DD2431 Machine Learning
Lab 2: Support Vector Machines

Örjan Ekeberg
Updated 2017 by Martin Hjelm & Nils Bore

January 16, 2017

1 Task

Your task is to build a Support Vector Machine for classification. You will make use of a library routine to solve the convex optimization problem which emerges in the dual formulation of the support vector machine. You need to write code for structuring the data so that the library routine can find the maximal-margin solution, and code for transforming this solution into a classifier which can process new data.

You will work in Python and use the `numpy` and `pylab` packages. You will also need the `cvxopt` package for convex optimization.

For reporting, you need to produce plots showing the decision boundary when using different kernel funcions. Optionally, you can also study how the introduction of slack variables change the boundary.

Note: The package `cvxopt` is not installed on the Sun computers in the lab rooms. Use the GNU/Linux machines in Grön, or log in remotely (via `ssh -X`) on either `u1.nada.kth.se` or `u2.nada.kth.se`.

2 Theory

The idea is to build a classifier which first makes an (optional) transformation of the input data, and then a linear separation where the decision boundary is placed to give maximal margins to the available data points. The location of the decision boundary is given by the weights ($\vec{w}$) and the bias (b) so the problem is to find the values for $\vec{w}$ and b which maximizes the margin, i.e. the distance to any datapoint.

The *primal* formulation of this optimization problem can be stated mathematically like this:

$$\min_{\vec{w},b} ||\vec{w}|| \tag{1}$$

under the constraints

$$t_i(\vec{w}^T \cdot \phi(\vec{x}_i) + b) \geq 1 \qquad \forall i \tag{2}$$

where we have used the following notation:

$\vec{w}$	Weight vector defining the separating hyperplane
b	Bias for the hyperplane
$\vec{x}_i$	The ith datapoint
t_i	Target class (-1 or 1) for datapoint i
$\phi(\ldots)$	Optional transformation of the input data

The constraints (2) enforce that all datapoints are not only correctly classified, but also that they stay clear of the decision boundary by a certain margin. Solving this optimization problem results in values for $\vec{w}$ and b which makes it possible to classify a new datapoint $\vec{x}^\star$ using this *indicator* function:

$$\text{ind}(\vec{x}^\star) = \vec{w}^T \cdot \phi(\vec{x}^\star) + b \tag{3}$$

If the indicator returns a positive value, we say that $\vec{x}^\star$ belongs to class 1, if it gives a negative value, we conclude that the class is -1. All the training data should have indicator values above 1 or below -1, since the interval between -1 and 1 constitutes the margin.

The bias variable, b, can be eliminated by a standard trick, i.e. by incorporating it as an extra element in the weight vector $\vec{w}$. We then need to let the $\phi(\ldots)$ function append a corresponding constant component, typically the value 1. Note that by using this trick we are actually slightly modifying the problem, since we are now also including the bias value in the cost function ($||\vec{w}||$). In practice, this will not make much difference, and we will use this "bias free" version from here on.

2.1 Dual Formulation

The optimization problem can be transformed into a different form, called the *dual problem* which has some computational advantages. In particular, it makes it possible to use the *kernel trick*, thereby eliminating the need for evaluating the $\phi(\ldots)$ function directly. This allows us to use transformations into very high-dimensional spaces without the penalty of excessive computational costs.

The dual form of the problem is to find the values α_i which minimizes:

$$\frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j t_i t_j \mathcal{K}(\vec{x}_i, \vec{x}_j) - \sum_i \alpha_i \tag{4}$$

subject to the constraints

$$\alpha_i \geq 0 \quad \forall i \tag{5}$$

The function $\mathcal{K}(\vec{x}_i, \vec{x}_j)$ is called a *kernel function* and computes the scalar value corresponding to $\phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$. This is, however, normally done implicitly,

i.e., without actually computing the two vectors and taking their scalar product (see section 2.4).

The indicator function now takes the form:

$$\text{ind}(\vec{x}^\star) = \sum_i \alpha_i t_i \mathcal{K}(\vec{x}^\star, \vec{x}_i) \tag{6}$$

For normal data sets, only a handful of the α's will be non-zero. Most of the terms in the indicator function will therefore be zero, and since this is known beforehand its evaluation can often be made very efficient.

2.2 Matrix Formulation

The dual problem can be expressed in a more compact form using vectors and matrices: find the vector $\vec{\alpha}$ which minimizes

$$\frac{1}{2}\vec{\alpha}^T P \vec{\alpha} - \vec{\alpha} \cdot \vec{1} \qquad \text{where } \vec{\alpha} \geq \vec{0} \tag{7}$$

$\vec{1}$ denotes a vector where all elements are one. Correspondingly, $\vec{0}$ is a vector with all zeros. We have also introduced the matrix P, with these elements:

$$P_{i,j} = t_i t_j \mathcal{K}(\vec{x}_i, \vec{x}_j) \tag{8}$$

In fact, this is a standard form for formulating quadratic optimization problems with linear constraints. This is a well known class of optimization problems where efficient solving algorithms are available.

2.3 Adding Slack Variables

The above method will fail if the training data are not linearly separable. In many cases, especially when the data contain some sort of noise, it is desirable to allow a few datapoints to be misclassified if it results in a substantially wider margin. This is where the method of *slack variables* comes in.

Instead of requiring that *every* datapoint is on the right side of the margin (equation 2) we will now allow for mistakes, quantified by variables ξ_i (one for each datapoint). These are called *slack variables*. The constraints will now be

$$t_i(\vec{w}^T \cdot \phi(\vec{x}_i)) \geq 1 - \xi_i \qquad \forall i \tag{9}$$

(Remember that we have already eliminated b from (2) by including it as a weight).

To make sense, we must ensure that the slack variables do not become unnecessarily large. This is easily achieved by adding a penalty term to the cost function, such that large ξ values will be penalized:

$$\min_{\vec{w},\vec{\xi}} ||\vec{w}|| + C \sum_i \xi_i \tag{10}$$

The new parameter C sets the relative importance of avoiding slack versus getting a wider margin. This has to be selected by the user, based on the character of the data. Noisy data typically deserve a low C value, allowing for more slack, since individual datapoints in strange locations should not be taken too seriously.

Fortunately, the dual formulation of the problem need only a slight modification to incorporate the slack variables. In fact, we only need to add an extra set of constraints to (5):

$$\alpha_i \geq 0 \quad \forall i \quad \text{and} \quad \alpha_i \leq C \quad \forall i \tag{11}$$

Equation (4) stays the same.

2.4 Selection of Kernel Function

One of the great advantages to support vector machines is that they are not restricted to linear separation. By transforming the input data non-linearly to a high-dimensional space, more complex decision boundaries can be utilized. In the dual formulation, these transformed data points $\phi(\vec{x}_i)$ always appear in pairs, and the only thing needed is the scalar product between the pair. This makes it possible to use what is often referred to as the *kernel trick*, i.e. we do not actually have to make the data transformation but, instead, we use a kernel function which directly returns the scalar product $\phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$.

Here are the most commonly used kernel functions:

- Linear kernel

$$\mathcal{K}(\vec{x}, \vec{y}) = \vec{x}^T \cdot \vec{y} + 1$$

  This kernel simply returns the scalar product between the two points. This results in a linear separation. Note the addition of 1 which comes from the elimination of the bias (b) by appending a constant element 1 to the data, resulting in an extra term 1×1 added to the scalar product.

- Polynomial kernels

$$\mathcal{K}(\vec{x}, \vec{y}) = (\vec{x}^T \cdot \vec{y} + 1)^p$$

  This kernel allows for curved decision boundaries. The exponent p (a positive integer) controls the degree of the polynomials. $p = 2$ will make quadratic shapes (ellipses, parabolas, hyperbolas). Setting $p = 3$ or higher will result in more complex shapes.

- Radial Basis Function kernels

$$\mathcal{K}(\vec{x}, \vec{y}) = e^{-\frac{(\vec{x}-\vec{y})^2}{2\sigma^2}}$$

  This kernel uses the explicit difference between the two datapoints, and often results in very good boundaries. The parameter σ can be used to control the smoothness of the boundary.

- Sigmoid kernels

$$\mathcal{K}(\vec{x}, \vec{y}) = \tanh(k\vec{x}^T \cdot \vec{y} - \delta)$$

This is yet another possible non-linear kernel. Parameters k and δ need to be tuned to get best performance.

3 Implementation

We will make use of a Python package for convex optimization[1] called `cvxopt`. In particular, we will call the function `qp` to solve our quadratic optimization problem. The `cvxopt` package relies on its own implementation of matrices so we must convert `numpy` arrays to `cvxopt` matrices.

Start by importing `qp` and `matrix` from `cvxopt`, and the other packages you will need:

```
from cvxopt.solvers import qp
from cvxopt.base import matrix

import numpy, pylab, random, math
```

`matrix` is a function which takes anything that can be interpreted as a matrix, for example a `numpy` array or an ordinary Python list of lists, and converts it into a `cvxopt` matrix which can be passed as a parameter to `qp`.

The call to `qp` can look like this:

```
r = qp(matrix(P), matrix(q), matrix(G), matrix(h))
alpha = list(r['x'])
```

This will find the $\vec{\alpha}$ which minimizes

$$\frac{1}{2}\vec{\alpha}^T P\vec{\alpha} + \vec{q}^T\vec{\alpha} \qquad \text{while } G\vec{\alpha} \leq \vec{h} \tag{12}$$

Here, P and G are matrices, while $\vec{q}$ and $\vec{h}$ are vectors.

As you can see, this is very similar to the problem we have. This is no coincidence, because we have formulated the problem in a standard way. To use `qp` for our problem we only need to build the necessary vectors and matrices; then `qp` will give us the optimal α's.

3.1 Things to implement

You will have to write code for:

[1] Quadratic problems are a special case of convex problems.

- A suitable kernel function

  The kernel function takes two data points as arguments and returns a "scalar product-like" similarity measure; a scalar value. Start with the linear kernel which is almost the same as an ordinary scalar product. Do not forget that you need to add 1 to the output, representing the extra "always-one" component.

- Build the P matrix from a given set of data points

  From the theory section we know that the P matrix should have the elements

  $$P_{i,j} = t_i t_j \mathcal{K}(\vec{x}_i, \vec{x}_j)$$

  Indices i and j run over all the data points. Thus, if you have N data points, P should be an $N \times N$ matrix.

- Build the $\vec{q}$ vector, G matrix, and $\vec{h}$ vector

  These vectors and matrices do not hold any actual data. Still, they need to be set up properly so that `qp` solves the right problem.

  By matching our problem (equation 7) with what `qt` solves (equation 12) we can see that $\vec{q}$ should be a N long vector containing only the number -1. Similarly, we realize that $\vec{h}$ must be a vector with all zeros.

  Note that the greater-than relation in (7) has to be made to match the less-than relation in (12). This can be achieved by creating a G matrix which has -1 in the diagonal and zero everywhere else (check this!).

- Call `qp`

  Make the call to `qp` as indicated in the code sample above. `qp` returns a dictionary data structure; this is why we must must use the string `'x'` as an index to pick out the actual α values.

- Pick out the non-zero α values

  If everything else is correct, only a few of the α values will be non-zero. Since we are dealing with floating point values, however, those that are supposed to be zero will in reality only be approximately zero. Therefore, use a low threshold (10^{-5} should work fine) to determine which are to be regarded as non-zero.

  You need to save the non-zero α_i's along with the corresponding data points ($\vec{x}_i$) in a separate data structure, for instance a list.

- Implement the indicator function

  Implement the indicator function (equation 6) which uses the non-zero α_i's together with their $\vec{x}_i$'s to classify new points.

4 Generating Test Data

In order to visualize the decision boundaries graphically we will restrict ourselves to two-dimensional data, i.e. points in the plane. The data will have the form of a vector of datapoints, where each datapoint is a triple of numbers. The first two numbers are the x and y coordiates and the last number is the class (-1 or 1).

We will use the function `random.normalvariate` to generate random numbers with a normal distribution. This is suitable for our task as we can build up more complex distributions by concatenating sample sets from multiple normal distributions.

```
# Uncomment the line below to generate
# the same dataset over and over again.
# numpy.random.seed(100)
classA = [(random.normalvariate(-1.5, 1),
           random.normalvariate(0.5, 1),
           1.0)
          for i in range(5)] + \
         [(random.normalvariate(1.5, 1),
           random.normalvariate(0.5, 1),
           1.0)
          for i in range(5)]

classB = [(random.normalvariate(0.0, 0.5),
           random.normalvariate(-0.5, 0.5),
           -1.0)
          for i in range(10)]

data = classA + classB
random.shuffle(data)
```

This will create ten datapoints for each class. The last line randomly reorders the points. You may want to change the values to move the clusters of data around.

In order to see your data, you can use the plot functions from `pylab`. This code will plot your two classes using blue and red dots.

```
pylab.hold(True)
pylab.plot([p[0] for p in classA],
           [p[1] for p in classA],
           'bo')
pylab.plot([p[0] for p in classB],
           [p[1] for p in classB],
           'ro')
pylab.show()
```

5 Plotting the Decision Boundary

Plotting the decision boundary is a great way of visualizing how the resulting support vector machine classifies new datapoints. The idea is to plot a curve in the input space (which is two dimensional here), such that all points on one side of the curve is classified as one class, and all points on the other side are classified as the other.

`pylab` has a function call `contour` that can be used to plot contour lines of a function given as values on a grid. Decision boundaries are special cases of contour lines; by drawing a contour at the level where the classifier has its threshold we will get the decision boundary.

What we will have to do is to call your indicator function at a large number of points to see what the classification is at those points. We then draw a contour line at level zero, but also countour lines at -1 and 1 to visualize the margin.

```
xrange=numpy.arange(-4, 4, 0.05)
yrange=numpy.arange(-4, 4, 0.05)

grid=matrix([[indicator(x, y)
              for y in yrange]
             for x in xrange])

pylab.contour(xrange, yrange, grid,
              (-1.0, 0.0, 1.0),
              colors=('red', 'black', 'blue'),
              linewidths=(1, 3, 1))
```

If everything is correct, the margins should touch some of the datapoints. These are the *support vectors*, and should correspond to datapoints with non-zero α's. You may want to plot datapoints with non-zero α's using a separate kind of marker to visualize this.

6 Running and Reporting

Once you have the linear kernel running, there are a number of things you can explore. Remember that support vector machines are especially good at finding a reasonable decision boundary from small sets of training data.

1. Move the clusters around to make it easier or harder for the classifier to find a decent boundary. Pay attention to when the `qt` function prints an error message that it can not find a solution.
2. Implement some of the non-linear kernels. you should be able to classify very hard datasets.
3. The non-linear kernels have parameters; explore how they influence the decision boundary. Reason about this in terms of the bias-variance trade-off.

7 Slack Implementation

You should now alter your program to include slack variables. They can quite easily be incorporated by adding the extra contraints (equation 11). This means that you have to extend the matrix G and vector $\vec{h}$ with additional rows corresponding to the $\alpha_i \leq C$ constraints:

- G should now be a $2N \times N$ matrix. Note that the constraint is now reversed, resulting in the additional lower part being negated as compared to previously.
- $\vec{h}$ should now be a $2N$ column vector.

Repeat the previous exercise with slack variables added. Answer the additional following questions:

1. Explore the role of the parameter C. What happens for very large/small values?
2. Imagine that you are given data that is not easily separable. When should you opt for more slack rather than going for a more complex model and vice versa?

8 The End

You are now done. Please make sure you answered all the questions and printed plots to support your reasoning.