DD2421 Machine Learning
Lab 2: Support Vector Machines

Örjan Ekeberg

January 29, 2018

1 Learning Objectives

The objective of this lab assignment is that you should get hands-on experience with implementing and using a support vector machine. In particular, you should be able to:

- use the mathematical formulation of the optimization task,
- formulate the indicator function and explain how it relates to the outcome of the classification,
- predict and explain the outcome of using different kernels,
- explain the effect of the C-value when using slack variables.

In addition, you will gain some experience in using Python together with the packages `numpy`, `scipy`, and `matplotlib`, commonly used for scientific computing.

2 Task

Your task is to build a Support Vector Machine for classification. You will make use of a library routine to solve the 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` package for efficient handling of numerical data. The optimization is done using a general purpose function from the package `scipy`. For plotting, you will use the `matplotlib` package. All three packages are freely available as open source and you can find more information, including detailed documentation, here `https://scipy.org`.

3 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	Offset (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 $\vec{s}$ using this *indicator* function:

$$\text{ind}(\vec{s}) = \vec{w}^T \cdot \phi(\vec{s}) - b \tag{3}$$

If the indicator returns a positive value, we say that $\vec{s}$ 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.

3.1 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

$$\alpha_i \geq 0 \quad \forall i \qquad \text{and} \qquad \sum_i \alpha_i t_i = 0 \tag{5}$$

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 3.3).

The indicator function now takes the form:

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

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 this can be utilized to make the classification of new points very efficient. The subset of the data points $\vec{x}_i$ corresponding to non-zero values of α_i are called the *support vectors* because they are located exactly *on* the margins, giving *support* in a mechanical sense.

The threshold value, b, does not appear in the dual problem and must therefore be calculated separately. This can be done from the fact that the indicator function for any support vector has a value equal to its target value, since we know that it is exactly *on* the margin.

$$b = \sum_i \alpha_i t_i \mathcal{K}(\vec{s}, \vec{x}_i) - t_{\vec{s}} \qquad \text{for any SV } \vec{s} \tag{7}$$

3.2 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 miss-classified if it results in a substantially wider margin. This is where the method of *slack variables* or *soft margins* comes in.

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

$$t_i(\vec{w}^T \cdot \phi(\vec{x}_i) - b) \geq 1 - \xi_i \qquad \forall i \tag{8}$$

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:

$$\min_{\vec{w},b,\vec{\xi}} ||\vec{w}|| + C \sum_i \xi_i \tag{9}$$

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 upper bound for the α-values to (5):

$$0 \leq \alpha_i \leq C \quad \forall i \qquad \text{and} \quad \sum_i \alpha_i t_i = 0 \tag{10}$$

Equation (4) stays the same.

3.3 Kernel Functions

One of the great advantages of support vector machines is that they are not restricted to linear separation in the input space. 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. In fact, the only operation ever needed is the scalar product between pairs. 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

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

  This kernel simply returns the scalar product between the two points. This results in a linear separation.

- 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 (RBF) kernels

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

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

4 Implementation

We will use the general purpose optimization function `minimize` available in the `scipy.optimize` library. This will work well for the small "toy problems" we are dealing with here. For more complex problems, perhaps with thousands of samples, it is better to use one of the special purpose optimizers developed to be efficient specifically for support vector machines.

Start by importing `minimize` from `scipy.optimize`, along with the other packages you will need[1]:

```
import numpy, random, math
from scipy.optimize import minimize
import matplotlib.pyplot as plt
```

The heart of your program will be a single call to the `minimize` function. A call to `minimize` should look like this:

```
ret = minimize( objective, start,
                bounds=B, constraints=XC )
alpha = ret['x']
```

This will find the vector $\vec{\alpha}$ which minimizes the function `objective` within the bounds `B` and the constraints `XC`.

`objective` is a function you have to define, which takes a vector $\vec{\alpha}$ as argument and returns a scalar value, effectively implementing the expression that should be minimized, in our case equation (4).

`start` is a vector with the initial guess of the $\vec{\alpha}$ vector. We can, e.g., simply use a vector of zeros: `numpy.zeros(N)`. N is here the number of training samples (note that each training sample will have a corresponding α-value).

`B` is a list of pairs of the same length as the $\vec{\alpha}$-vector, stating the lower and upper bounds for the corresponding element in $\vec{\alpha}$. To constrain the α values to be in the range $0 \leq \alpha \leq C$, we can set `bounds=[(0, C) for b in range(N)]`. To only have a lower bound, set the upper bound to `None` like this: `bounds=[(0, None) for b in range(N)]`.

Python hint: Here we make use of *List Comprehensions* which is a neat way of creating a new list by looping over an already existing list (or, in fact, any sequence). The code:

```
a = [ expr for x in seq ]
```

will construct a new list `a` of the same length as the sequence `seq`. Each element is computed by evaluating the expression `expr` while `x` temporarily contains the corresponding element from `seq`. Normally, `expr` is an expression which contains the variable `x` as a part.

[1] If you are running on your own computer you may need to install the packages first. Most GNU/Linux distributions have them prepackaged. For Windows and Mac you can download them from the `https://scipy.org`-site.

`XC` is used to impose other constraints, in addition to the bounds. We will use this to impose the equality constraint, that is, the second half of (10). This parameter is given as a dictionary with the fields `type` and `fun`, stating the type of the constraint and its implementation, respectively. You can write an equality constraint like this: `constraint={'type':'eq', 'fun':zerofun}`, where `zerofun` is a function you have defined which calculates the value which should be constrained to zero. Like `objective`, `zerofun` takes a vector as argument and returns a scalar value.

4.1 Things to implement

You will have to write code for:

- 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 the same as an ordinary scalar product, but also explore the other kernels in section 3.3.

- Implement the function `objective`

  Define a function which implements equation (4). This function will only receive the vector $\vec{\alpha}$ as a parameter. You can use global variables for other things that the function needs (t and $\mathcal{K}$ values).

  Python hint: This function will be called hundreds of times, so it makes sense to care about efficiency.

  You can pre-compute a matrix with these values:

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

  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. This matrix should be computed only once, outside of the function `objective`. Therefore, store it as a `numpy array` in a global variable.

  Inside the `objective` function, you can now make use of the functions `numpy.dot` (for vector-vector, vector-matrix, and matrix-vector multiplications), and `numpy.sum` (for summing the elements of a vector). This is much faster than explicit for-loops in Python.

- Implement the function `zerofun`

  This function should implement the equality constraint of (10). Also here, you can make use of `numpy.dot` to be efficient.

- Call `minimize`

  Make the call to `minimize` as indicated in the code sample above. Note that `minimize` 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. There are other useful indices that you can use; in particular, the index `'success'` holds a boolean value which is `True` if the optimizer actually found a solution.

- Extract the non-zero α values

  If the data is well separated, 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 ($\vec{x}_i$) and target values (t_i) in a separate data structure, for instance a list.

- Calculate the b value using equation (7).

  Note that you must use a point *on* the margin. This corresponds to a point with an α-value larger than zero, but less than C (if slack is used).

- 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 and t_i's to classify new points.

To test your support vector machine, you will also need code for generating test data and for visualizing the results. In the following sections you will be given code fragments that you can use directly in your program to achieve this.

5 Generating Test Data

To be able to visualize the decision boundaries graphically we will restrict ourselves to classifying two-dimensional data, i.e. points in the plane. The data will have the form of a $N \times 2$ array, `inputs`, where each row contains the (x, y)-coordinates of a datapoint. There is also a corresponding $N \times 1$ array, `targets`, which contains the classes, i.e. the t_i values, encoded as (-1 or 1).

We use the function `random.randn` to generate arrays with random numbers from a normal distribution with zero mean and unit variance. By multiplying with a number and adding a 2D-vector, we can scale and shift this cluster to any position. We can build up more complex distributions by concatenating several sets from multiple normal distributions.

This sample code below shows how you can generate data for one class (A) with ten points around $(1.5, 0.5)$ and ten around $(-1.5, 0.5)$, and another class (B) with twenty points around $(0.0, -0.5)$. The clusters all have a standard deviation of 0.2. The samples are stored in the array `inputs` and the corresponding class labels (1 and -1) are stored in the array `targets` at corresponding indices.

```
classA = numpy.concatenate(
    (numpy.random.randn(10, 2) * 0.2 + [1.5, 0.5],
     numpy.random.randn(10, 2) * 0.2 + [-1.5, 0.5]))
classB = numpy.random.randn(20, 2) * 0.2 + [0.0, -0.5]

inputs = numpy.concatenate((classA, classB))
targets = numpy.concatenate(
    (numpy.ones(classA.shape[0]),
     -numpy.ones(classB.shape[0])))

N = inputs.shape[0]  # Number of rows (samples)

permute=list(range(N))
random.shuffle(permute)
inputs = inputs[permute, :]
targets = targets[permute]
```

The last four lines randomly reorders the samples. You will later move the clusters around and change their spread by changing the values in this code sample.

Hint: If you insert a call "`numpy.random.seed(100)`" before the code that generates the data, you will get the same random data every time you run the program. This can help during debugging.

6 Plotting

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

```
plt.plot([p[0] for p in classA],
         [p[1] for p in classA],
         'b.')
plt.plot([p[0] for p in classB],
         [p[1] for p in classB],
         'r.')

plt.axis('equal') # Force same scale on both axes
plt.savefig('svmplot.pdf') # Save a copy in a file
plt.show() # Show the plot on the screen
```

The last two lines ensure that the plot is both saved in a file and shown on the screen. Every time you run the program you will overwrite the file, so you may want to rename it when you see something interesting in order to preserve it.

The formatting strings `'b.'` and `'r.'` controls the color (first character) and shape (second character) of the markers. You may want to use different colors or shapes to highlight which points are support vectors, e.g. with `'g+'` (green cross), `'yo'` (yellow large dot), etc.

6.1 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.

`matplotlib` has a function called `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, i.e. at zero, 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 contour lines at -1 and 1 to visualize the margin.

```
xgrid=numpy.linspace(-5, 5)
ygrid=numpy.linspace(-4, 4)

grid=numpy.array([[indicator(x, y)
                   for y in ygrid]
                  for x in xgrid])

plt.contour(xgrid, ygrid, 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.

7 Exploring 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 and change their sizes to make it easier or harder for the classifier to find a decent boundary. Pay attention to when the optimizer (`minimize` function) is not able to find a solution at all.
2. Implement the two non-linear kernels. You should be able to classify very hard data sets with these.
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.
4. Explore the role of the slack parameter C. What happens for very large/small values?
5. 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 (kernel) 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.