DS-6030 Homework Module 9

Tom Lever

07/24/2023

DS 6030 | Spring 2023 | University of Virginia

7. In this problem, you will use support vector approaches in order to predict whether a given car gets high or low gas mileage based on the `Auto` data set.

   (a) Create a binary variable that takes on a 1 for cars with gas mileage above the median, and a 0 for cars with gas mileage below the median.

```
library(ISLR2)
condition <- Auto$mpg > median(Auto$mpg)
vector_of_indicators_of_whether_gas_mileage_is_high <- ifelse(
    test = condition,
    yes = 1,
    no = 0
)
Auto$indicator_of_whether_gas_mileage_is_high <- as.factor(
    vector_of_indicators_of_whether_gas_mileage_is_high
)
head(Auto, n = 2)
```

```
#   mpg cylinders displacement horsepower weight acceleration year origin
# 1  18         8          307        130   3504         12.0   70      1
# 2  15         8          350        165   3693         11.5   70      1
#                        name indicator_of_whether_gas_mileage_is_high
# 1 chevrolet chevelle malibu                                        0
# 2         buick skylark 320                                        0
```

```
tail(Auto, n = 2)
```

```
#     mpg cylinders displacement horsepower weight acceleration year origin
# 396  28         4          120         79   2625         18.6   82      1
# 397  31         4          119         82   2720         19.4   82      1
#            name indicator_of_whether_gas_mileage_is_high
# 396 ford ranger                                        1
# 397  chevy s-10                                        1
```

   (b) Fit a support vector classifier to the data with various values of `cost`, in order to predict whether a car gets high or low gas mileage. Report the cross-validation errors associated with different values of this parameter. Comment on your results.

   In a p-dimensional space, a hyperplane is a flat affine subspace of dimension $p-1$. The word affine indicates that the subspace need not pass through the origin. For instance, in two dimensions, a hyperplane is a flat one-dimensional subspace; in other words, a line. In three dimensions, a hyperplane is a flat two-dimensional subspace; that is, a plane. In $p > 3$ dimensions, it can be hard to visualize a hyperplane, but the notion of a $(p-1)$-dimensional flat subspace still applies.

The mathematical definition of a hyperplane is quite simple. In two dimensions, a hyperplane is defined by the equation

$$h_* = \beta_0 + \beta_1 x_{*1} + \beta_2 x_{*2} = 0$$

for parameters β_0, β_1, and β_2. When we say that this equation defines the hyperplane, we mean that any $x = (x_{*1}, x_{*2})$ for which this equation holds is a point on the hyperplane. Note that this equation is simply the equation of a line, snice in two dimensions a hyperplane is a line.

This equation can be easily extended to the p-dimensional setting:

$$h_* = \beta_0 + \beta_1 x_{*1} + \beta_2 x_{*2} + \cdots + \beta_p x_{*p} = 0$$

defines a p-dimensional hyperplane, again the sense that if a point $x = (x_{*1}, x_{*2}, \cdots, x_{*p})$ in p-dimensional space (i.e., a vector of length p) satisfies this equation, then x lies on the hyperplane.

Now, suppose that x does not satify the above equation; rather,

$$h_* = \beta_0 + \beta_1 x_{*1} + \beta_2 x_{*2} + \cdots + \beta_p x_{*p} > 0$$

Then this tells us that x lies to one side of the hyperplane. On the other hand, if

$$h_* = \beta_0 + \beta_1 x_{*1} + \beta_2 x_{*2} + \cdots + \beta_p x_{*p} < 0$$

then x lies on the other side of the hyperplane. So we can think of the hyperplane as dividing p-dimensional space into two halves. One can easily determine on which side of the hyperplane a point lies by simply calculating the sign of the left-hand side of the above equation.

Suppose that we have an $n \times p$ data matrix $\boldsymbol{X}$ that consists of n training observations in p-dimensional space,

$$\boldsymbol{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}$$

and that these observations fall into two classes; that is,

$$\vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$$y_1, y_2, \cdots, y_n \in \{-1, 1\}$$

where -1 represents one class and 1 the other class. We also have a test observation, a p-vector of observed features $x_test = (x_{test}, x_{test,2}, \cdots, x_{test,p})$. Our goal is to develop a classifier based on the training data that will correctly classify the test observation using its feature measurements. We will now see an approach that is based upon the concept of a separating hyperplane.

Suppose that it is possible to construct a hyperplane that separates the training observations perfectly according to their class labels. We label the observations from the first class as $y_i = 1$ and those from the second class as $y_i = -1$. Then a separating hyperplane has the property that

$$h_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} > 0 if y_i = 1$$

$$h_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} < 0 if y_i = -1$$

Equivalently, a separating hyperplane has the property that

$$y_i h_i = y_i \cdot (\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}) > 0$$

for all $i = 1, \cdots, n$.

If a separating hyperplane exists, we can use it to construct a very natural classifier: a test observation is assigned a class depending on which side of the hyperplane it is located. That is, we classify the test observation x_{test} based on the sign of $h_{test} = \beta_0 + \beta_1 x_{test,1} + \beta_2 x_{test,2} + \cdots + \beta_p x_{test,p}$. If h_{test} is positive, then we assign the test observation to class 1, and if h_{test} is negative, then we assign the test observation to class -1. We can also make use of the magnitude of h_{test}. If \$$h_{test}$ is far from 0, then this means that x_{test} lies far from the hyperplane, and so we can be confident about our class assignment for x_{test}. On the other hand, if h_{test} is close to 0, then x_{test} is located near the hyperplane, and so we are less certain about the class assignment for x_{test}. Not surprisingly, a classifier that is based on a separating hyperplane leads to a linear decision boundary.

In general, if our data can be perfectly separated using a hyperplane, then there will in fact exist an infinite number of such hyperplanes. This is because a given separating hyperplane can usually be shifted a tiny bit up or down, without coming into contact with any of the observations. In order to construct a classifier based upon a separating hyperplane, we must have a reasonable way to decide which of the infinite possible separating hyperplanes to use.

A natural choice is the maximal margin hyperplane (also known as the optimal separating hyperplane), which is the separating hyperplane that is farthest from the training observations. That is, we can compute the perpendicular distance from each training observation to a given separating hyperplane; the smallest such distance is the minimal distance from the observations to the hyperplane, and is known as the margin. The maximal margin hyperplane is the separating hyperplane for which the margin is largest; that is, it is the hyperplane that has the farthest minimum distance to the training observations. We can then classify a test observation based on which side of the maximal margin hyperplane it lies. This is known as the maximal margin classifier. We hope that a classifier that has a large margin on the training data will also have a large margin on the test data, and hence will classify the test observations correctly. Although the maximal margin classifier is often successful, it can also lead to overfitting when p is large.

If β_0, β_1, $\cdots$, β_p are the coefficients of the maximal margin hyperplane, then the maximal margin classifier classifies the test observation x_{test} based on the sign of $h_{test} = \beta_0 + \beta_1 x_{test,1} + \beta_2 x_{test,2} + \cdots + \beta_p x_{test,p}$.

In a sense, the maximal margin hyperplane represents the mid-line of the widest "slab" that we can insert between the two classes.

For a maximal margin classifier, support vectors are training observations that are equidistant from the maximal margin hyperplane and lie along hyperplanes parallel to the maximal margin hyperplane indicating the width of the margin. Support vectors are vectors in p-dimensional space and they support the maximal margin hyperplane in the sense that if these points were moved slightly then the maximal margin hyperplane would move as well. Interestingly, the maximal margin hyperplane depends directly on the support vectors, but not on the other observations: a movement to any of the other observations would not affect the separating hyperplane, provided that the observation's movement does not cause it to cross the boundary set by the margin. The maximal margin hyperplane depends directly on only a small subset of the observations.

We now consider the task of constructing the maximal margin hyperplane based on a set of n training observations $x_1, x_2, \cdots, x_n \in \mathbb{R}^p$ and associated class labels $y_1, y_2, \cdots, y_n \in \{-1, 1\}$. Briefly, the maximal margin hyperplane is the solution to the optimization problem

"Maximize margin M,

by determining a maximal margin hyperplane,

with parameters β_0, β_1, $\cdots$, β_p,

and associated margin M,

subject to the length of vector of parameters $(\beta_0, \beta_1, \cdots, \beta_p)$ being 1, i.e.,

$\sum_{j=1}^{p} \left[\beta_j^2\right] = 1$, and

$y_i h_i = y_i \cdot (\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}) \geq M \ \forall i \in [1, n]$."

The penultimate constraint constrains the distance from the ith observation to the maximal margin hyperplane to be $y_i h_i$. The last constraint guarantees that each observation will be on the correct side of the hyperplane, with cushion M, provided that M is positive. Therefore, the last two constraints ensure that each observation is on the correct side of the hyperplane and at least a distance M from the hyperplane. Hence, M represents the margin of our hyperplane, and the optimization problem chooses $\beta_0, \beta_1, \cdots, \beta_p$ to maximize M. This is exactly the definition of the maximal margin hyperplane!

The maximal margin classifier is a very natural way to perform classification, if a separating hyperplane exists. However, in many cases no separating hyperplane exists, and so there is no maximal margin classifier. In this case, the optimization problem has no solution with $M > 0$. In this case, we cannot exactly separate the two classes. However, we can extend the concept of a separating hyperplane in order to develop a hyperplane that almost separates the classes, using a so-called soft margin. The generalization of the maximal margin classifier to the non-separable case is known as the support vector classifier.

Observations that belong to two classes are not necessarily separable by a hyperplane. In fact, even if a separating hyperplane does exist, then there are instances in which a classifier based on a separating hyperplane might not be desirable. A classifier based on a separating hyperplane will necessarily perfectly classify all of the training observations; this can lead to sensitivity to individual observations. An example in shown in Figure 9.5. The addition of a single observation in the right-hand panel of Figure 9.5 leads to a dramatic change in the maximal margin hyperplane. The resulting maximal margin hyperplane is not satisfactory; for one thing, it has only a tiny margin. This is problematic because as discussed previously, the distance of an observation from the hyperplane can be seen as a measure of our confidence that the observation was correctly classified. Moreover, the fact that a maximal margin hyperplane is extremely sensitive to a single observation suggests that it may have overfit the training data.

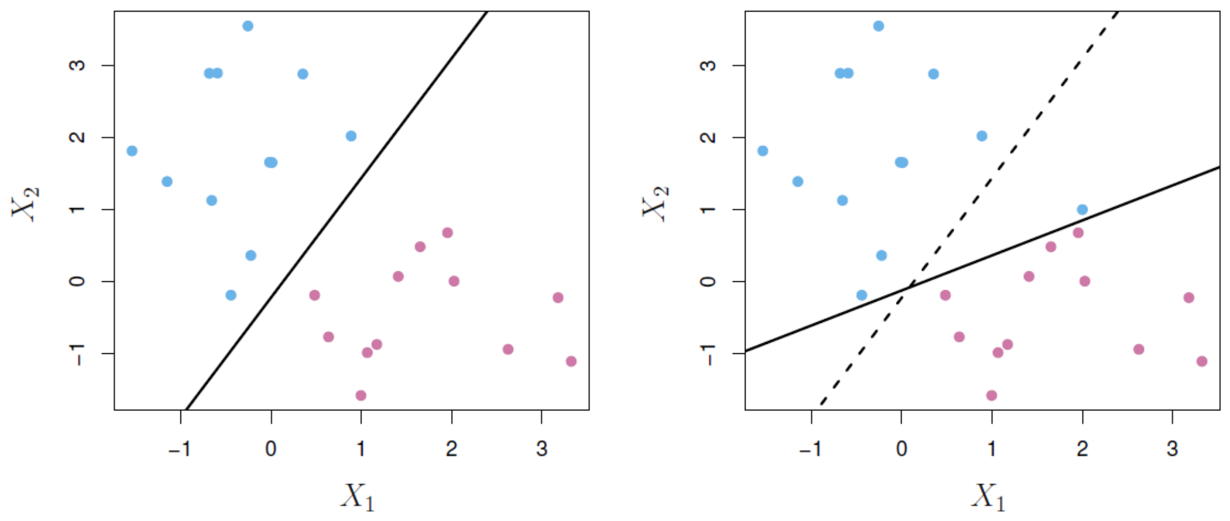


FIGURE 9.5. Left: *Two classes of observations are shown in blue and in purple, along with the maximal margin hyperplane.* Right: *An additional blue observation has been added, leading to a dramatic shift in the maximal margin hyperplane shown as a solid line. The dashed line indicates the maximal margin hyperplane that was obtained in the absence of this additional point.*

Figure 1: Sensitivity Of Maximal Margin Classifier

In this case, we might be willing to consider a classifier based on a hyperplane that does not perfectly separate the two classes, in the interest of greater robustness to individual observations, and better classification of most of the training observations. That is, it could be worthwhile to misclassify a few training observations in order to do a better job in classifying the remaining observations.

The support vector classifier, sometimes called a soft margin classifier, does exactly this. Rather than seeking the largest possible margin so that every observation is not only on the correct side of the hyperplane but also on the correct side of the margin, we instead allow some observations to be on the incorrect side of the margin, or even the incorrect side of the hyperplane. The margin is soft because it can be violated by some of the training observations.

An observation can be not only on the wrong side of the margin, but also on the wrong side of the hyperplane. In fact, when there is no separating hyperplane, such a situation is inevitable. Observations on the wrong side of the hyperplane correspond to training observations that are misclassified by the support vector classifier.

The support vector classifier classifies a test observation depending on which side of a hyperplane the test observation lies. The hyperplane is chosen to correctly separate most of the training observations into the two classes, but may misclassify a few observations. It is the solution to the optimization problem

“Maximize margin M,

by determining a maximal margin hyperplane,

with parameters β_0, β_1, $\cdots$, β_p, ϵ_1, ϵ_2, $\cdots$, ϵ_n,

and associated margin M,

subject to the length of vector of parameters $(\beta_0, \beta_1, \cdots, \beta_p)$ being 1, i.e.,

$\sum_{j=1}^{p} \left[\beta_j^2\right] = 1$,

$y_i h_i = y_i \cdot (\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}) \geq M \cdot (1 - \epsilon_i)\ \ \forall i \in [1, n]$,

$\epsilon_i \geq 0\ \forall i \in [1, n]$, and

$\sum_{i=1}^{n} \left[\epsilon_i\right] \leq C\ \forall i \in [1, n]$.”

Cost C is a nonnegative tuning parameter. As above, M is the width of the margin; we seek to make this quantity as large as possible. In the penultimate constraint above, $\epsilon_1, \epsilon_2, \cdots, \epsilon_n$ are slack variables that allow individual observations to be on the wrong side of the margin or the hyperplane. Once we have solved this optimization problem, we classify a test observation x_{test} as before, by simply determining on which side of the hyperplane it lies. That is, we classify the test observation based on the sign of h_{test}.

The slack variable ϵ_i tells us where the ith observation is located, relative to the hyperplane and relative to the margin. If $\epsilon_i = 0$ then the ith observation is on the correct side of the margin. If $\epsilon_i > 0$ then the ith observation is on the wrong side of the margin, and we say that the ith observation has violated the margin. If $\epsilon_i > 1$ then the ith observation is on the wrong side of the hyperplane.

We now consider the role of cost C. In the last constraint above, C bounds the sum of the slack variables ϵ_i, and so it determines the number and severity of the violations to the margin (and to the hyperplane) that we will tolerate. We can think of C as a budget for the amount that the margin can be violated by the n observations. If $C = 0$ then there is no budget for violations to the margin, and it must be the case that $\epsilon_1 = \epsilon_2 = \cdots \epsilon_n = 0$, in which case the optimization problem above simply amounts to the maximal margine hyperplane optimization problem. Of course, a maximal margin hyperplane exists only if the two classes are separable. For $C > 0$ no more than C observations can be on the wrong side of the hyperplane, because if an observation is on the wrong side of the hyperplane than $\epsilon_i > 1$, and the last constraint above requires that $\sum_{i=1}^{n} \epsilon_i \leq C$. As the budget C increases, we become more tolerant of violations to the margin, and so the margin will widen. Conversely, as C decreases, we become less tolerant of violations to the margin and so the margin narrows.

In practice, C is treated as a tuning parameter that is generally chosen via cross-validation. C controls the bias-variance trade-off of a support-vector classifier. When C is small, we seek narrow margins that are rarely violated; this amounts to a classifier that is highly fit to the data, which may have low bias but high variance. On the other hand, when C is larger, the margin is wide and we allow more violations to it; this amounts to fitting the data less hard and obtaining a classifier that is potentially more biased but may have lower variance.

The optimization problem above has a very interesting property: it turns out that only observations that either lie on the margin or that violate the margin will affect the hyperplane, and hence the classifier obtained. In other words, an observation that lies strictly on the correct side of the margin does not affect the support vector classifier! Changing the position of that observation would not change the classifier at all, provided that its position remains on the correct side of the margin. Observations that lie directly on the margin, or on the wrong side of the margin for their class, are known as support vectors. These observations do affect the support-vector classifier.

The fact that only support vectors affect the classifier is in line with our previous assertion that cost C controls the bias-variance trade-off of the support vector classifier. When the tuning parameter C is large, then the margin is wide, many observations violate the margin, and so there are many support vectors. In this case, many observations are involved in determining the hyperplane. This classifier has low variance (since many observations are support vectors) but potentially high bias. In contrast, if C is small, then there will be fewer support vectors andhence the resulting classifier will have low bias but high variance.

When C is large, then there is a high tolerance for observations being on the wrong side of the margin, and so the margin will be large. As C decreases, the tolerance for observations being on the wrong side of the margin decreases, and the margin narrows.

The fact that the support vector classifier's decision rule is based on a potentially small subset of the training observations (the support vectors) means that it is quite robust to the behavior of observations that are far away from the hyperplane. This property is distinct from other classification methods, such as linear discriminant analysis. Recall that the LDA classification rule depends on the mean of all of the observations within each class, as well as the within-class covariance matrix computed using all of the observations. In contrast, logistic regression, unlike LDA, has very low sensitivity to observations far from the decision boundary. In fact the support vector classifier and logistic regression are closely related.

A support vector machine with a linear kernel is a support vector classifier.

```r
set.seed(1)
the_tune <- e1071::tune(
    METHOD = e1071::svm,
    train.x = indicator_of_whether_gas_mileage_is_high ~ .,
    data = Auto,
    kernel = "linear",
    ranges = list(
        cost = c(0.01, 0.1, 1, 10, 100)
    )
)
summary(the_tune)
```

```
#
# Parameter tuning of 'e1071::svm':
#
# - sampling method: 10-fold cross validation
#
# - best parameters:
#  cost
#     1
#
# - best performance: 0.01025641
#
# - Detailed performance results:
```

```
#    cost      error dispersion
# 1 1e-02 0.07653846 0.03617137
# 2 1e-01 0.04596154 0.03378238
# 3 1e+00 0.01025641 0.01792836
# 4 1e+01 0.02051282 0.02648194
# 5 1e+02 0.03076923 0.03151981
```

A support vector classifier / support vector machine with linear kernel with a cost $C = 1$ has a lowest 10-fold cross validation error rate of 0.0103.

(c) Now repeat (b), this time using SVM's with radial and polynomial basis kernels, with different values of `gamma` and `degree` and `cost`. Comment on your results.

The support vector classifier is a natural approach for classification in the two-class setting, if the boundary between the two classes is linear. However, in practice we are sometimes faced with non-linear class boundaries.

We know that the performance of linear regression can suffer when there is a nonlinear relationship between the predictors and the outcome. In that case, we consider enlarging the feature space using functions of the predictors, such as quadratic and cubic terms, in order to address this non-linearity. In the case of the support vector classifier, we could address the problem of possibly non-linear boundaries between classes in a similar way, by enlarging the feature space using quadratic, cubic, and even higher-order polynomial functions of the predictors. For instance, rather than fitting a support vector classifier using p features x_{*1}, x_{*2}, $\cdots$, x_{*p}, we could instead fit a support vector classifier using the $2p$ features x_{*1}, x_{*1}^2, x_{*2}, x_{*2}^2, $\cdots$, x_{*p}, x_{*p}^2. Then our optimization problem would become

"Maximize margin M,
by determining a maximal margin hyperplane,
with parameters β_0, β_{11}, β_{12}, β_{21}, β_{22}, $\cdots$, β_{p1}, β_{p2}, ϵ_1, ϵ_2, $\cdots$, ϵ_n,
and associated margin M,
subject to the length of vector of parameters $(\beta_0, \beta_{11}, \beta_{12}, \beta_{21}, \beta_{22}, \cdots, \beta_{p1}, \beta_{p2})$ being 1, i.e.,
$\sum_{j=1}^{p} \left[\sum_{k=1}^{2} \left(\beta_{jk}^2 \right) \right] = 1$,
$y_i h_i = y_i \cdot \left(\beta_0 + \sum_{j=1}^{p} \left[\beta_{j1} x_{ij} \right] + \sum_{j=1}^{p} \left[\beta_{j2} x_{ij}^2 \right] \right) \geq M \cdot (1 - \epsilon_i) \;\; \forall i \in [1, n]$,
$\epsilon_i \geq 0 \; \forall i \in [1, n]$, and
$\sum_{i=1}^{n} [\epsilon_i] \leq C \; \forall i \in [1, n]$."

Why does this lead to a non-linear decision boundary? In the enlarged feature space, the deicison boundary that results from the above optimization problem is in fact linear. But in the original feature space, the decision boundary is of the form $q(x) = 0$, where q is a quadratic polynomial, and its solutions are generally non-linear. One might additionally want to enlarge the feature space with higher-order polynomial terms, or with interaction terms of the form $x_{*j} x_{*k}$ for $j \neq k$. Alternatively, other functions of the predictors could be considered rather than polynomials. It is not hard to see that there are many possible ways to enlarge the feature space, and that unless we are careful, we could end up with a huge number of features. Then computations would become unmanageable. The support vector machie allows us to enlarge the feature space used by the support vector classifier in a way that leads to efficient computations.

The support vector machine (SVM) is an extension of the support vector classifier that results from enlarging the feature space in a specific way, using kernels. We will now discuss this extension. The main idea is that we may want to enlarge our feature space in order to accommodate a non-linear boundary between the classes. The kernel approach that we describe here is simply an efficient computational appraoch for enacting this idea.

We have not discussed exactly how the support vector classifier is computed because the details become somewhat technical. However, it turns out that the solution to the support vector classifier problem involves only the inner products of the observations (as opposed to the observations themselves). The inner product of two p-vectors x_i and x_j is defined as $\langle x_i, x_j \rangle = \sum_{k=1}^{p} [x_{ij} x_{jk}]$. It can be shown that the linear support vector classifier can be represented as

$$f(x) = \beta_0 + \sum_{i=1}^{n} [\alpha_i \langle x, x_i \rangle]$$

where there are n training-observation parameters α_i, one per training observation. To estimate the parameters α_1, α_2, $\cdots$, α_n and β_0, all we need are the $\binom{n}{2}$ inner products $\langle x_i, x_j \rangle$ between all pairs of training observations. $\binom{n}{2} = \frac{n(n-1)}{2}$ gives the number of pairs among a set of n items.

Notice that in the representation of a linear support vector classifier, in order to evaluate te function $f(x)$, we need to compute the inner product between the new point x and each of the training points x_i. However, it turns out that α_i is nonzero only for the support vectors in the solution; that is, if a training observation is not a support vector, then its training-observation parameter α_i equals 0. So if S is the collection of indices of these support points, we can rewrite any solution function that is the representation of a linear support-vector classifier as

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

which typically involves far fewer terms than the number inner products between x and each of the training points x_i.

To summarize, in representing the linear support vector classifier $f(x)$, and in computing its coefficient β_0 and its training-observation parameters α_i, all we need are inner products.

Now suppose that every time the inner product $\langle x_i, x_j \rangle$ appears in the representation of a linear support vector classifier, or in a calculation of the solution for the linear support vector classifier, we replace the inner product with a generalization of the inner product of the form $K(x_i, x_j)$, where K is some function that we will refer to as a kernel. A kernel is a function that quantifies the similarity of two observations. For instance, we could simply take

$$K(x_i, x_j) = \sum_{k=1}^{p} [x_{ik} x_{jk}]$$

which would just give us back the linear support vector classifier. This equation is known as a linear kernel because the support vector classifier is linear in the features; the linear kernel essentially quantifies the similarity of a pair of observations using Pearson (standard) correlation. But one could instead choose another form for $K(x_i, x_j)$. For instance, one could define kernel

$$K(x_i, x_j) = \left(1 + \sum_{k=1}^{p} [x_{ik} x_{jk}]\right)^d$$

This is known as a polynomial kernel of degree d, where d is a positive integer. Using such a kernel with $d > 1$, instead of the standard linear kernel, in the support vector classifier algorithm leads to a much more flexible decision boundary. It essentially amounts to fitting a support vector classifier in a higher-dimensional space involving polynomials of degree d, rather than in the original feature space. When the support vector classifier is combined with a non-linear kernel such as a polynomial kernel, the resulting classifier is known as a support vector machine. Note that in this case the non-linear function has the form

$$f(x) = \beta_0 + \sum_{i \in S} [\alpha_i K(x, x_i)]$$

When $d = 1$, then the SVM reduces to the support vector classifier seen earlier in this chapter.

The polynomial kernel presented above is one example of a possible non-linear kernel, but alternatives abound. Another populat choice is the radial kernel, which takes the form

$$K\left(x_i, x_j\right) = exp\left\{-\gamma \sum_{k=1}^{p}\left[\left(x_{ik} - x_{jk}\right)^2\right]\right\}$$

For this radial kernel, γ is a positive constant.
How does the radial kernel actually work? If a given test observation $x_{test} = \left(x_{test,1}, x_{test,2}, \cdots, x_{test,p}\right)$ is far from a training observation x_i in terms of Euclidean distance, then $\sum_{k=1}^{p}\left[\left(x_{ik} - x_{jk}\right)^2\right]$ will be large, and so $K\left(x_{test}, x_i\right)$ will be tiny. This means that in the above formula for a support vector machine, x_i will play virtually no role in $f\left(x_{test}\right)$. Recall that the predicted class label for the test observation x_{test} is based on the sign of $f\left(x_{test}\right)$. In other words, training observations that are far from x_{test} will play essentially no role in the predicted class label for x_{test}. This means that the radial kernel has very local behavior, in the sense that only nearby training observations have an effect on the class label of a test observation.
What is the advantage of using a kernel rather than simply enlarging the feature space using functions of the original features? One advantage is computation, and it amounts to the fact that using kernels, one need only compute $K\left(x_i, x_j\right)$ for all distinct pairs of i and j. This can be done without explicitly working in the enlarged feature space. This is important because in many applications of SVM's, the enlarged feature space is so large that computations are intractable. For some kernels, such as the radial kernel, the feature space is implicit and infinite-dimensional, so we could never do the computations there anyway!

```
set.seed(1)
the_tune <- e1071::tune(
    METHOD = e1071::svm,
    train.x = indicator_of_whether_gas_mileage_is_high ~ .,
    data = Auto,
    kernel = "polynomial",
    ranges = list(
        cost = c(0.01, 0.1, 1, 10, 100),
        degree = c(2, 3, 4)
    )
)
summary(the_tune)
```

```
#
# Parameter tuning of 'e1071::svm':
#
# - sampling method: 10-fold cross validation
#
# - best parameters:
#  cost degree
#   100      2
#
# - best performance: 0.3013462
#
# - Detailed performance results:
#     cost degree     error dispersion
# 1  1e-02      2 0.5511538 0.04366593
# 2  1e-01      2 0.5511538 0.04366593
# 3  1e+00      2 0.5511538 0.04366593
# 4  1e+01      2 0.5130128 0.08963366
# 5  1e+02      2 0.3013462 0.09961961
# 6  1e-02      3 0.5511538 0.04366593
# 7  1e-01      3 0.5511538 0.04366593
# 8  1e+00      3 0.5511538 0.04366593
# 9  1e+01      3 0.5511538 0.04366593
# 10 1e+02      3 0.3446154 0.09821588
# 11 1e-02      4 0.5511538 0.04366593
# 12 1e-01      4 0.5511538 0.04366593
# 13 1e+00      4 0.5511538 0.04366593
# 14 1e+01      4 0.5511538 0.04366593
# 15 1e+02      4 0.5511538 0.04366593
```

A support vector machine with polynomial kernel of degree $d = 2$ with a cost $C = 100$ has a lowest 10-fold cross validation error rate of 0.301.

```r
set.seed(1)
the_tune <- e1071::tune(
    METHOD = e1071::svm,
    train.x = indicator_of_whether_gas_mileage_is_high ~ .,
    data = Auto,
    kernel = "radial",
    ranges = list(
        cost = c(0.01, 0.1, 1, 10, 100, 1000),
        gamma = c(0.01, 0.1, 1, 10, 100, 1000)
    )
)
summary(the_tune)
```

```
#
# Parameter tuning of 'e1071::svm':
#
# - sampling method: 10-fold cross validation
#
# - best parameters:
#  cost gamma
#   100  0.01
#
# - best performance: 0.01282051
#
# - Detailed performance results:
#     cost gamma      error dispersion
# 1  1e-02 1e-02 0.55115385 0.04366593
# 2  1e-01 1e-02 0.08929487 0.04382379
# 3  1e+00 1e-02 0.07403846 0.03522110
# 4  1e+01 1e-02 0.02557692 0.02093679
# 5  1e+02 1e-02 0.01282051 0.01813094
# 6  1e+03 1e-02 0.02820513 0.02549818
# 7  1e-02 1e-01 0.21711538 0.09865227
# 8  1e-01 1e-01 0.07903846 0.03874545
# 9  1e+00 1e-01 0.05371795 0.03525162
# 10 1e+01 1e-01 0.03076923 0.03375798
# 11 1e+02 1e-01 0.03583333 0.02759051
# 12 1e+03 1e-01 0.03583333 0.02759051
# 13 1e-02 1e+00 0.55115385 0.04366593
# 14 1e-01 1e+00 0.55115385 0.04366593
# 15 1e+00 1e+00 0.06384615 0.04375618
# 16 1e+01 1e+00 0.05884615 0.04020934
# 17 1e+02 1e+00 0.05884615 0.04020934
# 18 1e+03 1e+00 0.05884615 0.04020934
# 19 1e-02 1e+01 0.55115385 0.04366593
# 20 1e-01 1e+01 0.55115385 0.04366593
# 21 1e+00 1e+01 0.51794872 0.05063697
# 22 1e+01 1e+01 0.51794872 0.04917316
# 23 1e+02 1e+01 0.51794872 0.04917316
# 24 1e+03 1e+01 0.51794872 0.04917316
# 25 1e-02 1e+02 0.55115385 0.04366593
# 26 1e-01 1e+02 0.55115385 0.04366593
# 27 1e+00 1e+02 0.55115385 0.04366593
# 28 1e+01 1e+02 0.55115385 0.04366593
# 29 1e+02 1e+02 0.55115385 0.04366593
# 30 1e+03 1e+02 0.55115385 0.04366593
# 31 1e-02 1e+03 0.55115385 0.04366593
# 32 1e-01 1e+03 0.55115385 0.04366593
# 33 1e+00 1e+03 0.55115385 0.04366593
# 34 1e+01 1e+03 0.55115385 0.04366593
# 35 1e+02 1e+03 0.55115385 0.04366593
# 36 1e+03 1e+03 0.55115385 0.04366593
```

A support vector machine with radial kernel with constant $\gamma = 0.01$ and cost $C = 100$ has a lowest 10-fold cross validation error rate of 0.013.

(d) Make some plots to back up your assertions in (b) and (c).

Hint: In the lab, we used the plot() function for svm objects only in cases with $p = 2$. When $p > 2$, you can use the `plot` function to create plots displaying pairs of variables at a time. Essentially, instead of typing

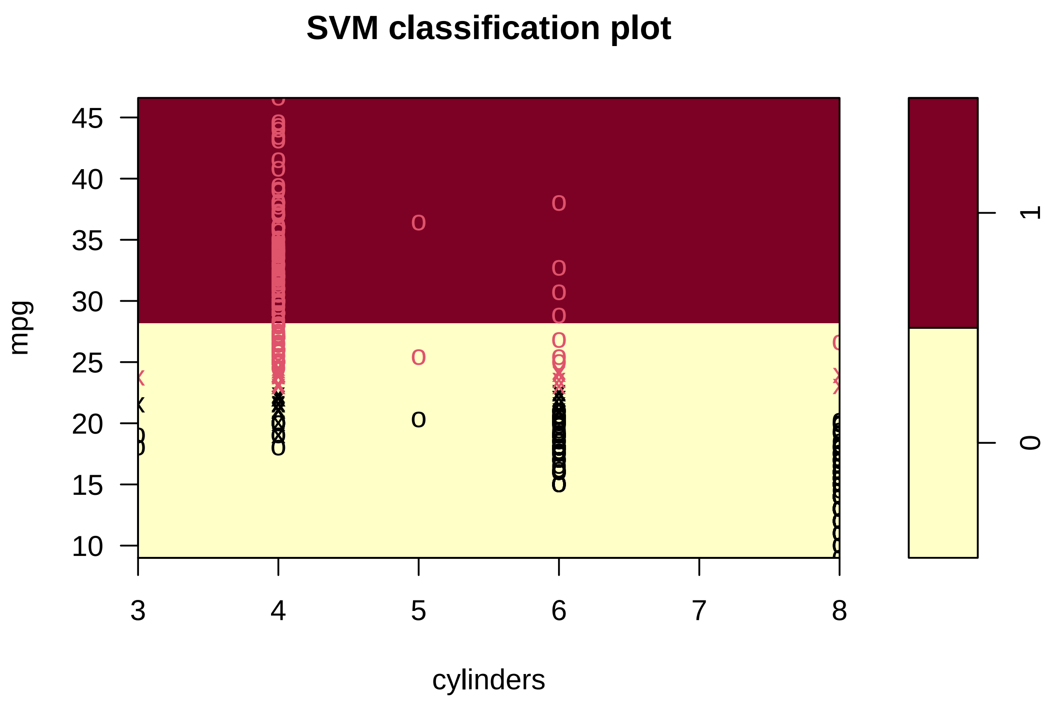
```
plot(svmfit, dat)
```

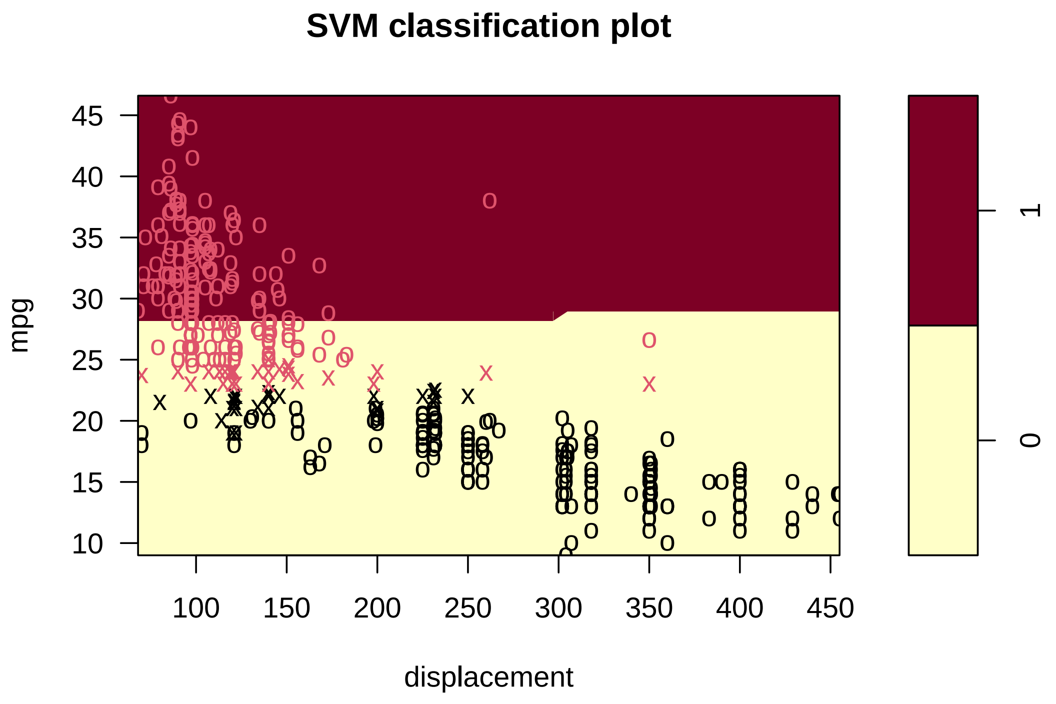
where `svmfit` contains your fitted model and `dat` is a data frame containing your data, you can type

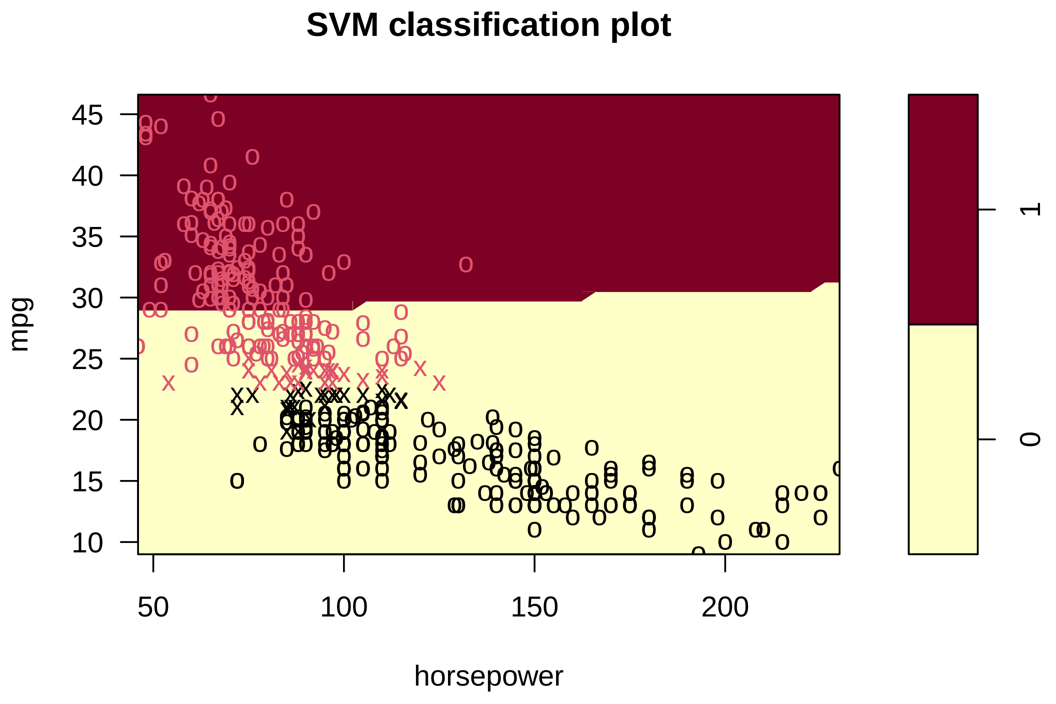
```
plot(svmfit, dat, x1 ~ x4)
```

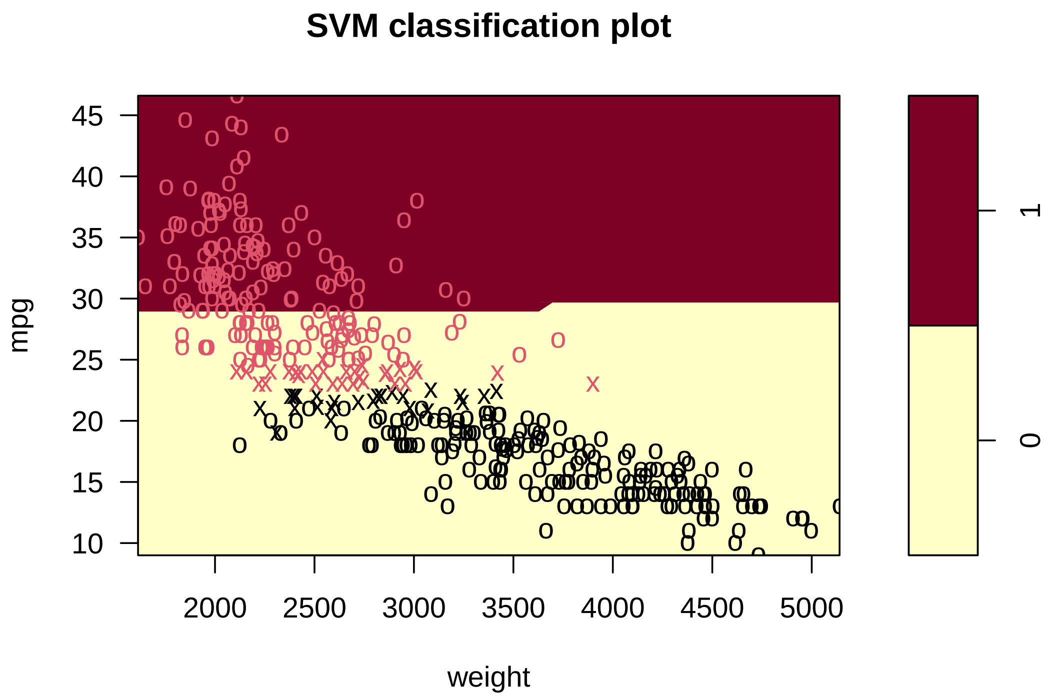
in order to plot just the first and fourth variables. However, you must replace `x1` and `x4` with the correct variable names. To find out more, type `?plot.svm`.

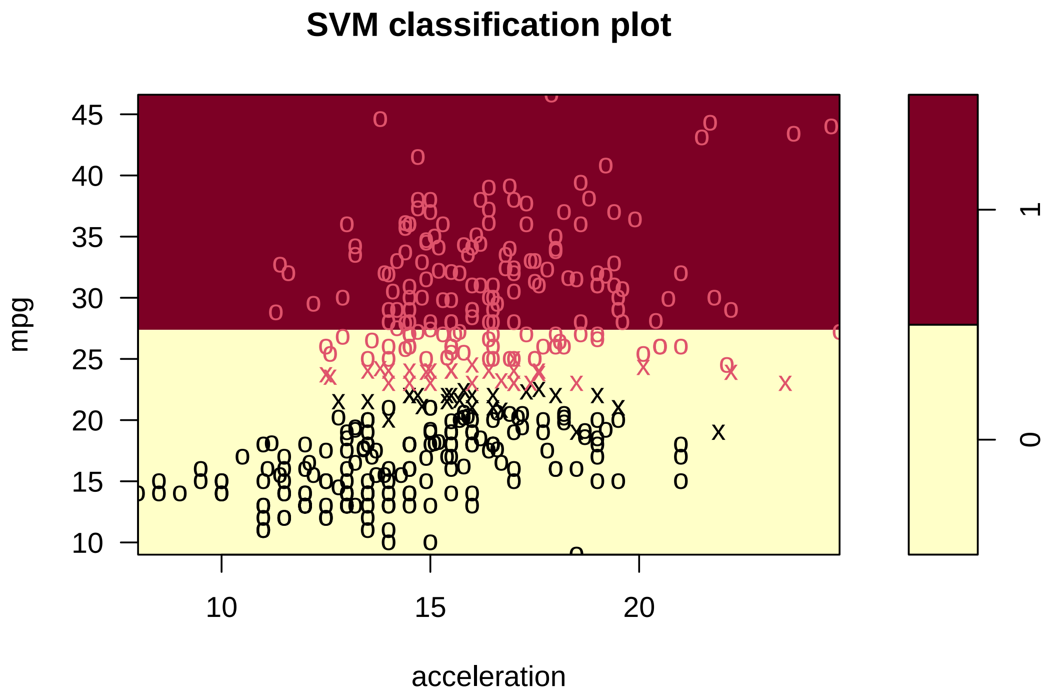
```
linear_SVM <- e1071::svm(
    formula = indicator_of_whether_gas_mileage_is_high ~ .,
    data = Auto,
    kernel = "linear",
    cost = 1
)
polynomial_SVM <- e1071::svm(
    formula = indicator_of_whether_gas_mileage_is_high ~ .,
    data = Auto,
    kernel = "polynomial",
    cost = 100,
    degree = 2
)
radial_SVM <- e1071::svm(
    formula = indicator_of_whether_gas_mileage_is_high ~ .,
    data = Auto,
    kernel = "radial",
    cost = 100,
    gamma = 0.01
)
plot_gas_mileage_vs_non_factor_predictor = function(SVM) {
    for (
        name_of_column in c(
            "cylinders",
            "displacement",
            "horsepower",
            "weight",
            "acceleration",
            "year",
            "origin"
        )
    ) {
        formula_as_string = paste("mpg ~ ", name_of_column, sep = "")
        formula = as.formula(formula_as_string)
        plot(x = SVM, data = Auto, formula = formula)
    }
}
plot_gas_mileage_vs_non_factor_predictor(linear_SVM)
```

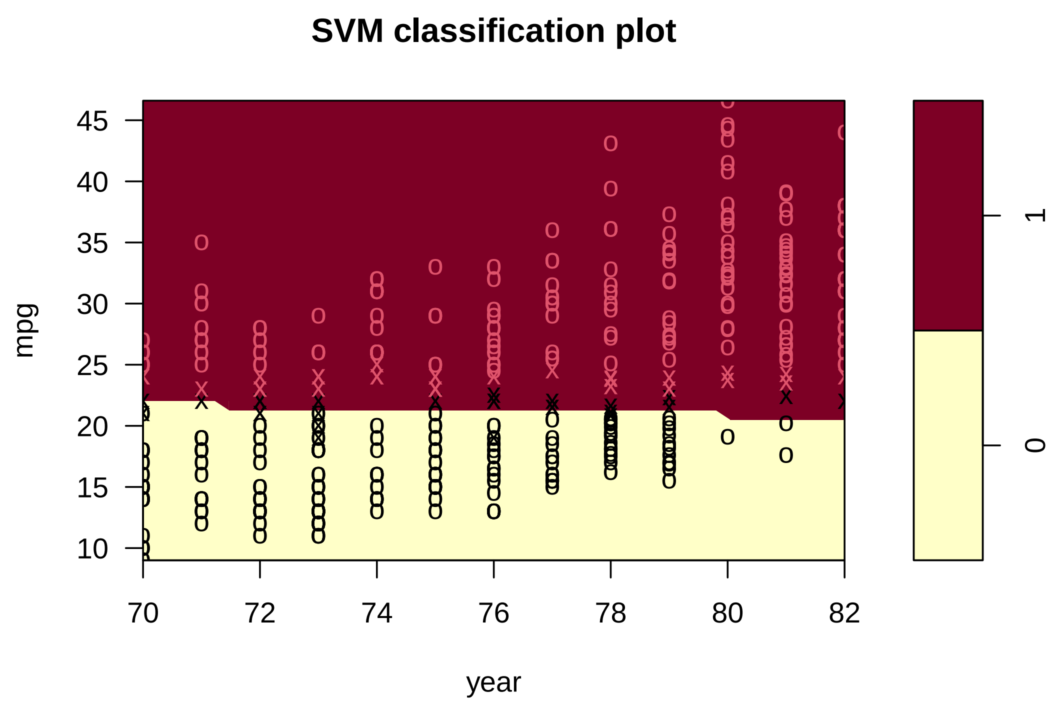


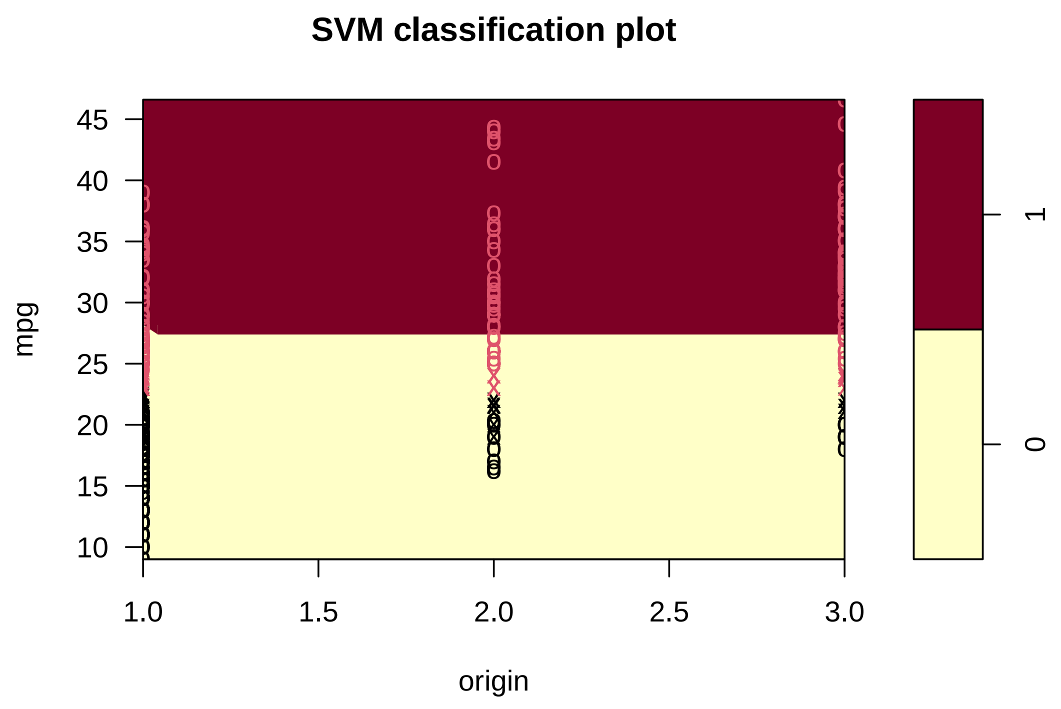

SVM classification plot
mpg
displacement
45
40
35
30
25
20
15
10
100
150
200
250
300
350
400
450
1
0


SVM classification plot
mpg
horsepower
45
40
35
30
25
20
15
10
50
100
150
200
1
0

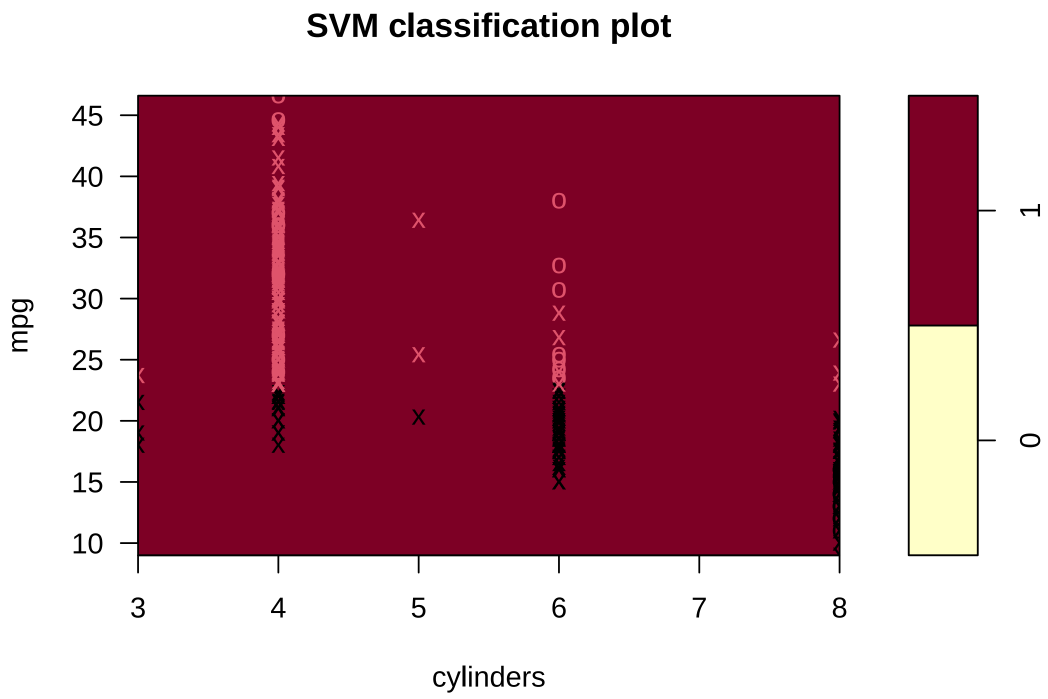

SVM classification plot
mpg
weight
45
40
35
30
25
20
15
10
2000
2500
3000
3500
4000
4500
5000
1
0

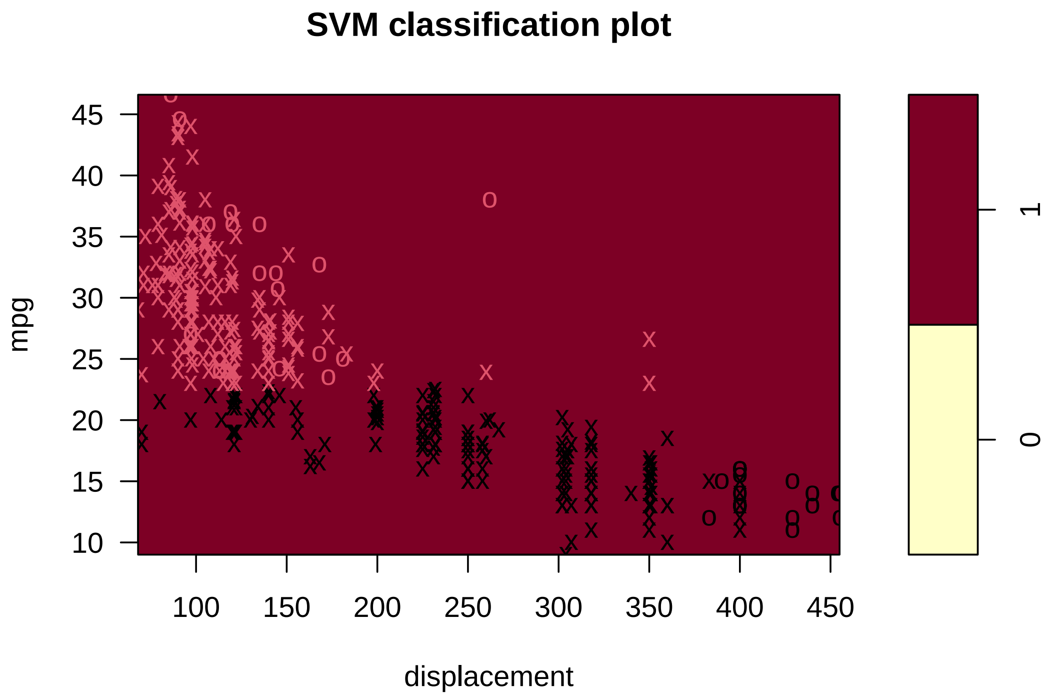

SVM classification plot
mpg
acceleration
45
40
35
30
25
20
15
10
10
15
20
1
0

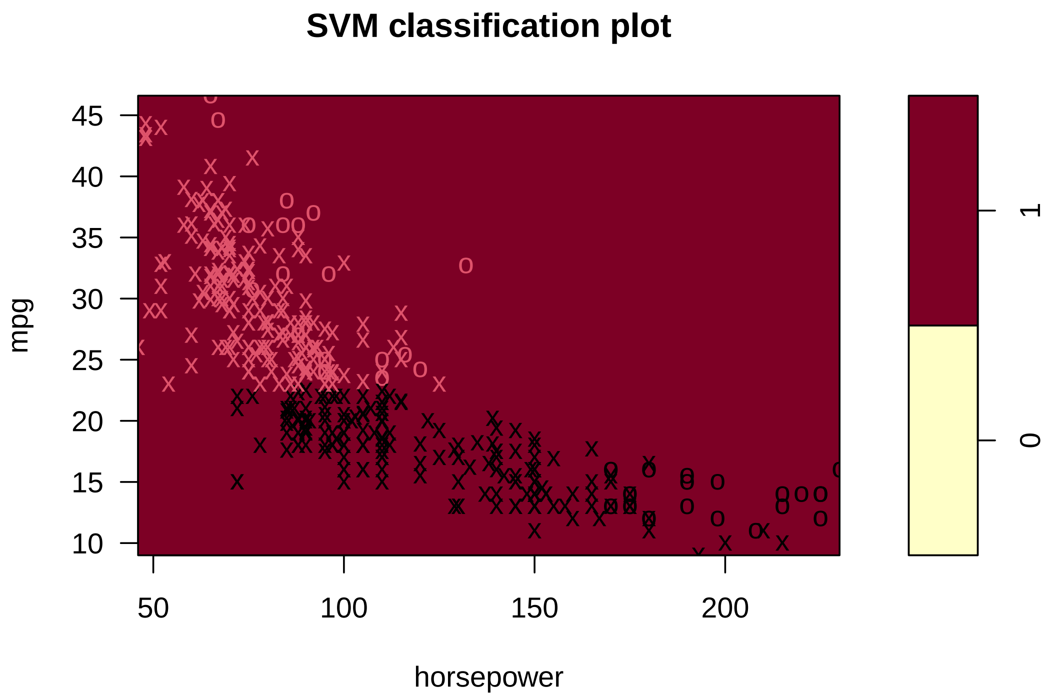


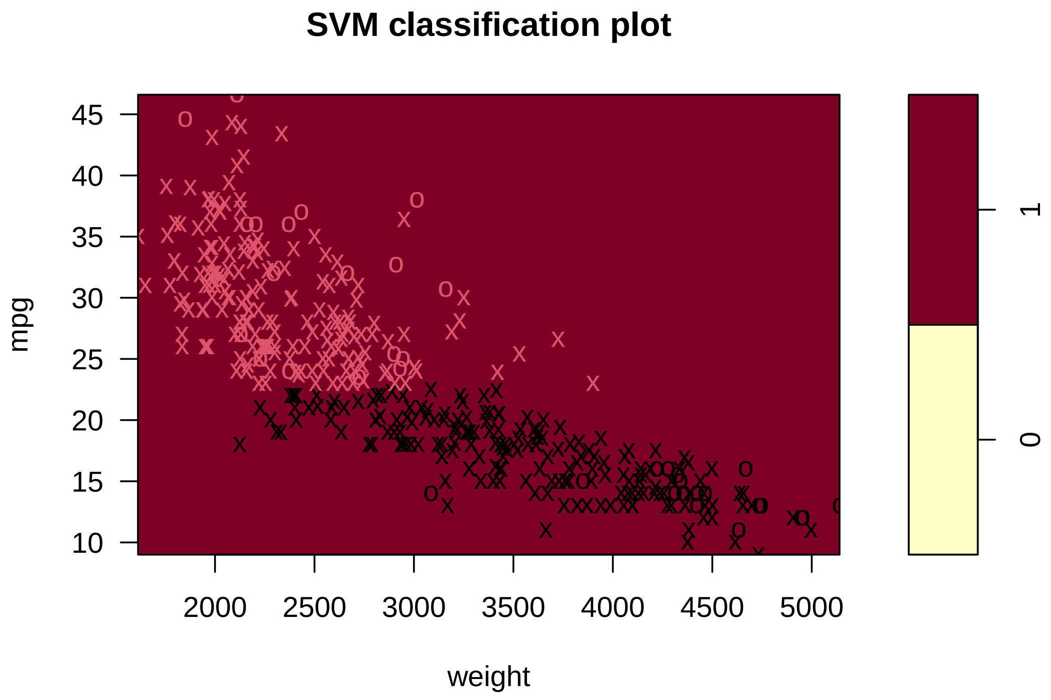


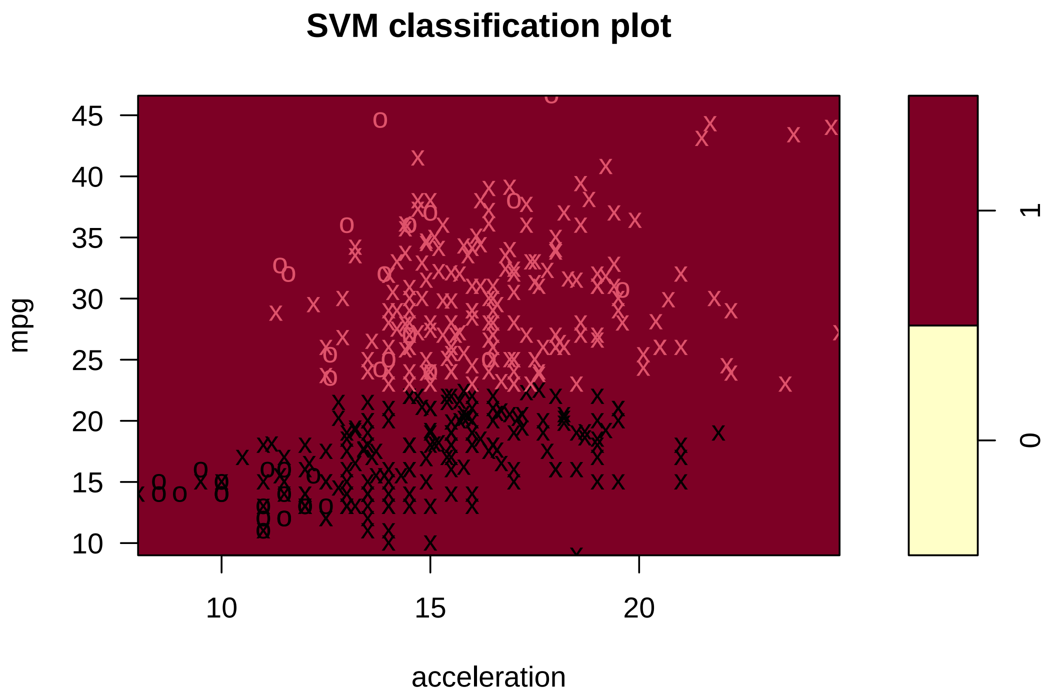
```r
plot_gas_mileage_vs_non_factor_predictor(polynomial_SVM)
```


SVM classification plot
mpg
cylinders
45
40
35
30
25
20
15
10
3
4
5
6
7
8
1
0

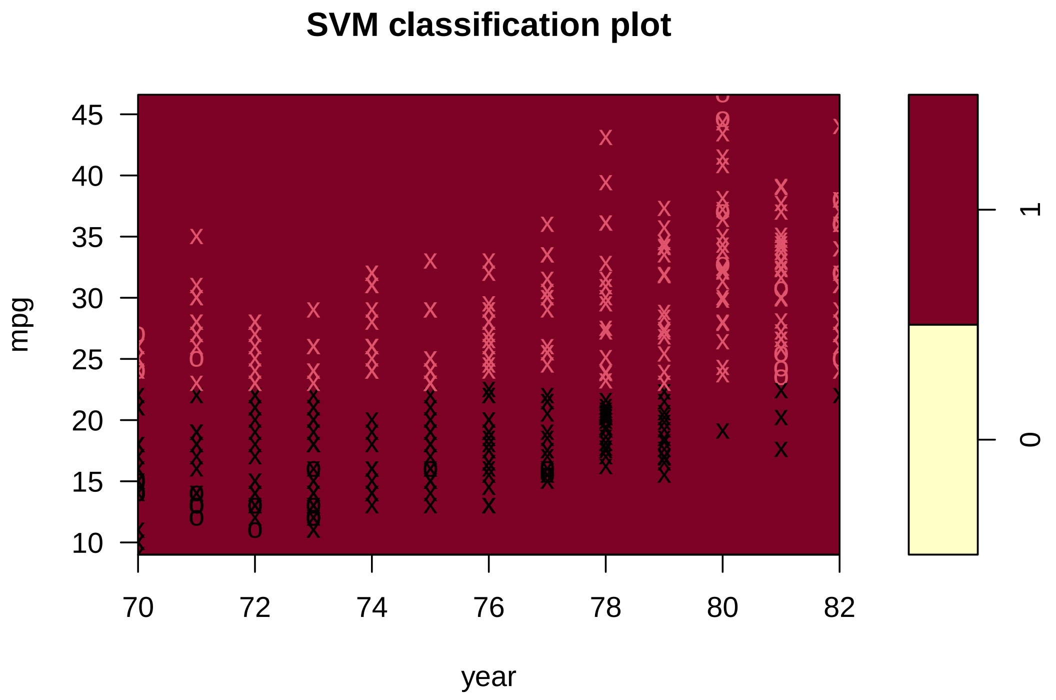

SVM classification plot
mpg
displacement
45
40
35
30
25
20
15
10
100
150
200
250
300
350
400
450
1
0


SVM classification plot
mpg
horsepower
45
40
35
30
25
20
15
10
50
100
150
200
1
0

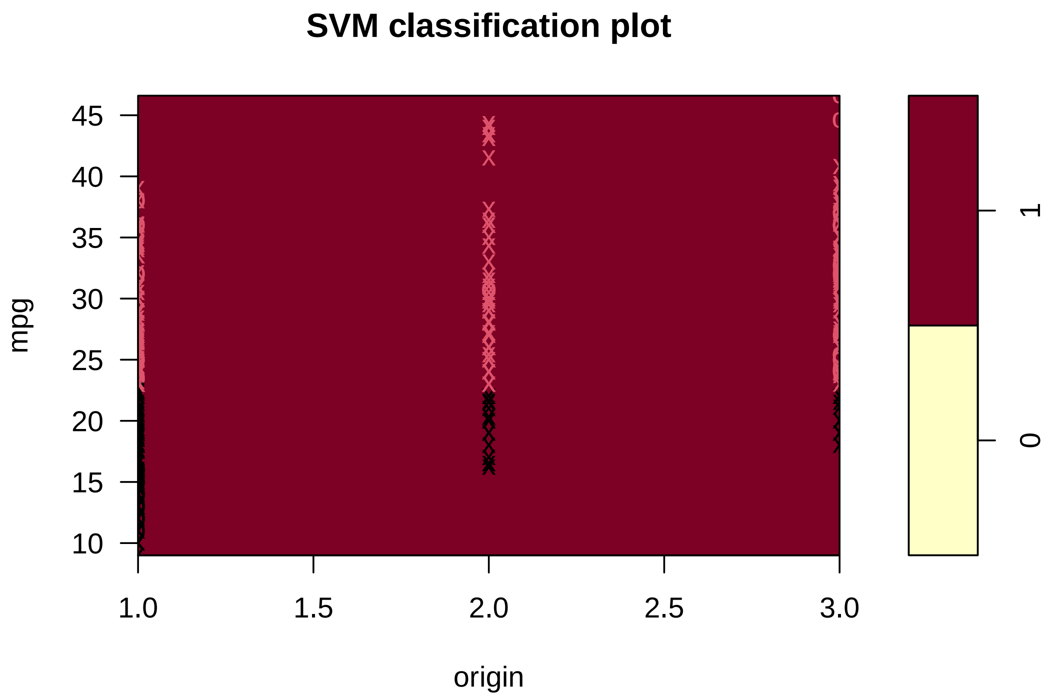

SVM classification plot
mpg
weight
45
40
35
30
25
20
15
10
2000
2500
3000
3500
4000
4500
5000
1
0



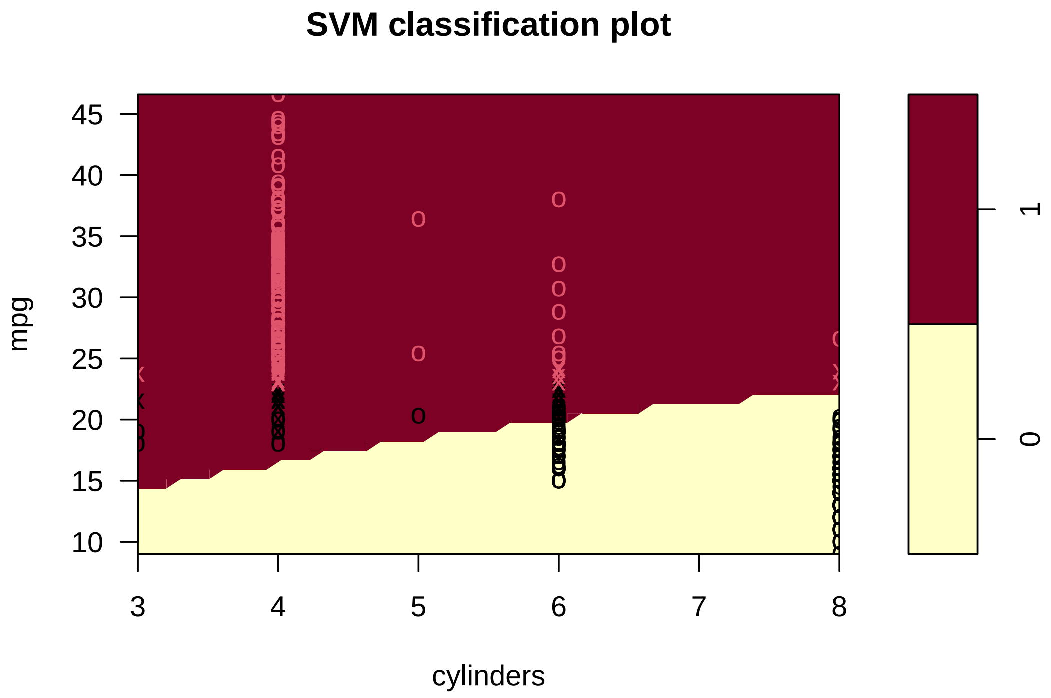
SVM classification plot
mpg
acceleration
45
40
35
30
25
20
15
10
10
15
20
1
0

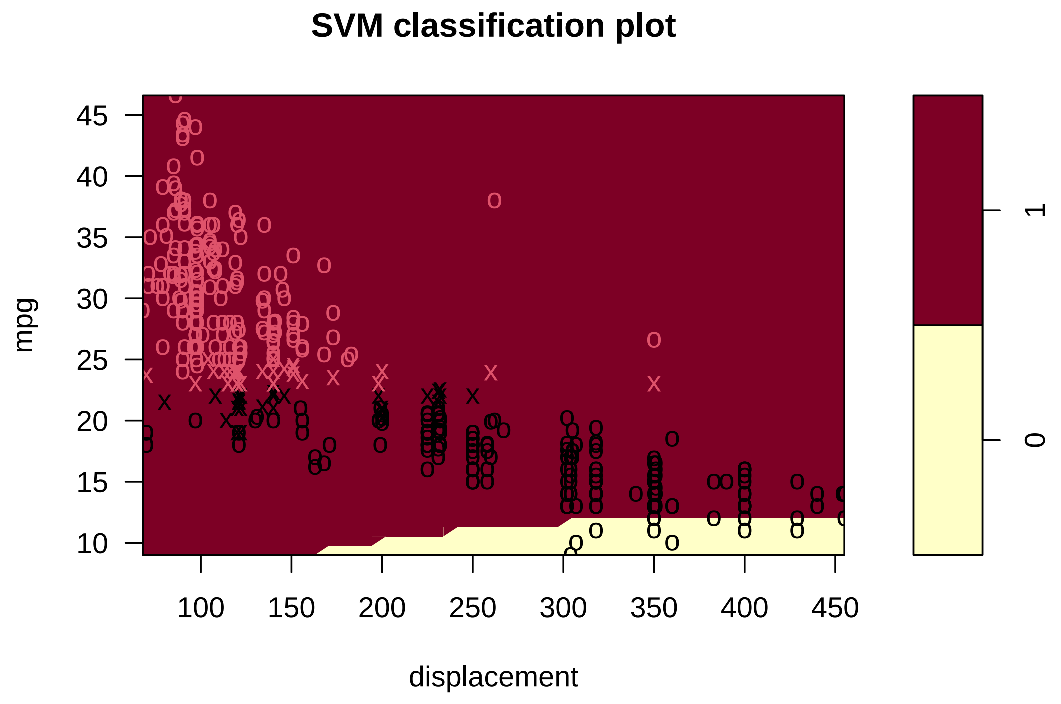


SVM classification plot
mpg
year
45
40
35
30
25
20
15
10
70
72
74
76
78
80
82
1
0

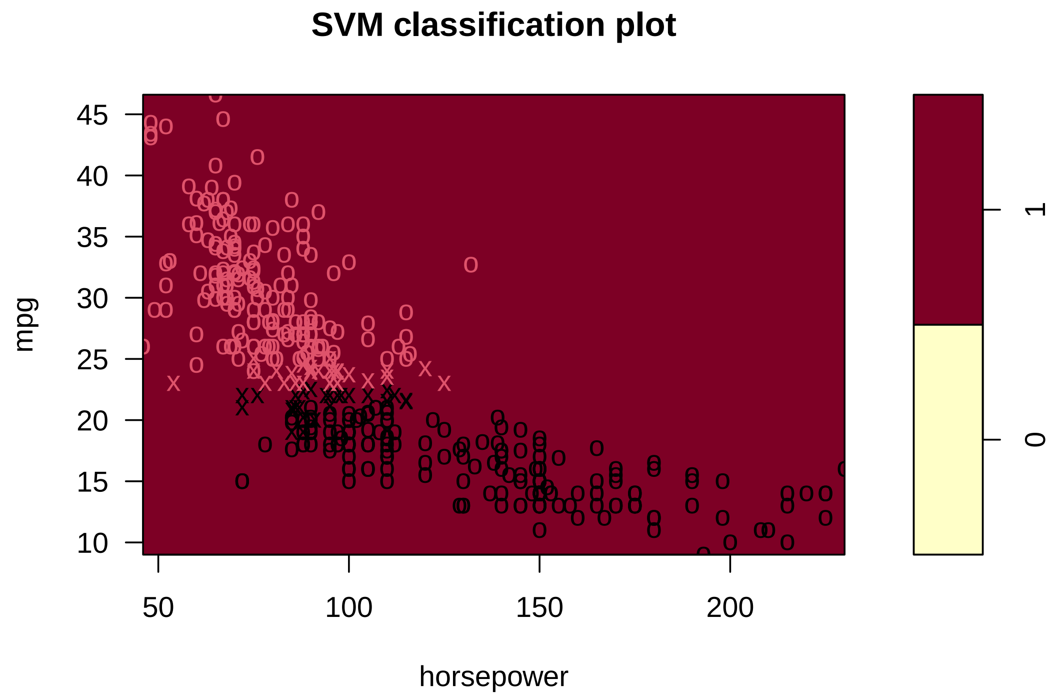


```r
plot_gas_mileage_vs_non_factor_predictor(radial_SVM)
```

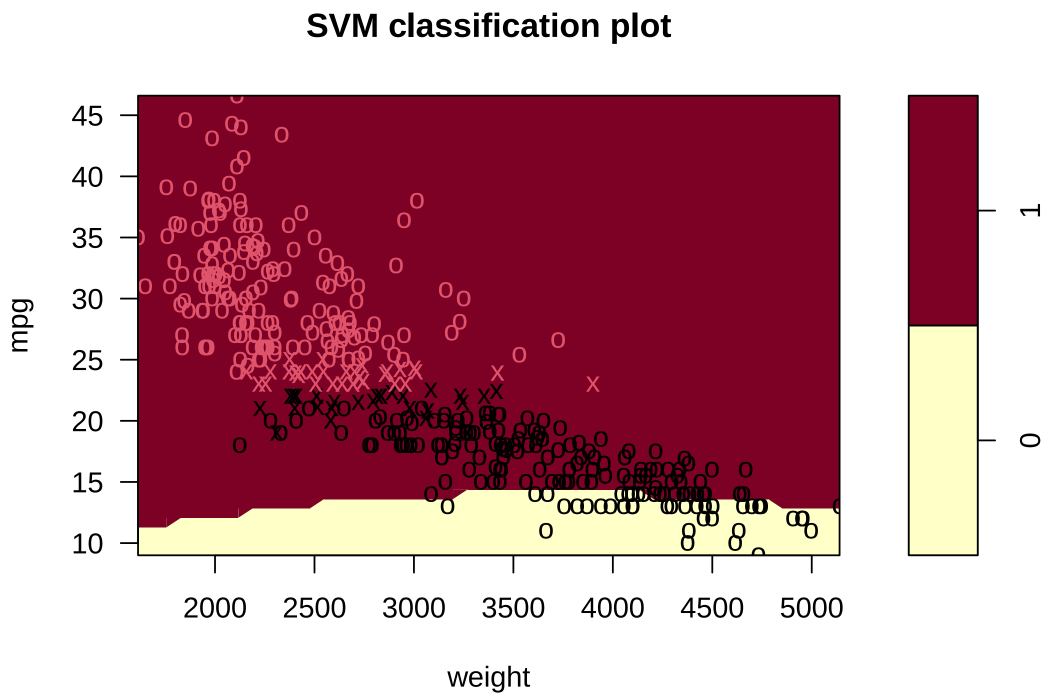


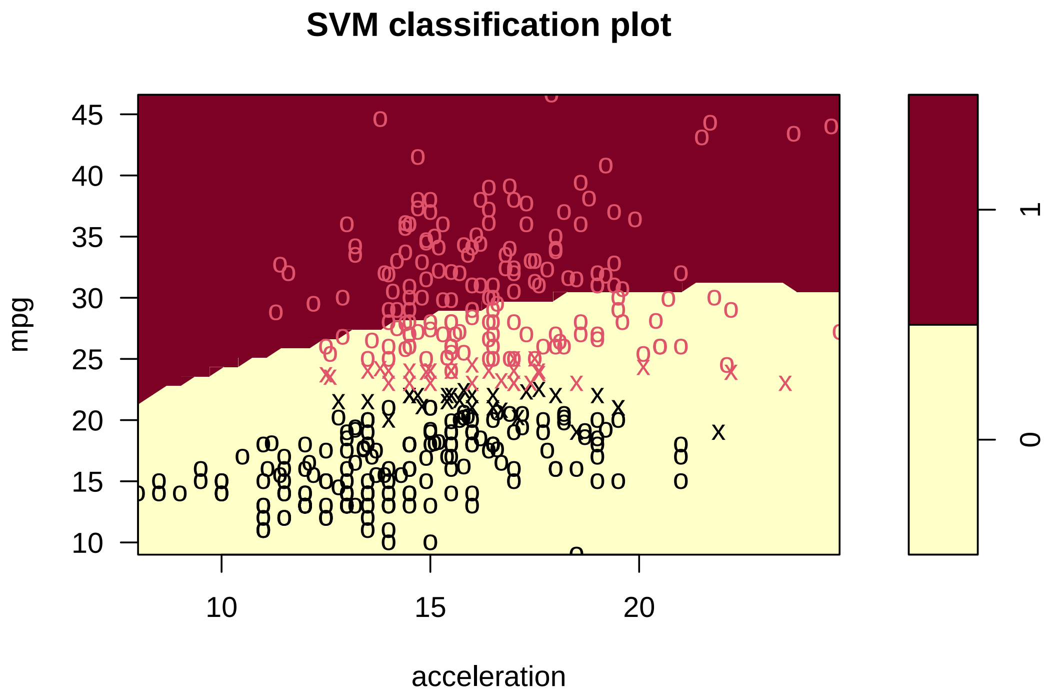


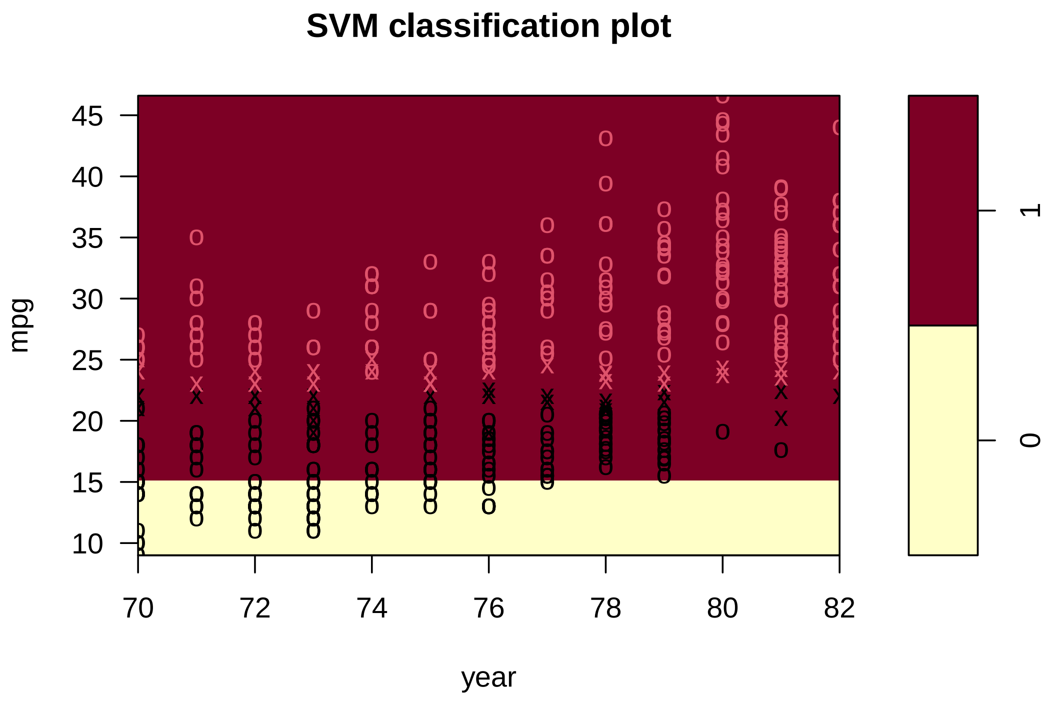
SVM classification plot
mpg
displacement
45
40
35
30
25
20
15
10
100
150
200
250
300
350
400
450
1
0

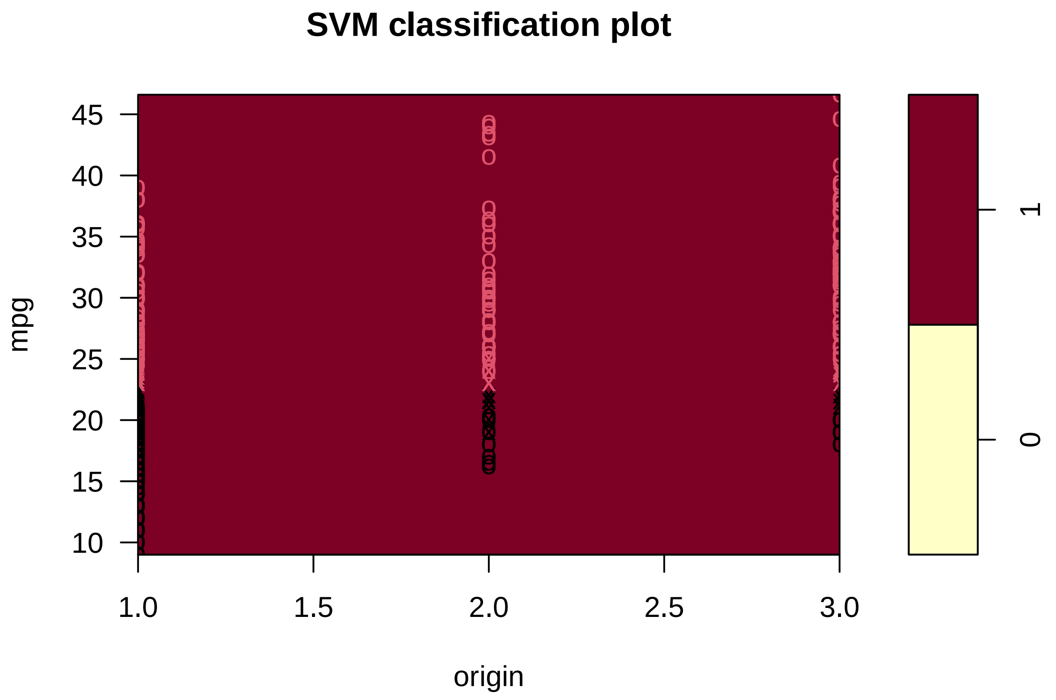


SVM classification plot
mpg
horsepower
45
40
35
30
25
20
15
10
50
100
150
200
1
0


SVM classification plot
mpg
weight
1
0


SVM classification plot
mpg
acceleration
1
0





X's indicate support vectors. Light pink indicates a point with an actual indicator of whether gas mileage is high of 1. Black indicates a point with an actual indicator of whether gas mileage is high of 0. Dark pink indicates a point with with a predicted indicator of whether gas mileage is high of 1. Yellow indicates a point with an predicted indicator of whether gas mileage is high of

0.

Our Linear SVM Classification Plots back up our assertions by showing that for non-factor predictors other than *year*, all points with an actual indicator of whether gas mileage is high of 0 are correctly classified and most points with an actual indicator of whether gas mileage is high of 1 are correctly classified.

Our Polynomial SVM Classification Plots suggest that a point is always classified as having having an indicator of whether gas mileage is high of 1.

Our Radial SVM Classification Plot for *mpg* and *acceleration* backs up our assertions similarly to our Linear SVM Classification Plots. Our Radial SVM Classification Plots for *mpg* and *horsepower* and *mpg* and *origin* suggests that a point is always classified as having an indicator of whether gas mileage is high of 1. Our Radial SVM Classification Plots for *mpg* and *cylinders*, *mpg* and *displacement*, *mpg* and *weight*, and *mpg* and *year* suggest that all points with an actual indicator of whether gas mileage is high of 1 are correctly classified and most points with an actual indicator of whether gas mileage is high of 0 are classified as having actual indicator of whether gas mileage is high of 1.

8. This problem involves the OJ data set which is part of the ISLR2 package.

   (a) Create a training set containing a random sample of 800 observations, and a test set containing the remaining observations.

```
training_and_testing_data <-
    TomLeversRPackage::split_data_set_into_training_and_testing_data(
        ISLR2::OJ,
        number_of_training_data = 800
    )
training_data <- training_and_testing_data$training_data
testing_data <- training_and_testing_data$testing_data
head(training_data, n = 2)
```

```
#     Purchase WeekofPurchase StoreID PriceCH PriceMM DiscCH DiscMM SpecialCH
# 133       CH            272       2    1.86    2.18      0   0.06         0
# 294       MM            273       2    1.86    2.18      0   0.06         0
#     SpecialMM  LoyalCH SalePriceMM SalePriceCH PriceDiff Store7 PctDiscMM
# 133         0 0.955260        2.12        1.86      0.26     No  0.027523
# 294         0 0.000317        2.12        1.86      0.26     No  0.027523
#     PctDiscCH ListPriceDiff STORE
# 133         0          0.32     2
# 294         0          0.32     2
```

```
head(testing_data, n = 2)
```

```
#     Purchase WeekofPurchase StoreID PriceCH PriceMM DiscCH DiscMM SpecialCH
# 37        CH            258       1    1.76    2.18      0      0         0
# 863       CH            256       7    1.86    2.18      0      0         0
#     SpecialMM LoyalCH SalePriceMM SalePriceCH PriceDiff Store7 PctDiscMM
# 37          0 0.70816        2.18        1.76      0.42     No         0
# 863         0 0.40000        2.18        1.86      0.32    Yes         0
#     PctDiscCH ListPriceDiff STORE
# 37          0          0.42     1
# 863         0          0.32     0
```

   (b) Fit a support vector classifier to the training data using `cost=0.01`, with `Purchase` as the response and the other variables as predictors. Use the `summary()` function to produce summary statistics, and describe the results obtained.

```r
set.seed(1)
linear_SVM <- e1071::svm(
    Purchase ~ .,
    data = training_data,
    kernel = "linear",
    cost = 0.01
)
summary(linear_SVM)
```

```
#
# Call:
# svm(formula = Purchase ~ ., data = training_data, kernel = "linear",
#     cost = 0.01)
#
#
# Parameters:
#    SVM-Type:  C-classification
#  SVM-Kernel:  linear
#        cost:  0.01
#
# Number of Support Vectors:  436
#
#  ( 219 217 )
#
#
# Number of Classes:  2
#
# Levels:
#  CH MM
```

For our support vector classifier / support vector machine with linear kernel, 437 of 800 training points are support vectors. 218 of the support vectors correspond to purchases of Minute-Maid orange juice. 219 of the support vectors correspond to purchases of Citrus Hill orange juice.

(c) What are the training and test error rates?

```r
vector_of_predicted_purchases <- predict(linear_SVM, training_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    training_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                         vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                      CH 443  56
#                      MM  71 230
#
# $error_rate
# [1] 0.15875
```

The training error rate is immediately above.

```
vector_of_predicted_purchases <- predict(linear_SVM, testing_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    testing_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                       vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                      CH 126  28
#                      MM  27  89
#
# $error_rate
# [1] 0.2037037
```

The testing error rate is immediately above.

(d) Use the `tune()` function to select an optimal `cost`. Consider values in the range 0.01 to 10.

```
set.seed(1)
the_tune <- e1071::tune(
    METHOD = e1071::svm,
    train.x = Purchase ~ .,
    data = training_data,
    kernel = "linear",
    ranges = list(
        cost = 10^seq(from = -2, to = 1, by = 0.25)
    )
)
the_summary <- summary(the_tune)
the_summary
```

```
#
# Parameter tuning of 'e1071::svm':
#
# - sampling method: 10-fold cross validation
#
# - best parameters:
#        cost
#  0.01778279
#
# - best performance: 0.1575
#
# - Detailed performance results:
#           cost   error dispersion
# 1   0.01000000 0.16500 0.04199868
# 2   0.01778279 0.15750 0.03446012
# 3   0.03162278 0.16125 0.04016027
# 4   0.05623413 0.16000 0.03622844
# 5   0.10000000 0.16375 0.03884174
# 6   0.17782794 0.16750 0.03736085
# 7   0.31622777 0.16500 0.03855011
# 8   0.56234133 0.16375 0.03972562
# 9   1.00000000 0.16750 0.04005205
# 10  1.77827941 0.16875 0.04135299
# 11  3.16227766 0.16750 0.04174992
# 12  5.62341325 0.16500 0.03899786
# 13 10.00000000 0.16625 0.03866254
```

```r
best_cost <- the_summary$best.parameters["cost"]
best_cost
```

```
#         cost
# 2 0.01778279
```

The optimal cost is immediately above.

(e) Compute the training and test error rates using this new value for `cost`.

```r
linear_SVM <- e1071::svm(
    Purchase ~ .,
    data = training_data,
    kernel = "linear",
    cost = best_cost
)
vector_of_predicted_purchases <- predict(linear_SVM, training_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    training_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                         vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                       CH 445  54
#                       MM  69 232
#
# $error_rate
# [1] 0.15375
```

The training error rate is given immediately above.

```r
vector_of_predicted_purchases <- predict(linear_SVM, testing_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    testing_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                         vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                       CH 126  28
#                       MM  27  89
#
# $error_rate
# [1] 0.2037037
```

The testing error rate is given immediately above.

(f) Repeat parts (b) through (e) using a support vector machine with a radial kernel. Use the default value for gamma.

```
set.seed(1)
radial_SVM <- e1071::svm(
    Purchase ~ .,
    data = training_data,
    kernel = "radial",
    cost = 0.01
)
summary(radial_SVM)
```

```
#
# Call:
# svm(formula = Purchase ~ ., data = training_data, kernel = "radial",
#     cost = 0.01)
#
#
# Parameters:
#    SVM-Type:  C-classification
#  SVM-Kernel:  radial
#        cost:  0.01
#
# Number of Support Vectors:  605
#
#  ( 304 301 )
#
#
# Number of Classes:  2
#
# Levels:
#  CH MM
```

For our support vector machine with radial kernel, 639 of 800 training points are support vectors. 320 of the support vectors correspond to purchases of Minute-Maid orange juice. 319 of the support vectors correspond to purchases of Citrus Hill orange juice.

```
vector_of_predicted_purchases <- predict(radial_SVM, training_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    training_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                          vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                       CH 499   0
#                       MM 301   0
#
# $error_rate
# [1] 0.37625
```

The training error rate is immediately above.

```r
vector_of_predicted_purchases <- predict(radial_SVM, testing_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    testing_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                          vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                       CH 154   0
#                       MM 116   0
#
# $error_rate
# [1] 0.4296296
```

The testing error rate is immediately above.

```r
set.seed(1)
the_tune <- e1071::tune(
    METHOD = e1071::svm,
    train.x = Purchase ~ .,
    data = training_data,
    kernel = "radial",
    ranges = list(
        cost = 10^seq(from = -2, to = 1, by = 0.25)
    )
)
the_summary <- summary(the_tune)
the_summary
```

```
#
# Parameter tuning of 'e1071::svm':
#
# - sampling method: 10-fold cross validation
#
# - best parameters:
#       cost
#  0.3162278
#
# - best performance: 0.16375
#
# - Detailed performance results:
#            cost   error dispersion
# 1   0.01000000 0.37625 0.06467064
# 2   0.01778279 0.37625 0.06467064
# 3   0.03162278 0.37500 0.06871843
# 4   0.05623413 0.20375 0.03120831
# 5   0.10000000 0.17250 0.03162278
# 6   0.17782794 0.16750 0.03238227
# 7   0.31622777 0.16375 0.03653860
# 8   0.56234133 0.16500 0.02993047
# 9   1.00000000 0.16500 0.02486072
# 10  1.77827941 0.16625 0.02766993
# 11  3.16227766 0.16750 0.03073181
# 12  5.62341325 0.17000 0.03446012
# 13 10.00000000 0.17375 0.03606033
```

```
best_cost <- the_summary$best.parameters["cost"]
best_cost
```

```
#        cost
# 7 0.3162278
```

The optimal cost is immediately above.

```
radial_SVM <- e1071::svm(
    Purchase ~ .,
    data = training_data,
    kernel = "radial",
    cost = best_cost
)
vector_of_predicted_purchases <- predict(radial_SVM, training_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    training_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                         vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                       CH 455  44
#                       MM  76 225
#
# $error_rate
# [1] 0.15
```

The training error rate is given immediately above.

```
vector_of_predicted_purchases <- predict(radial_SVM, testing_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    testing_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                         vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                       CH 133  21
#                       MM  33  83
#
# $error_rate
# [1] 0.2
```

The testing error rate is given immediately above.

(g) Repeat parts (b) through (e) using a support vector machine with a polynomial kernel. Set `degree=2`.

```r
set.seed(1)
polynomial_SVM <- e1071::svm(
    Purchase ~ .,
    data = training_data,
    kernel = "polynomial",
    cost = 0.01,
    degree = 2
)
summary(polynomial_SVM)
```

```
#
# Call:
# svm(formula = Purchase ~ ., data = training_data, kernel = "polynomial",
#     cost = 0.01, degree = 2)
#
#
# Parameters:
#    SVM-Type:  C-classification
#  SVM-Kernel:  polynomial
#        cost:  0.01
#      degree:  2
#      coef.0:  0
#
# Number of Support Vectors:  606
#
#  ( 305 301 )
#
#
# Number of Classes:  2
#
# Levels:
#  CH MM
```

For our support vector machine with polynomial kernel, 619 of 800 training points are support vectors. 307 of the support vectors correspond to purchases of Minute-Maid orange juice. 312 of the support vectors correspond to purchases of Citrus Hill orange juice.

```r
vector_of_predicted_purchases <- predict(polynomial_SVM, training_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    training_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                          vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                      CH 499   0
#                      MM 301   0
#
# $error_rate
# [1] 0.37625
```

The training error rate is immediately above.

```r
vector_of_predicted_purchases <- predict(polynomial_SVM, testing_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    testing_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                         vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                       CH 154   0
#                       MM 116   0
#
# $error_rate
# [1] 0.4296296
```

The testing error rate is immediately above.

```r
set.seed(1)
the_tune <- e1071::tune(
    METHOD = e1071::svm,
    train.x = Purchase ~ .,
    data = training_data,
    kernel = "polynomial",
    ranges = list(
        cost = 10^seq(from = -2, to = 1, by = 0.25),
        degree = 2
    )
)
the_summary <- summary(the_tune)
the_summary
```

```
#
# Parameter tuning of 'e1071::svm':
#
# - sampling method: 10-fold cross validation
#
# - best parameters:
#  cost degree
#    10      2
#
# - best performance: 0.175
#
# - Detailed performance results:
#            cost degree   error dispersion
# 1    0.01000000      2 0.37625 0.06467064
# 2    0.01778279      2 0.36000 0.06582806
# 3    0.03162278      2 0.35125 0.06136469
# 4    0.05623413      2 0.32250 0.04241004
# 5    0.10000000      2 0.30750 0.04533824
# 6    0.17782794      2 0.24375 0.04340139
# 7    0.31622777      2 0.19375 0.03547789
# 8    0.56234133      2 0.19375 0.03346329
# 9    1.00000000      2 0.18875 0.03508422
# 10  1.77827941      2 0.18250 0.03184162
# 11  3.16227766      2 0.18125 0.03240906
# 12  5.62341325      2 0.18000 0.03184162
# 13 10.00000000      2 0.17500 0.03280837
```

```r
best_cost <- the_summary$best.parameters["cost"]
best_cost
```

```
#    cost
# 13   10
```

The optimal cost is immediately above.

```r
polynomial_SVM <- e1071::svm(
    Purchase ~ .,
    data = training_data,
    kernel = "polynomial",
    cost = best_cost,
    degree = 2
)
vector_of_predicted_purchases <- predict(polynomial_SVM, training_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    training_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                         vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                       CH 464  35
#                       MM  77 224
#
# $error_rate
# [1] 0.14
```

The training error rate is given immediately above.

```r
vector_of_predicted_purchases <- predict(polynomial_SVM, testing_data)
summary_of_performance <- TomLeversRPackage::summarize_performance_of_model(
    testing_data$Purchase,
    vector_of_predicted_purchases
)
summary_of_performance
```

```
# $confusion_matrix
#                         vector_of_predicted_values
# vector_of_actual_values  CH  MM
#                       CH 139  15
#                       MM  35  81
#
# $error_rate
# [1] 0.1851852
```

The testing error rate is given immediately above.

(h) Overall, which approach seems to give the best results on this data?

For one evaluation of test error rate for support vector machines with linear, radial, and polynomial kernels, the support vector machines with linear and radial kernels have lowest test error rates of 0.1814815.