Modeling the Exposure and the Outcome: Support Vector Machines

Ashley I Naimi

Oct 2022

Contents

In this section, we discuss the basics behind support vector machines. This tutorial is adapted from the (excellent) book by Andriy Burkov (Burkov, 2019). There, he illustrates the general process behind a supervised learning algorithm via support vector machines with a spam email data example. Here, we'll try to provide a bit more depth on the subject, but follow the general overall approach that Burkov takes. To keep our discussion grounded, let's use data the `spam` data available in R from the `kernlab` package:

```
library(kernlab)

data(spam)

spam <- as_tibble(spam)

dim(spam)
```

```
## [1] 4601   58
```

Note these data are not the same as in the book. Rows in this dataset represent emails. The first 48 columns in the dataset contain the frequency of the variable name. For example, the "make" column is a numeric variable that represents the proportion of times the work "make" appears in the email, relative to all the words in the email.

If the variable name starts with num (e.g., num650) the it indicates the frequency of the corresponding number (e.g., 650). The variables 49-54 indicate the frequency of the characters ';', '(', '[', '!', '$', and '#'. The variables 55-57 contain the average, longest and total run-length of capital letters. Variable 58 indicates the type of the mail and is either "nonspam" or "spam".

These data were introduced by Hastie et al (2009). Here is a reproduction of their Table 1.1, of variables that are most different across spam and nonspam emails:

```
spam %>%
    group_by(type) %>%
    summarise(george = mean(george), you = mean(you),
        your = mean(your), hp = mean(free),
```

```
        hpl = mean(hpl), `!` = mean(charExclamation),
        our = mean(our), re = mean(re), edu = mean(edu),
        remove = mean(remove))
```

```
## # A tibble: 2 x 11
##   type     george   you  your     hp     hpl   `!`   our    re    edu  remove
##   <fct>     <dbl> <dbl> <dbl>  <dbl>   <dbl> <dbl> <dbl> <dbl>  <dbl>   <dbl>
## 1 nonspam 1.27     1.27 0.439 0.0736 0.432   0.110 0.181 0.416 0.287  0.00938
## 2 spam    0.00155  2.26 1.38  0.518  0.00917 0.514 0.514 0.125 0.0147 0.275
```

1 Basics of SVMs

First, we'll recode the outcome variable (or label) `type` to be either `+1` ("spam") or `-1` ("not spam"):

```
library(e1071)
```

```
##
## Attaching package: 'e1071'
```

```
## The following object is masked from 'package:Hmisc':
##
##     impute
```

```
library(kernlab)

data(spam)

spam <- spam %>%
    mutate(type = if_else(type == "spam",
        1, -1))

spam %>%
    count(type)
```

```
##   type    n
## 1   -1 2788
## 2    1 1813
```

The regression equation for a SVM looks like this:

$$y = \text{sign}(\mathbf{wx} - b)$$

where y is the outcome, in our case either -1 or $+1$, $\mathbf{wx}$ is a feature vector and parameters. In our dataset above, $\mathbf{wx}$ could be rewritten as:

$$w_1 \times \text{make} + w_2 \times \text{address} + w_3 \times all + \ldots + w_{57} \times \text{capitalTotal} - b,$$

where w_1 to w_{57}, and b (the intercept) are parameter's we'd like to estimate. The sign() operator is a function that returns a -1 if the sign of the argument in the parentheses is negative, and a $+1$ otherwise.

The objective of the fitting procedure is to find a set of values for $\mathbf{w}$ and b that make the output from $\text{sign}(\mathbf{wx} - b)$ as close to the observed values of y as possible. In the book, these values are denoted $\mathbf{w}^*$ and b^*.

Finding the values $\mathbf{w}^*$ and b^* is an **optimization problem**. An illustration of this optimization problem is given in Figure 1.1. in the text, which I've mimicked here:

```
set.seed(123)
library(e1071)
library(kernlab)

# let's create some data that look like
# those in Figure 1.1
dat <- as_tibble(data.frame(x1 = c(rnorm(9,
    8, 1), rnorm(13, 2, 1)), x2 = c(rnorm(9,
    3, 1), rnorm(13, 7, 2.5)), y = c(rep(-1,
    9), rep(1, 13))))

dat
```

```
## # A tibble: 22 x 3
##       x1    x2     y
##    <dbl> <dbl> <dbl>
##  1  7.44  1.97    -1
##  2  7.77  2.27    -1
##  3  9.56  2.37    -1
##  4  8.07  1.31    -1
##  5  8.13  3.84    -1
##  6  9.72  3.15    -1
##  7  8.46  1.86    -1
##  8  6.73  4.25    -1
##  9  7.31  3.43    -1
## 10  1.55  6.26     1
## # ... with 12 more rows
```

```
# here's a simple svm representation of
# figure 1.1

svm_simple = svm(as.factor(y) ~ ., data = dat,
    kernel = "linear", scale = FALSE)

svm_simple
```

```
##
## Call:
## svm(formula = as.factor(y) ~ ., data = dat, kernel = "linear", scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1
##
## Number of Support Vectors:  3
```

```
# Plot Results

png(here("figures", "svm_plot2.png"), width = 450,
    height = 450, units = "px")
plot(svm_simple, dat, symbolPalette = c("red",
    "blue"), color.palette = cm.colors)
dev.off()
```

```
## pdf
##   2
```

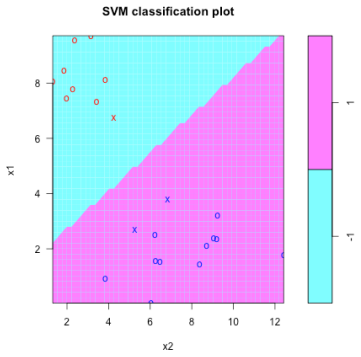

Figure 1: SVM Example in Two Dimensions, replicated from page 6 of The 100 Page ML Book, replicated via plot.svm

(#fig:fig:svm-simple)

Let's use this basic figure to identify some important concepts behind support vector machines. The first to understand is the maximal margin, which is the space that's centered on the line separating the cases from the noncases. The line separating cases from noncases in the Figure is generally referred to as the separating hyperplane.[1] The margin space is defined as a function of the *support vectors*, which are denoted x in Figure **??**. Support vectors are the datapoints that yield the largest margin between the cases and noncases.

[1] Note we have a line in two dimensions, a plane in three dimensions, and a hyperplane in >3 dimensions.

The objective of the optimization algorithm for a support vector machine is to find the values of $\mathbf{w}$ that separate the cases and the noncases by the largest margin. Because the margin is defined as $\frac{2}{||\mathbf{w}||}$, we need to find the smallest values of $\mathbf{w}$ that are compatible with the data. How do we measure compatibility with the data? We can use constraints such as $y_i(\mathbf{w}\mathbf{x}_i - b) \geq 1$.

In this simple linear example, it's easier to understand the process behind SVMs. The problem is that one would not typically encounter such a clean

dataset in real life, where the data points are perfectly separated by a single straight line. So let's look at the spam data to get a more realistic working example.

Here is a plot similar to the simple example above, but with the x axis representing the frequency of the word "our", and the y axis representing the frequency of the word "you".

```
p1 <- spam %>%
    ggplot(.) + theme(text = element_text(size = 25)) +
    geom_point(aes(y = you, x = our, group = as.factor(type),
        color = as.factor(type)), size = 5,
        alpha = 0.25)

png(here("figures", "svm_plot3.png"), width = 450,
    height = 450, units = "px")
p1
dev.off()
```

```
## pdf
##   2
```

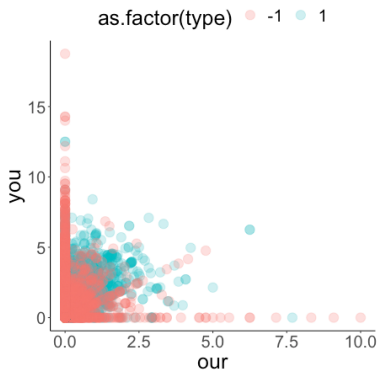

Figure 2: More complex data for fitting SVMs, showing spam and nonspam emails as a function of the frequency of 'our' and 'you' in the email.

(#fig:fig:svm-simple2)

Figure **??** shows a much more complicated relation between the frequency of the words "our" and "you", and whether the email is spam or not. In this particular case, it seems clear that a linear separator will do a very poor job at classifying the emails. It turns out that there are several other types of (curvi-

linear) separators. Two (of many) options are polynomial separator, and the radial basis function separator.

```
spam2 <- spam %>%
    select(type, you, our)

svm_polynomial = svm(as.factor(type) ~ .,
    data = spam2, kernel = "polynomial",
    gamma = 1, cost = 1)

svm_radial = svm(as.factor(type) ~ ., data = spam2,
    kernel = "radial", gamma = 1, cost = 1)

# Plot Results
png(here("figures", "svm_plot4.png"), width = 450,
    height = 450, units = "px")
plot(svm_polynomial, spam2, symbolPalette = c("red",
    "blue"), color.palette = cm.colors)
dev.off()
```

```
## pdf
##   2
```

```
png(here("figures", "svm_plot5.png"), width = 450,
    height = 450, units = "px")
plot(svm_radial, spam2, symbolPalette = c("red",
    "blue"), color.palette = cm.colors)
dev.off()
```

```
## pdf
##   2
```

In the code above, we fit two different support vector machines. The first uses a **polynomial** separator, and the second uses a **radial basis function**. In the process, we've also introduced two different parameters that we've set equal to one in both cases: **gamma**, and **cost**. The Figures below demonstrate how emails are classified when these two types of separators are used.

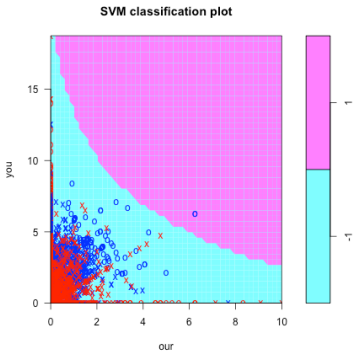

Figure 3: More complex data for fitting SVMs, showing spam and nonspam emails as a function of the frequency of 'our' and 'you' in the email, classified with a polynomial separator in SVM

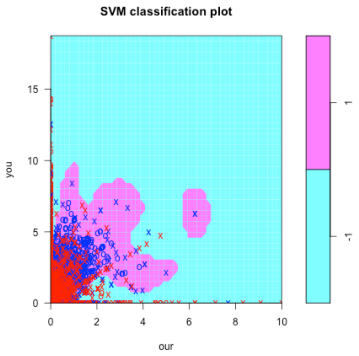

Figure 4: More complex data for fitting SVMs, showing spam and nonspam emails as a function of the frequency of 'our' and 'you' in the email, classified with a radial basis function separator in SVM

As the names imply, the polynomial separator fits a polynomial curve to the data in order to separate cases from noncases. In the case of the `svm` function from the `e1071` package, the default polynomial is cubic.[2] Alternatively, the radial basis function encircles to the data to separate cases from noncases.

With both the polynomial and radial basis functions, the two additional parameters that are important to consider are the `cost` and `gamma` arguments.

The `cost` argument essentially captures the price you want to pay for incorrectly classifying an email. reflect a penalty for placing outcomes on the wrong side of the line, plane, or hyperplane. The higher the cost, the more careful the algorithm will be at drawing these, but the greater the risk of overfitting the data. We can see this in action if we set `cost= = 100` and refit the algorithm:

[2] This default can be changed using the `degree` argument in the `svm` function.

```
svm_radial = svm(as.factor(type) ~ ., data = spam2,
    kernel = "radial", gamma = 1, cost = 100)
```

```
png(here("figures", "svm_plot6.png"), width = 450,
    height = 450, units = "px")
plot(svm_radial, spam2, symbolPalette = c("red",
    "blue"), color.palette = cm.colors)
dev.off()
```

```
## pdf
##   2
```

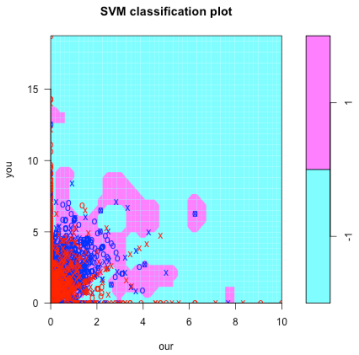

Figure 5: More complex data for fitting SVMs, showing spam and nonspam emails as a function of the frequency of "our" and "you" in the email, classified with a radial basis function separator in SVM with a cost of 100

Gamma is the kernel parameter for the polynomial or radial basis function used to define the line, plane, or hyperplane. In effect, it determines how flexible (or "wiggly") the line, plane, or hyperplane will be. Higher values of gamma result in a closer fit to the data (but incurs a penalty that results from overfitting, which we'll discuss later). We can see this gamma parameter in action if we set `gamma = 100` and refit the algorithm:

```
svm_radial = svm(as.factor(type) ~ ., data = spam2,
    kernel = "radial", gamma = 100, cost = 1)

png(here("figures", "svm_plot7.png"), width = 450,
    height = 450, units = "px")
plot(svm_radial, spam2, symbolPalette = c("red",
    "blue"), color.palette = cm.colors)
dev.off()
```

```
## pdf
```

```
##      2
```

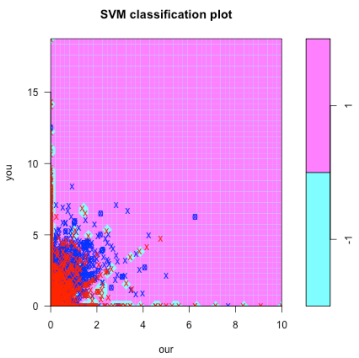

Figure 6: More complex data for fitting SVMs, showing spam and nonspam emails as a function of the frequency of “our” and “you” in the email, classified with a radial basis function separator in SVM with a gamma value of 100

These parameters (cost and gamma) are often referred to as hyperparameters or tuning parameters, since they have to be selected, and they determine how well the algorithm will perform. Hyperparameters differ from regular old parameters in that they have to be set for the algorithm to work, instead of simply estimated from the data in the usual way (in fact, they can be “estimated” with data, but this requires some additional work). Hyperparameters are not unique to SVMs, but are generally found in any machine learning algorithm. So naturally an important question arises: what values should one choose to get a good performing algorithm? This question is so important (since sub-optimal selection can lead to disastrous performance), that it’s spawned a whole set of ML algorithms and techniques of its own. We will cover these methods in depth over the course of this book club.

References

A Burkov. *The Hundred Page Machine Learning Book*. Andriy Burkov, 2019.

Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, NY, 2009.