# Support Vector Machines with the mlr package

**hefinioanrhys** **10th Oct 2019** **Uncategorized**

## *What is the support vector machine (SVM) algorithm?*

Imagine you would like to predict whether your boss will be in a good mood or not (a very important machine learning application). Over a couple of weeks, you record the number of hours you spend playing games at your desk and how much money you make the company each day. You also record your boss' mood the next day as good or bad (they're very binary). You decide to use the SVM algorithm to build a classifier that will help you decide whether you need to avoid your boss on a particular day! The SVM algorithm will learn a linear hyperplane that separates the days your boss is in a good mood form the days they are in a bad mood. The SVM algorithm is also able add an extra dimension to the data to find the best hyperplane.

## *SVMs for linearly-separable data*

Take a look at the data shown in figure 1. The plots show the data you recorded on the mood of your boss, based on how hard you're working and how much money you're making the company.

The SVM algorithm finds an optimal linear hyperplane that separates the classes. A hyperplane is a surface that exists in as many dimensions as there are variables in a dataset. For a two-dimensional feature space, such as in the example in figure 1, a hyperplane is simply a straight line. For a three-dimensional feature space, a hyperplane is a surface. It's hard to picture hyperplanes with four or more dimensions, but the principle is the same: they are surfaces that cut through the feature space.

For problems where the classes are fully separable, there may be many different hyperplanes which do just as good a job at separating the classes in the training data. To find an optimal hyperplane (which will, hopefully, generalize better to

unseen data), the algorithm finds the hyperplane which maximizes the *margin* around itself. The margin is a distance around the hyperplane which touches the fewest training cases. The cases in the data that touch the margin are called *support vectors* because they support the position of the hyperplane (hence the name of the algorithm).

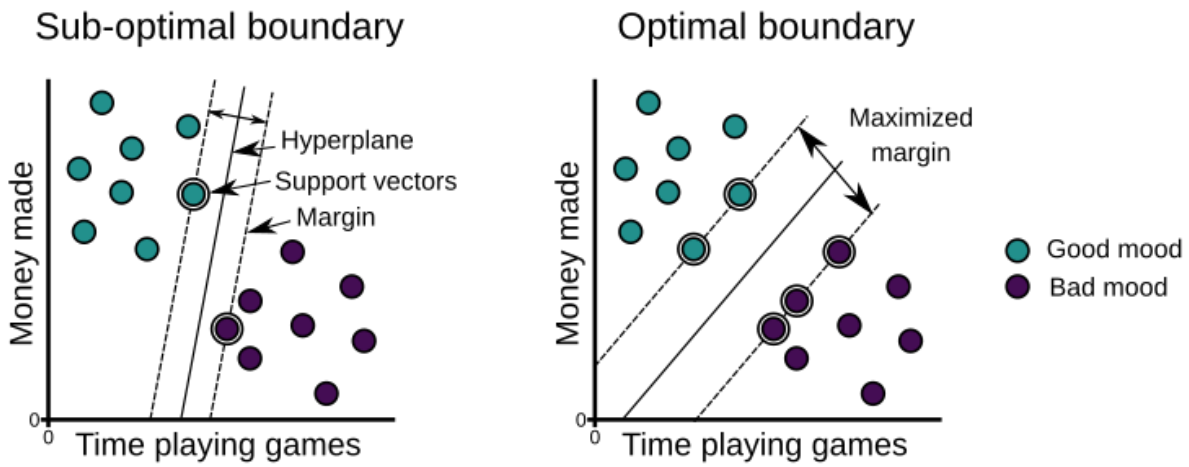


Figure 1. The SVM algorithm finds a hyperplane (solid line) in as many dimensions as there are predictor variables. An optimal hyperplane is one that maximizes the margin around itself (dotted lines). The margin is a region around the hyperplane that touches the fewest cases. Support vectors are shown with double circles.

The support vectors are the most important cases in the training set because they define the boundary between the classes. Not only this, the hyperplane that the algorithm learns is entirely dependent on the position of the support vectors, and none of the other cases in the training set. Take a look at figure 2. If we move the position of one of the support vectors, then we move the position of the hyperplane. If, however, we were to move a non-support vector case, there is no influence on the hyperplane at all!

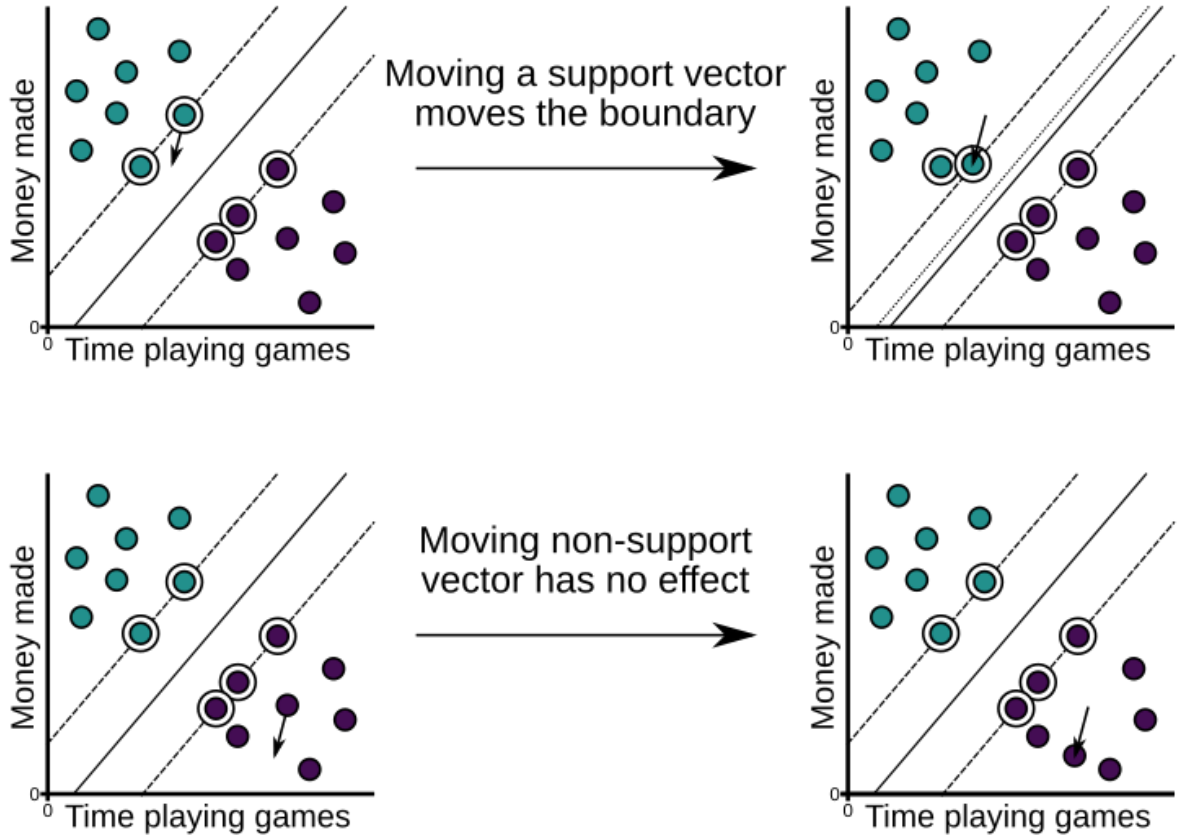


Figure 2. The position of the hyperplanes are entirely dependent on the position of support vectors. Moving a support vector moves the hyperplane from its original position (dotted line) to a new position (top two plots). Moving a non-support vector has no impact on the hyperplane (bottom two plots).

SVMs are extremely popular right now. That's mainly for three reasons:

They are good at finding ways of separating non-linearly separable classes

They tend to perform well for a wide variety of tasks

We now have the computational power to apply them to larger, more complex datasets

This last point is important because it highlights a potential downside of SVMs: they tend to be more computationally expensive to train than many other classification algorithms. For this reason, if you have a very large dataset and computational power is limited, it may be economical for you to try cheaper algorithms first and see how they perform.

TIP: Usually, we favor predictive performance over speed. But a computationally cheap algorithm that performs well enough for your problem may be preferable to you than one which is very expensive. Therefore, I usually try cheaper algorithms before trying the expensive ones.

## *SVMs for non-linearly separable data*

Great! So far the SVM algorithm seems quite simple, and for linearly separable classes like in our boss mood example, it is. But I mentioned that one of the strengths of the SVM algorithm is that it can learn decision boundaries between classes that are *not* linearly separable. As I've told you that the algorithm learns linear hyperplanes, this seems like a contradiction. Well here's what makes the SVM algorithm so powerful: it can add an extra dimension to your data, to find a linear way to separate non-linear data.

Take a look a look at the example in figure 3. The classes are not linearly separable using the two predictor variables. The SVM algorithm adds an extra dimension to the data, such that a linear hyperplane can separate the classes in this new, higher dimensional space. We can visualize this as a sort of deformation or stretching of the feature space. The extra dimension is called a *kernel*.

> ### Why is it called a kernel?
>
> The word kernel may confuse you (it certainly confuses me). It has nothing to do with the kernels in computing (the bit of your operating system that directly interfaces with the computer hardware), or kernels in corn or fruit.
>
> The truth is that reason they are called kernels is a little murky. In 1904, a German mathematician called David Hilbert published *Grundzüge einer allgemeinen theorie der linearen integralgleichungen* (Principles of a general theory of linear integral equations). In this book, Hilbert uses the word *kern* to mean the *core* of an integral equation. In 1909, an American mathematician called Maxime Bôcher published *An introduction to the study of integral equations* in which he translates Hilberts use of the word kern, to *kernel*.
>
> The mathematics of kernel functions evolved from the work in these publications, and took the name kernel with them. The extremely confusing thing is that multiple, seemingly unrelated concepts in mathematics have the word kernel in them!

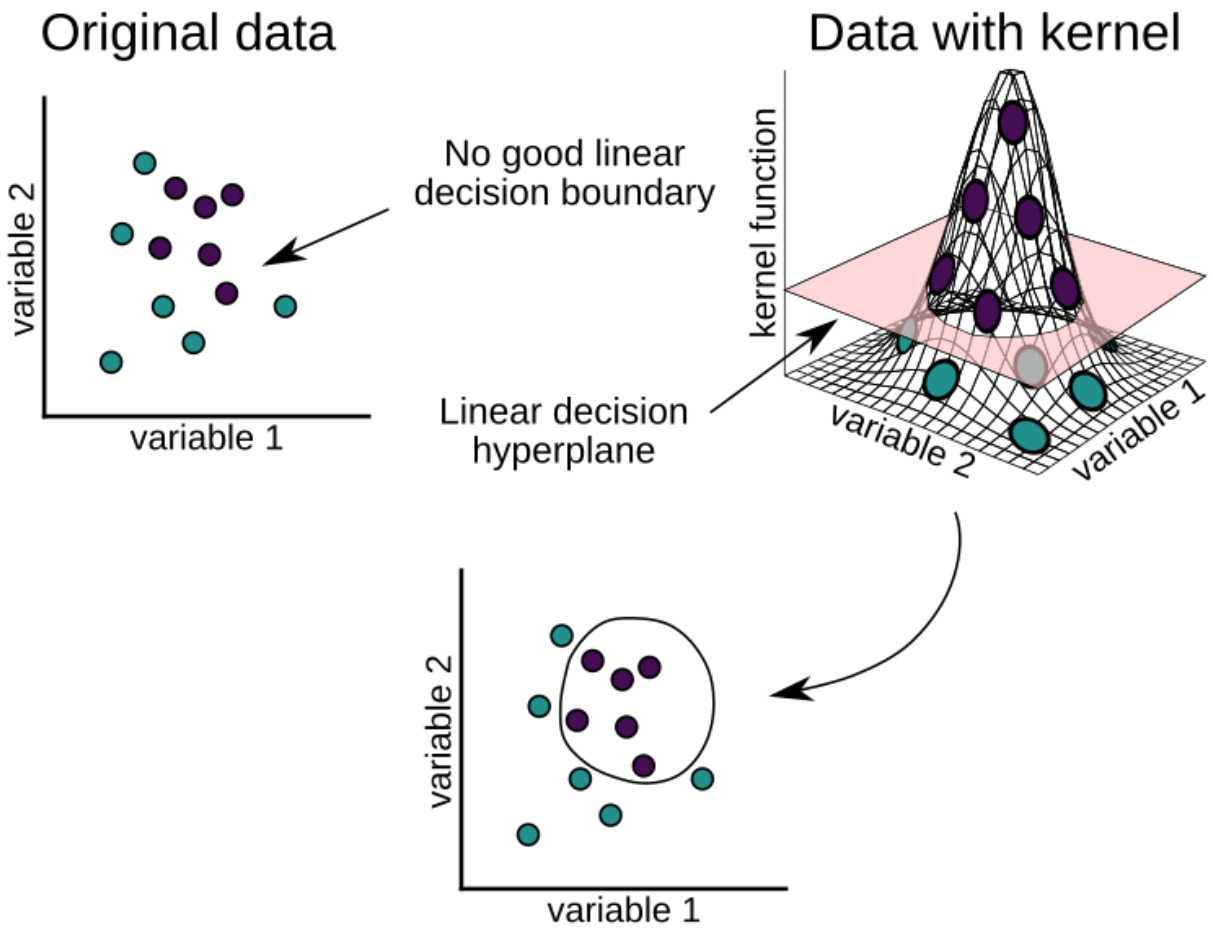


Figure 3. The SVM algorithm adds an extra dimension to linearly separate the data. The classes in the original data are not linearly separable. The SVM algorithm adds an extra dimension that, in a twodimensional feature space, can be illustrated as a "stretching" of the data into a third dimension. This additional dimension allows the data to be linearly separated. When this hyperplane is projected back onto the original two dimensions, it appears as a curved decision boundary.

So how does the algorithm find this new kernel? It uses a mathematical transformation of the data called a *kernel function*. There are many kernel functions to choose from, each of which applies a different transformation to the data and is suitable for finding linear decision boundaries for different situations. Figure 4 shows examples of situations where some common kernel functions can separate non-linearly separable data. These include:

- linear kernel (equivalent to no kernel)
- polynomial kernel
- Gaussian radial basis kernel
- sigmoid kernel

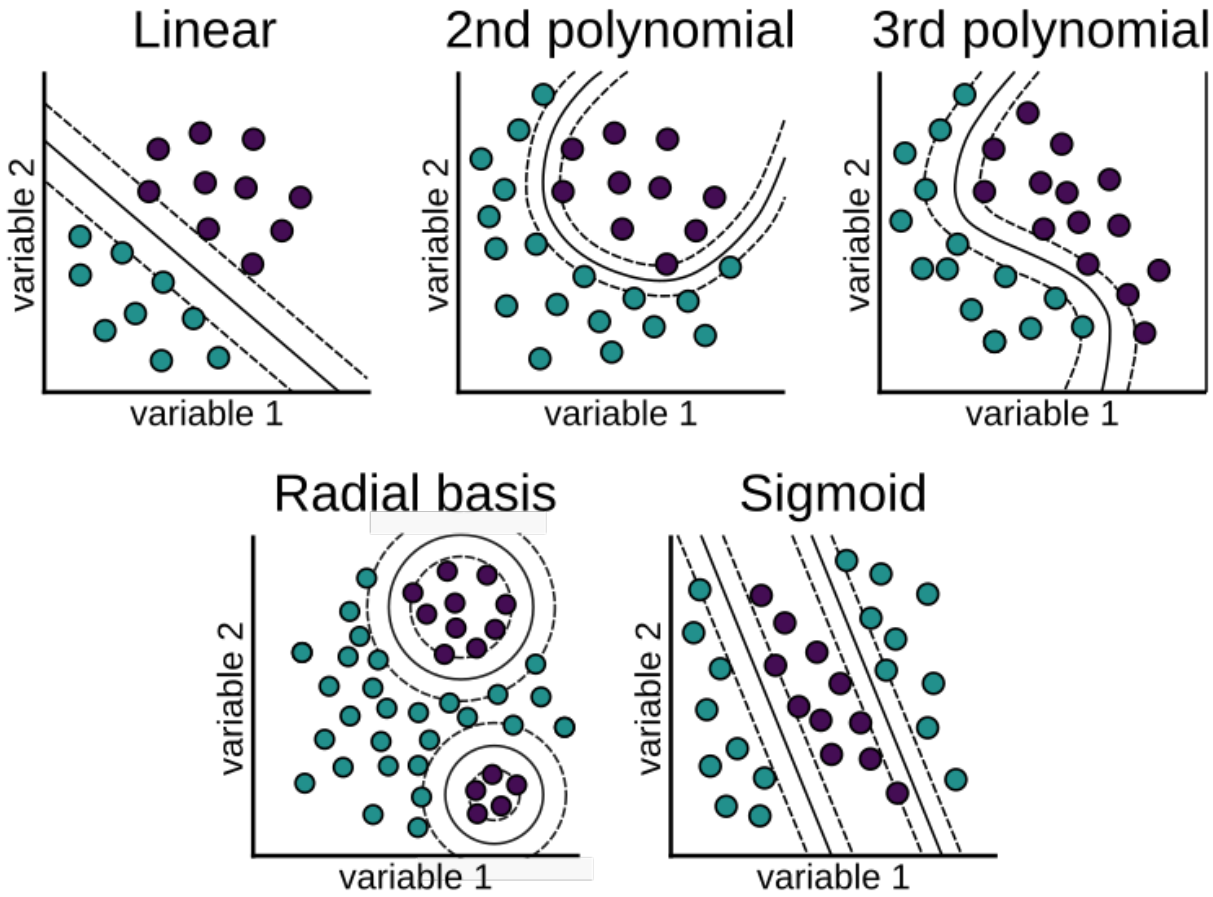


Figure 4. Examples of kernel functions. For each example, the solid line indicates the decision boundary (projected back onto the original feature space) and the dashed lines indicate the margin. With the exception of the linear kernel, imagine the cases of one of the groups is raised off the page in a third dimension.

Now the type of kernel function for a given problem isn't learned from the data, we have to specify it. Because of this, the choice of kernel function is a *categorical hyperparameter* (a hyperparameter which takes discrete, not continuous values). Therefore, the best approach for choosing the best performing kernel is with hyperparameter tuning.

## *Hyperparameters of the SVM algorithm*

This is where SVMs become fun/difficult/painful depending on your problem, computational budget and sense of humor. We need to tune quite a lot of hyperparameters when building an SVM. This, coupled with the fact that training a single model can be moderately expensive, can make training an optimally performing SVM take quite a long time. You'll see this in the worked example later in the article.

So the SVM algorithm has quite a few hyperparameters to tune, but the most important ones to consider are:

- the kernel (shown in figure 4)
- the *degree* hyperparameter, which controls how "bendy" the decision

boundary will be for the polynomial kernel (shown in figure 4)
- the *cost* or *C* hyperparameter, which controls how "hard" or "soft" the margin is (shown in figure 5)
- the *gamma* hyperparameter, which controls how much influence individual cases have on the position of the decision boundary (shown in figure 5)

The effect of the kernel function, and degree hyperparameters are shown in figure 4. Note the difference in the shape of the decision boundary between the 2nd and 3rd degree polynomials.

NOTE: The higher the degree of polynomial, the more bendy and complex a decision boundary can be learned, but this has the potential to over-fit the training set.

In the illustrations I've shown you so far, the classes have been fully separable.

This is so I could clearly show you how the positioning of the hyperplane is chosen to maximize the margin. But what about situations where the classes are not completely separable? How can the algorithm find a hyperplane when there is no margin that won't have cases *inside* it? This is where the cost (also called C) hyperparameter comes in.

The cost hyperparameter assigns a cost or penalty to having cases inside the margin or, put another way, tells the algorithm how bad it is to have cases inside the margin. A low cost tells the algorithm that it's acceptable to have more cases inside the margin and will result in wider margins less influenced by local differences near the class boundary. A high cost imposes a harsher penalty on having cases inside the boundary and will result in narrower margins more influenced by local differences near the class boundary. The effect of cost is illustrated for a linear kernel in the top part of figure 4.

IMPORTANT: Cases inside the margin are also support vectors, as moving them would change the position of the hyperplane.

The gamma hyperparameter controls the influence each case has on the position of the hyperplane, and is used by all the kernel functions except the linear kernel. Think of each case in the training set jumping up and down shouting "me! me! classify me correctly!" The larger gamma is, the more attention-seeking each case is, and the more granular the decision boundary will be (potentially leading to overfitting). The smaller gamma is, the less attention-seeking each case will be, and the less granular the decision boundary will be (potentially leading to underfitting). The effect of gamma is illustrated for a Gaussian radial basis kernel in the bottom part of figure 5.

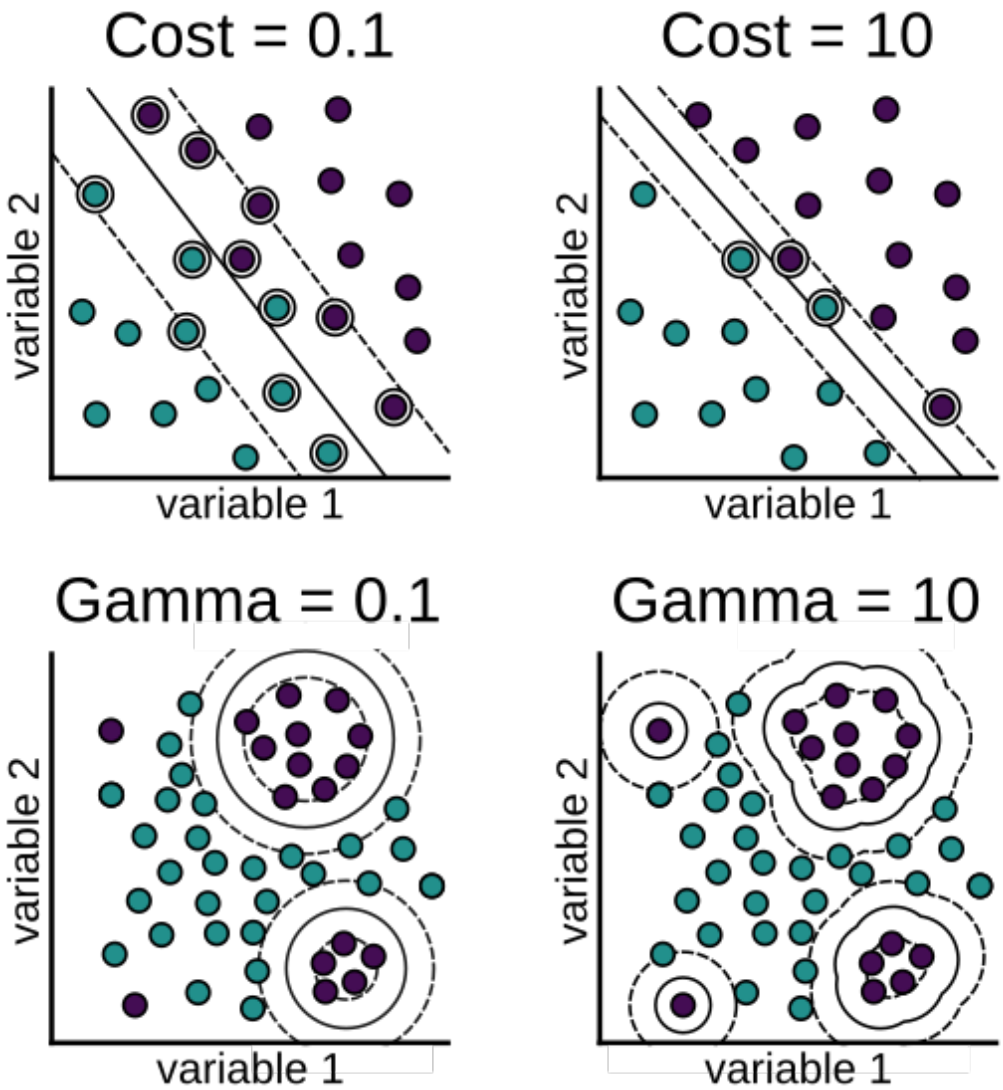


Figure 5. The impact of the cost and gamma hyperparameters. Larger values of the cost hyperparameter give greater penalization for having cases inside the margin. Larger values of the gamma hyperparameter mean individual cases have greater influence on the position of the decision boundary, leading to more complex decision boundaries.

So the SVM algorithm has multiple hyperparameters to tune! I'll show you how we can tune these simultaneously using mlr in a little bit.

## *What if I have more than two classes?*

So far, I've only shown you examples of two-class classification problems. This is because the SVM algorithm is inherently geared towards separating two classes. But can we use it for multiclass problems (where we're trying to predict more than two classes)? Absolutely! When there are more than two classes, instead of creating a single SVM, we make multiple models, and let them fight it out to predict the most likely class for new data. There are two ways of doing this:

- *one versus all*
- *one versus one*

In the one versus all (also called one versus rest) approach, we create as many SVM models as there are classes. Each SVM model describes a hyperplane that best separates one class from *all the other classes*. Hence the name, one versus all. When we classify new, unseen cases, the models play a game of *winner takes all*. Put simply, the model that puts the new case on the "correct" side of its hyperplane (the side with the class it separates from all the others) wins. The case is then assigned to the class that model was trying to separate from the others. This is illustrated in the left-side plot of figure 6.

In the one versus one approach, we create an SVM model for every pair of classes. Each SVM model describes a hyperplane that best separates one class from *one other class*, ignoring data from the other classes. Hence the name, one versus one. When we classify new, unseen cases, the models each cast a vote. For example, if one model separates classes A and B, and the new data falls on the B side of the decision boundary, that model will vote for B. This continues for all the models, and the majority class vote wins. This is illustrated in the right-side plot of figure 6.

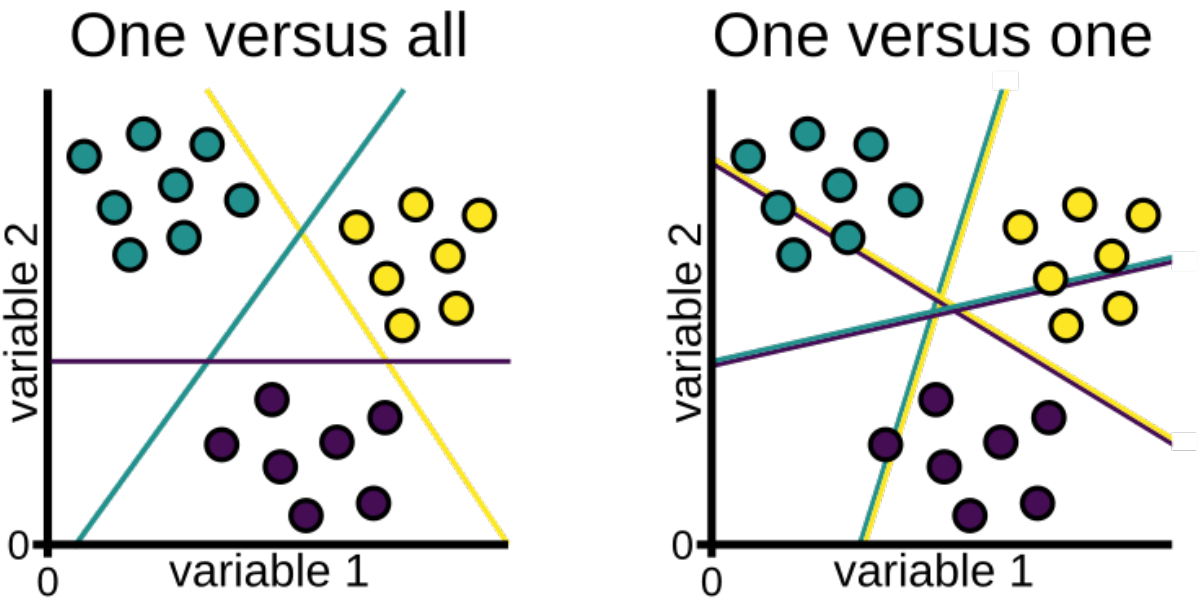


Figure 6. One versus all and one versus one approaches to multi-class SVMs. In the one versus all approach, a hyperplane is learned per class, separating it from all the other cases. In the one versus one approach, a hyperplane is learned for every pair of classes, separating them while ignoring the data from the other classes.

Which do we choose? Well in practice, there is usually little difference in the performance of the two methods. Despite training more models (for more than three classes), the one versus one is sometimes less computationally expensive that one versus all. This is because, although we're training more models, the training sets are smaller (because of the ignored cases). The implementation of the SVM algorithm called by mlr uses the one versus one approach.

There is, however, a problem with these approaches. There will often be regions of the feature space in which none of the models give a clear winning class. Can you see the triangular space between the hyperplanes in the left-side plot of 6? If a new

case appeared inside this triangle, none of the three models would clearly win outright. This is a sort of classification no-man's land. Though not as obvious in figure 6, this occurs with the one versus one approach also.

If there is no outright winner when predicting a new case, a technique called *Platt Scaling* is used (named after computer scientist John Platt). Platt scaling takes the distances of the cases from each hyperplane, and converts them into probabilities using the logistic function. The logistic function maps a continuous variable to probabilities that range between 0 and 1. Using Platt scaling to make predictions proceeds like this:

1. for every hyperplane (whether we use one versus all or one versus one):
    1. measure the distance of each case from the hyperplane
    2. use the logistic function to convert these distances into probabilities
2. classify new data as belonging to the class of the hyperplane which has the highest probability

If this seems confusing, take a look at figure 7. We're using the one versus all approach in the figure, and have generated three separate hyperplanes (one to separate each class from the rest). The dashed arrows in the figure indicate distance in either direction, away from the hyperplanes. Platt scaling takes these distances, and converts them into probabilities using the logistic function (the class each hyperplane separates from the rest has positive distance).

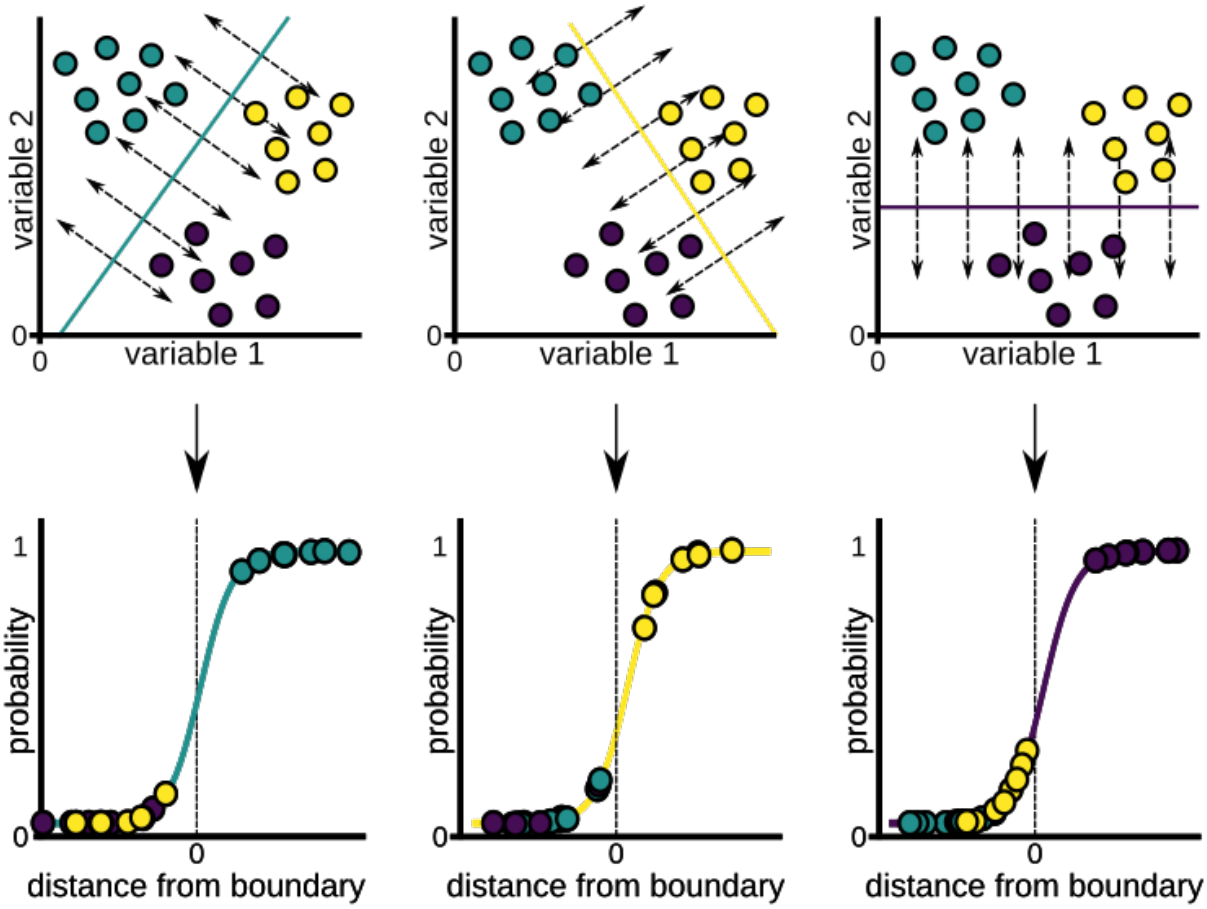


Figure 7. How Platt scaling is used to get probabilities for each hyperplane. This example shows a one versus all approach (also applies to one versus one). For each hyperplane, the distance of each case from the hyperplane is recorded (indicated by dashed arrows). These distances are converted into probabilities using the logistic function.

When we classify new, unseen data, the distance of the new data is converted into a probability using each of the three "s"-shaped curves, and the one that gives the highest probability is what the case is classified as. Handily, all of this is taken care of for us in the implementation of SVM called by mlr. If we supply a three-class classification task, we will get a one versus one SVM model with Platt scaling, without having to change our code.

## *Building our first SVM model*

In this section I'll teach you how to build an SVM model and tune multiple hyperparameters simultaneously. Imagine you're sick and tired of receiving so many spam emails (maybe you don't need to imagine!). It's difficult for you to be productive because you get so many emails requesting your bank details for a mysterious Ugandan inheritance, and trying to sell you Viagra.

You decide to perform a feature extraction on the emails you receive over a few months, which you manually class as spam or not spam. These features include things like the number of exclamation marks and the frequency of certain words. With this data you decide to make an SVM that will classify new emails as spam or

not spam, that you can use as a spam filter.

Let's learn how to train an SVM model and tune multiple hyperparameters simultaneously. We'll start by loading the mlr and tidyverse packages.

```
install.packages("mlr", dependencies = TRUE)

install.packages("tidyverse")

library(mlr)

library(tidyverse)
```

## *Loading and exploring the spam dataset*

Let's define our task and learner. In the mlr package the task consists of the data, and what we want do to with it. In this case, the data is `spamTib` and we want to classify the data with the `type` variable as the target variable. The learner is simply the name of the algorithm we plan to use, along with any additional arguments the algorithm accepts. In this example, we're using the SVM algorithm for classification, so our learner is `"classif.svm"`. Now, let's load the data, which is built into the kernlab package, convert it into a tibble (with as_tibble()) and explore it a little.

NOTE: The kernlab package should have been installed along with mlr as a suggested package. If you get an error when trying to load the data, you may need to install it with install.packages("kernlab").

We have a tibble containing 4601 emails and 58 variables extracted from emails. Our goal is to train a model which can use the information in these variables to predict whether a new email is spam or not.

NOTE: Except the factor type, which denotes whether an email is spam or not, all of the variables are continuous as the SVM algorithm cannot handle categorical predictors.

Listing 1. Loading and exploring the spam dataset

```
data(spam, package = "kernlab")

spamTib <- as_tibble(spam)

spamTib

# A tibble: 4,601 x 58
    make address   all num3d   our  over remove interne
   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>    <dbl
 1  0       0.64  0.64     0  0.32  0      0         0
 2  0.21    0.28  0.5      0  0.14  0.28   0.21      0.0
 3  0.06    0     0.71     0  1.23  0.19   0.19      0.1
 4  0       0     0        0  0.63  0      0.31      0.6
 5  0       0     0        0  0.63  0      0.31      0.6
 6  0       0     0        0  1.85  0      0         1.8
 7  0       0     0        0  1.92  0      0         0
 8  0       0     0        0  1.88  0      0         1.8
 9  0.15    0     0.46     0  0.61  0      0.3       0
10  0.06    0.12  0.77     0  0.19  0.32   0.38      0
# … with 4,591 more rows, and 48 more variables...
```

TIP: We have a lot of features in this dataset! I'm not going to discuss the meaning of each one, but you'll find a description of what they mean by running ?kernlab::spam.

## *Tuning our hyperparameters*

Let's define our task and learner. This time, we supply "classif.svm" as the argument to makeLearner() to specify we're going to use SVM.

Listing 2 Creating the task and learner

```
spamTask <- makeClassifTask(data = spamTib, target = "ty

svm <- makeLearner("classif.svm")
```

Now before we train our model, we need to tune our hyperparameters. To find out which hyperparameters are available for tuning for an algorithm, we simply pass the name of the algorithm in quotes to getParamSet(). For example, code listing 3 shows how to print the hyperparameters for the SVM algorithm. I've removed some rows and columns of the output to make it fit, but the most important columns are there:

- the row name is the name of the hyperparameter
- Type is whether the hyperparameter takes numeric, integer, discrete or logical values
- Def is the default value (the value that will be used if you don't tune it)
- Constr defines the constraints for the hyperparameter, either a set of specific

values or a range of acceptable values
- Req defines whether the hyperparameter is required by the learner
- Tunable is logical and defines whether that hyperparameter can be tuned (some algorithms have options that cannot be tuned but can be set by the user)

Listing 3 Printing available SVM hyperparameters

```
getParamSet("classif.svm")

                  Type     Def              Constr  Req
cost           numeric       1            0 to Inf    Y
kernel        discrete  radial  [lin,poly,rad,sig]    -
degree         integer       3            1 to Inf    Y
gamma          numeric       -            0 to Inf    Y
scale    logicalvector    TRUE                   -    -
```

TIP: The SVM algorithm is sensitive to variables being on different scales, and so it's usually a good idea to scale the predictors first. Notice the scale hyperparameter in code listing 3? This tells us the algorithm will scale the data for us by default.

Extracting the possible values for a hyperparameter

While the getParamSet() function is useful, I don't find it particularly simple to extract information from. If you call str(getParamSet("classif.svm")), you'll see that it has a reasonably complex structure. To extract information about a particular hyperparameter, you need to call getParamSet("classif.svm")$pars$[HYPERPAR] (where [HYPERPAR] is replaced by the hyperparameter you're interested in). To extract the possible values for that hyperparameter, you append $values to the call. For example, to extract the possible kernel functions:

```
getParamSet("classif.svm")$pars$kernel$values

$linear
[1] "linear"

$polynomial
[1] "polynomial"

$radial
[1] "radial"

$sigmoid
[1] "sigmoid"
```

The most important hyperparameters for us to tune are:

- the kernel

- the cost
- the degree
- gamma

Let's define the hyperparameters we want to tune in code listing 4. We're going to start by defining a vector of kernel functions we wish to tune.

TIP: Notice I omit the linear kernel? This is because the linear kernel is the same as the polynomial kernel with degree = 1, so we'll just make sure we include 1 as a possible value for the degree hyperparameter. Including the linear kernel *and* the 1st degree polynomial kernel is simply a waste of computing time.

Next, we use the makeParamSet() function to define the hyperparameter space we wish to tune over. To the makeParamSet() function, we supply the information needed to define each hyperparameter we wish to tune, separated by columns. Let's break this down line by line:

- the kernel hyperparameter takes discrete values (the name of the kernel function), so we use the makeDiscreteParam() function to define its values as the vector of kernels we created
- the degree hyperparameter takes integer values (whole numbers), so we use the makeIntegerParam() function and define its lower and upper values we wish to tune over
- the cost and gamma hyperparameters take numeric values (any number between zero and infinity), so we use the makeNumericParam() function to define their lower and upper values we wish to tune over

For each of these functions, the first argument is the name of the hyperparameter given by getParamSet("classif.svm"), in quotes.

Listing 4 Defining the hyperparameter space for tuning

```
kernels <- c("polynomial", "radial", "sigmoid")

svmParamSpace <- makeParamSet(
  makeDiscreteParam("kernel", values = kernels),
  makeIntegerParam("degree", lower = 1, upper = 3),
  makeNumericParam("cost", lower = 0.1, upper = 10),
  makeNumericParam("gamma", lower = 0.1, 10))
```

When searching for the best-performing combination of hyperparameters, one approach is to train models using every combination of hyperparameters, exhaustively. We then evaluate the performance of each of these models and choose the best-performing one. This is called a _grid search_. We used the grid search procedure to try every value of *k* that we defined, during tuning. This is what the grid search method does: tries every combination of the hyperparameter space

you define, and finds the best performing combination.

Grid search is great because, provided you specify a sensible hyperparameter space to search over, it will always find the best performing hyperparameters. But look at the hyperparameter space we defined for our SVM. Let's say we wanted to try the values for the cost and gamma hyperparameters in steps of 0.1, that's 100 values of each. We have three kernel functions we're trying, and four values of the degree hyperparameter. To perform a grid search over this parameter space would require training a model 120,000 times! If, in such a situation, you have the time, patience, and computational budget for such a grid search, then good for you. I, for one, have better things I could be doing with my computer!

Instead, we can employ a technique called *random search*. Rather than trying every possible combination of parameters, random search proceeds as follows:

1. Randomly select a combination of hyperparameter values
2. Use cross-validation to train and evaluate a model using those hyperparameter values
3. Record the performance metric of the model (usually mean misclassification error for classification tasks)
4. Repeat (iterate) steps 1 to 3 as many times as your computational budget allows
5. Select the combination of hyperparameter values that gave you the best performing model

Unlike grid search, random search isn't guaranteed to find the best set of hyperparameter values. However, with enough iterations, it can usually find a good combination that performs well. By using random search, we can run 500 combinations of hyperparameter values, instead of all 120,000 combinations.

Let's define our random search using the makeTuneControlRandom() function. We tell the function how many iterations of the random search procedure we want to use, with the maxit argument. You should try to set this as high as your computational budget allows, but in this example we'll stick to 20 to prevent the example from taking too long. Next, I describe our cross-validation procedure. I usually prefer k-fold cross-validation to hold-out cross-validation (as the former gives more stable estimates of model performance), unless the training procedure is computationally expensive. Well, this is computationally expensive, so I'm compromising by using hold-out cross-validation instead.

Listing 5. Defining the random search

```
randSearch <- makeTuneControlRandom(maxit = 20)

cvForTuning <- makeResampleDesc("Holdout", split = 2/3)
```

TIP: There's something else we can do to speed up this process. R, as a language, doesn't make that much use of multi-threading (using multiple CPUs simultaneously to accomplish a task). However, one of the benefits of the mlr package is that it allows multi-threading to be used with its functions. This helps you use multiple cores/CPUs on your computer to accomplish tasks such as hyperparameter tuning and cross-validation, much more quickly. If you don't know how many cores your computer has, you can find out in R by running parallel::detectCores(). If your computer only has one core, the 90s called and they want their computer back.

To run an mlr process in parallel, we place its code between the parallelStartSocket() and parallelStop() functions from the parallelMap package. To start our hyperparameter tuning process, we call the tuneParams() function and supply as arguments:

- the first argument is the name of the learner
- task = the name of our task
- resampling = our cross validation procedure (defined in code listing 5)
- par.set = our hyperparameter space (defined in code listing 4)
- control = the search procedure (random search, defined in code listing 5)

This code, between the parallelStartSocket() and parallelStop() functions is shown in code listing 6.

WARNING: The computer I'm writing this on has four cores and the code below takes nearly a minute to run on it. It is of the utmost importance that you go make a cup of tea while it runs. Milk and no sugar, please.

Listing 6. Performing hyperparameter tuning

```
library(parallelMap)
library(paralell)

parallelStartSocket(cpus = detectCores())

tunedSvmPars <- tuneParams("classif.svm", task = spamTa
                      resampling = cvForTuning,
                      par.set = svmParamSpace,
                      control = randSearch)

parallelStop()
```

TIP: The degree hyperparameter only applies to the polynomial kernel function, and the gamma hyperparameter doesn't apply to the linear kernel. Does this create errors when the random search selects combinations that don't make sense? Nope. If the random search selects the sigmoid kernel, for example, it simply ignores the

value of the degree hyperparameter.

Welcome back after our interlude! You can print the best-performing hyperparameter values and the performance of the model built with them by calling tunedSvm, or extract just the named values themselves (so you can train a new model using them) by calling tunedSvm$x. Looking at code listing 7, we can see that the 1st degree polynomial kernel function (equivalent to the linear kernel function), with a cost of 5.8 and gamma of 1.56, gave the best performing model.

Listing 7 Extracting the winning hyperparameter values from tuning

```
tunedSvmPars

Tune result:
Op. pars: kernel=polynomial; degree=1; cost=5.82; gamma
mmce.test.mean=0.0645372

tunedSvmPars$x
$kernel
[1] "polynomial"

$degree
[1] 1

$cost
[1] 5.816232

$gamma
[1] 1.561584
```

IMPORTANT: Are your values different to mine? They probably are! This is the nature of the random search. It may find different winning combinations of hyperparameter values each time it's run. To reduce this variance, we should commit to increasing the number of iterations it makes.

## *Training the model with the tuned hyperparameters*

Now we've tuned our hyperparameters, let's build our model using the best performing combination. We use the setHyperPars() function to combine a learner with a set of pre-defined hyperparameter values. The first argument is the learner we want to use, and the par.vals argument is the object containing our tuned hyperparameter values. We then train a model using our tunedSvm learner with the train() function.

Listing 8 Training the model with tuned hyperparameters

```
tunedSvm <- setHyperPars(makeLearner("classif.svm"),
                         par.vals = tunedSvmPars$x)

tunedSvmModel <- train(tunedSvm, spamTask)
```

TIP: As we already defined our learner in code listing 2, we could simply have run setHyperPars(svm, par.vals = tunedSvmPars$x) to achieve the same result.

That's all for this article. If you want to learn more about the book, check it out on liveBook **here** and see this **slide deck**.

Get more blogs at **http://www.r-bloggers.com** !



**hefinioanrhys** **10th Oct 2019** **Uncategorized**

## Leave a Reply