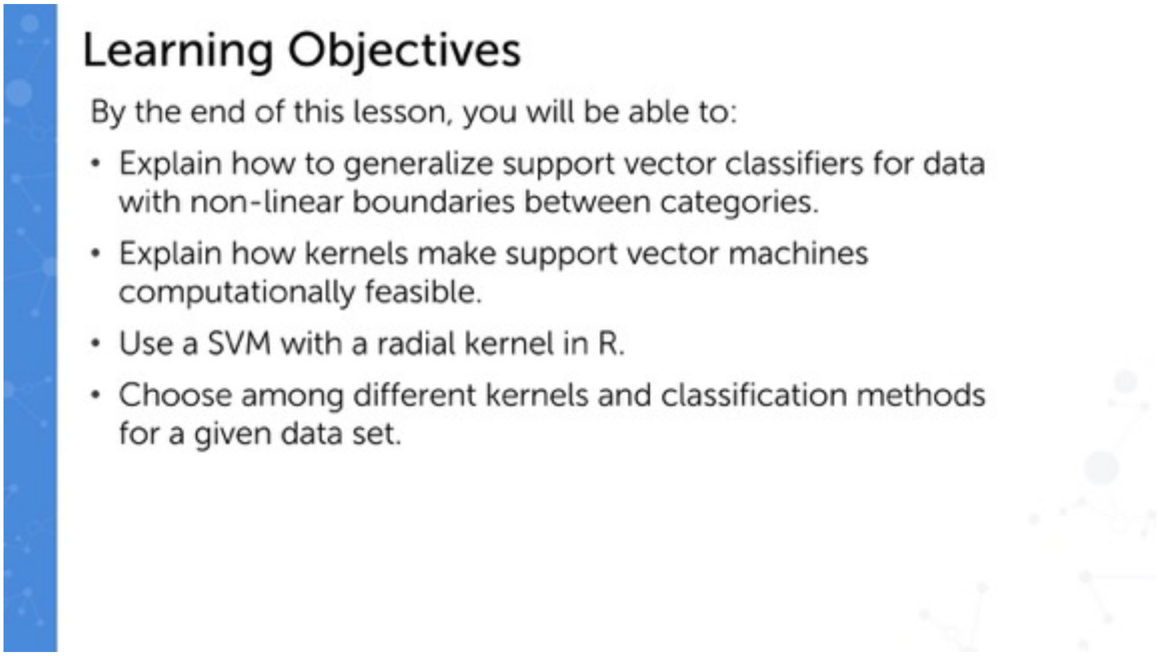
UNIVERSITY OF WISCONSIN
DATA SCIENCE

DS 740

Data Mining

Support Vector Machines (SVMs)

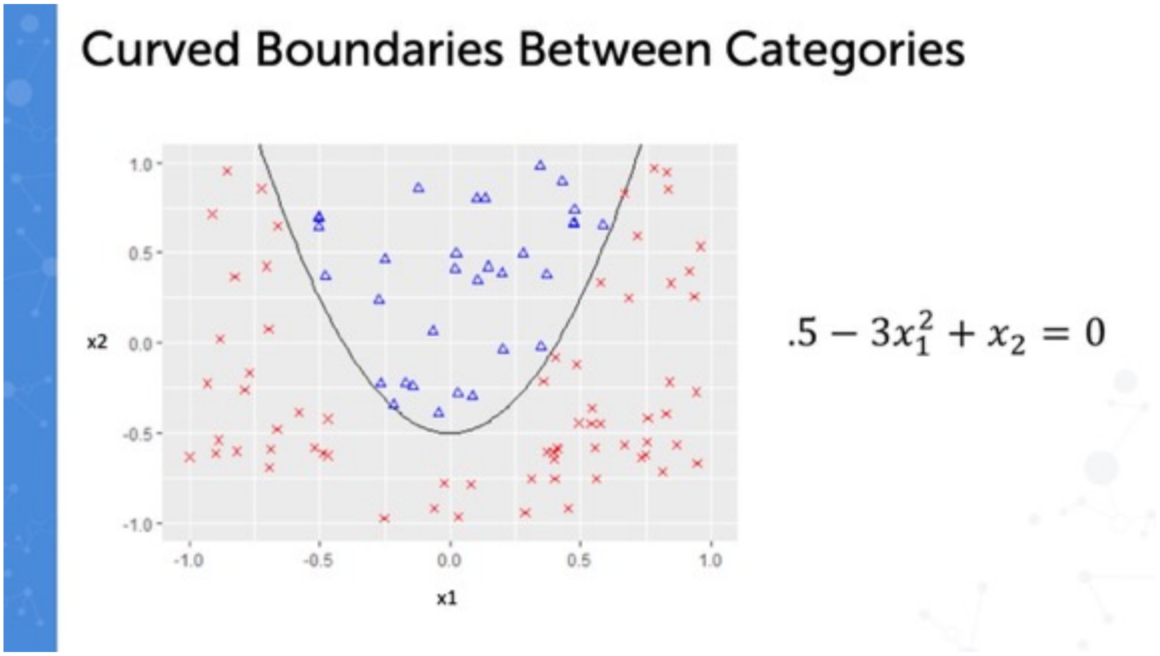
Important note: Transcripts are **not** substitutes for textbook assignments.



Learning Objectives

By the end of this lesson, you will be able to:

- Explain how to generalize support vector classifiers for data with non-linear boundaries between categories.
- Explain how kernels make support vector machines computationally feasible.
- Use a SVM with a radial kernel in R.
- Choose among different kernels and classification methods for a given data set.



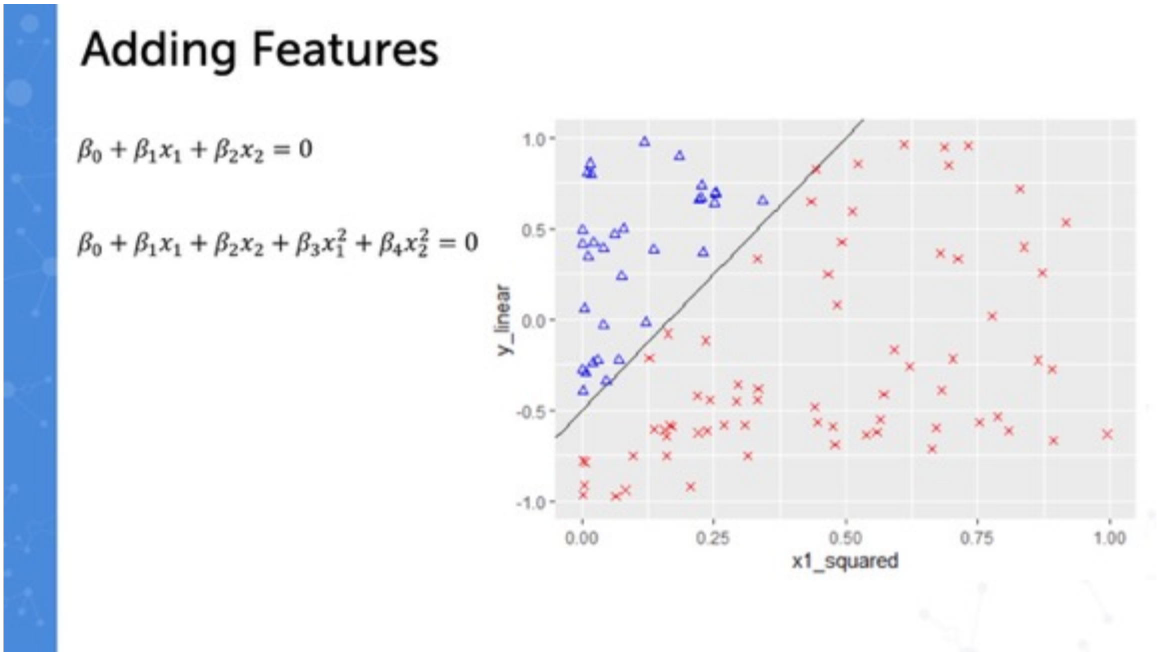
Curved Boundaries Between Categories

$$.5 - 3x_1^2 + x_2 = 0$$

A support vector machine is a generalization of a support vector classifier that allows for curved boundaries between categories. For example, the data set shown here can't be separated using a straight line. But it can be separated using a parabola. In this case, the parabola 0.5 minus 3 times x1 squared plus x2 equals 0.

Notes:

```
n = 100

set.seed(516)
poly_example = data.frame(x1 = runif(n, -1, 1),
                          x2 = runif(n, -1, 1))
poly_example <- poly_example %>%
  mutate(y = factor(ifelse(.5 - 3*x1^2 + x2 > 0, 1, -1)))

polynomial = data.frame(x_poly = seq(-1, 1, by = .01))
polynomial <- polynomial %>%
  mutate(y_poly = -.5 + 3*x_poly^2)

gf_line(y_poly ~ x_poly, data = polynomial) %>%
  gf_point(x2 ~ x1, color =~ y, pch =~ y, data = poly_example) %>%
  gf_refine(scale_shape_manual(values = c(4,2)),
            scale_color_manual(values = c("red", "blue")),
            coord_cartesian(ylim = c(-1,1)))
```



Adding Features

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 = 0$$

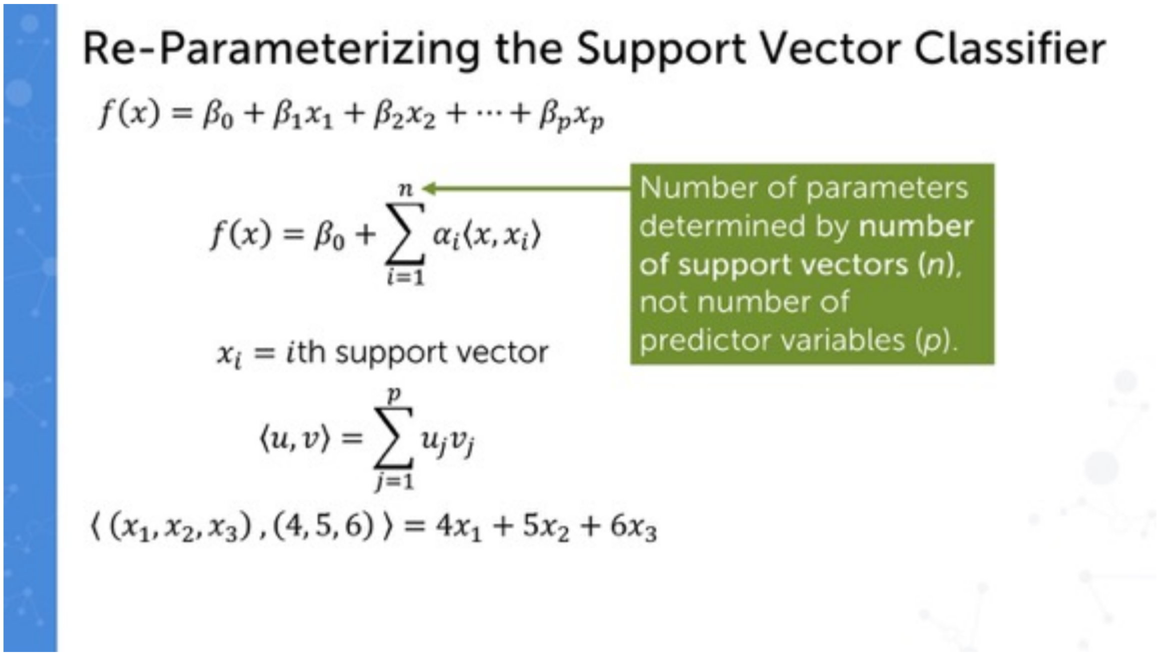
$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_2^2 = 0$$

In this case, we can convert our curved boundary between categories into a linear boundary by graphing x sub 2 versus the square of x sub 1. Essentially, we're converting our support vector machine problem into a support vector classifier problem by adding features or predictor variables. In this case, x sub 1 squared. So instead of the equation for a line shown here, we have the equation for a curve involving x sub 1 squared and x sub 2 squared. But you could also think of this as the equation for a hyperplane where x sub 1 squared is itself a feature or predictor variable. This is very similar to adding higher order or interaction terms in multiple linear regression.

But you can see that with a complicated boundary between categories, this could get very complicated very fast by having to add lots and lots of features. And we don't want our problem to become so complicated that we have to spend hours searching for an optimal separator between the categories.

Notes:

```
poly_example <- poly_example %>%
  mutate(x1_squared = x1^2)

linear = data.frame(x1_squared = seq(-.25, 1, by = .01))
linear <- linear %>%
  mutate(y_linear = -.5 + 3*x1_squared)

gf_line(y_linear ~ x1_squared, data = linear) %>%
  gf_point(x2 ~ x1_squared, color =~ y, pch =~ y,
           data = poly_example) %>%
  gf_refine(scale_shape_manual(values = c(4,2)),
            scale_color_manual(values = c("red", "blue")),
            coord_cartesian(xlim = c(0,1), ylim = c(-1,1)))
```



Re-Parameterizing the Support Vector Classifier

$$f(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p$$

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

Number of parameters determined by **number of support vectors (*n*)**, not number of predictor variables (*p*).

$x_i = i$th support vector

$$\langle u, v \rangle = \sum_{j=1}^{p} u_j v_j$$

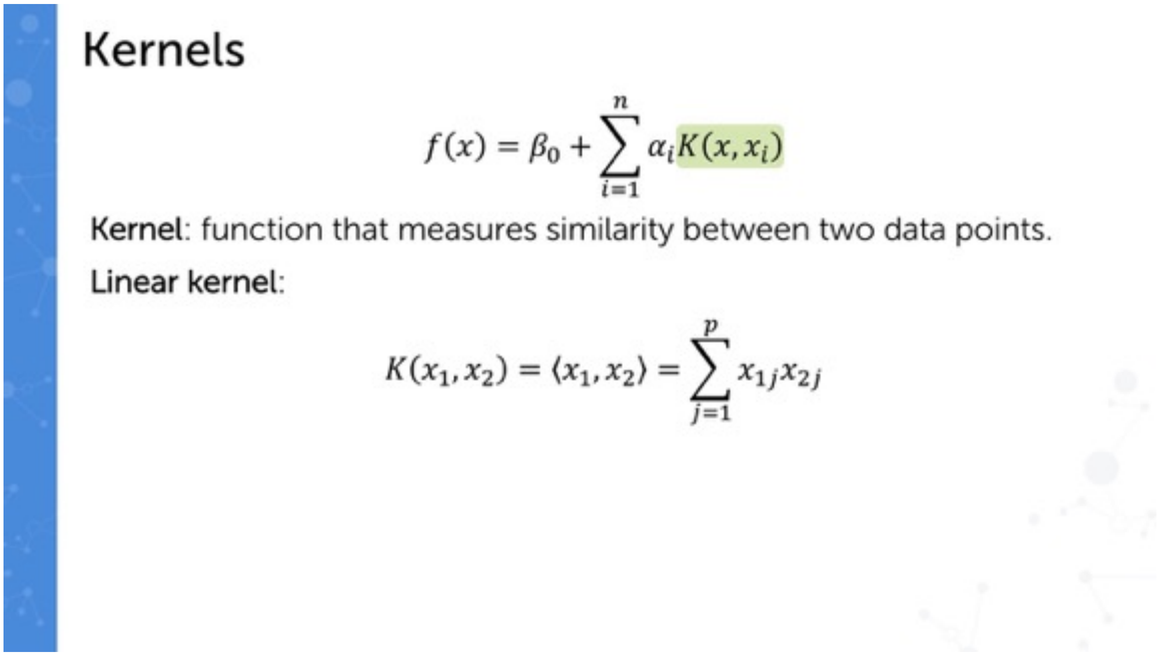
$$\langle (x_1, x_2, x_3), (4, 5, 6) \rangle = 4x_1 + 5x_2 + 6x_3$$

Fortunately, we can keep our problem computationally feasible by reparameterizing the support vector classifier. Previously, we've written the support vector classifier as a function f of x with the format shown here, where the number of coefficients, the betas, is determined by p, the number of predictor variables.

So this is the number of parameters we have to optimize over in order to choose the best hyperplane. However, this function can also be written in this format, where x is, again, a variable representing whichever point we're trying to classify. And x sub i now represents the i-th support vector. The angle brackets represent the inner product, which is the sum of the products of the corresponding elements of each of two vectors.

So for example, the inner product of x1, x2, x3 with the vector 4, 5, 6 is 4 times x1 plus 5 times x2, plus 6 times x3. Using this new way of writing the support vector classifier, we can represent the same hyperplane as we did before. But now the parameters were trying to optimize over are beta sub 0 and the alpha sub i's. So the number of parameters is now determined by n, the number of support vectors, not p, the number of predictor variables.

So now the complexity of our optimization problem is limited by the number of support vectors, not the number of predictor variables. That's good news if you have lots of predictor variables. Potentially even more predictor variables then you have data points.



Kernels

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

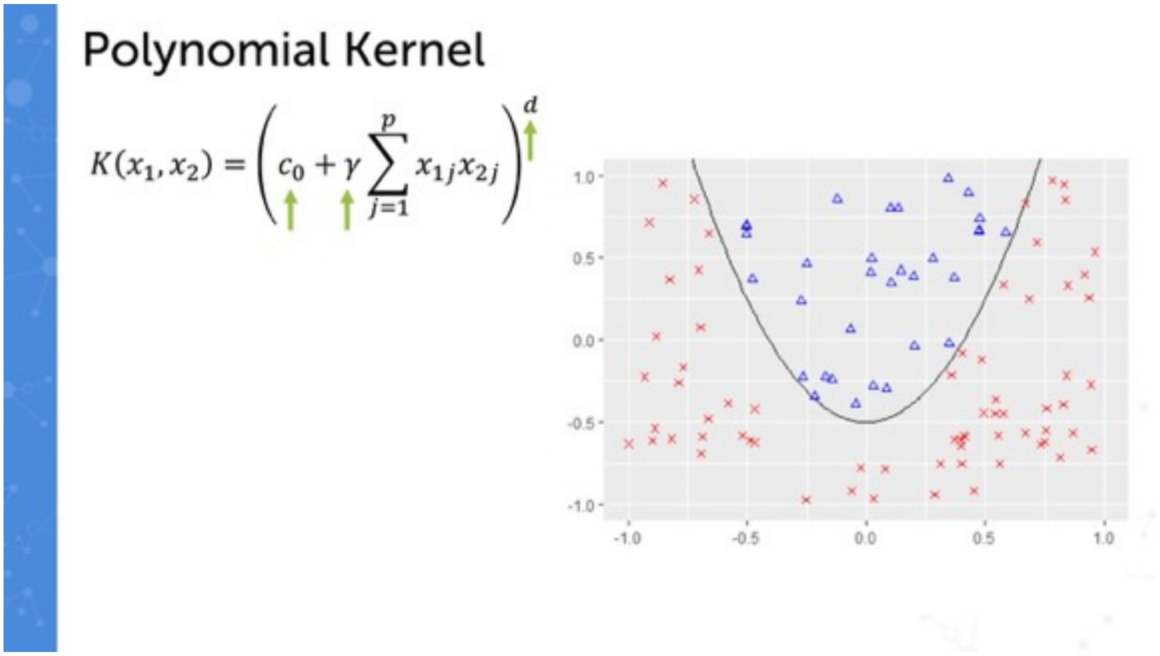
Kernel: function that measures similarity between two data points.

Linear kernel:

$$K(x_1, x_2) = \langle x_1, x_2 \rangle = \sum_{j=1}^{p} x_{1j} x_{2j}$$

We can make this reparameterization more general using kernels. Here, we've written the same function f of x as we saw in the previous slide, but we've replaced the inner product with a kernel function, k. A kernel is any function that measures the similarity between two data points, x and x sub i.

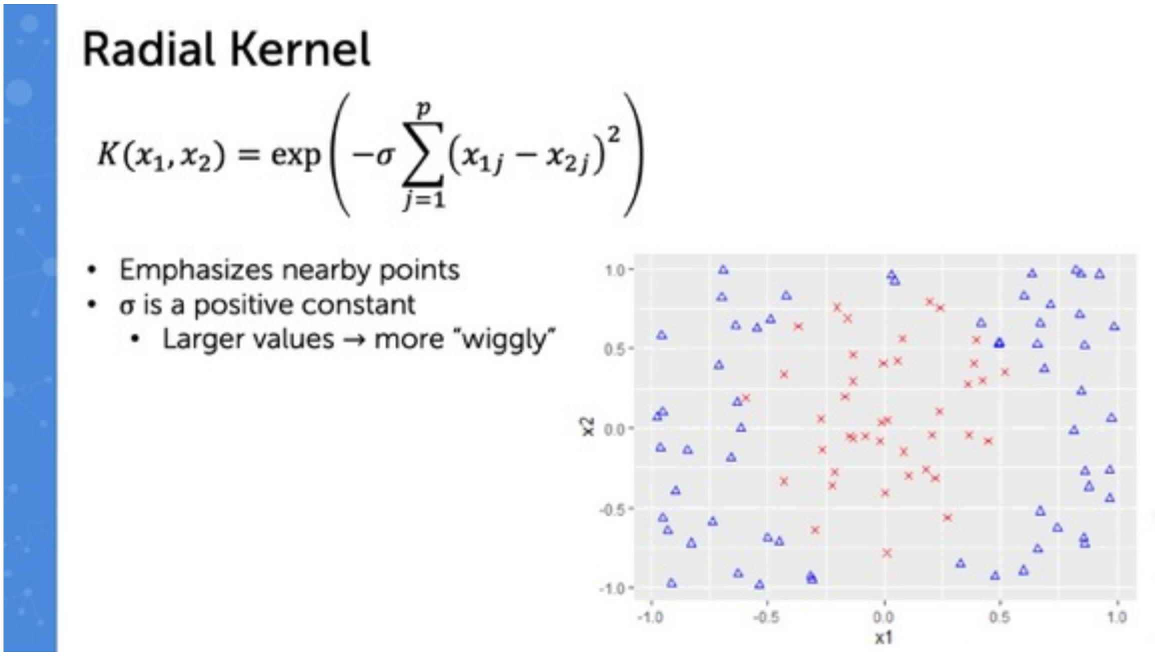
The linear kernel is just the inner product that we've already seen. And if you substitute this in for k of x, x sub i, you get back the same parametrization of the support vector classifier that we saw in the previous slide. The linear kernel measure similarity between two data points because it's basically an unscaled version of the Pearson correlation coefficient.



Polynomial Kernel

$$K(x_1, x_2) = \left(\underset{\uparrow}{c_0} + \underset{\uparrow}{\gamma} \sum_{j=1}^{p} x_{1j}x_{2j}\right)^{\overset{d}{\uparrow}}$$

Another option for k is the polynomial kernel, as shown here. This is useful when the boundary between the two classes is a parabola, a cubic function, a quartic function, or so on. In the polynomial kernel, we have three parameters to tune using cross validation; the cost, the scale parameter gamma, and d, the degree of the polynomial. So this would be 2 for a parabola, 3 for a cubic, and so on.



Radial Kernel

$$K(x_1, x_2) = \exp\left(-\sigma \sum_{j=1}^{p} (x_{1j} - x_{2j})^2\right)$$

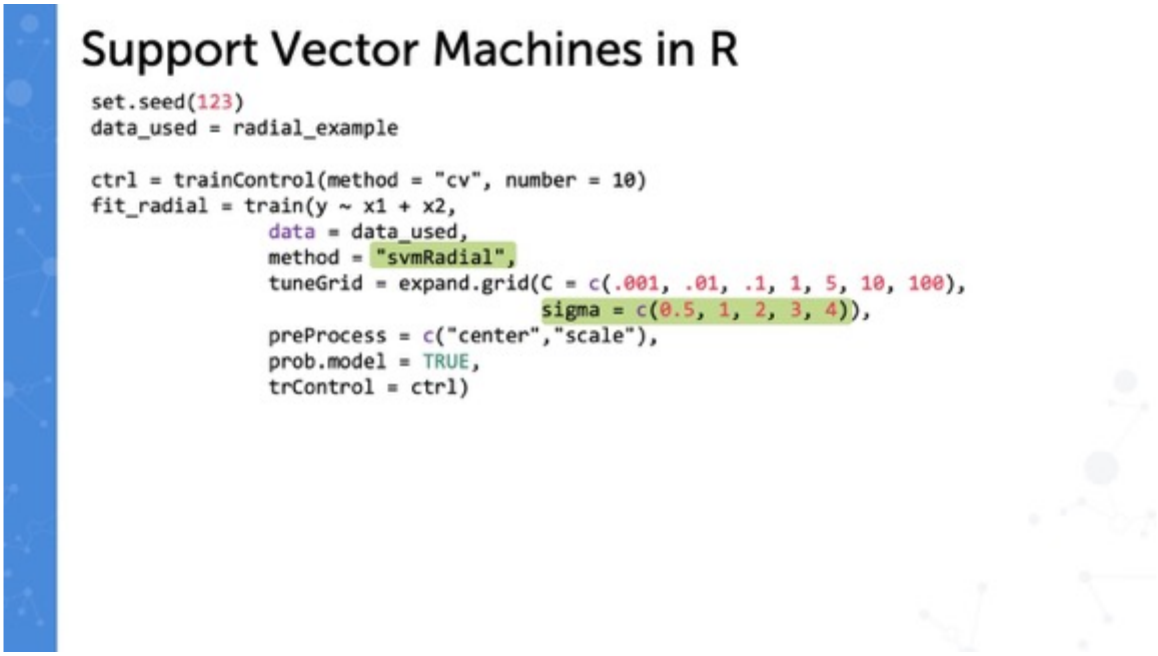
- Emphasizes nearby points
- σ is a positive constant
  - Larger values → more "wiggly"

Another popular choice for kernels is the radial kernel, also known as the radial basis function kernel, which has the format shown here. This kernel emphasizes nearby points as being most similar to each other. So it's especially good when you have clusters of points in a category with curved boundaries between clusters, like in the example shown here. In the radial kernel, sigma is a positive constant, which you can tune using cross-validation, along with the cost.

Larger values of sigma tend to make the boundary between the categories more wiggly, so they fit the training data better, but raise a risk of over-fitting.

Notes:

```
n = 100
set.seed(515)
x = matrix(runif(n*2, -1,1),nc=2)
y = rep("red", n)
y[which(-.75+2*(x[,1])^2+(x[,2])^2>0)] = "blue"

mypch = rep(4,n)
mypch[y=="blue"] = 2
plot(x, pch = mypch, col = y, las=1, cex.axis=1.2, xlab="x1", ylab =
"x2")
```



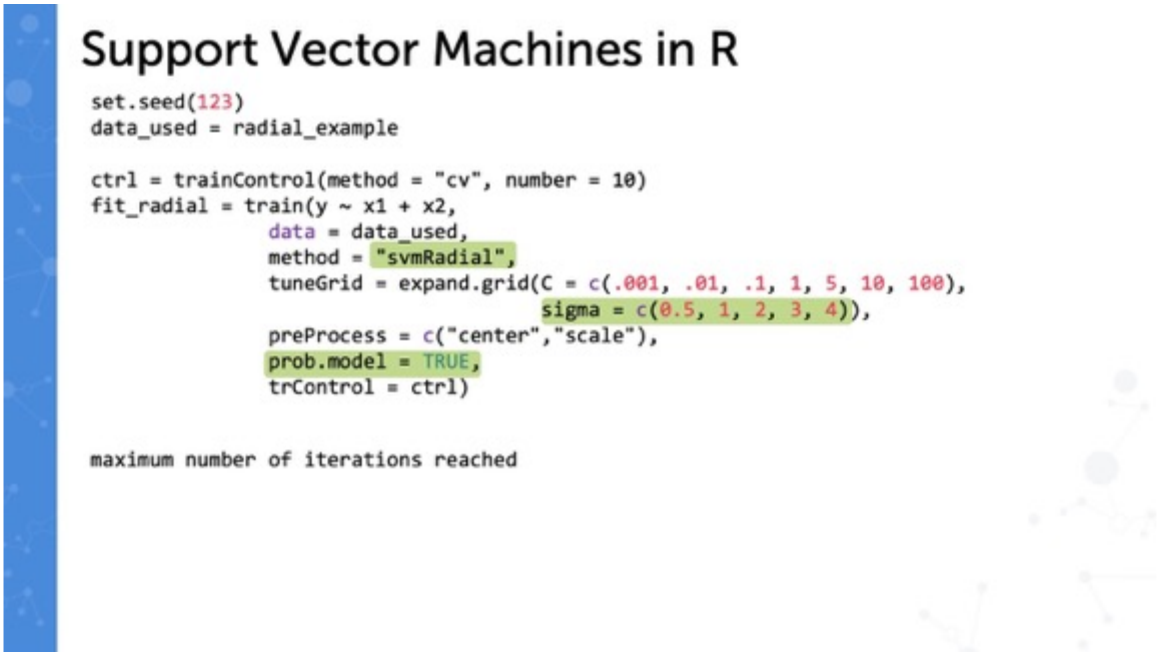
Support Vector Machines in R

```
set.seed(123)
data_used = radial_example

ctrl = trainControl(method = "cv", number = 10)
fit_radial = train(y ~ x1 + x2,
                   data = data_used,
                   method = "svmRadial",
                   tuneGrid = expand.grid(C = c(.001, .01, .1, 1, 5, 10, 100),
                                          sigma = c(0.5, 1, 2, 3, 4)),
                   preProcess = c("center","scale"),
                   prob.model = TRUE,
                   trControl = ctrl)
```

To fit a support vector machine in R, we can use caret, similar to how we fit our support vector classifier with a linear kernel. The main difference is that we use method equals SVM radial for a radial kernel, or SVM poly a polynomial kernel. We also have an additional tuning parameter of sigma for a radial kernel, or the degree and scale for a polynomial kernel.

Notes:

```
set.seed(123)
data_used = radial_example

ctrl = trainControl(method = "cv", number = 10)
fit_radial = train(y ~ x1 + x2,
                   data = data_used,
                   method = "svmRadial",
                   tuneGrid = expand.grid(C = c(.001, .01, .1, 1, 5, 10,
100),
                                          sigma = c(0.5, 1, 2, 3, 4)),
                   preProcess = c("center","scale"),
                   prob.model = TRUE,
                   trControl = ctrl)
```



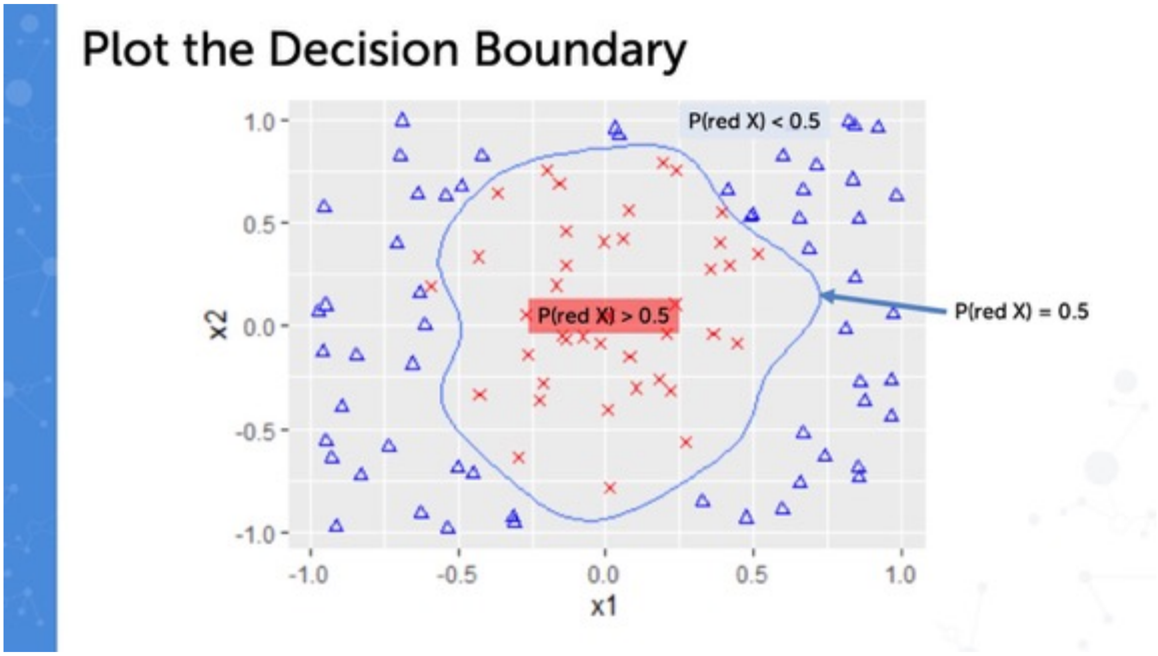
Support Vector Machines in R

```
set.seed(123)
data_used = radial_example

ctrl = trainControl(method = "cv", number = 10)
fit_radial = train(y ~ x1 + x2,
                   data = data_used,
                   method = "svmRadial",
                   tuneGrid = expand.grid(C = c(.001, .01, .1, 1, 5, 10, 100),
                                          sigma = c(0.5, 1, 2, 3, 4)),
                   preProcess = c("center","scale"),
                   prob.model = TRUE,
                   trControl = ctrl)


maximum number of iterations reached
```

In this case, I'm also using an optional argument, prob dot model equals true, which will allow us to compute probabilities from our model, which is useful for graphing the model. The disadvantage of using prob model equal to true is that it can cause the model to fail to converge, in which case, you'll get the warning message-- maximum number of iterations reached. If this happens, you may want to have prob model equal to true to create your graphs, and then fit the model a second time with prob model equal to false, or just omit the argument, in order to get the most accurate predictions of classifications.

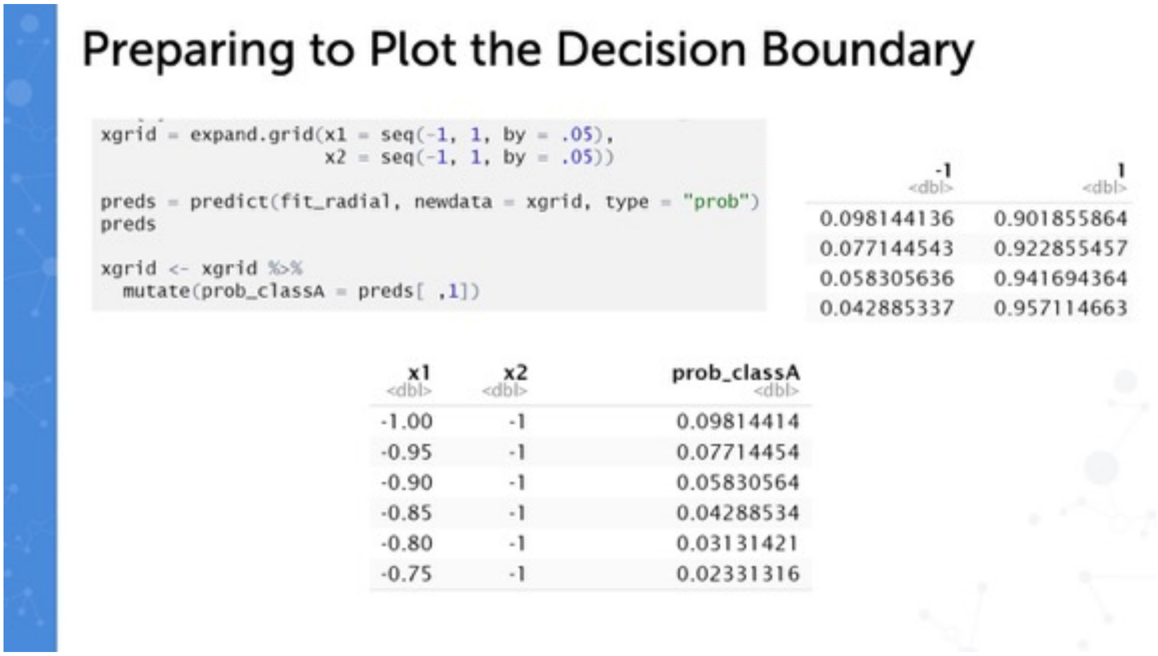
Notes:

```
set.seed(123)
data_used = radial_example

ctrl = trainControl(method = "cv", number = 10)
fit_radial = train(y ~ x1 + x2,
                   data = data_used,
                   method = "svmRadial",
                   tuneGrid = expand.grid(C = c(.001, .01, .1, 1, 5, 10,
100),
                                          sigma = c(0.5, 1, 2, 3, 4)),
                   preProcess = c("center","scale"),
                   prob.model = TRUE,
                   trControl = ctrl)
```



Plot the Decision Boundary

When you're using a non-linear kernel, the decision boundary between the classes ends up being curved. In order to plot this, we'll create a grid of example points and use the model to predict the probability that each of the points belongs to one of the classes, say, red x’s. Then, we'll graph the curve of all the points where the probability of being a red x is exactly 0.5. Then everything on either side of the curve will have a probability either greater than 0.5, meaning it's a red x, or a probability less than 0.5, meaning it's predicted to be a blue triangle.



Preparing to Plot the Decision Boundary

```
xgrid = expand.grid(x1 = seq(-1, 1, by = .05),
                    x2 = seq(-1, 1, by = .05))

preds = predict(fit_radial, newdata = xgrid, type = "prob")
preds

xgrid <- xgrid %>%
  mutate(prob_classA = preds[ ,1])
```

-1 <dbl>	1 <dbl>
0.098144136	0.901855864
0.077144543	0.922855457
0.058305636	0.941694364
0.042885337	0.957114663

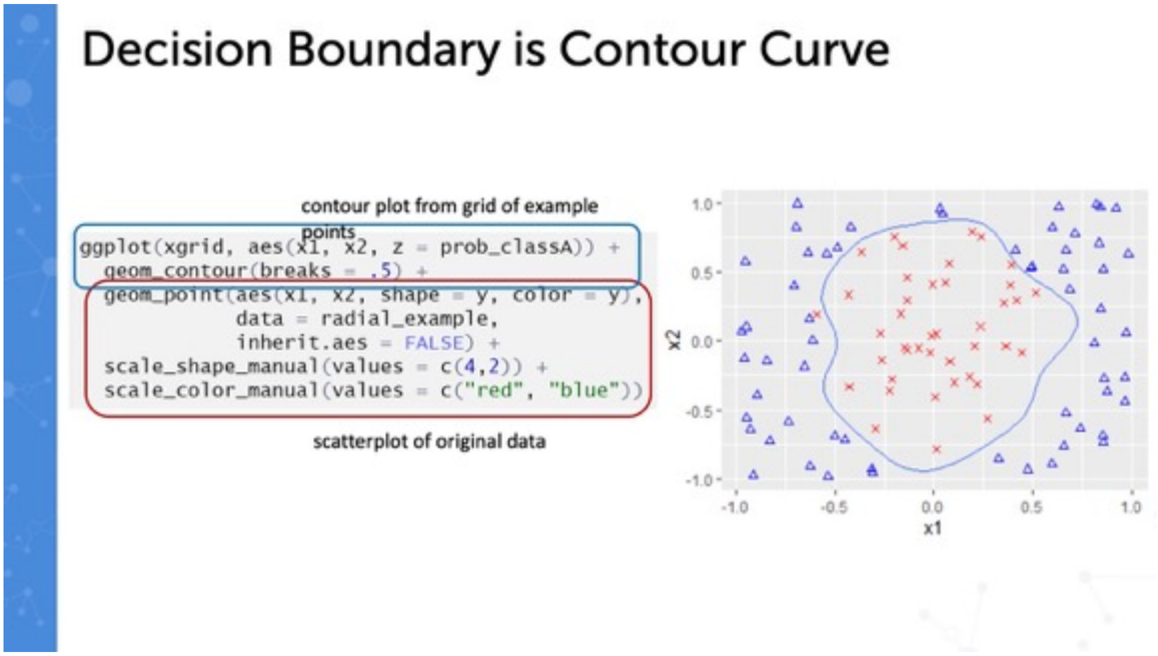
x1 <dbl>	x2 <dbl>	prob_classA <dbl>
-1.00	-1	0.09814414
-0.95	-1	0.07714454
-0.90	-1	0.05830564
-0.85	-1	0.04288534
-0.80	-1	0.03131421
-0.75	-1	0.02331316

Here, we're using the expand dot grid function to create a data frame of example points with values of x1 and x2, ranging between negative 1 and 1. So this looks like this, with all possible combinations of values from negative 1 to 1 in increments of 0.05. Then we'll use the predict function with the argument type equals prob to predict the probabilities that each data point in our example grid belongs to each of the categories.

This will give us a data frame of two columns containing the probability of belonging to class negative 1 and class 1. We only need one of those two columns, so we can add it to our x grid data frame by taking just the first column and calling that prob class A, the probability of belonging to class A.

Notes:

```
xgrid = expand.grid(x1 = seq(-1, 1, by = .05),
                    x2 = seq(-1, 1, by = .05))

preds = predict(fit_radial, newdata = xgrid, type = "prob")
preds

xgrid <- xgrid %>%
  mutate(prob_classA = preds[ ,1])
```



Decision Boundary is Contour Curve

contour plot from grid of example points

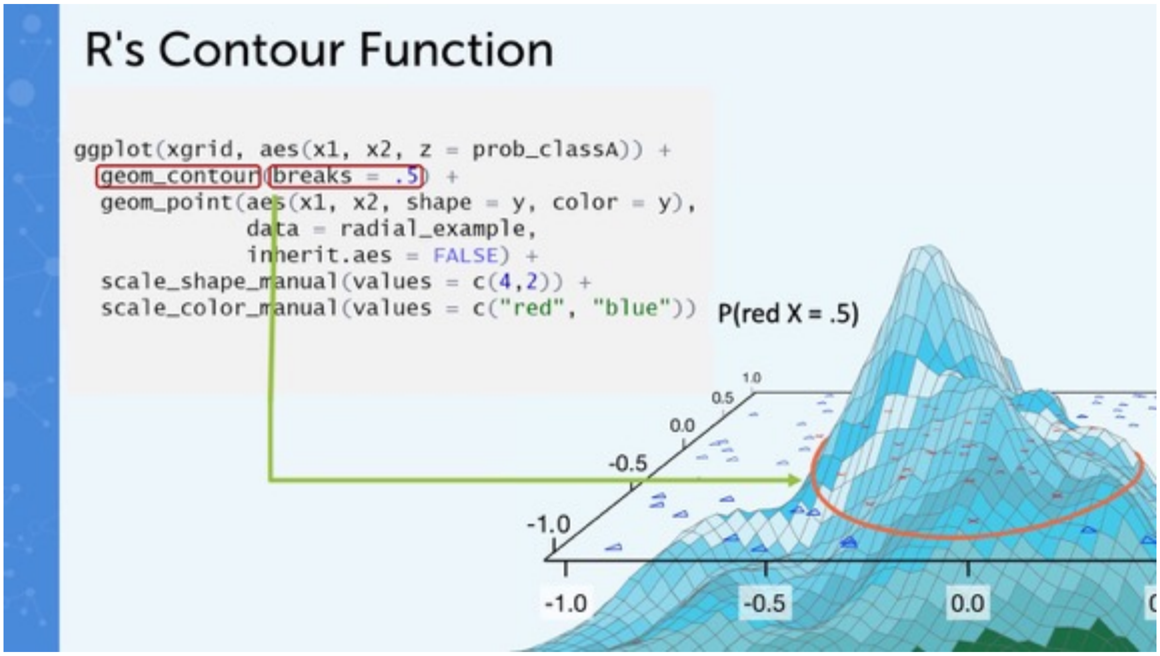
```
ggplot(xgrid, aes(x1, x2, z = prob_classA)) +
  geom_contour(breaks = .5) +
  geom_point(aes(x1, x2, shape = y, color = y),
             data = radial_example,
             inherit.aes = FALSE) +
  scale_shape_manual(values = c(4,2)) +
  scale_color_manual(values = c("red", "blue"))
```

scatterplot of original data

We can then use the geom point function from ggplot2 to make a scatterplot of our original data, and the geom contour function to make a contour curve from the grid of example points.

Notes:

```
ggplot(xgrid, aes(x1, x2, z = prob_classA)) +
  geom_contour(breaks = .5) +
  geom_point(aes(x1, x2, shape = y, color = y),
             data = radial_example,
             inherit.aes = FALSE) +
  scale_shape_manual(values = c(4,2)) +
  scale_color_manual(values = c("red", "blue"))
```



R's Contour Function

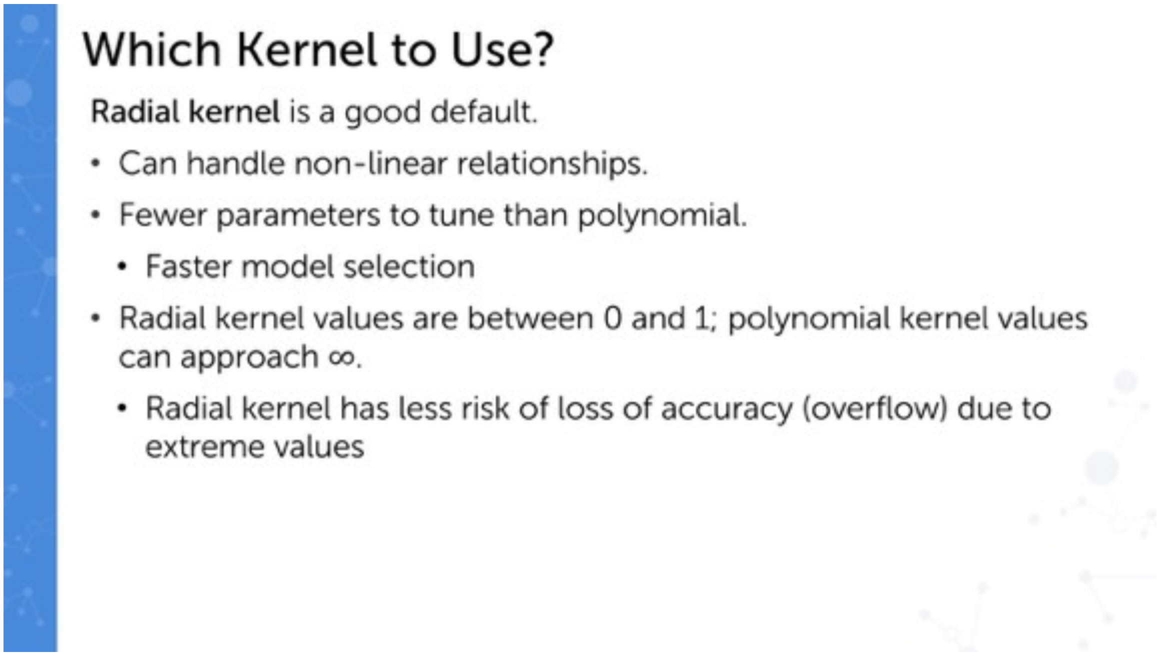
```
ggplot(xgrid, aes(x1, x2, z = prob_classA)) +
  geom_contour(breaks = .5) +
  geom_point(aes(x1, x2, shape = y, color = y),
             data = radial_example,
             inherit.aes = FALSE) +
  scale_shape_manual(values = c(4,2)) +
  scale_color_manual(values = c("red", "blue"))
```

We can think of the probabilities as being a third dimension lying above the plane of x1 and x2-- a set of hills and valleys. The geom contour function identifies all of the points that are at the same level, in this case, a height of 0.5.

So all of the points that are on one side of the curve will have probabilities greater than 0.5, meaning they belong to the class that we selected. All of the points on the other side of the curve will have probabilities less than 0.5, meaning they belong to the other class.

Notes:

```
ggplot(xgrid, aes(x1, x2, z = prob_classA)) +
  geom_contour(breaks = .5) +
  geom_point(aes(x1, x2, shape = y, color = y),
             data = radial_example,
             inherit.aes = FALSE) +
  scale_shape_manual(values = c(4,2)) +
  scale_color_manual(values = c("red", "blue"))
```

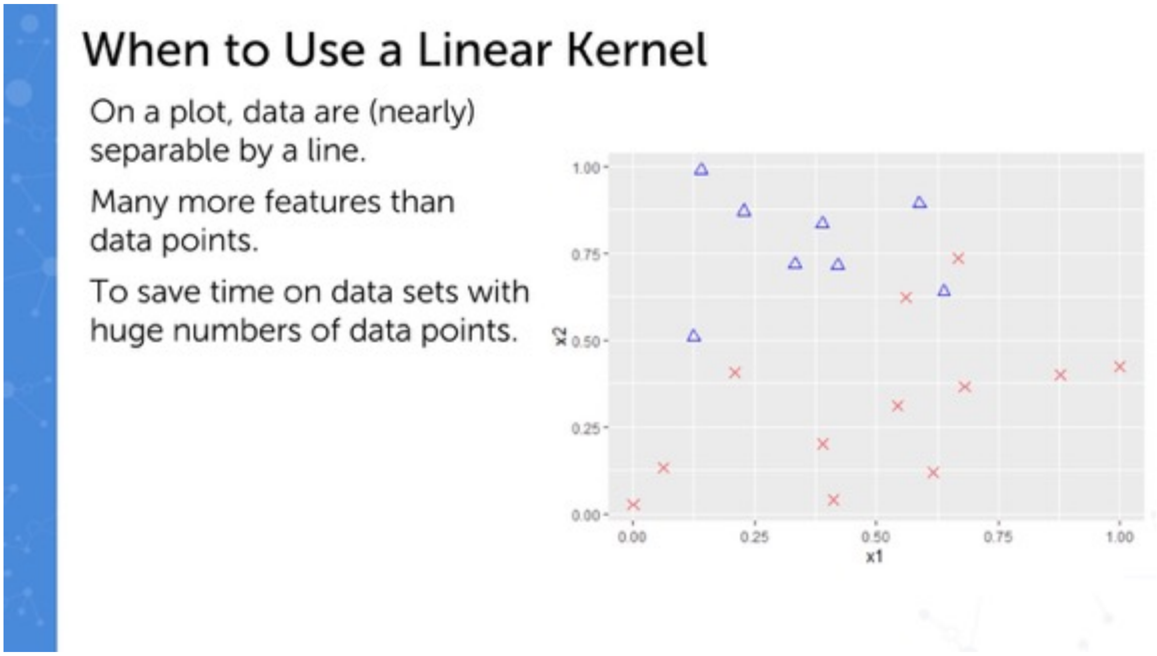


Which Kernel to Use?

Radial kernel is a good default.

- Can handle non-linear relationships.
- Fewer parameters to tune than polynomial.
  - Faster model selection
- Radial kernel values are between 0 and 1; polynomial kernel values can approach ∞.
  - Radial kernel has less risk of loss of accuracy (overflow) due to extreme values

Notes:
For a summary of the advantages of the radial kernel, see *A Practical Guide to Support Vector Classificationby Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin (PDF)*



When to Use a Linear Kernel

On a plot, data are (nearly) separable by a line.

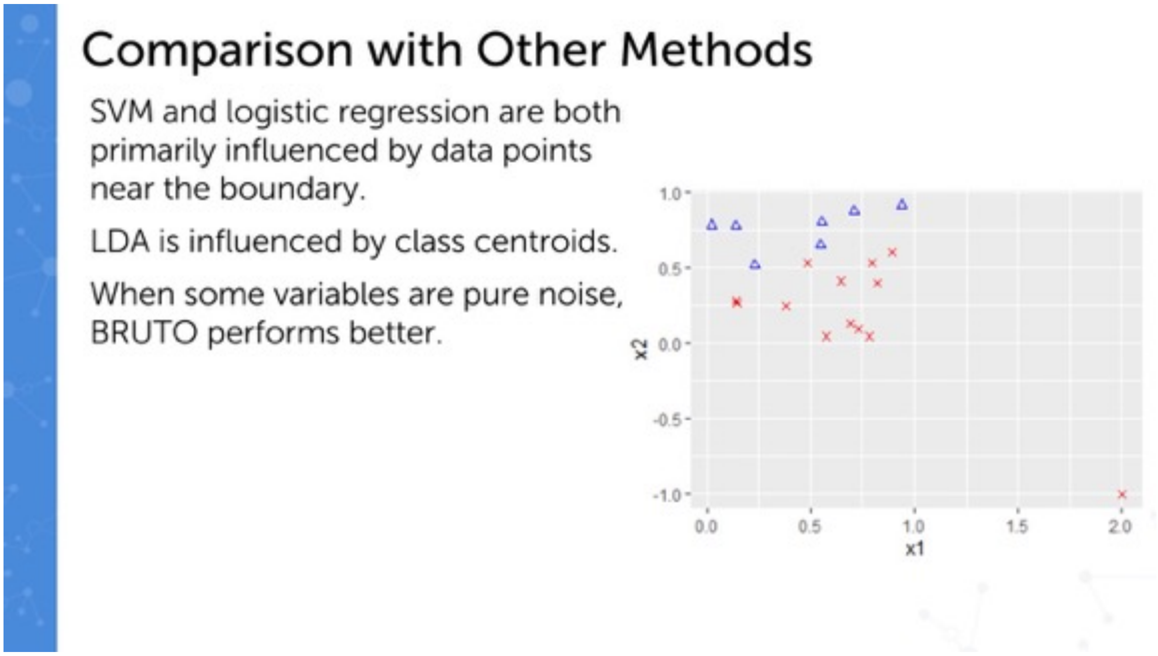
Many more features than data points.

To save time on data sets with huge numbers of data points.

While a radial kernel is a good general default, there are some situations in which a linear kernel is preferred. One of these is if you plot the data and can see that they're nearly separable by a line, as in the example shown here. This works best if you only have two predictor variables. But if you have a small number of predictor variables, it might be worth plotting each predictor variable versus each of the others in order to get an understanding of the shape of the separation between the categories.

Another situation in which a linear kernel is likely to be successful is if you have many more features or predictor variables than you have data points. In this case, the data points are likely to be spaced far apart in our multidimensional feature space. So it's probably going to be easy to find a way to separate them, possibly with a hyperplane. This is the same reason why we don't want to use k nearest neighbors when we have more features than data points.

The third situation in which we might want to use a linear kernel is when working with a data set with a huge number of data points. In this case, it may simply be computationally infeasible in terms of memory or time to use a radial kernel.



Comparison with Other Methods

SVM and logistic regression are both primarily influenced by data points near the boundary.

LDA is influenced by class centroids.

When some variables are pure noise, BRUTO performs better.

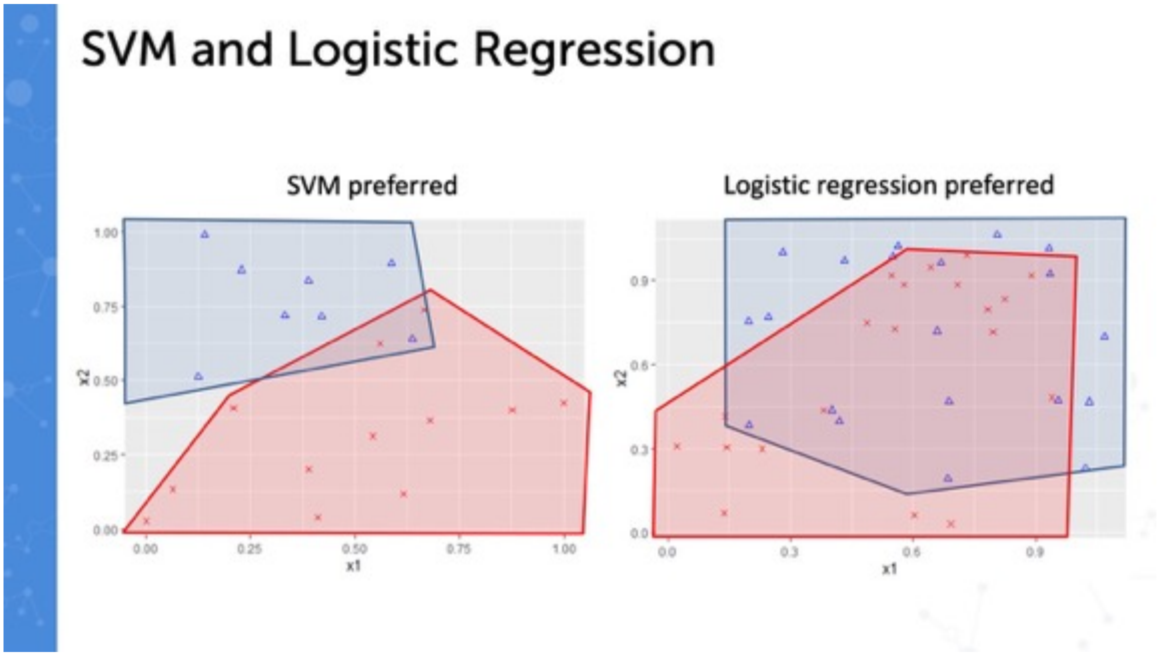
Support vector machines actually have a lot in common with logistic regression. And, in fact, you can use the idea of kernels in logistic regression to handle non-linear relationships between categories in a computationally efficient manner. Both of these methods are primarily influenced by data points near the boundary between the categories. Support vector machines are only influenced by the support vectors. And in a logistic regression, the influence of points on the boundary tapers off as you get farther away from the boundary between the categories.

In contrast to these, linear discriminant analysis is influenced by the class centroids, which can be quite far from the boundary between the categories. For this reason, you may prefer to use either support vector machines or logistic regression in a situation like the one pictured here, where we have outliers that are far away from the boundary between the classes.

One of the advantages of support vector machines is that they tend to perform well even when the number of features is large relative to the number of data points. However, they don't do so well when some of the variables are pure noise, don't contribute anything to the decision between which category a point belongs to. In this situation, it may be better to use a variable selection technique, such as BRUTO.

Notes:
For more about kernels in logistic regression, see *The Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman.

For a summary of BRUTO, see “Flexible Discriminant Analysis by Optimal Scoring” by Trevor Hastie, Robert Tibshirani, and Andreas Buja (PDF).



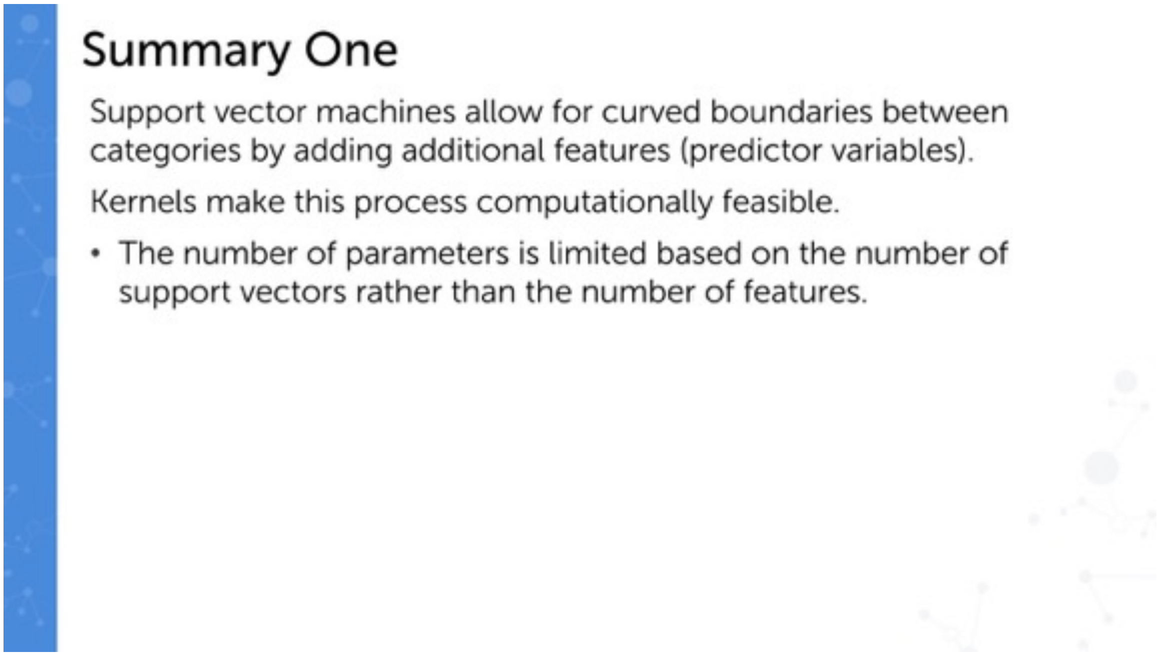
SVM and Logistic Regression

SVM preferred

Logistic regression preferred

Support vector machines and logistic regression often perform very similarly. However, support vector machines frequently prefer better when there is very little overlap between the categories. And logistic regression typically performs better when there's a lot of overlap between the categories.

Notes:
For more about kernels in logistic regression, see *The Elements of Statistical Learning by Hastie, Tibshirani, and Friedman (PDF)*

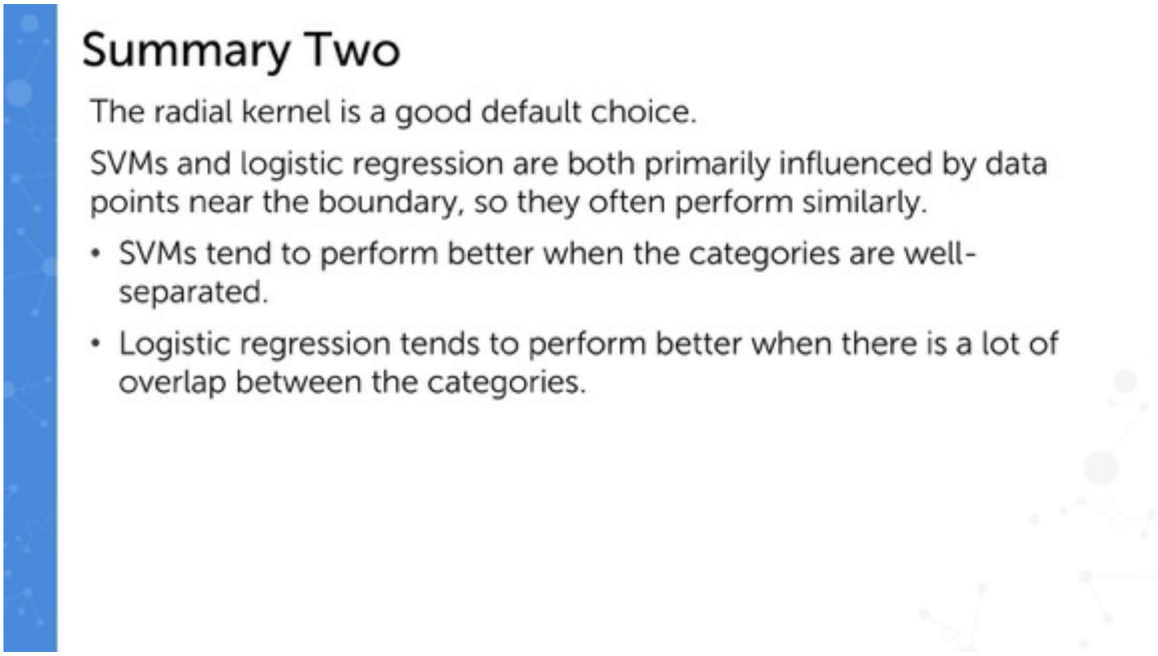


Summary One

Support vector machines allow for curved boundaries between categories by adding additional features (predictor variables).

Kernels make this process computationally feasible.

- The number of parameters is limited based on the number of support vectors rather than the number of features.



Summary Two

The radial kernel is a good default choice.

SVMs and logistic regression are both primarily influenced by data points near the boundary, so they often perform similarly.

- SVMs tend to perform better when the categories are well-separated.
- Logistic regression tends to perform better when there is a lot of overlap between the categories.