Machine Learning for Finance Applications - Support Vector Machines

Brian Clark

2019

Contents

1 Introduction

This chapter introduces support vector machines (SVM) based methods for finance applications. The topics closely follow Chapter 9 of James et al. (2013).[1] Another resource is Chapter 12 of Friedman, Hastie, and Tibshirani (2001), which provides a more technical discussion.[2] SVMs have been around for decades and are commonly used in many machine learning (ML) applications. Unlike trees, SVMs are intended to be classifiers.

The advantage of SVMs is that they are flexible and can be highly non-linear. Of course the downside to the flexibility is that it leads to a penchant for over-fitting. This vignette will follow the notation of James et al. (2013) and provide a brief introduction to SVM's followed by some applicaitons in R. Similar to other ML algorithms, SVMs can be enhanced by ensamble methods such as bagging and boosting.

2 Support Vector Machines Explained

2.1 Hyperplanes and the Maximal Margin Classifier

SVMs are derived from the idea of a hyperplane. A hyperplane in a $p-dimension$ space is defined as a "flat affine subspace of dimension $p-1$" (page 338, James et al. (2013)). For teh case of two dimensions, this means a line; three dimensions, a flat plane; etc. We can define a hyperplane in p dimensions as:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p = 0. \tag{1}$$

[1] The full text is available at http://www-bcf.usc.edu/ gareth/ISL/.
[2] The full text is available at https://web.stanford.edu/ hastie/Papers/ESLII.pdf.

SVMs are derived from the idea that if a hyperplane exists and the data is linearly separable, then there are an infinte number of hyperplances that can separate the data (note that this is unrealistic since any real dataset will NOT be completely linearly separable). However, it's a good place to start to understand where SVM's come from.

Let's say we have a datapoint $X = (X_1, X_2, ..., X_p)^T$. If that point satisfies Eq. (1), it will lie on the hyperplane. If not, there are one of two possibilities:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p > 0, \tag{2}$$

or

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p < 0. \tag{3}$$

In other words, the point is on one side of the hyperplane or the other. For example (again unrealistic), all defaulted accounts might satisfy Eq. (2) and all good accounts might satisfy Eq. (3).

If the data is completely seperable (meaning that there exists such a line that perfectly classifies the data), then we could choose a classification rule based on a hyperplane. Because there will be an infinite number of hyperplanes that separate the data (if one exists), we want the one that best separates the data, i.e., the "Maximal Margin Classifier."

How do we get this maximial margin classifier? Ideally, we want the hyperplane that maximizes the distance between the nearest points. The term *margin* refers to the shrotest distance of all the datapoints to the hyperplane. The maximal margin classifier refers to the hyperplane that maximizes the minimum distance. Essentially, the maximal margin hyperplane represents the locus of points that make up the midpoint between the widest set of parallel hyperplanes that separate the data. These hyperplanes eventually intersect with certain datapoints, which we term *support vectors*. It is important to note that the support vectors are the only points that affect the classifier.

Section 9.1.4 of James et al. (2013) outlines how to find the maximal margin classifier. However, since it rarely exists in real data, it suffices to understand the intuition.

2.2 Support Vector Classifiers

In most applications, the data cannont be completely separated - or at least it is not optimal to separate all the data. Therefore, we try to find the best classifier and call this the *support vector classifier*. The basic idea to to classify as many instances as possible. As with most approaches, we sacrifice some bias for better variance properties.

The support vector classifier can be written as an optimization problem as follows (again following the notation of James et al. (2013)):

$$\max_{\beta_0,...,\beta_p,\epsilon_1,...,\epsilon_n,M} M \tag{4}$$

subject to,

$$\sum_{j=1}^{p} \beta_j^2 = 1, \tag{5}$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i,2} + ... + \beta_p x_{ip}) \geq M(1 - \epsilon_i), \tag{6}$$

$$\epsilon_i \geq 0, \sum_{i=1}^{n} \epsilon_i \leq C, \tag{7}$$

where, M is the margin and C is the tuning parameter. $\epsilon_1, ..., \epsilon_n$ are called slack parmaters and tell which side of the margin observations are located:

- $\epsilon_i = 0$: the observation is on the correct side of the margin
- $\epsilon_i > 0$: the observation is on the wrong side of the margin
- $\epsilon_i > 1$: the observation is on the wrong side of the hyperplane (and margin).

Combined with Eqs. (4) - (7), the definition of ϵ_i shows why C is the tuning parameter. As C increases, the algortihm allows more observaitons to be mis-classified, thus reducing the variance of the classifier. At the other extreme, as C goes to zero, we get the maximal margin hyperplane (if it exists). C can be chosen via cross valiation or a similar method.

The term support vector again comes up because only the instances within the margin or those that are misclassifed affect the classifier. As C increases so do the number of support vectors which lowers the variance (because more instances are used in fitting the model). The cost is that some bias is sacrificed. The discussion thus far has implicitly focused on linear hyperplanes.

2.3 Support Vectore Machines

The basic idea of support vector machines (SVM's) is to expand the feature space in such a way to make the decision boundary non-linear. The most simplistic case would be a parallel to making linear regression non-linear by including transformed variables. For exmaple, we could transform the feature space by a factor of two and make the decision boundary quadratic. It would then amount to solving the following problem:

$$\max_{\beta_0, \beta_{11}, \beta_{12}, ..., \beta_{p1}, \beta_{p2}, \epsilon_1, ..., \epsilon_n, M} M \tag{8}$$

subject to,

$$y_i(\beta_0 + \sum_{j=1}^{p} \beta_{j1} x_{ij} + \sum_{j=1}^{p} \beta_{j2} x_{ij}) \geq M(1 - \epsilon_i), \tag{9}$$

$$\epsilon_i \geq 0, \sum_{i=1}^{n} \epsilon_i \leq C, \tag{10}$$

$$\sum_{j=1}^{p} \sum_{k=1}^{2} \beta_{jk}^2 = 1. \tag{11}$$

Similar to a regression approach, the above problem will result in a linear decision boudary in the enhanced feature space. However, when mapped back to the original $X_1, X_2, ..., X_p$ space it will be non-linear. This is the central idea behind SVMs - to expand the feature space such that non-linear decision boundaries can be estimated. Again, the degree of non-linearity will need to be tuned to control the bias-variance tradeoff.

2.3.1 Kernels

The basic idea is to expand the feature space so when the decision boundary is mapped back to the original feature space, it becomes nonlinear. See sectinm 9.3 of James et al. (2013) for details. Kernels are described **here** (it's a Wikipedia link - but it's a good source for learning). Kernals are one method for expanding the feature space. A convenient property of SVMs and kernels is that they only require the inner products of the observations - which saves on computational time.

The inner product of two vectors of length n is defined as $\langle x, y\rangle = \sum_{i=1}^{n} x_i y_i$ (in R, this is done using basic math operations; i.e., `Inner.Product <- sum(x*y)`). In the context of SVMs, we can defined the inner product of two observations as:

$$\langle x_i, x_{i'}\rangle = \sum_{j=1}^{p} x_{ij} x_{i'j} \tag{12}$$

Generalizing Eq. (9), we can write the linear SV classifier as follows:

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x_i, x_{i'}\rangle, \tag{13}$$

where there are n parameters α_i, one for each training observation. In other words, there is a huge number of parameters. However, the solution is simplified because α_i's are only non-zero for the support vectors. That is, the α_i's associated with the correctly classified instances are all equal to zero. Defining S as the set of indices of support vectors, Eq. (13) simplifies to the following:

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i \langle x, x_i\rangle, \tag{14}$$

Kernels come in as generalizations of the simple inner products,

$$K(x_i, x_{i'}). \tag{15}$$

A kernel is a function that measures the similarity between two observations. Some common kernels are as follows:

1. Linear kernal:
$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} x_{i'j} \tag{16}$$

2. Polynomial kernel of degree d:
$$K(x_i, x_{i'}) = (1 + \sum_{j=1}^{p} x_{ij} x_{i'j})^d \tag{17}$$

3. Radial kernel:
$$K(x_i, x_{i'}) = exp\left[-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2\right] \tag{18}$$

4. Neural network kernel:

$$K(x_i, x_{i'}) = tanh(\kappa_1 \sum_{j=1}^{p} x_{ij} x_{i'j}) + \kappa_2) \tag{19}$$

We can use any of the above kernels to define the SVM classifier function for each observation x as follows:

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i K(x, x_i). \tag{20}$$

The advantage using kernels as opposed to transformations of the actual feature space as in Eqs. (8) - (11) is computational. For details on why this is so, see Section 9.3 of James et al. (2013) and for a much more detailed descirption of the problem, Section 12.3 of Friedman, Hastie, and Tibshirani (2001).

2.3.2 SVM's as *Loss + Penalty* Funtions and Relation to Logistic Regression

SVM's can be re-written in the usual *Loss + Penalty* formulation (see pages 420-428 of Friedman, Hastie, and Tibshirani (2001) for a detailed description of why this is so and Section 9.5 of James et al. (2013) for a ess technical discussion). In particular, the problem descibed by Eq.'s (8) - (11) for fitting the support vector classifier $f(X) = \beta_0 + \beta_1 X_1 + ... + \beta_p X_p$ can be written as follows:

$$\min_{\beta_0, \beta_1, ..., \beta_p} \left\{ \sum_{i=1}^{n} [1 - y_i f(x_i)]_+ + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}, \tag{21}$$

where, λ is a non-negative tuning parameter that controls the bias-variance tradeoff. Large values of λ tip the scale in favor of a more biased but lower variance classifier. Small values of λ do the opposite. Practically speaking, λ is calibrated using a cross-validation or similar approach.

To show the relation between SVM's and logistic regresson, let's start by rewriting the the problem in the *Loss + Penalty* form:

$$\min_{\beta_0, \beta_1, ..., \beta_p} \left\{ L(\mathbf{X}, \mathbf{y}, \beta) + \lambda P(\beta) \right\} \tag{22}$$

where $L(\mathbf{X}, \mathbf{y}, \beta)$ is some loss function and $P(\beta)$ is the penalty function. Notice that in Eq. (21), $P(\beta)$ is the ridge penalty. Tying back to logistic regression with ridge and lasso penalties, the logistic regression loss function is

$$L(\mathbf{X}, \mathbf{y}, \beta) = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij} \beta_j \right)^2. \tag{23}$$

The lasso and ridge penalty functions are:

$$P_{Ridge}(\beta) = ||\beta||^2 \tag{24}$$

and

$$P_{Lasso}(\beta) = ||\beta||^1. \tag{25}$$

The loss function for the SVM classifier is:

$$L(\mathbf{X}, \mathbf{y}, \beta) = \sum_{i=1}^{n} [1 - y_i(\beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip})]_+. \quad (26)$$

The above SVM loss function is called the *hinged* loss function because it has a hinged shape (similar to the payout of an option). The logistic regression loss function is similar, except it has a smoother shape. The relation between the two is that observations far away from the decision boundary have liitle impact on the classifier. The SVM loss function takes this to the extreme in that only the support vectors have any influence where as observations that are correctly classified have no impact. That is, for any observation x_i where $y_i(\beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip}) \geq 1$, $L(\mathbf{X}, \mathbf{y}, \beta) = 0$. Figure 9.12 of James et al. (2013) plots the relationship.

The takeaway is that SVM's are in fact closely related to logistic regression and other methods. Also, formulating the problem as a loss plus penalty problem means that we can use similar model selection and calibration techniques to the other methods ML methods such as logistic regression, decision trees, etc. The remainder of the document shows how to implement SVM's using R.

3 Implementation of SVM's using R

The examples below follow the SVM lab in Section 9.6 of James et al. (2013). We will continue to use the Fannie Mae mortgage data.

3.1 Import the Mortgage Data

First, initialize the R session and load the data. Note that you can skip this subsection if you have already processed this data in one of the other modules.

```
rm(list=ls()) # clear the memory
setwd("C:/Users/CLARKB2/Documents/Classes/ML Course")
library("ggplot2")
library("reshape")
library("plm")
library("rpart")
library("zoo")
library("plyr")
library("dplyr")
library("stringr")
library("reshape2")
library("ggplot2")
library("pander")
library("DataCombine")
library("plm")
library("quantmod")

# Import the mortgage data:
load("Mortgage_Annual.Rda")
```

The next step is to process the data. Within the .Rda file is a data frame called `p.mort.dat.annual`. As a matter of preference, rename `p.mort.dat.annual` as `df`. Set the data as a panel dataset (`pdata.frame()`) based on `LOAN_ID` and `year`.

```
# Rename the data (matter of preference):
df <- p.mort.dat.annual
rm(p.mort.dat.annual)
```

```
df <- pdata.frame(df, index=c("LOAN_ID","year"),
                  stringsAsFactors = F)

# Print the class of the variable:
class(df)
```

```
## [1] "pdata.frame" "data.frame"
```

Next, we can generate variables that we need. First, define default as the first instance of 180+ days delinquent, which is given in the data by `F180_DTE` (to refer to the variable in `df`, use `df$F180_DTE`). `F180_DTE` is the date in which a loan first becomes 180+ delinquent. If the loan never becomes delinquent, it is missing (i.e., `NA`). To make the default variable, first find the indices where `df$F180_DTE == df$date` and save it as a vector, `tmp`. The line `df$def[tmp] <- 1` sets the new variable `def = 1` for the year in which the loan defaults.

```
# Generate Variables we want:
# 1. Default 1/0 indicator (180+ DPD):
df$def <- 0
# Save the indices (rows) of
tmp <- which(df$F180_DTE == df$date)
df$def[tmp] <- 1
```

We may want to generate some other variables. For example, the variable `NUM_UNIT` gives the number of units is a house. Use `table(df$NUM_UNIT)` to print a frequency table of values of the number of units per house. Then, define a new variable to be a dummy if the number of units is greater than one (`MULTI_UN`).

```
# 2. Replace NUM_UNIT with MULTI_UNIT dummy:
table(df$NUM_UNIT)
```

```
##
##      1      2      3      4
## 305028   6452   1051   1085
```

```
df$MULTI_UN <- 0
tmp <- which(df$NUM_UNIT > 1)
df$MULTI_UN[tmp] <- 1
```

Finally, we can compress the data down to a single observation per loan. If you wanted to conduct a time-series or panal data analysis, you would skip this step. First, print the number of unique loans.

```
# 3. Count the number of loans:
print(length(unique(df$LOAN_ID)))
```

```
## [1] 66704
```

Next, compress the data down to a single observation per loan. First, we need to generate a variable equal to one if the loan ever defaulted. We can do this using the `plm` package and grouping the data by `LOAN_ID`. Make a new dataset `df.annual` with a few additional variables: i) `def.max` and ii) `n` which is the row number to be used for keeping observaitons.

```
# Compress the data to single loans:
df.annual <-df %>%
  group_by(LOAN_ID) %>%
  mutate(def.max = max(def)) %>%
  mutate(n = row_number()) %>%
  ungroup()
```

```
## Warning: `as_dictionary()` is soft-deprecated as of rlang 0.3.0.
```

```
## Please use `as_data_pronoun()` instead
## This warning is displayed once per session.
```

```
## Warning: `new_overscope()` is soft-deprecated as of rlang 0.2.0.
## Please use `new_data_mask()` instead
## This warning is displayed once per session.
```

```
## Warning: The `parent` argument of `new_data_mask()` is deprecated.
## The parent of the data mask is determined from either:
##
##   * The `env` argument of `eval_tidy()`
##   * Quosure environments when applicable
## This warning is displayed once per session.
```

```
## Warning: `overscope_clean()` is soft-deprecated as of rlang 0.2.0.
## This warning is displayed once per session.
```

```
# Print the variable names in df.annual
names(df.annual)
```

```
##  [1] "year"               "ZIP_3"              "V1"
##  [4] "LOAN_ID"            "ORIG_CHN"           "Seller.Name"
##  [7] "ORIG_RT"            "ORIG_AMT"           "ORIG_TRM"
## [10] "ORIG_DTE"           "FRST_DTE"           "OLTV"
## [13] "OCLTV"              "NUM_BO"             "DTI"
## [16] "CSCORE_B"           "FTHB_FLG"           "PURPOSE"
## [19] "PROP_TYP"           "NUM_UNIT"           "OCC_STAT"
## [22] "STATE"              "MI_PCT"             "Product.Type"
## [25] "CSCORE_C"           "MI_TYPE"            "RELOCATION_FLG"
## [28] "Monthly.Rpt.Prd"    "Servicer.Name"      "LAST_RT"
## [31] "LAST_UPB"           "Loan.Age"           "Months.To.Legal.Mat"
## [34] "Adj.Month.To.Mat"   "Maturity.Date"      "MSA"
## [37] "Delq.Status"        "MOD_FLAG"           "Zero.Bal.Code"
## [40] "ZB_DTE"             "LPI_DTE"            "FCC_DTE"
## [43] "DISP_DT"            "FCC_COST"           "PP_COST"
## [46] "AR_COST"            "IE_COST"            "TAX_COST"
## [49] "NS_PROCS"           "CE_PROCS"           "RMW_PROCS"
## [52] "O_PROCS"            "NON_INT_UPB"        "REPCH_FLAG"
## [55] "TRANSFER_FLAG"      "CSCORE_MN"          "ORIG_VAL"
## [58] "PRIN_FORG_UPB"      "MODTRM_CHNG"        "MODUPB_CHNG"
## [61] "Fin_UPB"            "modfg_cost"         "C_modir_cost"
## [64] "C_modfb_cost"       "Count"              "LAST_STAT"
## [67] "lpi2disp"           "zb2disp"            "INT_COST"
## [70] "total_expense"      "total_proceeds"     "NET_LOSS"
## [73] "NET_SEV"            "Total_Cost"         "Tot_Procs"
## [76] "Tot_Liq_Ex"         "LAST_DTE"           "FMOD_DTE"
## [79] "FMOD_UPB"           "FCE_DTE"            "FCE_UPB"
## [82] "F180_DTE"           "F180_UPB"           "VinYr"
## [85] "ActYr"              "DispYr"             "MODIR_COST"
## [88] "MODFB_COST"         "MODTOT_COST"        "d.HPI"
## [91] "date"               "n"                  "n.obs"
## [94] "n.year"             "n.year.max"         "def"
## [97] "MULTI_UN"           "def.max"
```

Finally, we can save only one observation per loan.

```
# keep one obs per loan:
tmp <- which(df.annual$n == 1)
df.annual <- df.annual[tmp,]
dim(df.annual)
```

```
## [1] 66704    98
```

Notice that the number of rows is equal to the number of unique loans as shown above. Now, retain only the variables needed for the analysis.

```
# Keep only relevant variables for default analysis:
my.vars <- c("ORIG_CHN","ORIG_RT",
             "ORIG_AMT","ORIG_TRM","OLTV",
             "DTI","OCC_STAT",
             "MULTI_UN",
             "CSCORE_MN",
             "ORIG_VAL",
             "VinYr","def.max")
df.model <- subset(df.annual,select=my.vars)
names(df.model)
```

```
##  [1] "ORIG_CHN"  "ORIG_RT"   "ORIG_AMT"  "ORIG_TRM"  "OLTV"
##  [6] "DTI"       "OCC_STAT"  "MULTI_UN"  "CSCORE_MN" "ORIG_VAL"
## [11] "VinYr"     "def.max"
```

```
# Print the number of defaults/non-defaults
table(df.model$def.max)
```

```
##
##     0     1
## 64397  2307
```

```
tmp <- table(df.model$def.max)
df.rate <- tmp[2]/sum(tmp)*100
message(sprintf("The default rate is: %4.2f%%",df.rate))
```

```
## The default rate is: 3.46%
```

The last line prints the default rate. You can control the formatting just as you would in Matlab. The next step is to plot the data. We will use the `ggplot2()` package.

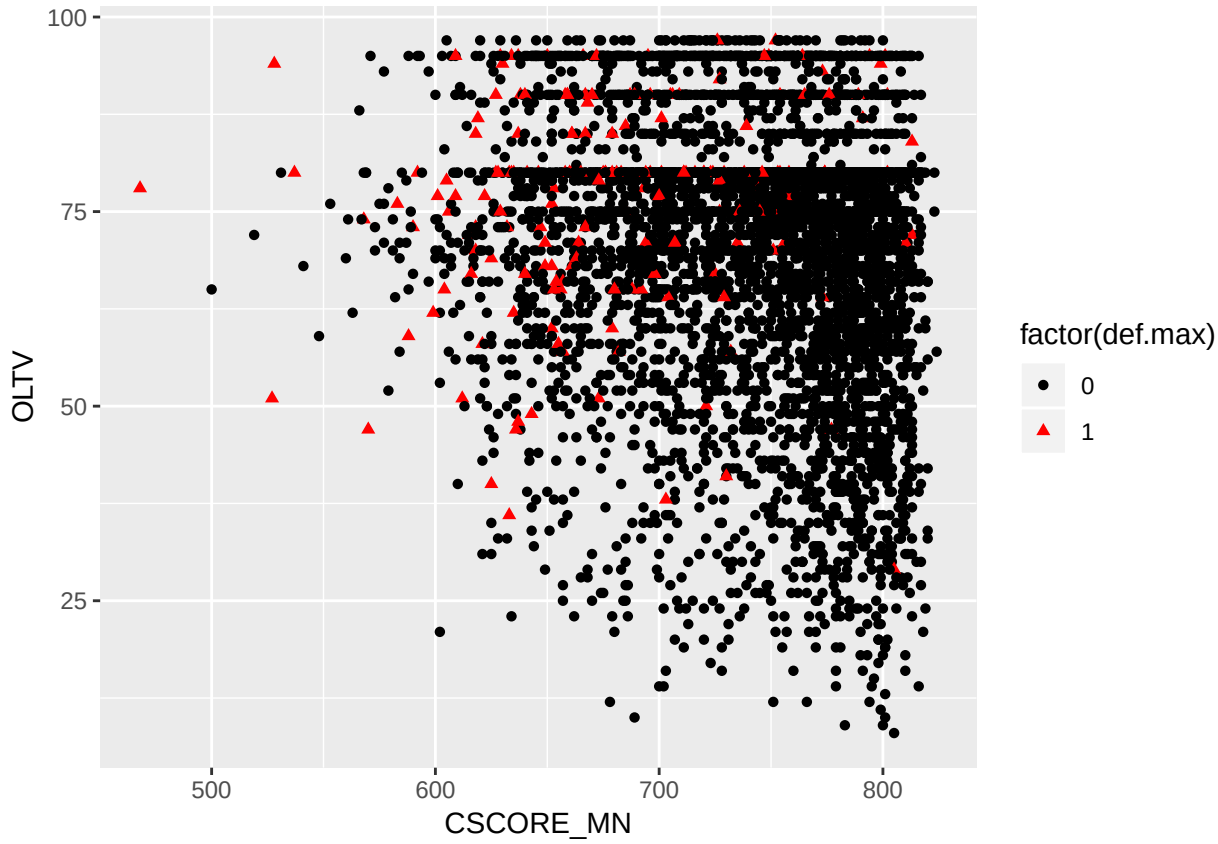
```
# --------------------
# Plot the data:
# Set the colors for the points:
mycolor <- c("black","red")

# Generate a small sample for plotting:
df.model.small <- df.model[sample(dim(df.model)[1],5000),]

# Plot Defaults vs. CSCORE_MN and OLTV:
sp <- ggplot(df.model.small, aes(x=CSCORE_MN, y=OLTV,
                                 color=factor(def.max))) +
  geom_point(aes(shape=factor(def.max))) +
  scale_fill_manual(values=mycolor) +
  scale_colour_manual(values=mycolor)
```

The above code saves the plot object as `sp`. To show the plot, simply print it. `ggplot()` has a nice feature that we can simply add to plots (similar to `hold on` in Matlab) as follows. As an example, we can make the

size of the plotted points larger for defaulted loans (i.e., make the red dots bigger).

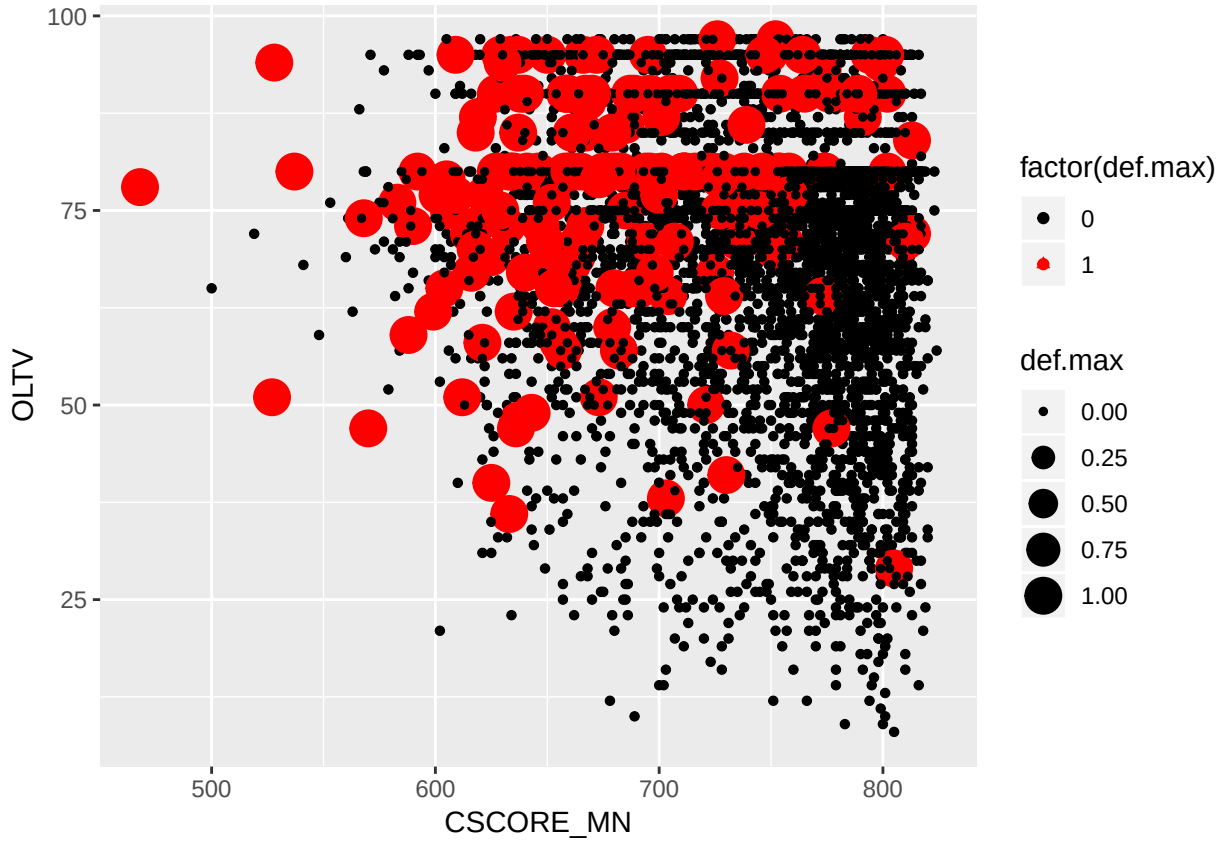
```
print(sp)
```

```
## Warning: Removed 16 rows containing missing values (geom_point).
```




```
# Add some new formatting:
sp <- sp + geom_point(aes(size=def.max))
print(sp)
```

```
## Warning: Removed 16 rows containing missing values (geom_point).

## Warning: Removed 16 rows containing missing values (geom_point).
```



Notice that the defaults tend to cluster more hevily for higher original LTV's and lower credit scores (note that `CSCORE_MN` is the minimum credit score of the borrowers). Finally, we can remove all unnecessary ojects.

```
# Print the objects in memory:
ls()
```

```
## [1] "df"             "df.annual"      "df.model"       "df.model.small"
## [5] "df.rate"        "my.vars"        "mycolor"        "sp"
## [9] "tmp"
```

```
# Remove all but df.model
rm(list=setdiff(ls(), "df.model"))
ls()
```

```
## [1] "df.model"
```

Now we only have a single object in memory - `df.model`. Note that in many ML applicaitons, memory becomes crucial. For this example, the data is pretty small so memory isn't a concern but in "big data" applications, the above commands could prove useful.

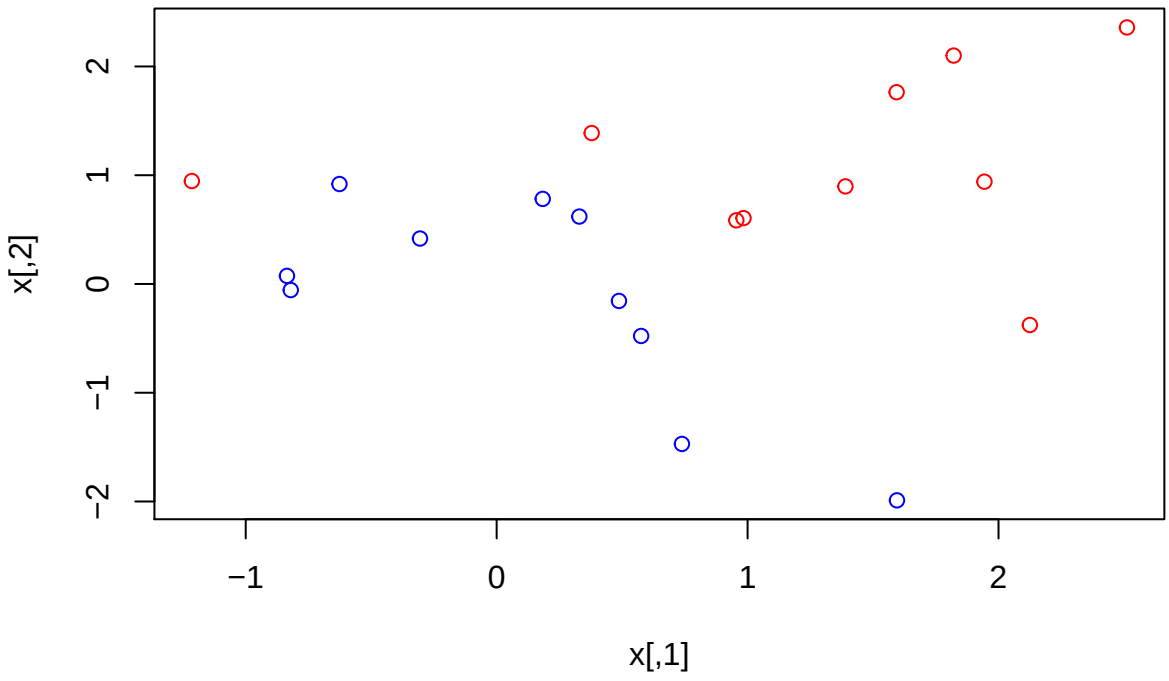
3.2 Implementing SVM's using CARET

This section follows the examples in Section 9.6 of James et al. (2013) except we use the `caret` package. The `caret` packge is a wrapper that calls underlying packages such as `e1071` which is used by James et al. (2013). In other words, we will replicate the lab in Section 9.6 of James et al. (2013) using the `caret` wrapper. The reason is that `caret` is a wrapper for many ML algorithms so getting used to its syntax will undoubtedly save time and frustration in the long run.

```
library("caret")
library("ggplot2")
library("e1071")
```

Start with a two dimensional example.

```
# Set the RNG seed
set.seed(1)

# Generate some random normal data (and store as a matrix):
x <- matrix(rnorm(20*2),ncol=2)

# generate vector  y = +/-1
y <- c(rep(-1,10),rep(1,10))

# SHift half the observations (to make them close to linearly separable)
x[y==1,] <- x[y==1] + 1

# Plot the data:
plot(x, col=(3-y))
```



Clearly the classes are not linearly separable. The next step is to fit a SV classifier but first have to make teh response (y) a factor.

```
class(x)
```

```
## [1] "matrix"
```

```r
class(y)
```

```
## [1] "numeric"
```

```r
dat <- data.frame(x=x,y=as.factor(y))

# Fir a model using e1071 via caret:
trControl <- trainControl(method="repeatedcv",number=10,repeats=10)
grid <- expand.grid(.cost=10)
svm.fit <- train(y~., data=dat, trControl=trControl,
                 method="svmLinear2",
                 tuneGrid=grid)
# List the variables in the summary:
names(svm.fit)
```

```
##  [1] "method"       "modelInfo"    "modelType"    "results"
##  [5] "pred"         "bestTune"     "call"         "dots"
##  [9] "metric"       "control"      "finalModel"   "preProcess"
## [13] "trainingData" "resample"     "resampledCM"  "perfNames"
## [17] "maximize"     "yLimits"      "times"        "levels"
## [21] "terms"        "coefnames"    "xlevels"
```

```r
# Print the support vectors:
svm.fit$finalModel$SV
```

```
##           x.1         x.2
## 1  -1.2235317  0.39564544
## 2  -0.4709149  0.26839033
## 5  -0.3354003  0.11744992
## 7  -0.1886843 -0.60383768
## 14 -1.7700389  0.42095638
## 16  0.2457719  0.08506883
## 17  0.2724758  0.10432305
```

```r
# Summarize the model to get some basic model info:
summary(svm.fit)
```

```
##
## Call:
## svm.default(x = as.matrix(x), y = y, kernel = "linear", cost = param$cost,
##     probability = classProbs)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##       gamma:  0.5
##
## Number of Support Vectors:  7
##
##  ( 4 3 )
##
##
## Number of Classes:  2
##
```

```
## Levels:
##  -1 1
```

Note, to find the model output for which you are looking print `str(svm.fit)`. This will show all the objects in the model file. I dod not print it here because it is very long.

The above brief summary says that for cost=10, there are seven support vectors: four (4) in class x_1 and three (3) in class x_2.

Let's repeat the above example for a small cost parameter, `cost = 0.1`.

```
# Fit a model using e1071 via caret:
trControl <- trainControl(method="repeatedcv",number=10,repeats=10)
grid <- expand.grid(.cost=0.1)
svm.fit.dot1 <- train(y~., data=dat, trControl=trControl,
                 method="svmLinear2",
                 tuneGrid=grid)
# List the variables in the summary:
names(svm.fit)
```

```
##  [1] "method"       "modelInfo"    "modelType"    "results"
##  [5] "pred"         "bestTune"     "call"         "dots"
##  [9] "metric"       "control"      "finalModel"   "preProcess"
## [13] "trainingData" "resample"     "resampledCM"  "perfNames"
## [17] "maximize"     "yLimits"      "times"        "levels"
## [21] "terms"        "coefnames"    "xlevels"
```

```
# Print the support vectors:
svm.fit.dot1$finalModel$SV
```

```
##           x.1         x.2
## 1  -1.2235317  0.39564544
## 2  -0.4709149  0.26839033
## 3  -1.4178646 -0.38961436
## 4   0.8405600 -2.30894999
## 5  -0.3354003  0.11744992
## 7  -0.1886843 -0.60383768
## 9  -0.1066010 -0.90361072
## 10 -0.9252475 -0.07029192
## 12  0.6496993  0.37540502
## 13 -0.2896433  0.83150681
## 14 -1.7700389  0.42095638
## 15  1.3326289 -0.80960179
## 16  0.2457719  0.08506883
## 17  0.2724758  0.10432305
## 18  1.1643837  0.41583390
## 20  0.8392784  1.18070621
```

```
# Summarize the model to get some basic model info:
summary(svm.fit.dot1)
```

```
##
## Call:
## svm.default(x = as.matrix(x), y = y, kernel = "linear", cost = param$cost,
##     probability = classProbs)
##
##
## Parameters:
```

```
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.1
##       gamma:  0.5
##
## Number of Support Vectors:  16
##
##  ( 8 8 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

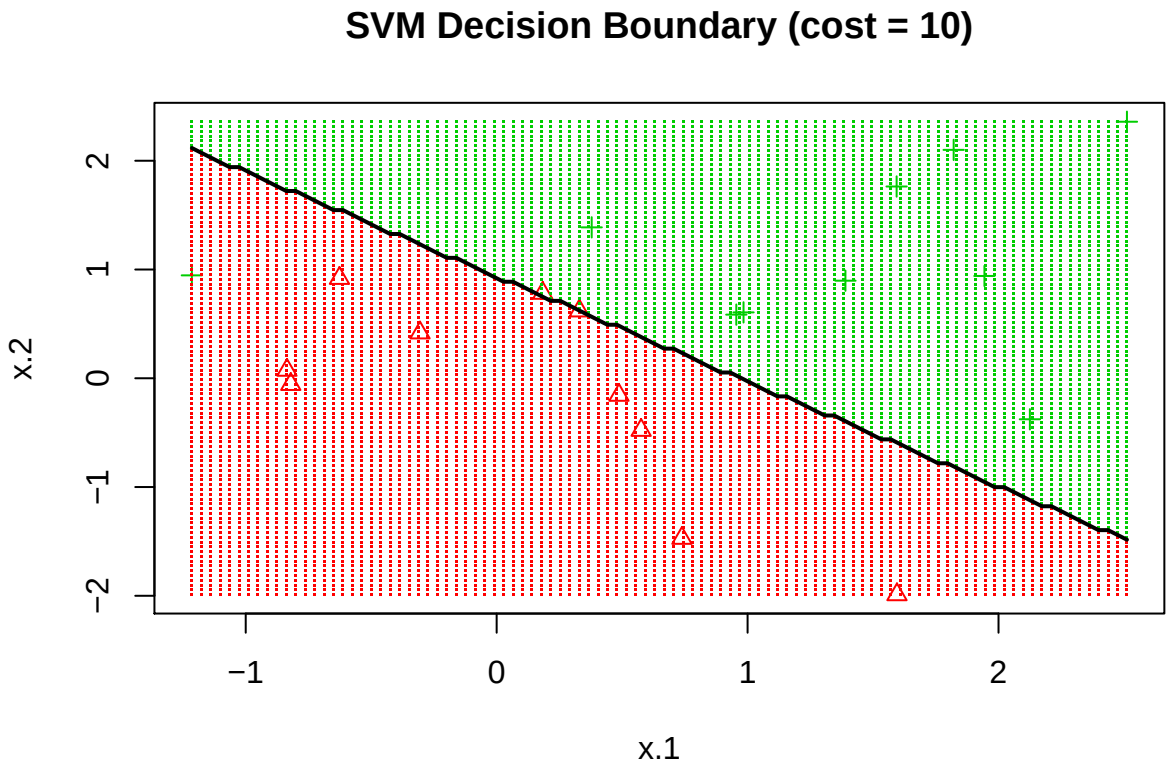
There are now more support vectors, because the cost penalty is lower. This is as expected because the margin is wider. We can plot the decision boundary using Michael Hahsler's `decisionplot()` function which can be found **here**.

```r
# Define the function:
decisionplot <- function(model, data, class = NULL, predict_type = "class",
  resolution = 100, showgrid = TRUE, ...) {

  if(!is.null(class)) cl <- data[,class] else cl <- 1
  data <- data[,1:2]
  k <- length(unique(cl))

  plot(data, col = as.integer(cl)+1L, pch = as.integer(cl)+1L, ...)

  # make grid
  r <- sapply(data, range, na.rm = TRUE)
  xs <- seq(r[1,1], r[2,1], length.out = resolution)
  ys <- seq(r[1,2], r[2,2], length.out = resolution)
  g <- cbind(rep(xs, each=resolution), rep(ys, time = resolution))
  colnames(g) <- colnames(r)
  g <- as.data.frame(g)

  ### guess how to get class labels from predict
  ### (unfortunately not very consistent between models)
  # p <- predict(model, g, type = predict_type)
  p <- predict(model,newdata=g)
  if(is.list(p)) p <- p$class
  p <- as.factor(p)

  if(showgrid) points(g, col = as.integer(p)+1L, pch = ".")

  z <- matrix(as.integer(p), nrow = resolution, byrow = TRUE)
  contour(xs, ys, z, add = TRUE, drawlabels = FALSE,
    lwd = 2, levels = (1:(k-1))+.5)

  invisible(z)
}
```

Call the above function to plot the decision boundary. Note that teh above function had to be amended slightly from the one directly given on Michael Hahsler's webpage. In particular, the line `## p <- predict(model,`

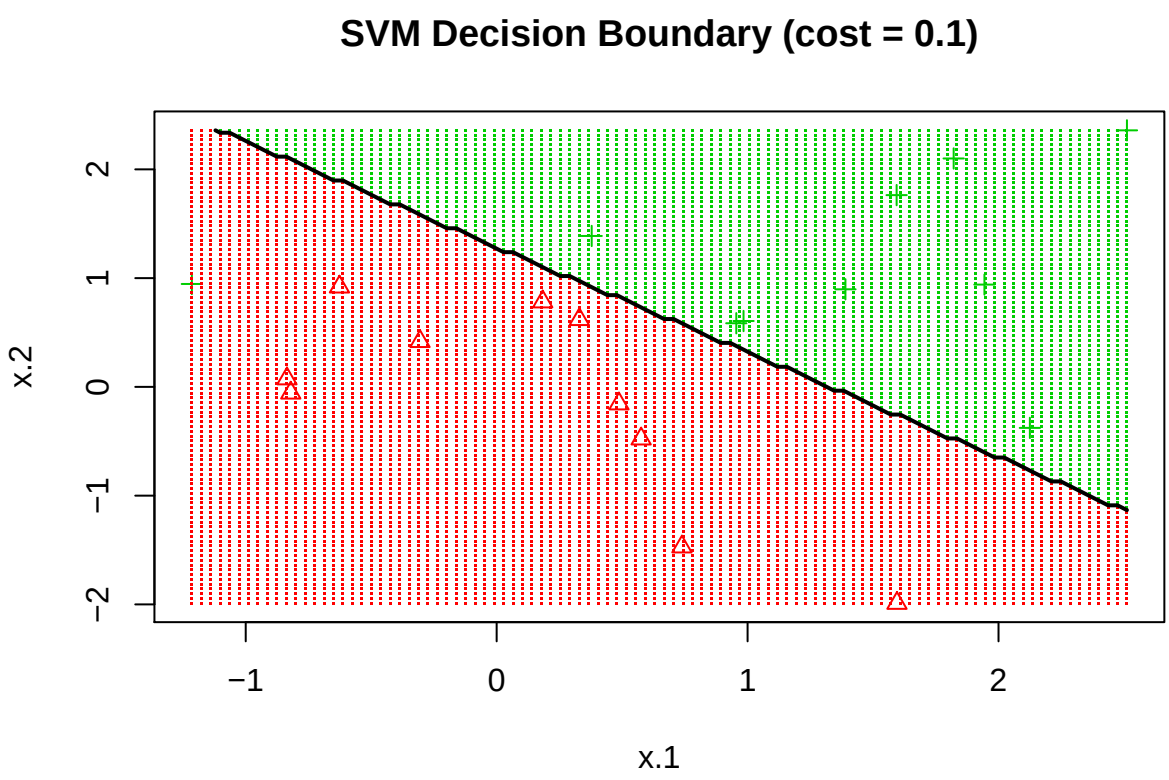
`g, type = predict_type)` was replaced with `## p <- predict(model,newdata=g)`. The reason is that the original function was not intended for use with `caret`. This change *should* make the `decisionplot()` function more robust in the sense that it will work for various types of models when using `caret`, but I have yet to verify.

```
# Call the amended function for the cost=10 model
decisionplot(svm.fit,dat,class="y",main="SVM Decision Boundary (cost = 10)")
```



Repeat for the low cost estimator.

```
# Call the amended function for the cost=0.1 model
decisionplot(svm.fit.dot1,dat,class="y",main="SVM Decision Boundary (cost = 0.1)")
```



Note that the shape is very similar but the boundary is shifted up and to the right a bit. What is not shown is the margin. This would require changing the above code to include three types of classes ($y = 1, y = -1$, and those in the margin). Note that the basic `svm()` function in the `e1071` package does not return the decision boundary so this step is not included in the chapter 9 lab in James et al. (2013).

Let's now consider an experiment to fir the model for multiple values of the cost parameter. The first step is to re-define the grid variable and call it in teh `train` function. We will keep the cross-validation as above.

```
# Fit a model using e1071 via caret:
trControl <- trainControl(method="repeatedcv",number=10,repeats=10)

# Iterate through various cost paramters:
grid <- expand.grid(.cost=c(0.001,0.01,0.1,1,5,10,100))
print(grid)
```

```
##   .cost
## 1 1e-03
## 2 1e-02
## 3 1e-01
## 4 1e+00
## 5 5e+00
## 6 1e+01
## 7 1e+02
```

Note that `expand.grid` makes a data frame of all possible combinations of the inputs. In this case, it simply converts the `.cost` vector to a column of a data frame. However, if there was another tuning parameter, say `.weights` and we wanted to iterate through 5 different values, then `expand.grid` would make a data frame with 2 columns and 35 (= 5 x 7) rows, each with a unique combination of `.weights` and `.cost`.

```
svm.fit.tune <- train(y~., data=dat, trControl=trControl,
                      method="svmLinear2",
                      tuneGrid=grid)

# Summarize the model to get some basic model info:
svm.fit.tune$results
```

```
##    cost Accuracy Kappa AccuracySD   KappaSD
## 1 1e-03    0.750  0.50  0.2512595 0.5025189
## 2 1e-02    0.750  0.50  0.2512595 0.5025189
## 3 1e-01    0.945  0.89  0.1572330 0.3144660
## 4 1e+00    0.895  0.79  0.2046801 0.4093602
## 5 5e+00    0.895  0.79  0.2046801 0.4093602
## 6 1e+01    0.895  0.79  0.2046801 0.4093602
## 7 1e+02    0.885  0.77  0.2114763 0.4229526
```

The model with cost = 0.1 results in the best accuracy, which is our test metric. We could have used Kappa or even a custom metric as the target metric by setting `metric = "Kappa"` (or another value) in teh `train()` function.

Note that the `caret` package automatically saves the best model in the `finalModel` object. To see the optimal cost parameter, we could simply print it out.

```
print(svm.fit.tune$finalModel$cost)
```

```
## [1] 0.1
```

The above model statistics are in sample. If we instead wanted to predict a test sample (which is of more relvance in practice) we can easily amend out code to do so. First generate a sample of random test data.

```
# random input obeservations:
x.test <- matrix(rnorm(20*2), ncol=2)

# random vector of outcomes:
y.test <- sample(c(-1,1),20,rep=T)

# Shift the output:
x.test[y.test==1,] <- x.test[y.test==1,] + 1

# Put it all in a data frame:
test.dat <- data.frame(x=x.test, y=as.factor(y.test))

# Make the prediciton:
y.pred <- predict(svm.fit.tune, newdata = test.dat)
table(predict=y.pred,truth=test.dat$y)
```

```
##        truth
## predict -1 1
##      -1  8 3
##      1   2 7
```

3.3 Non-linear Kernels

We can easily fir SVM's with non-linear kernels using the `caret` package. We will do two examples: polynomial with $degree = 2$ and a radial kernel with $\gamma = 1$.

3.3.1 Polynomial Kernel

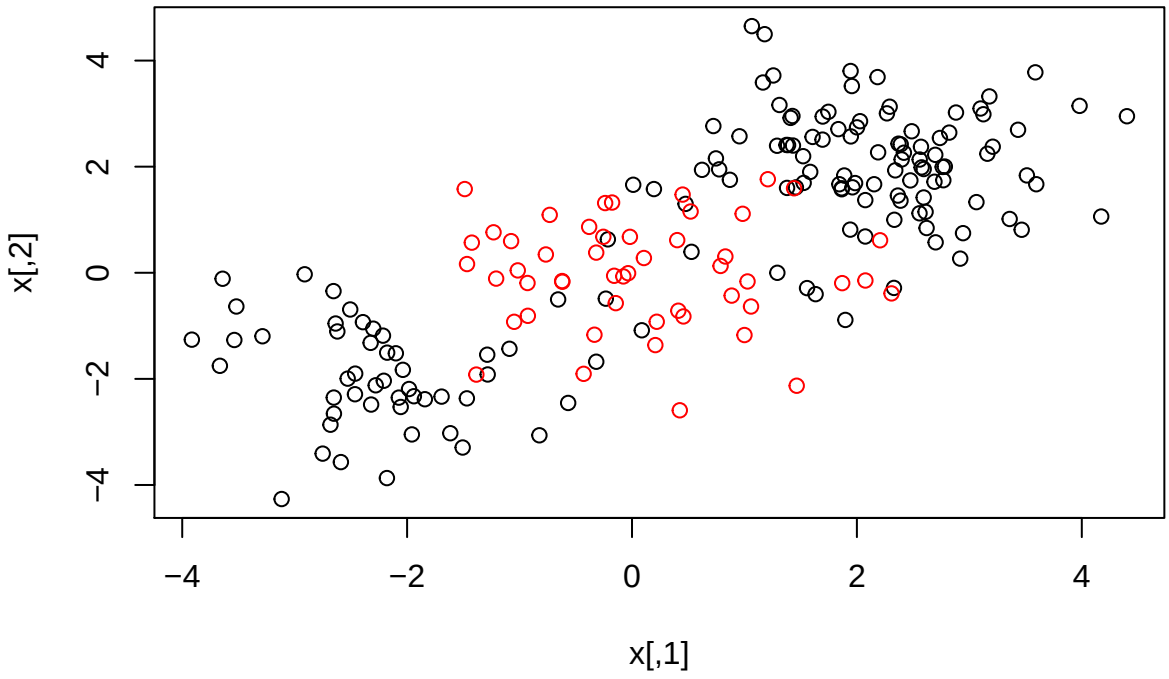
First implement and plot the polynomial kernel of *degree* = 2. We will do so using the `method = 'svmPoly'` which is part of the `kernlab` package.

```
library("kernlab")
```

```
##
## Attaching package: 'kernlab'
```

```
## The following object is masked from 'package:ggplot2':
##
##     alpha
```

```
set.seed(1)
x <- matrix(rnorm(200*2), ncol=2)
x[1:100,] <- x[1:100,] + 2
x[101:150,] <- x[101:150,] - 2
y <- c(rep(1,150),rep(2,50))
dat <- data.frame(x=x, y=as.factor(y))

plot(x,col=y)
```




```
# Fit a model using e1071 via caret:
trControl <- trainControl(method="repeatedcv",number=10,repeats=10)

# Change the tuning parameters
grid <- expand.grid(.C=1,
```

```
                    .scale=TRUE,
                    .degree=3)

# Fit the model
svm.fit.poly.3 <- train(y~., data=dat, trControl=trControl,
                  method="svmPoly",
                  tuneGrid=grid)
# List the variables in the summary:
names(svm.fit.poly.3)
```

```
##  [1] "method"       "modelInfo"    "modelType"    "results"
##  [5] "pred"         "bestTune"     "call"         "dots"
##  [9] "metric"       "control"      "finalModel"   "preProcess"
## [13] "trainingData" "resample"     "resampledCM"  "perfNames"
## [17] "maximize"     "yLimits"      "times"        "levels"
## [21] "terms"        "coefnames"    "xlevels"
```

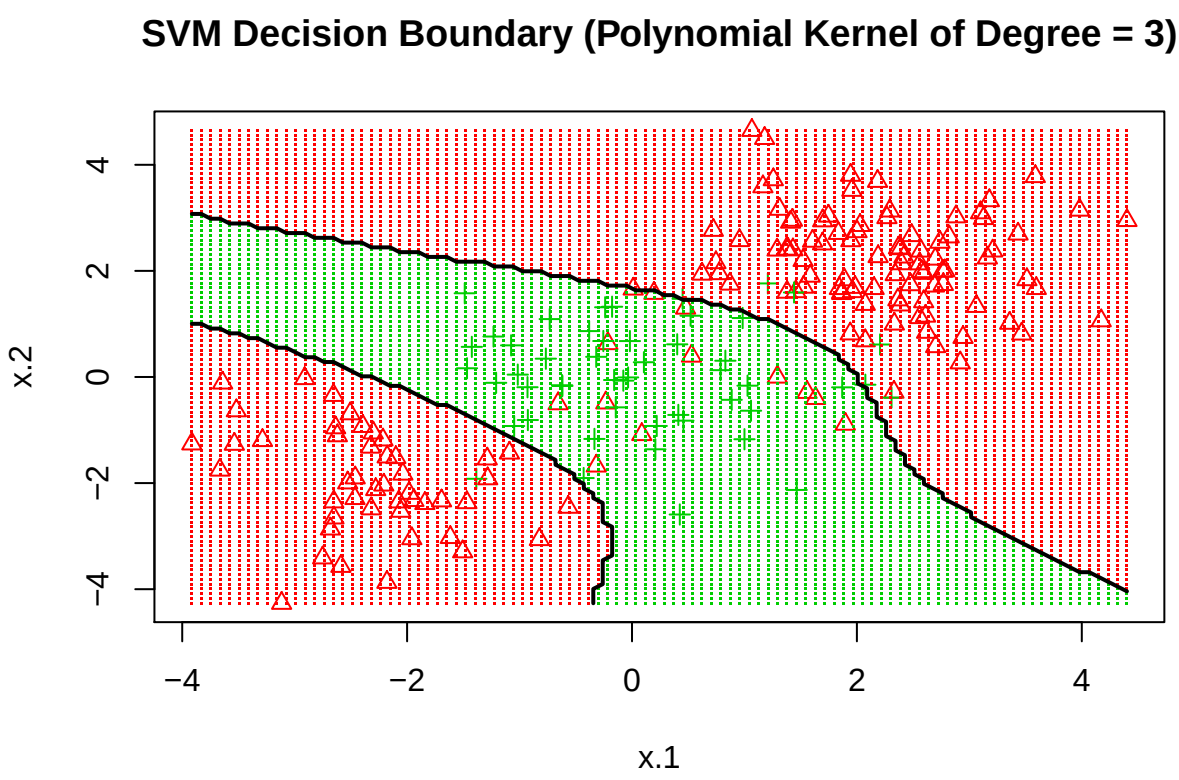
```
# Print the support vectors:
svm.fit.poly.3$finalModel
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Polynomial kernel function.
##  Hyperparameters : degree =  3  scale =  TRUE  offset =  1
##
## Number of Support Vectors : 49
##
## Objective Function Value : -43.0466
## Training error : 0.095
```

```
# Summarize the model to get some basic model info:
summary(svm.fit.poly.3)
```

```
## Length  Class   Mode
##      1   ksvm     S4
```

```
# Call the amended function for the cost=10 model
decisionplot(svm.fit.poly.3,dat,class="y",main="SVM Decision Boundary (Polynomial Kernel of Degree = 3)
```



Note the non-linear boundary above.

3.3.2 Radial Kernel

We will implement the radial kernel using the same `kernlab` package and `method='svmRadialSigma'`. The method reauries a parameter *sigma* which is the the thing as what we called γ in Eq. (18).

```
library("kernlab")

# Fit a model using e1071 via caret:
trControl <- trainControl(method="repeatedcv",number=10,repeats=10)

# Change the tuning parameters
grid <- expand.grid(.C=1,
                    .sigma=1)

# Fit the model
svm.fit.radial.1 <- train(y~., data=dat, trControl=trControl,
                 method="svmRadialSigma",
                 tuneGrid=grid)
# List the variables in the summary:
names(svm.fit.radial.1)
```

```
##  [1] "method"       "modelInfo"    "modelType"    "results"
##  [5] "pred"         "bestTune"     "call"         "dots"
##  [9] "metric"       "control"      "finalModel"   "preProcess"
```

```
## [13] "trainingData" "resample"     "resampledCM"  "perfNames"
## [17] "maximize"     "yLimits"      "times"        "levels"
## [21] "terms"        "coefnames"    "xlevels"
```

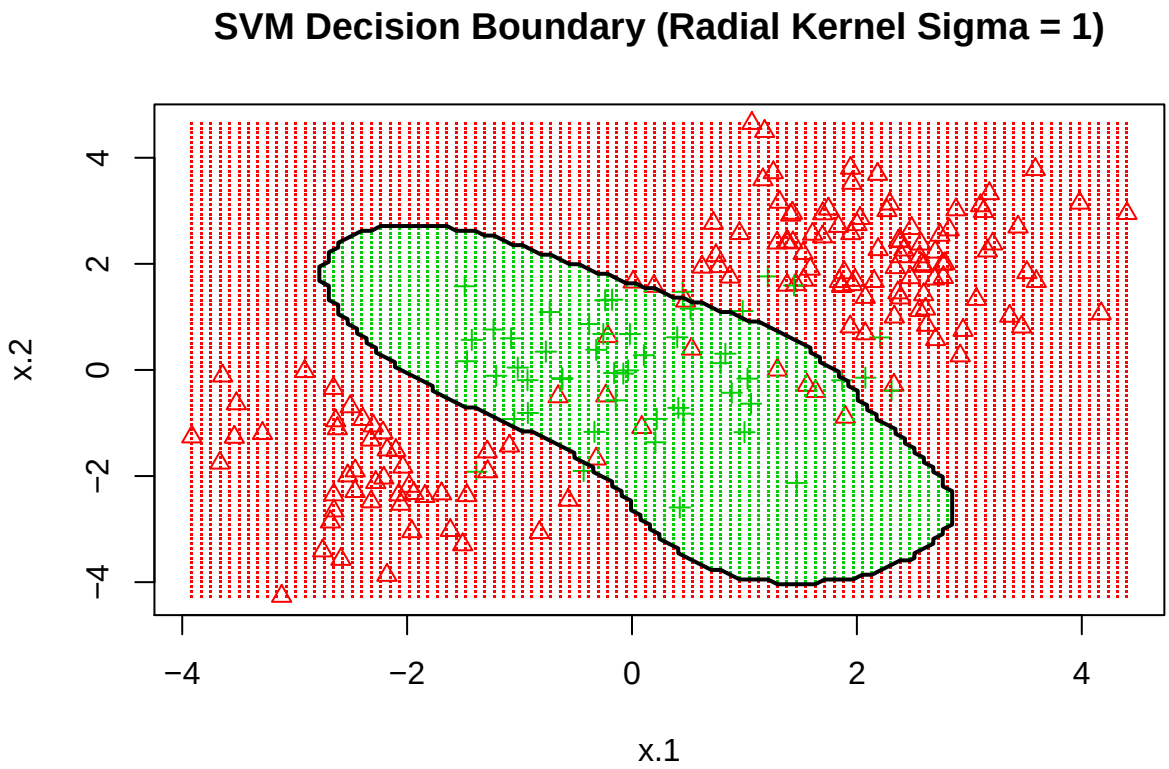
```
# Print the support vectors:
svm.fit.radial.1$finalModel
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  1
##
## Number of Support Vectors : 63
##
## Objective Function Value : -48.6061
## Training error : 0.1
```

```
# Summarize the model to get some basic model info:
summary(svm.fit.radial.1)
```

```
## Length  Class   Mode
##      1   ksvm     S4
```

```
# Call the amended function for the cost=10 model
decisionplot(svm.fit.radial.1,dat,class="y",main="SVM Decision Boundary (Radial Kernel Sigma = 1)")
```



Notice that the radial plot uses a nonlinear recombining decision rule.

3.4 Application to Mortgage Data

We can repeat the above analysis for the mortgage data. To make teh models work better in terms fo accuracy, we will down-sample the data. That is, we will consider all defaults and then randomly sample non-defaulted accounts so we have te same number of each class.

```
# Mortgage Data:
```

References

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2001. *The Elements of Statistical Learning*. Vol. 1. 10. Springer series in statistics New York.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An Introduction to Statistical Learning*. Vol. 112. Springer.