Support Vector Machines

Albert Mata

30/11/2018

Contents

Black Box Methods - Support Vector Machines

We talk about **black box** processes when the mechanism that transforms the input into the output is obfuscated by an imaginary box (mainly due to the complex mathematics allowing them to function). But it's dangerous to apply black box models blindly, so we'll peek inside the box and see how SVMs work.

Understanding Support Vector Machines

A **Support Vector Machine (SVM)** can be imagined as a surface that creates a boundary between points of data plotted in a multidimensional space that represent examples and their feature values. The goal of a SVM is to create a flat boundary called a **hyperplane** that divides the space to create fairly homogeneous partitions on either side. It can be seen as a combination of nearest neighbours + linear regression models. The maths behind SVMs are somewhat difficult, but the basic concepts are easy to understand. SVMs work well for both classification (the part we'll explore) and numeric prediction as in:

- Classification of microarray gene expression data.
- Text categorization (language, subject, etc).
- Detection of rare yet important events like combustion engine failure, security breaches or eartquakes.

Classification with hyperplanes

SVMs use a boundary called a hyperplane to partition data into groups of similar class values. When data can be separated perfectly by the hyperplane, they are said to be **linearly separable** (but that obviously isn't always the case).

But given that there are multiple hyperplanes that can divide data into separated groups, the algorithm chooses between them searching for the **Maximum Margin Hyperplane (MMH)** that creates the greatest separation between the groups, as it is likely that it will be the hyperplane that will generalize the best to future data.

The **support vectors** are the points (at least one) from each class that are the closest to the MMH. Using them alone, it is possible to define the MMH, so they provide a very compact way to store a classification model.

When the classes are linearly separable, it's quite easy to understand how to find the MMH: it is as far away as possible from the outer boundaries of the two groups of data points. These outer boundaries are known as the **convex hull**. The MMH is the perpendicular bisector of the shortest line between the two convex hulls (found through **quadratic optimization**).

When the classes are not linearly separable, we use a **slack variable** that creates a soft margin that allows some points to fall on the incorrect side of the margin. A cost value *C* is applied to all points on the wrong side and, rather than finding the maximum margin, the algorithm attempts to minimize the total cost. Modifying this value *C* will adjust the penalty for falling on the wrong side of the hyperplane. The greater the cost parameter *C*, the harder the optimization will try to achieve 100% separation. A lower cost parameter *C* will place the emphasis on a wider overall margin. It is important to strike a balance between these two in order to create a model that generalizes well to future data.

Using kernels for non-linear spaces

In many real-world applications, the relationships between variables are nonlinear. As we've just seen, SVM can still be trained on such data adding a slack variable. But there's another possible approach: SVMs can map the problem into a higher dimension space using a process known as the **kernel trick**. Doing so, a nonlinear relationship may suddenly appear to be quite linear. As in the example where we have sunny/snowy observations based on latitude and longitude and then we add the new dimension altitude and data becomes linearly separable.

SVMs with nonlinear kernels add additional dimensions to the data in order to create separation in this way. The kernel trick involves a process of constructing new features that express mathematical relationships between measured characteristics. This allows SVM to learn concepts that were not explicitly measured in the original data.

SVMs with nonlinear kernels are extremely powerful classifiers, although they do have some downsides as well.

Strengths:

- Can be used for classification or numeric prediction problems.
- Not overly influenced by noisy data and not very prone to overfitting.
- May be easier to use than neural networks, particulary due to the existence of several well-supported SVM algorithms.
- Gaining popularity due to its high accuracy and high-profile wins in data mining competitions.

Weaknesses:

- Finding the best model requires testing of various combinations of kernels and model parameters.
- Can be slow to train, particulary if the input dataset has a large number of features or examples.
- Results in a complex black box model that is difficult, if not impossible, to interpret.

A few of the most commonly used kernel functions are the linear kernel, the polynomial kernel (of degree *d*), the sigmoid kernel (that results in a SVM model somewhat analogous to a neural network using a sigmoid activation function) or the Gaussian RBF kernel. There is no reliable rule to match a kernel to a particular learning task. The fit depends heavily on the concept to be learned as well as the amount of training data and the relationships among the features. A bit of trial and error is often

required by training and evaluating several SVMs on a validation dataset. Although in many cases the performance may vary only slightly.

Example - performing Optical Character Recognition with SVMs

Step 1 - collecting data

Image processing is a difficult task for many types of ML algorithms. The relationships linking patterns of pixels to higher concepts are extremely complex and hard do define. And image data is often noisy. But given the strengths stated a moment ago, this looks like an ideal scenario for SVMs.

OCR software divides the paper into a matrix such that each cell contains a single glyph. Then it tries to match each glyph to a set of all characters it recognizes. Finally, individual characters are combined into words that can even be spell-checked against a dictionary.

In this example we'll simulate a process that involves matching glyphs to one of the 26 letters (A-Z). We'll utilize data containing 20,000 examples of 26 English alphabet capital letters as printed using 20 different randomly reshaped and distorted black and white fonts (Frey and Slate 1991).

Step 2 - exploring and preparing the data

```
letters <- read.csv("letterdata.csv")
str(letters)
```

```
## 'data.frame':    20000 obs. of  17 variables:
##  $ letter: Factor w/ 26 levels "A","B","C","D",..: 20 9 4 14 7 19 2 1 10 13 ...
##  $ xbox  : int  2 5 4 7 2 4 4 1 2 11 ...
##  $ ybox  : int  8 12 11 11 1 11 2 1 2 15 ...
##  $ width : int  3 3 6 6 3 5 5 3 4 13 ...
##  $ height: int  5 7 8 6 1 8 4 2 4 9 ...
##  $ onpix : int  1 2 6 3 1 3 4 1 2 7 ...
##  $ xbar  : int  8 10 10 5 8 8 8 8 10 13 ...
##  $ ybar  : int  13 5 6 9 6 8 7 2 6 2 ...
##  $ x2bar : int  0 5 2 4 6 6 6 2 2 6 ...
##  $ y2bar : int  6 4 6 6 6 9 6 2 6 2 ...
##  $ xybar : int  6 13 10 4 6 5 7 8 12 12 ...
##  $ x2ybar: int  10 3 3 4 5 6 6 2 4 1 ...
##  $ xy2bar: int  8 9 7 10 9 6 6 8 8 9 ...
##  $ xedge : int  0 2 3 6 1 0 2 1 1 8 ...
##  $ xedgey: int  8 8 7 10 7 8 8 6 6 1 ...
##  $ yedge : int  0 4 3 2 5 9 7 2 1 1 ...
```

```
##  $ yedgex: int  8 10 9 8 10 7 10 7 7 8 ...
```

According to the documentation provided by the authors, when the glyphs are scanned, they are converted into pixels and 16 statistical attributes are recorded (features 2 to 17). As expected, letter has 26 levels (A-Z).

SVM learners require all features to be numeric and scaled to a fairly small interval. All features in our dataset are numeric, but some of the ranges appear fairly wide. We'd need to normalize or standardize the data. However, we'll skip this step for now, as the R package we will use for fitting the SVM model will perform the rescaling automatically.

We need to partition the already randomly sorted data into a training set with 80 percent of the examples and a testing set with the remaining 20 percent:

```
letters_train <- letters[1:16000, ]
letters_test <- letters[16001:20000, ]
```

Step 3 - training a model on the data

Packages `e1071` and `klaR` provide nice SVM implementations in R. But we'll be working with the SVM functions in the `kernlab` package, which can be used with the `caret` package we've already been using.

Let's begin by training a simple linear SVM classifier:

```
library(kernlab)
letter_classifier <- ksvm(letter ~ ., data = letters_train, kernel = "vanilladot")
```

```
##  Setting default kernel parameters
```

```
letter_classifier
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Linear (vanilla) kernel function.
##
## Number of Support Vectors : 7037
##
## Objective Function Value : -14.1746 -20.0072 -23.5628 -6.2009 -7.5524 -32.7694 -49.9786 -18.1824 -62.1111
## Training error : 0.130062
```

Step 4 - evaluating model performance

We use our classifier to generate predictions and then compare them to the true values:

```
letter_predictions <- predict(letter_classifier, letters_test)
head(letter_predictions)
```

```
## [1] U N V X N H
## Levels: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

As we haven't specified the type parameter, `type = "response"` default was used and we get a vector containing a predicted letter for each row of values in the test data.

We can now compare the predicted letter to the true letter in the testing dataset using the `table()` function:

```
table(letter_predictions, letters_test$letter)
```

```
##
## letter_predictions   A   B   C   D   E   F
##                  A 144   0   0   0   0   0
##                  B   0 121   0   5   2   0
##                  C   0   0 120   0   4   0
##                  D   2   2   0 156   0   1
##                  E   0   0   5   0 127   3
##                  F   0   0   0   0   0 138
```

The diagonal values indicate the total number of records where the predicted letter matches the true value. Number 5 in second row fourth column indicates that there were five cases where the letter D was misindentified as a B. Looking at each type of mistake individually may reveal some interesting patterns about the specific types of letters the model has trouble with, but this is time consuming. We can simplify our evaluation instead by calculating the overall accuracy.

```
agreement <- letter_predictions == letters_test$letter
table(agreement)
```

```
## agreement
## FALSE  TRUE
##   643  3357
```

```
prop.table(table(agreement))
```

```
## agreement
##   FALSE    TRUE
## 0.16075 0.83925
```

The classifier correctly identified the letter in 3,357 out of the 4,000 test records (about 84% accuracy).

Step 5 - improving model performance

But we can try to improve our results using a more complex kernel function (that is, mapping the data into a higher dimensional space and potentially obtaining a better model fit). A popular convention is to begin with the Gaussian RBF kernel, which has been shown to perform well for many types of data:

```
set.seed(12345)
letter_classifier_rbf <- ksvm(letter ~ ., data = letters_train, kernel = "rbfdot")
letter_predictions_rbf <- predict(letter_classifier_rbf, letters_test)
agreement_rbf <- letter_predictions_rbf == letters_test$letter
prop.table(table(agreement_rbf))
```

```
## agreement_rbf
##   FALSE    TRUE
## 0.06875 0.93125
```

Changing the kernel function we've been able to increase the accuracy of our model from 84% to 93%. We could try to improve it even further using other kernels or varying the value for the cost of constraints parameter *C* so to modify the width of the decision boundary.

References

Frey, Peter W, and David J Slate. 1991. "Letter Recognition Using Holland-Style Adaptive Classifiers." *Machine Learning* 6: 161.

Lantz, Brett. 2015. *Machine Learning with R*. Packt Publishing Ltd.