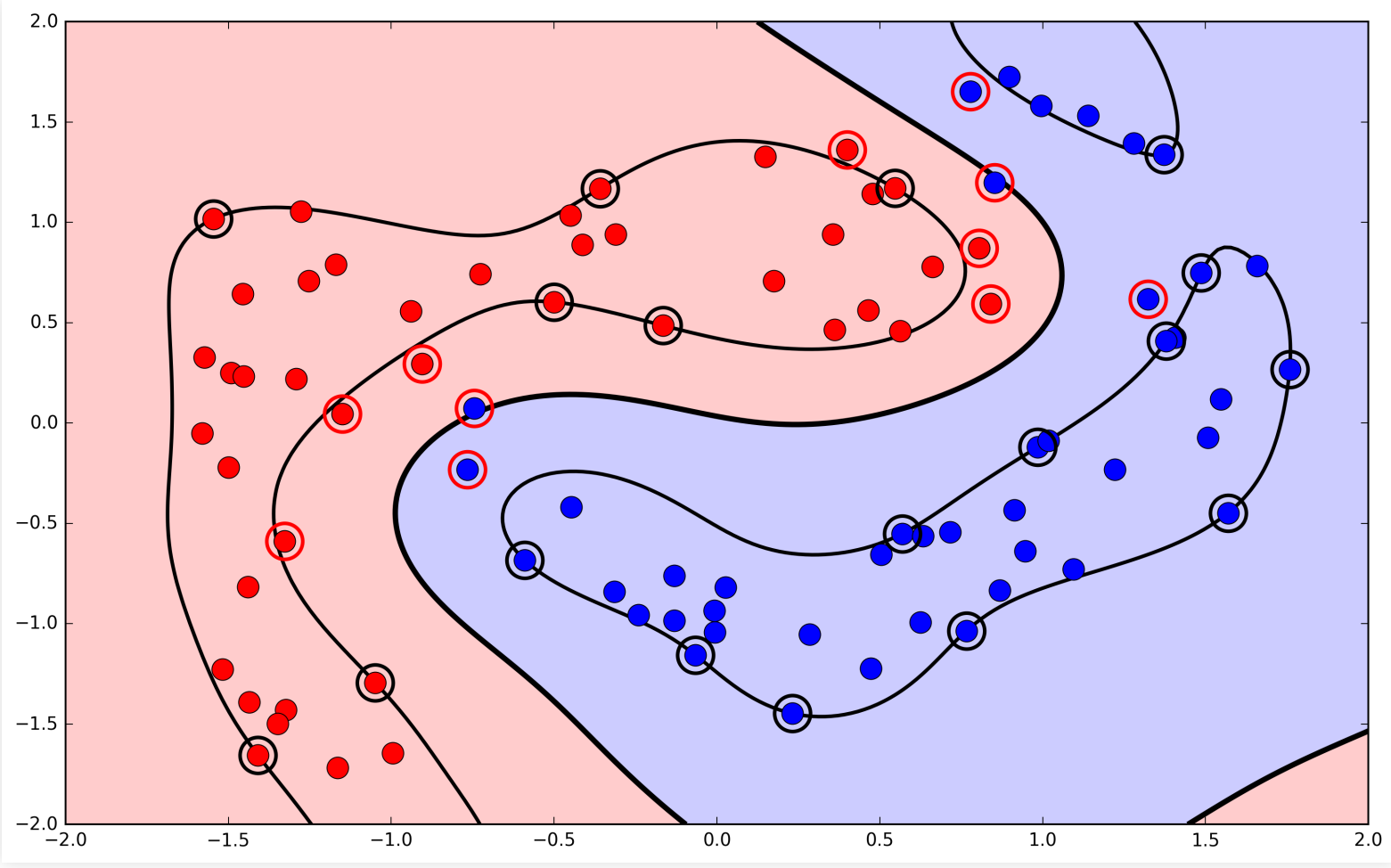
Kernel trick

- Gaussian RBF Kernel, $\gamma = 0.5$, C = 1



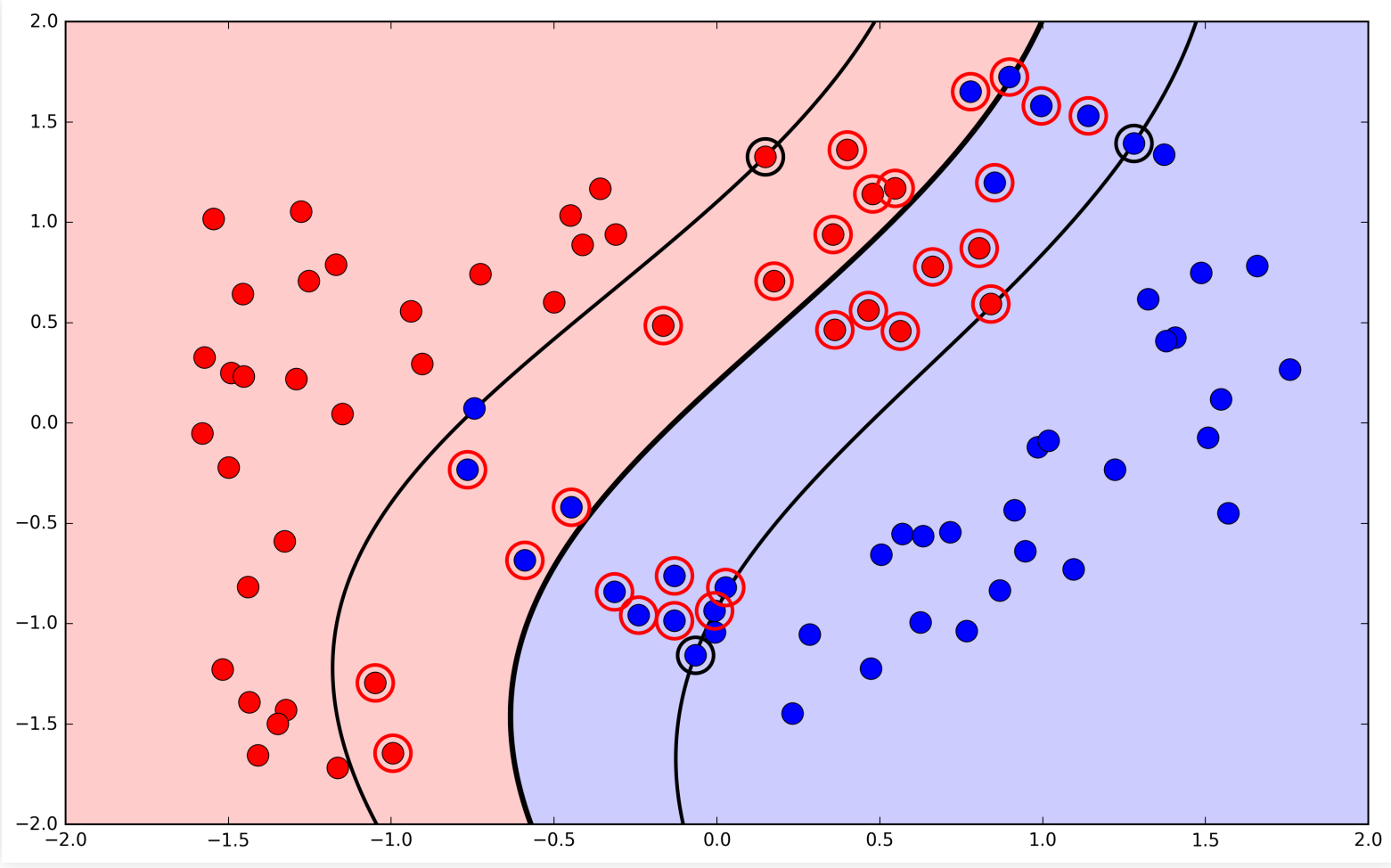
Kernel trick

- Gaussian RBF Kernel, $\gamma = 2$, C = 1



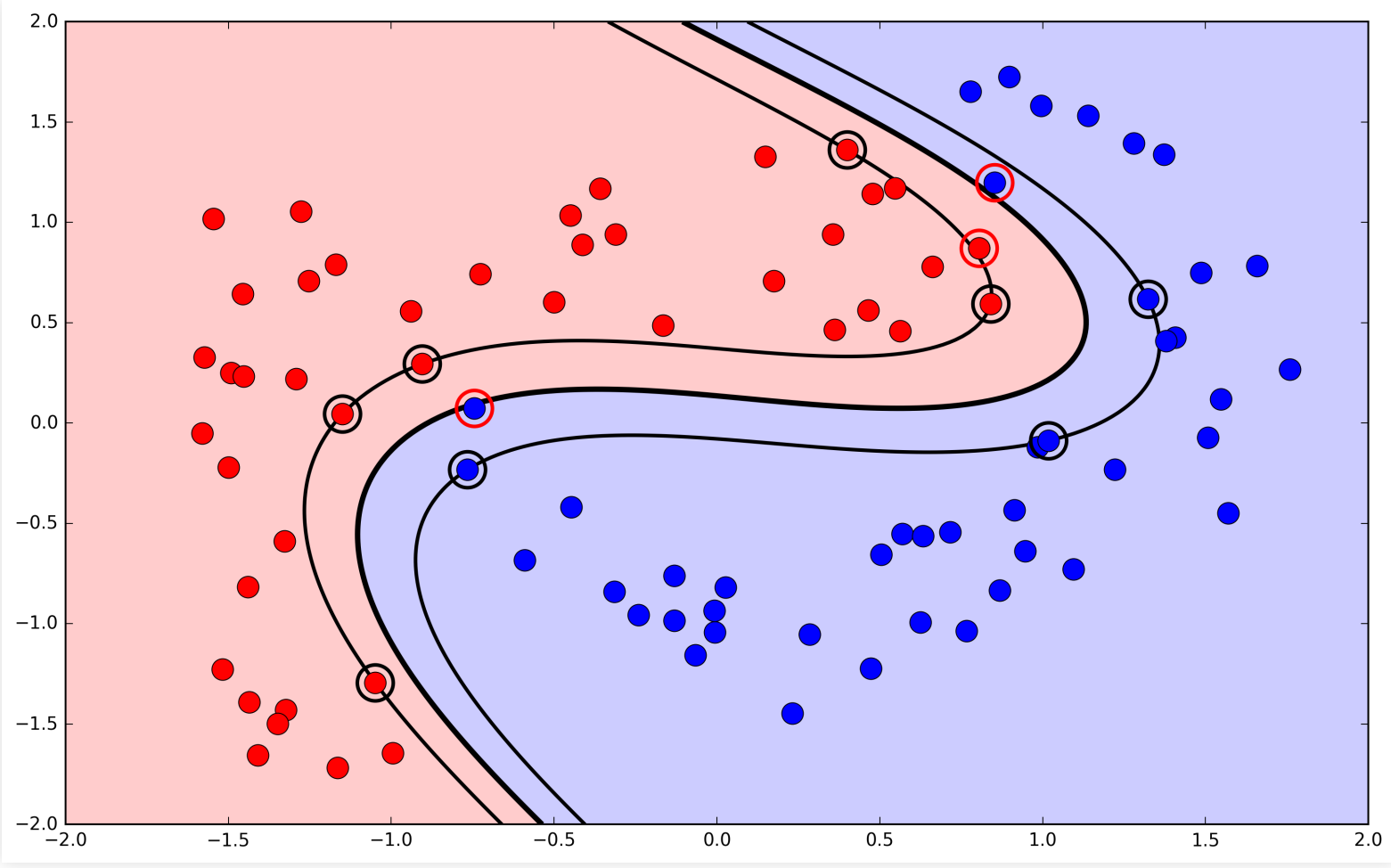
Kernel trick

- Gaussian RBF Kernel,, $\gamma = 0.01$, C = 1000



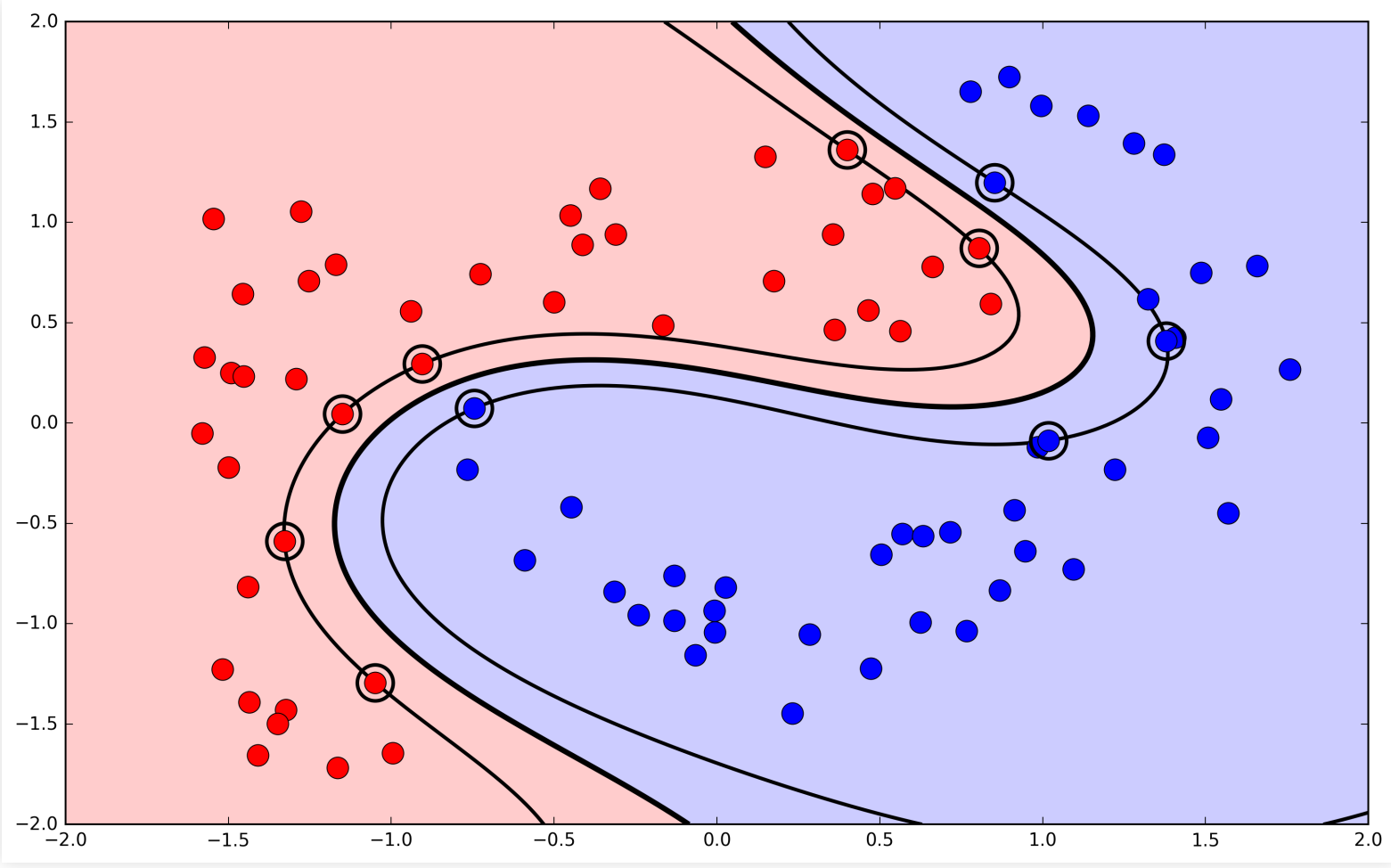
Kernel trick

- Gaussian RBF Kernel,, $\gamma = 0.1$, C = 1000



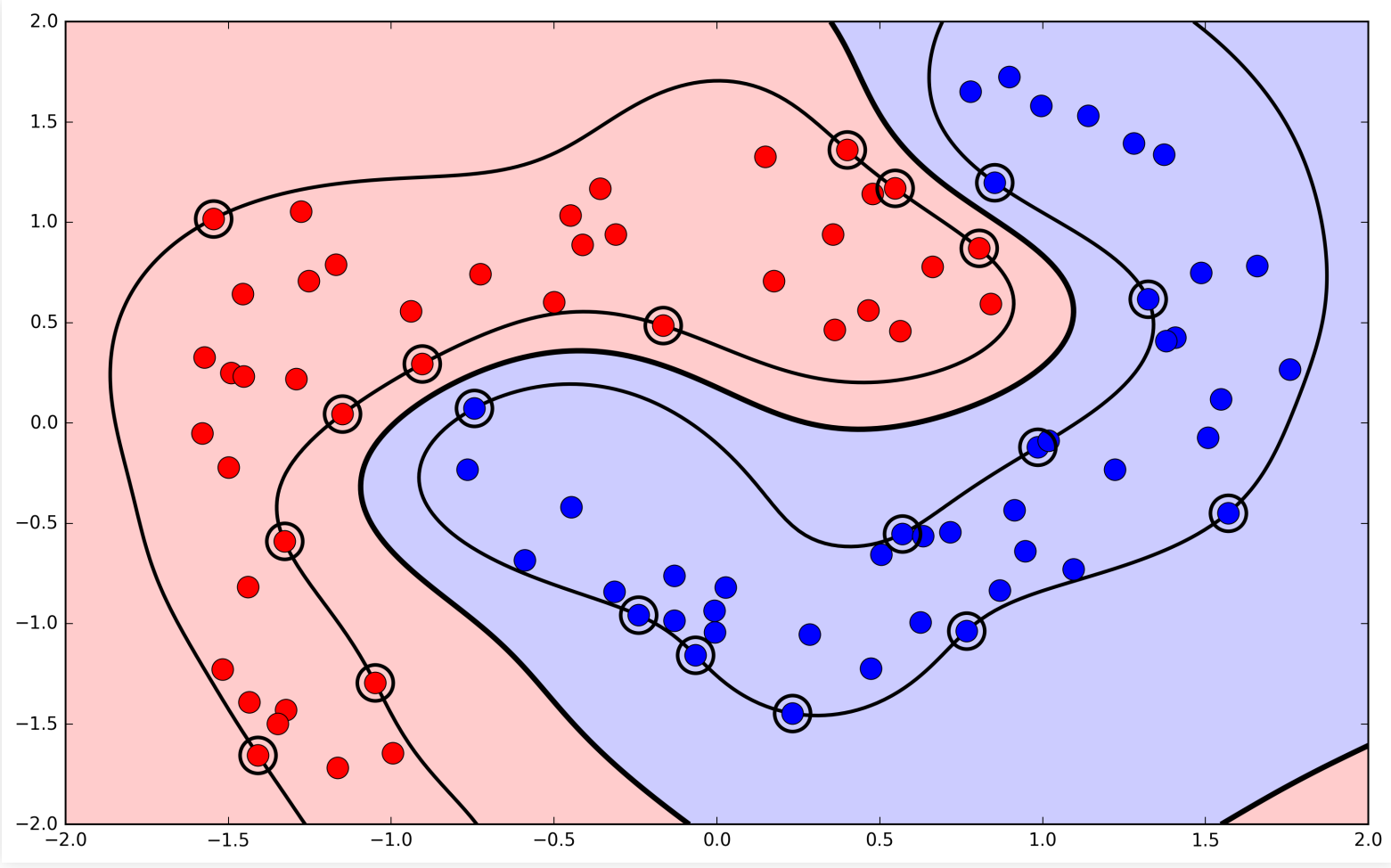
Kernel trick

- Gaussian RBF Kernel,, $\gamma = 0.5$, C = 1000



Kernel trick

- Gaussian RBF Kernel,, $\gamma = 2$, C = 1000



Assignment 1

Assignment 1

Classify bank notes

- Four features (from scans of bank notes), continuous
- Do not expand (that was for illustrating the concept)
- Three classifiers:

- Naïve Bayes with KDE (yours), Gaussian NB and SVM (sklearn)

- Implement Naïve Bayes using KDE from Scikit-Learn
- Optimize parameters (KDE bandwidth and gamma for SVM)
- Compare using approximate normal and McNemar's test
- Submit code, plots and answered question form

- English or Português

- Optional: optimize gamma and C for SVM, compare
- Check page for instructions (and updates)

Summary

SVM-2

Summary

- Soft margin
  - Use slack variables ξ
  - End result is same, but with upper limit C on α
- Non-linear classification with SVM
  - Kernel trick: use function of inner products
  - Kernel examples, sklearn.svm
- Regularization: soft margins

Further reading

- Alpaydin, Sections 13.1 - 13.8
- Marsland, Chapter 5
- Bishop, Section 7.1