MTAT.03.227 Machine Learning (Spring 2013) Exercise session XIV: Support Vector Machines

Konstantin Tretyakov

May 13, 2013

The aim of this exercise session is to get acquainted with the inner workings of support vector machine classification and regression. As usual, for all exercises you need to write a brief explanation and, for most of them, also a short piece of code demonstrating the result. You can submit your whole solution as a single *decently commented* R file, provided it is sequentially readable and executable.

We shall use a slightly modified version of the familiar base code from Exercise Session VI. Fetch it at `svm_base.R`.

Maximal margin

Exercise 1 (2pt). We begin by analyzing a trivialized dataset consisting of just four points. Use the functions `load.data` and `plot.data` to visualize it. Use manual examination of the data to derive the parameters $(\mathbf{w}, b)$ of the canonical[1] maximal-margin classifier.

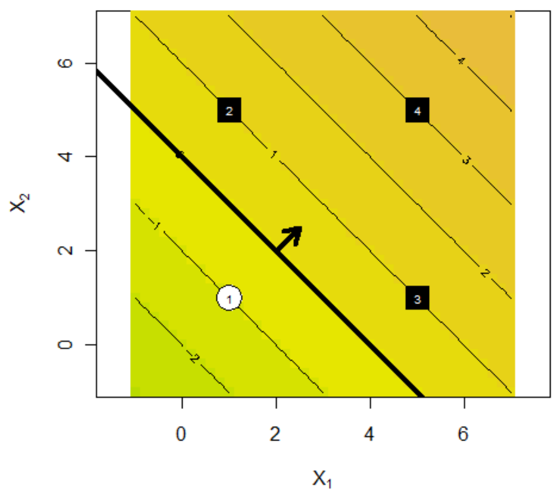
Hints.

1. Use pen and paper to sketch the location of the maximal margin separating hyperplane for the given dataset and its two margin-defining isolines $f(\mathbf{x}) = \pm 1$.
2. By visual examination, guess the coordinates of the weight vector $\mathbf{w}$ (up to a constant). Use `plot.classifier(w)` and `plot.data(data, add=T)` to verify your guess.
3. Next, find the length of the normal, $|\mathbf{w}|$, such that would assure that the closest training points are located at the isolines ± 1. For this, note that the distance between those two isolines is equal to $d = 2/|\mathbf{w}|$.
4. Rescale the $\mathbf{w}$ you guessed in Step 2 to have length you computed in Step 3.
5. Finally, compute the value for the bias parameter b by using the fact that $f(\mathbf{x}_1) = -1$.
6. Visualize the computed $(\mathbf{w}, b)$ using `plot.classifier`.

[1] A canonical maximal-margin classifier is the classifier for which the functional margin of the closest training example is equal to 1. It is what we were talking about on the lecture.

Solution.

1. A brief look at the data should convince you that what we are looking for is the following separating hyperplane:



2. The normal of the separating hyperplane seems to be exactly diagonal, hence the direction of the normal is given by $\mathbf{w} = (1,1)^T$.

3. Looking at the figure above we can compute that the geometric distance d between the two desired margins is $2\sqrt{2}$. The desired normal vector $\mathbf{w}$ must therefore have length $|\mathbf{w}| = 2/d = 1/\sqrt{2}$.

4. The length of our first guess, $(1,1)^T$ is $\sqrt{2}$, hence we have to scale it by a factor of $1/2$ to obtain $\mathbf{w} = (0.5, 0.5)^T$, which has the required length.

5. Now that we know $\mathbf{w}$ we can find the bias w_0, which positions the separating line so that the first training example $\mathbf{x}_1 = (1,1)$ is exactly at the isoline $f(\mathbf{x}) = -1$. For that we simply solve:

$$\begin{aligned} f(\mathbf{x}_1) &= -1 \\ \mathbf{w}^T\mathbf{x}_1 + w_0 &= -1 \\ (0.5, 0.5) \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} + w_0 &= -1 \\ 0.5 + 0.5 + w_0 &= -1 \\ w_0 &= -2. \end{aligned}$$

6. Visualization above demonstrates that we found the correct parameters $\mathbf{w} = (0.5, 0.5)^T$, $w_0 = -2$ for the desired separating line.

Next, we shall use R's quadratic programming facilities to find the maximal margin classifier for the same dataset. There are several packages which we could use, but the simplest option seems to be the `quadprog` package and its `solve.QP` function. Load the package using the command `library(quadprog)`.

For the following you need to know that the invocation

```
> solve.QP(D, d, t(A), b)
```

will numerically solve the following quadratic programming problem:

$$\text{argmin}_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T\mathbf{D}\mathbf{x} - \mathbf{d}^T\mathbf{x},$$

$$\text{s.t. } \mathbf{A}\mathbf{x} \geq \mathbf{b}.$$

Exercise 2 (3pt). Recall that the hard-margin SVM optimization problem is: *Primal form*

$$\text{argmin}_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2,$$

$$\text{s.t. } \forall i \quad (\mathbf{w}^T\mathbf{x}_i + b)y_i \geq 1.$$

Use `solve.QP` to compute a solution to this problem for our sample dataset.

Hints.

1. Determine what is $\mathbf{x}$ in this problem.
2. Rewrite the objective in the form $\frac{1}{2}\mathbf{x}^T\mathbf{D}\mathbf{x} - \mathbf{d}^T\mathbf{x}$. What are $\mathbf{D}$ and $\mathbf{d}$?
3. How many constraints are there? Rewrite them in the form $\mathbf{A}\mathbf{x} \geq \mathbf{b}$.
4. Finally, define the necessary variables in R and invoke `solve.QP` as shown above.
5. The quadratic programming solver will probably complain that the matrix $\mathbf{D}$ is not positive semidefinite. You can address by adding a tiny value on the diagonal of this matrix:

   ```
   D = D + 1e-10*diag(nrow(D))
   ```

6. Confirm that the solution matches the one you obtained manually.

Solution.

1. We need to find the parameters w_0 and $\mathbf{w}$. The optimization variable $\mathbf{x}$ is thus a concatenation of the two: $\mathbf{x} := (w_0, \mathbf{w}^T) = (w_0, w_1, w_2)^T$.

2. Observe that

$$\frac{1}{2}\|\mathbf{w}\|^2 = \frac{1}{2}\mathbf{w}^T\mathbf{w} = \frac{1}{2}\mathbf{w}^T\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\mathbf{w} = \frac{1}{2}(w_0, \mathbf{w}^T)\left(\begin{array}{c|cc}0 & 0 & 0\\ \hline 0 & 1 & 0\\ 0 & 0 & 1\end{array}\right)\begin{pmatrix}w_0\\ \mathbf{w}\end{pmatrix},$$

and thus

$$\mathbf{D} = \left(\begin{array}{c|cc}0 & 0 & 0\\ \hline 0 & 1 & 0\\ 0 & 0 & 1\end{array}\right), \qquad \mathbf{d} = \begin{pmatrix}0\\0\\0\end{pmatrix}.$$

3. There is a single constraint for each data point, thus there is a total of four constraints. Let us rearrange them into the form $\mathbf{Ax} \geq \mathbf{b}$:

$$\begin{cases}(\mathbf{w}^T\mathbf{x}_1 + w_0)y_1 \geq 1\\ (\mathbf{w}^T\mathbf{x}_2 + w_0)y_2 \geq 1\\ (\mathbf{w}^T\mathbf{x}_3 + w_0)y_3 \geq 1\\ (\mathbf{w}^T\mathbf{x}_4 + w_0)y_4 \geq 1\end{cases} \Leftrightarrow \begin{cases}(y_1, y_1\mathbf{x}_1^T)\begin{pmatrix}w_0\\ \mathbf{w}\end{pmatrix} \geq 1\\ (y_2, y_2\mathbf{x}_2^T)\begin{pmatrix}w_0\\ \mathbf{w}\end{pmatrix} \geq 1\\ (y_3, y_3\mathbf{x}_3^T)\begin{pmatrix}w_0\\ \mathbf{w}\end{pmatrix} \geq 1\\ (y_4, y_4\mathbf{x}_4^T)\begin{pmatrix}w_0\\ \mathbf{w}\end{pmatrix} \geq 1\end{cases} \Leftrightarrow \begin{pmatrix}y_1 & y_1\mathbf{x}_1^T\\ y_2 & y_2\mathbf{x}_2^T\\ y_3 & y_3\mathbf{x}_3^T\\ y_4 & y_4\mathbf{x}_4^T\end{pmatrix}\begin{pmatrix}w_0\\ \mathbf{w}\end{pmatrix} \geq \begin{pmatrix}1\\1\\1\\1\end{pmatrix}$$

4-6.
```
data = load.data()
D = diag(3)
D[1, 1] = 0
D = D + 1e-10*diag(3)
d = c(0, 0, 0)
A = cbind(data$y, data$X * data$y)
b = c(1, 1, 1, 1)
library(quadprog)
sol = solve.QP(D, d, t(A), b)
```

```
> sol$solution
[1] -2.0 0.5  0.5
```

Exercise 3 (3pt). Now let us solve the SVM in *dual* form. Some formalities aside, the dual form is obtained by representing the weight vector $\mathbf{w}$ as a linear combination of training points:

Dual form

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i,$$

where α_i are the unknowns (the *dual variables*) we shall be seeking for instead of $\mathbf{w}$.

In the dual form the optimization problem turns into[2]:

$$\text{argmin}_{\boldsymbol{\alpha}} \left(\frac{1}{2} \boldsymbol{\alpha}^T (\mathbf{K} \circ \mathbf{Y}) \boldsymbol{\alpha} - \mathbf{1}^T \boldsymbol{\alpha} \right)$$

$$\begin{aligned} \text{s.t. } \boldsymbol{\alpha} &\geq 0, \\ \mathbf{y}^T \boldsymbol{\alpha} &= 0, \end{aligned}$$

where

$\mathbf{K} = \mathbf{X}\mathbf{X}^T$ is the *kernel matrix*,

$\mathbf{Y} = \mathbf{y}\mathbf{y}^T$,

$\mathbf{1}$ is a vector of ones, and

$\circ$ denotes elementwise multiplication.

Recast this formulation in the format suitable for `solve.QP` and find both the dual solution $\boldsymbol{\alpha}$ and the corresponding bias term b. Finally, use the obtained $\boldsymbol{\alpha}$ values to compute $\mathbf{w}$ and compare your result to two previous attempts.

Hints.

1. Similarly to the previous exercise, you first need to recast the problem in terms of $\mathbf{D}$, $\mathbf{d}$, $\mathbf{A}$, $\mathbf{b}$. Note that there is one equality constraint now. Include its coefficients as the first row of the matrix $\mathbf{A}$, its right side as the first element of vector $\mathbf{b}$ and provide the parameter `meq=1` to `solve.QP`.

2. Similarly to the previous exercise, if the solver complains about the lack of positive semidefiniteness, add a small value to the diagonal of $\mathbf{D}$.

3. Compute $\mathbf{w}$ as
$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i.$$
Note that this can be done concisely matrix multiplication.

[2]Note that in the lecture there was also the constraint $\boldsymbol{\alpha} \leq C$. This constraint is not present in the hard-margin case, however.

4. To find the bias term b find any *support vector* and use the fact that for a support vector $\mathbf{x}_i$, in the case of hard-margin classification, it must hold that
$$y_i(\mathbf{w}^T\mathbf{x}_i + b) = 1.$$

Examine the values α that you obtained. Try the following visualization:

```
plot.classifier(w, b)
plot(data, add=T)
text(data$X[,1], data$X[,2], alphas)
```

Solution.

1. It is not hard to see directly from the optimization problem that:
$$\mathbf{x} := \boldsymbol{\alpha}, \qquad \mathbf{D} = \mathbf{K} \circ \mathbf{Y}, \qquad \mathbf{d} = \mathbf{1}.$$
The inequality constraints (of which there will be 4) in matrix form are simply
$$\mathbf{I}\boldsymbol{\alpha} \geq 0,$$
where $\mathbf{I}$ is the 4×4 identity matrix. There is also one equality constraint which is already given in matrix form. To pass both types of constraints to `solve.QP` we need to stack them all together into a single matrix, with the equality constraint on the top, and use the `meq = 1` parameter. Thus:
$$\mathbf{A} = \begin{pmatrix} y_1 & y_2 & y_3 & y_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad \mathbf{b} = \mathbf{0}.$$

2.
```
K = data$X %*% t(data$X)
Y = data$y %*% t(data$y)
D = K*Y + 1e-10*diag(4)
d = c(1, 1, 1, 1)
b = c(0, 0, 0, 0, 0)
A = rbind(data$y, diag(4))
sol = solve.QP(D, d, t(A), b, meq=1)
```

```
> sol$solution
[1] 0.2500010 0.1249990 0.1250010 0.0000000
```

3. We can compute the primal weight vector as $\mathbf{w} = \mathbf{X}^T(\boldsymbol{\alpha} \circ \mathbf{y})$:

```
> w = t(data$X) %*% (sol$solution*data$y)
> w
        [,1]
x1 0.5000029
x2 0.4999952
```

4. A support vector (for the hard margin case) is any training example with the corresponding $\alpha \neq 0$. We can use any of them to compute w_0.

```
support_vectors = which(sol$solution != 0) # 1, 2, 3
s = support_vectors[1]                     # Pick any
w0 = data$y[s] - t(w) %*% data$X[s,]   # -1.999998
```

By examining the α values of the support vectors, observe how the "weight" of the constraints is balanced on both sides of the separating line.

Finding the bias term in dual

Exercise 4* (5pt). In the case of hard-margin classification you may always find w_0 by relying on the fact that all support vectors are always located exactly on the margin (i.e. they satisfy $f(\mathbf{x}_i) = y_i$). In the case of soft margin classification this idea can also sometimes be exploited. Namely, all support vectors which have $0 < \alpha < C$ are also necessarily on the margin. However, this need not always be the case – it may turn out so that *all* support vectors of a soft-margin SVM are violating the margin.

How should you compute w_0 in this case? Provide a proof of your claim.

Solution. Consider the primal form of the soft-margin SVM optimization task:

$$\text{argmin}_{\mathbf{w},w_0} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i (1 - (\mathbf{w}^T\mathbf{x}_i + w_0)y_i)_+$$

After we have found the dual solution $\boldsymbol{\alpha}$, we can use it to compute and fix the correct weight vector $\mathbf{w} := \mathbf{w}^*$. We then consider the optimization problem again with a fixed value of $\mathbf{w}$, this time to find just the bias term:

$$\text{argmin}_{w_0} \frac{1}{2}\|\mathbf{w}^*\|^2 + C\sum_i (1 - (\mathbf{w}^{*T}\mathbf{x}_i + w_0)y_i)_+.$$

We can simplify it by observing that for the purposes of optimization with respect to w_0, the (now fixed) value of $\|\mathbf{w}^*\|^2$ plays no role. In addition, we can drop all those terms from the sum, which correspond to non-support vectors $\mathbf{x}_i$, as we know that those play no role in determining the separating line, as long as they lie outside the margin:

$$\text{argmin}_{w_0} \sum_{s\in\text{SV}} (1 - (\mathbf{w}^{*T}\mathbf{x}_s + w_0)y_s)_+.$$

$$\text{s.t. } \forall k \notin \text{SV} \quad (\mathbf{w}^{*T}\mathbf{x}_k + w_0)y_k \geq 1 \tag{*}$$

Next, note that for all support vectors the expression $1 - (\mathbf{w}^{*T}\mathbf{x}_s + w_0)y_s$ at the optimal value of w_0 is always non-negative. We can thus safely replace the $(\cdot)_+$ function with $|\cdot|$, which is "more stringent" for negative inputs only.

$$\text{argmin}_{w_0} \sum_{s\in\text{SV}} |1 - (\mathbf{w}^{*T}\mathbf{x}_s + w_0)y_s|.$$

We rearrange slightly:

$$|1-(\mathbf{w}^{*T}\mathbf{x}_s+w_0)y_s| = |y_s-(\mathbf{w}^{*T}\mathbf{x}_s+w_0)| = |(y_s-\mathbf{w}^{*T}\mathbf{x}_s)-w_0|,$$

and see that the final optimization is just the min-absolute-value problem:

$$\text{argmin}_{w_0}\sum_{s\in\text{SV}}|e_s-w_0|.$$

where $e_s = y_s - \mathbf{w}^{*T}\mathbf{x}_s$.

We know from Exercise Session V that the solution to this problem is the median of e_s. Thus, the correct way to obtain w_0 is to compute e_s for all support vectors, and take a median.

The additional condition (*) can sometimes play a role. Namely, when the number of support vectors is even[3], the median is not uniquely defined – it is a range of values, some of which may not satisfy the condition (*).

The fact that the bias term w_0 may not be uniquely defined has an intuitive explanation. Indeed, if there are no support vectors on the margin, then shifting the line slightly will decrease penalty paid by the support vectors on one side, and compensate it by increasing the penalty of the support vectors on the other side by exactly the same amount. Thus any positions of the separating line are allowed, as long as support vectors are within the margins and non-support vectors – outside the margins.

This suggests a practical way of computing the bias term directly from the constraints. We know that

$$\begin{cases} y_i(\mathbf{w}^{*T}\mathbf{x}_i+b)\le 1, & \text{for } i\in\text{SV},\\ y_i(\mathbf{w}^{*T}\mathbf{x}_i+b)\ge 1, & \text{for } i\notin\text{SV},\end{cases}$$

which can be written compactly as

$$y_i(\mathbf{w}^{*T}\mathbf{x}_i+b)\overset{\text{SV}}{\lesseqgtr} 1.$$

Then, multiplying both sides by y_i and keeping track of the sign:

$$\begin{cases} \mathbf{w}^{*T}\mathbf{x}_i+b\overset{\text{SV}}{\lesseqgtr} y_i, & \text{if } y_i=1\\ \mathbf{w}^{*T}\mathbf{x}_i+b\overset{\text{SV}}{\gtreqless} y_i, & \text{if } y_i=-1.\end{cases}$$

which, abusing notation even further, is:

$$\begin{aligned} \mathbf{w}^{*T}\mathbf{x}_i+b &\overset{\text{SV xor }-1}{\lesseqgtr} y_i,\\ b &\overset{\text{SV xor }-1}{\lesseqgtr} y_i-\mathbf{w}^T\mathbf{x}_i,\end{aligned}$$

[3]This, by the way, is a necessary condition for a situation where no support vectors are on the margin, think why.

and thus the complete solution for b (in the case of *no* support vectors being on the margin) is:

$$b \in \left[\max_{i\in LB} e_i, \quad \min_{i\in UB} e_i\right],$$

where

$$UB = \{i : i \in SV \text{ xor } y_i = -1\},$$
$$LB = \{i : i \notin SV \text{ xor } y_i = -1\},$$
$$e_i = y_i - \mathbf{w}^{*T}\mathbf{x}_i.$$

It is worth noting that some implementations use the mean of e_s as the estimate of w_0 rather than the median[4] (which is correct for L2-SVMs, but not for the "classical", L1-SVMs).

Exercise 5 (1pt). Finally, let us use a third-party tool to fit an SVM model. The de-facto standard implementation in R is provided by the package `e1071` and its `svm` function[5]. Invoke it as follows: *e1071*

```
m = svm(data$X, data$y, scale=FALSE,
              kernel="linear", type="C-classification", cost=9e9)
```

Examine the output. Find the $\boldsymbol{\alpha}$ vector and the bias term. Do they match your previous results? Compute the $\mathbf{w}$ vector.

Solution.

```
m = svm(data$X, data$y, scale=FALSE,
              kernel="linear", type="C-classification", cost=9e9)
alphas = m$coefs          # Only nonzero alphas in this vector
w = t(m$SV)%*% alphas
w0 = -m$rho
```

You might notice that the `svm` function internally reverses the classes – this is a consequence of treating y as a factor and its first value (which in our case is -1) as the positive class.

Exercise 6 (1pt). So far we have only examined hard-margin classification (note that the `svm` function does not have a specific hard-margin mode, but specifying $C = 9 \cdot 10^9$, as we did in the previous exercise, is more-or-less as good as prohibiting any margin violations). Now let us study the effect of "relaxing" the margin. *Soft-margin*

First, add a new datapoint $\mathbf{x}_5 = (0, 5)^T$ of class $y_5 = -1$ to the dataset:

[4]Probably because the topic of computing w_0 is swept under the rug in many SVM textbook treatments.

[5]This is actually a packaged version of the LibSVM C library.

```
data = load.data()
data$X = rbind(data$X, c(0, 5))
data$y = c(data$y, -1)
```

Now run `svm` with `cost = 9e9` and plot the resulting classifier using `plot.classifier` and `plot.data`. Try to guess what happens to the separating line once you reduce the cost to 1 and then further to 0.1. Verify your guesses.

Solution. As we reduce the cost of misclassification slightly (say, to 10 or so), at first nothing changes. However, at some point, as the cost becomes small enough, the SVM decides to "let go" of the single "most annoying" training example (#2 in our case) in return for getting a wider margin. If we continue reducing the penalty, at some point another example will be sacrificed, and so on. Such "incremental letting go" of the parameters is actually a property of all models which involve sharp corners (like the hinge loss does) in their objective function.

You may observe that one of the negative effects of having a very low regularization penalty is reduction in the sparseness of the model – the more training examples are allowed to violate the margin, the larger is the total number of support vectors.

On the other hand (this is something you do not see in this example but is commonly observed on large datasets), the less stringent is the margin penalty, the faster will the SVM optimizer converge.

RBF kernel

Exercise 7* (2pt). A *kernel* (to be covered in the upcoming lecture) is a generalization of an *inner product*. It turns out that everywhere where we use an inner product $\mathbf{x}_i^T\mathbf{x}_j$, we may instead use a nonlinear function $K(\mathbf{x}_i, \mathbf{x}_j)$, which will serve the same purpose but will make our model non-linear.

One of the most popular kernels is the *RBF* kernel:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2\right).$$

Study its properties on our toy example and answer the following questions.

1. The linear SVM in dual form, as you should know, corresponds to the following functional:

   $$f(\mathbf{x}) = \sum_i \alpha_i y_i \mathbf{x}_i^T\mathbf{x} + w_0.$$

   Is this functional linear in $\mathbf{x}$? Prove it.

2. What is the corresponding functional of an SVM with an RBF kernel? Is it linear in $\mathbf{x}$?

3. What is the value of $K(\mathbf{x}, \mathbf{z})$ for any point $\mathbf{z}$ that is geometrically very close to $\mathbf{x}$? What is the value of $K(\mathbf{x}, \mathbf{z})$ for points $\mathbf{z}$ that are far from $\mathbf{x}$?

4. Use the `svm` function to train an SVM classifier with an RBF kernel for the data you used in the previous exercise. Let $\gamma = 1$ and $C = 9 \cdot 10^9$.

5. Use the function `plot.svm.functional`, provided in the base code, to visualize the resulting functional[6]. Locate the decision boundary on the resulting plot.

6. Try different values of $\gamma \in \{0.001, 0.01, 0.1, 1, 10\}$ and visualize the results. What do you observe?

7. For $\gamma = 0.1$ try different values of $C \in \{0.1, 1, 10\}$. What do you observe?

8. What, do you think, is a good way for picking suitable values of γ and C in real-life applications?

Solution.

1. Recall that a functional is linear if it satisfies $f(a\mathbf{x} + \mathbf{y}) = af(\mathbf{x}) + f(\mathbf{y})$. It is easy to see that the given functional does not satisfy this property (unless $w_0 = 0$). However, it is none the less an *affine* functional and corresponds to a linear classifier.

2. In the case of the RBF kernel, the functional has the form:

$$f(\mathbf{x}) = \sum_i \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + w_0 = \sum_i \alpha_i y_i \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}\|^2) + w_0,$$

   and is obviously not linear in $\mathbf{x}$.

3. The function $K(\mathbf{x}, \cdot)$ is essentially a "hump" centered at $\mathbf{x}$. For a point $\mathbf{z}$ close to the center of the hump, the value of $K(\mathbf{x}, \mathbf{z})$ is close to 1. For points $\mathbf{z}$ far from the center of the hump the value of $K(\mathbf{x}, \mathbf{z})$ is close to 0. This can provide some intuitive understanding of how the RBF classifier works. Ignoring the w_0 term it is essentially,

$$f(\mathbf{z}) = \sum_i \alpha_i y_i K(\mathbf{x}_i, \mathbf{z}) \approx \sum_{\substack{i \\ y_i = 1 \\ \mathbf{z} \text{ is close to } \mathbf{x}_i}} \alpha_i - \sum_{\substack{j \\ y_j = -1 \\ \mathbf{z} \text{ is close to } \mathbf{x}_j}} \alpha_j,$$

   Intuitively, the classifier sums the contributions α from the support vectors near the query point, and decides on its class based on what class dominates among the nearby highly contributing support vectors.

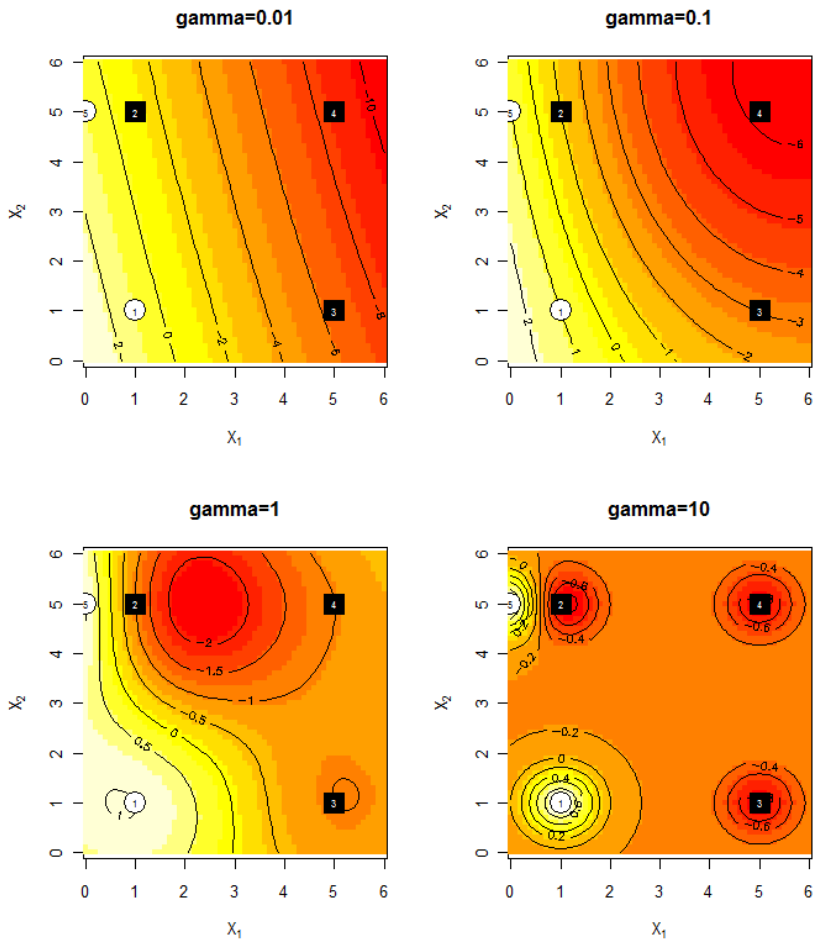
4-5. The training and visualization of the RBF model is performed as follows:

[6] Note that there is a function `plot.svm` in the `e1071` package, but it does not exactly do what we need here.

```
m = svm(data$X, data$y, kernel="radial",
    type="C-classification", gamma=1, cost=9e9)
plot.svm.functional(m, data)
```

Notice that the decision boundary is curved now, and that all data points are support vectors. Examine which isolines the datapoints are located at.

6. The following is a visualization of the resulting RBF functional for several values of γ. Notice that small values of γ correspond to simpler decision boundaries and smaller number of support vectors (i.e. high bias, low variance situation), whereas very high values of γ correspond to severe overfitting.



7. What you should notice from varying the C parameter is that in our

situation it does not influence the general shape of the functional as much as the γ parameter does (a fairly common effect with RBF SVMs). It does, however, affect the decision boundary – for very small C the classifier will simply classify everything as the dominant class.

8. The most popular way of selecting the γ and C parameters is via cross-validation. Note that choosing two parameters means that we have to assess the algorithm performance for all *pairs* of (γ, C), from a preselected grid. This can often be quite computation-intensive.

Exercise 8* (1pt). In the lecture we did not get to cover the concept of *Support vector regression (SVR)*. Read about this approach on your own. As the answer to this exercise, provide the formulation of the SVR optimization problem in primal form, explaining the meaning of the parameters.

Support vector regression

Solution. Support vector regression differs from support vector classification in that y_i are not $+1/-1$ class labels but rather arbitrary real values, and that the *hinge loss* penalty is replaced by the ε*-insensitive loss* penalty.

$$\mathrm{loss}_\varepsilon(\mathbf{x}_i, y_i) = (|y_i - (\mathbf{w}^T\mathbf{x}_i + w_0)| - \varepsilon)_+.$$

Here ε defines the distance that the regression line may have to the true y_i values without incurring any penalty.

The optimization objective is then the familiar combination:

$$\mathrm{argmin}_{\mathbf{w},w_0} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum \mathrm{loss}_\varepsilon(\mathbf{x}_i, y_i).$$

Of course, analogously to the usual SVMs, there is an easily kernelizable dual reformulation.

Exercise 9* (2pt). Finally, let us consider a simple case study. We shall use the `spam` dataset from the package `ElemStatLearn`:

Case study

```
> install.packages("ElemStatLearn")
> library(ElemStatLearn)
> data(spam)
```

Each row in the `spam` data frame corresponds to an email. The emails have been preprocessed and frequencies of certain words have been extracted – those frequencies form the input features. You need to train a classifier to predict the value of the output feature (the `spam` attribute).

Leave 25% of the training examples for final testing, and use the 75% for training. Try SVM and at least one other classifier. For SVM, try at least the linear and the RBF kernel and do not forget to tune the the `gamma` and `cost` parameters. Explain how you proceeded. Once you settle on the best model, evaluate it once on your held-out 25% and report the result.

Solution. This is a fairly open-ended task and this sample solution is not *the* only way to go. It is rather meant to illustrate a couple of main ideas necessary to try the application of SVMs in the basic analysis of the data. We shall compare the performance of the logistic model with the linear-kernel and RBF-kernel SVM classifiers. We shall use cross-validation for tuning the model parameters.

Data examination

First we load the data and separate a 25% holdout set for final testing.

```
install.packages("ElemStatLearn")
library(ElemStatLearn)
data(spam)
train_idx = sample(1:nrow(spam), 0.75*nrow(spam))
spam_train = spam[train_idx,]
spam_holdout = spam[-train_idx,]
```

Modern spam classification is a problem with highly imbalanced class probabilities – there is way more spam in your inbox than valid email. Such problems always require careful treatment. Luckily, the toy dataset we are considering has a pretty balanced class distribution. The reason is that it comes from the good old times when spam was in its infancy. Thus, we do not have to bother about this issue here.

```
> table(spam$spam)
email  spam
 2788  1813
```

We also ignore the issue of having different penalties for false positive and false negative misclassifications, which is otherwise very important in real-life spam detection. You are usually ready to get a couple of spam emails in your inbox now and then, in return for higher confidence that no valid emails will be put into the spam folder.

Before going on with the analysis it also makes sense to check the distribution of the values in the dataset using `boxplot`, visualize the data using `prcomp`, look for NA values and outliers, consider centering or normalizing the attributes, etc. I leave this part for your personal entertainment. For the purposes of this example the dataset seems to be good as-is.

Performance evaluation

We shall be using 3-fold cross-validation[7] for assessing performance of various algorithms. As of today, R does not seem to have a nice generic cross-validation package available[8], but there are specific implementations here and there. In particular, there is the `tune` method in the `e1071` package which works nicely with the `svm`, `randomForest`, `rpart` and a couple of other learners. For the logistic model we are better off using the `cv.glm` method in `boot` package.

[7]Of course, 10-fold is more conventional, but it is also way slower, especially for grid searches and is not worth the trouble. We have enough data.

[8]Anyone interested in a simple, yet enormously useful project?

For simplicity of this exposition, we shall use the *classification error* (i.e. $1 -$ accuracy) as our target performance measure, although it must be noted again, that in tasks such as spam classification, *precision* and *recall* usually matter more than accuracy.

Classifier learning and parameter tuning
Let us start with the logistic model.

```
m = glm(spam~., spam_train, family=binomial)
cv.results = cv.glm(spam_train, m,
              function(y, yhat) { mean(y!=round(yhat)) }, 3)
cv.results$delta[1]   # 0.07652174
```

The result is approximately 7.7%. Given that the size of the dataset `nrow(spam_train)` used to perform cross-validation is 3450, we should remember to treat this estimate as having a potential error of about $\approx 1/\sqrt{3450} \approx 1.7\%$ or so[9].

Logistic regression does not have any tunable parameters, but we could further experiment by attempting to apply logistic regression on transformed (e.g. log-transformed) data. I'll leave it for your personal examination.

Next, let's try linear SVMs:

```
tune.results = tune(svm, spam~., data=spam_train,
                tunecontrol=tune.control(cross=3),
                ranges=list(kernel="linear"))
```

The result is a similar 7.9%. What about RBF SVM? Let us check it with a range of values[10] for $\gamma \in \{0.001, 0.01, 0.1\}$ and[11] $C \in \{1, 10, 100\}$:

```
tune.results = tune(svm, spam~., data=spam_train,
           tunecontrol=tune.control(cross=3),
           ranges=list(kernel="radial",
                       gamma=c(0.001, 0.01, 0.1),
                       cost=c(1, 10, 100)))
```

```
> tune.results

Parameter tuning of 'svm':

- sampling method: 3-fold cross validation

- best parameters:
 kernel gamma cost
```

[9] To be more precise, the 95% confidence interval of this estimate is $\pm 1.96\sqrt{0.9(1-0.9)/n} \approx \pm 1\%$.

[10] The small ranges are selected to allow the example to run in reasonable time. In practice you might often want to check larger parameter spaces and leave the evaluation running for a night or two. Sometimes a week or two.

[11] I should admit I made a couple of preliminary runs to rule out large γ and small C values.

```
 radial 0.001 100

- best performance: 0.06840792
```

This result is only slightly (but most probably statistically significantly) better than the linear solutions, hence we settle on it as our best choice.

We can now perform the final evaluation of the selected model by training it on the whole training set and evaluating on the held-out set:

```
m = svm(spam~., spam_train, kernel="radial",
        gamma=0.001, cost=100)
p = predict(m, spam_holdout)
performance = mean(p != spam_holdout$spam) # 0.05299739
```

Nice. Can we do even better on this dataset? Yes, we can, but using something else than SVMs! Want to know how? Look for the answer in the textbook "Elements of Statistical Learning" by Hastie *et al.* [12].

---

[12]http://www.stanford.edu/~hastie/local.ftp/Springer/OLD/ESLII_print4.pdf