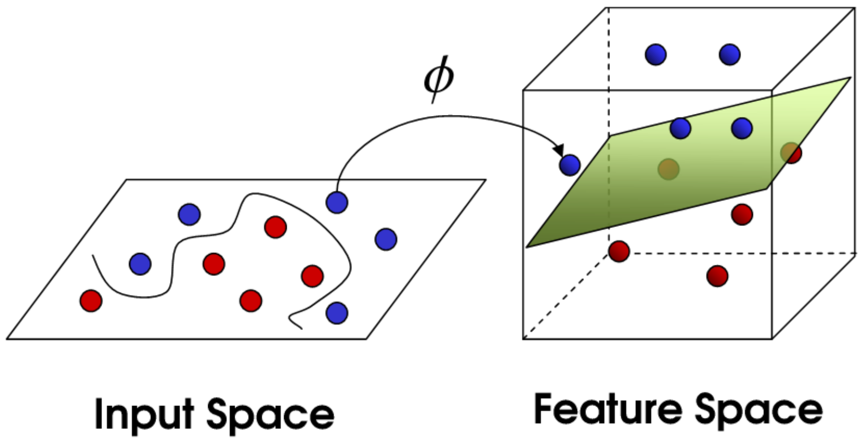
The Kernel Trick in Support Vector Classification

Drew Wilimitis Dec 12, 2018 · 7 min read



The kernel trick seems to be one of the most confusing concepts in statistics and machine learning; it first appears to be genuine mathematical sorcery, not to mention the problem of lexical ambiguity (does kernel refer to: a non-parametric way to estimate a probability density (statistics), the set of vectors **v** for which a linear transformation T maps to the zero vector — i.e. T(**v**) = 0 (linear algebra), the set of elements in a group G that are mapped to the identity element by a homomorphism between groups (group theory), the core of a computer operating system (computer science), or something to do with the seeds of nuts or fruit?).

Although there are some obstacles to understanding the kernel trick, it is highly important to understand how kernels are used in support vector classification. For practical reasons, it is important to understand because implementing support vector classifiers requires specifying a kernel function, and there are not established, general rules to know what kernel will work best for your particular data.

More conceptually, the kernel trick also illustrates some fundamental ideas about different ways to represent data and how machine learning algorithms "see" these different data representations. And finally, the seeming mathematical sleight of hand in the kernel trick just begs one to further explore what it actually means.

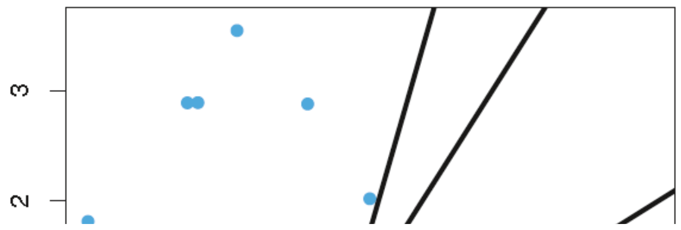
I will certainly not be able to fully explain the kernel trick in this post. I believe a truly deep understanding of the kernel trick requires a rigorous mathematical treatment, and this can't be done in what is labeled as a seven minute read, let alone by someone who realistically has only been learning about the kernel trick and SVMs for a short period of time. However, I will attempt to do the following:

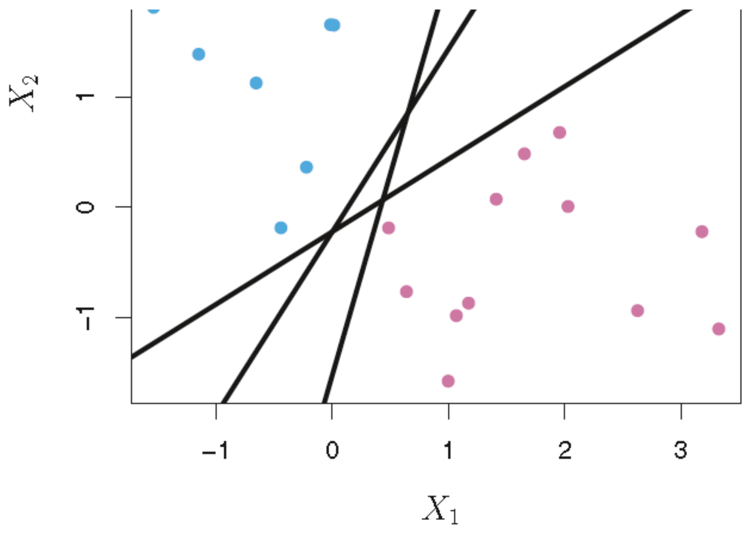
- briefly introduce support vector classification
- visualize some non-linear transformations in the context of support vector classification
- introduce the idea that the benefit of the kernel trick in training support vector classifiers lies in a unique data representation

Intro to Support Vector Classification

Support vector classification is based on a very natural way that one might attempt to classify data points into various target classes. If the classes in our training data can be separated by a line or some boundary, then we can just classify the data depending on what side of this decision boundary the data lies on.

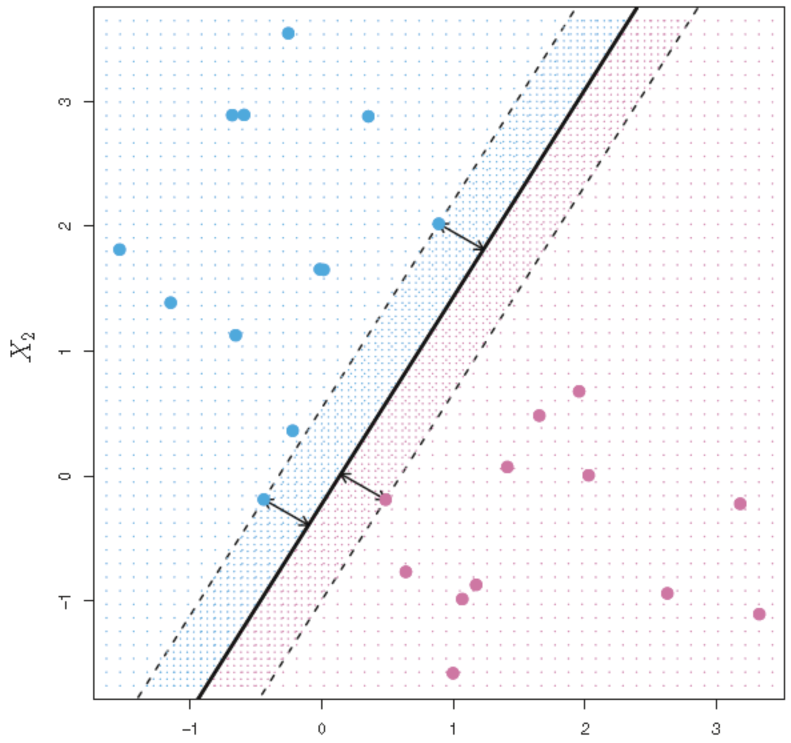
In the following 2-d example, we can separate the data with any of the three lines, and then just assign classes based on whether the observation lies above or below the line. The data are 2-dimensional vectors specified by the features X1 and X2 with class labels as either y =1 (blue) or y = 0 (red).

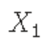




An example dataset showing classes that can be linearly separated.

Training a linear support vector classifier, like nearly every problem in machine learning, and in life, is an optimization problem. We maximize the *margin* — the distance separating the closest pair of data points belonging to opposite classes. These points are called the support vectors, because they are the data observations that "support", or determine, the decision boundary. To train a support vector classifier, we find the *maximal margin hyperplane*, or *optimal separating hyperplane*, which optimally separates the two classes in order to generalize to new data and make accurate classification predictions.





The support vectors are the points on the dashed lines. The distance from the dashed line to the solid line is the margin, represented by the arrows.

Support vector machines are much harder to interpret in higher dimensions. It is much harder to visualize how the data can be linearly separable, and what the decision boundary will look like. A hyperplane in p-dimensions is a p-1 dimensional "flat" subspace that lies inside the larger p-dimensional space. In 2 dimensions, the hyperplane is just a line. In 3 dimensions, the hyperplane is a regular 2-d plane. Mathematically, we have the following:

Equation of a Hyperplane in 2 dimensions

$$\beta_0 + \beta_1 \mathbf{X_1} + \beta_2 \mathbf{X_2} = 0$$

Equation of a Hyperplane in p-dimensions

$$\beta_0 + \beta_1 \mathbf{X_1} + \ldots + \beta_p \mathbf{X_p} = 0$$

We have points on the hyperplane:

$$\mathbf{X} = [\mathbf{X_1}, \ldots, \mathbf{X_p}]^T$$

where $\mathbf{X}$ is a feature vector

We can use this to make classifications by considering one class defined by:
$$\beta_0 + \beta_1 \mathbf{X_1} + \ldots + \beta_p \mathbf{X_p} > 0$$

With the other class defined by:
$$\beta_0 + \beta_1 \mathbf{X_1} + \ldots + \beta_p \mathbf{X_p} < 0$$

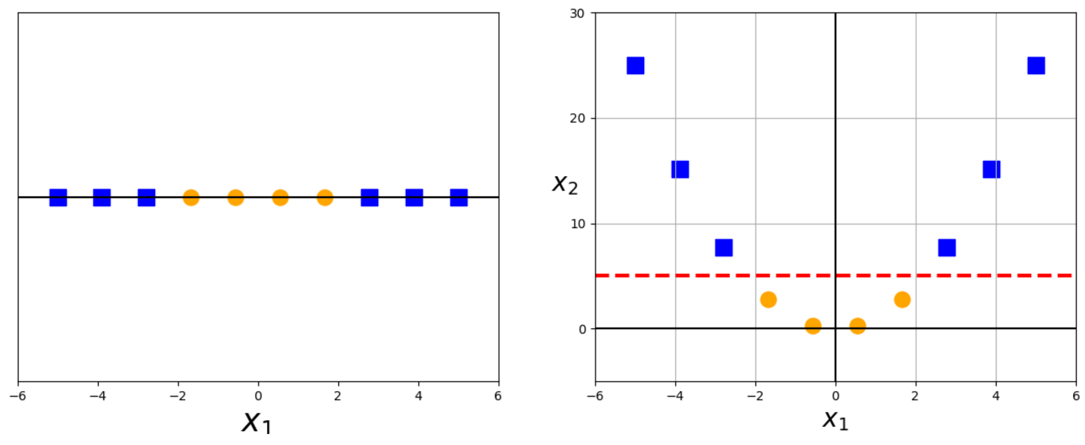
Equations defining a hyperplane and the classification rules defined by the decision boundary.

Support vector classification relies on this notion of linearly separable data. "Soft margin" classification can accommodate some classification errors on the training data, in the case where data is not perfectly linearly separable. However, in practice data is often very far from being linearly separable, and we need to transform the data into a higher dimensional space in order to fit a support vector classifier.

Non-linear transformations

If the data is not linearly separable in the original, or input, space then we apply transformations to the data, which map the data from the original space into a higher dimensional feature space. The goal is that after the transformation to the higher dimensional space, the classes are now linearly separable *in this higher dimensional feature space*. We can then fit a decision boundary to separate the classes and make predictions. The decision boundary will be a hyperplane in this higher dimensional space.

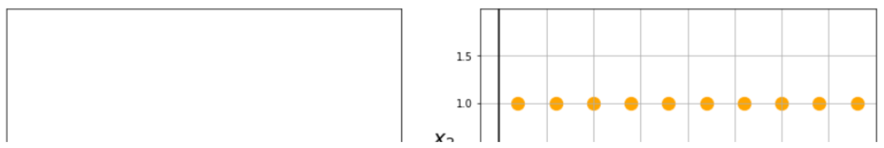
It is obviously hard to visualize higher dimensional data, and so we first focus on some transformations applied to 1-dimensional data. In this example, the picture on the left shows our original data points. In 1-dimension, this data is not linearly separable, but after applying the transformation $\phi(x) = x^2$ and adding this second dimension to our feature space, the classes become linearly separable.

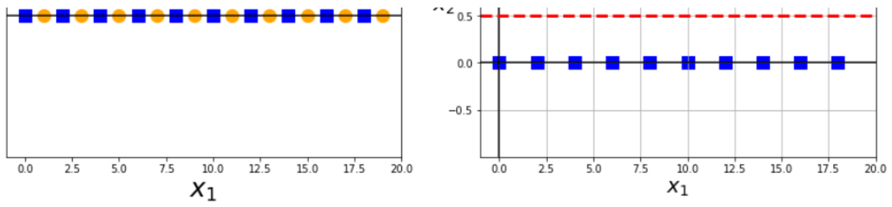


This data becomes linearly separable after a quadratic transformation to 2-dimensions.

For now, we are just examining transformations of the original data to higher dimensions that allow the data to be linearly separated. These are just functions, and there are many possible functions that can map the data to any number of higher dimensions.

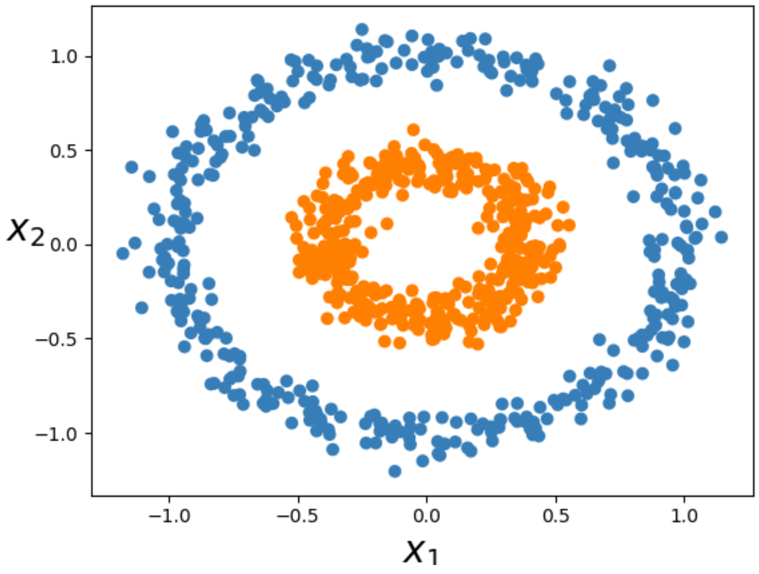
Here we apply the transformation $\phi(x) = x \bmod 2$





This transformation allows us to linearly separate the even and odd X1 values in 2 dimensions.

Now let's look at an example where our original data is not linearly separable in two dimensions. Here is our original data, which cannot be linearly separated.



After the following transformation:

Our data becomes linearly separable (by a 2-d plane) in 3-dimensions.

Linearly separable data in 3-d after applying the 2nd-degree polynomial transformation

There can be many transformations that allow the data to be linearly separated in higher dimensions, but not all of these functions are actually kernels. The kernel function has a special property that makes it particularly useful in training support vector models, and the use of this property in optimizing non-linear support vector classifiers is often called the kernel trick.

The Kernel Trick

We have seen how higher dimensional transformations can allow us to separate data in order to make classification predictions. It seems that in order to train a support vector classifier and optimize our objective function, we would have to perform operations with the higher dimensional vectors in the transformed feature space. In real applications, there might be many features in the data and applying transformations that involve many polynomial combinations of these features will lead to extremely high and impractical computational costs.

The **kernel trick** provides a solution to this problem. The "trick" is that kernel methods represent the data only through a set of pairwise similarity comparisons between the original data observations $\mathbf{x}$ (with the original coordinates in the lower dimensional space), instead of explicitly applying the transformations $\phi(\mathbf{x})$ and representing the data by these transformed coordinates in the higher dimensional feature space.

In kernel methods, the data set **X** is represented by an n x n kernel matrix of pairwise similarity comparisons where the entries (i, j) are defined by the **kernel function:** k(xi, xj). This kernel function has a special mathematical property. The kernel function acts as a modified dot product. We have:

Our kernel function accepts inputs in the original lower dimensional space and returns the dot product of the transformed vectors in the higher dimensional space. There are also theorems which guarantee the existence of such kernel functions under certain conditions.

It can somewhat help to understand how the kernel function is equal to the dot product of the transformed vectors by considering that each coordinate of the transformed vector $\phi(\mathbf{x})$ is just some function of the coordinates in the corresponding lower dimensional vector **x**.

For example, the kernel trick for the 2nd-degree polynomial is illustrated below, and we visualized this transformation in 3-d in a previous figure. The transformed vectors have coordinates that are functions of the two components x1 and x2. so the dot product will only involve components x1 and x2 as well. The kernel function will also take inputs x1, x2 and return a real number. The dot product always returns a real number too.

On the left-hand side, we have the dot product of the transformed feature vectors, which is equal to our 2nd-degree polynomial kernel function.

The kernel function here is the polynomial kernel k(a,b) = (a^T * b)²

The ultimate benefit of the kernel trick is that the objective function we are optimizing to fit the higher dimensional decision boundary only includes the dot product of the transformed feature vectors. Therefore, we can just substitute these dot product terms with the kernel function, and we don't even use $\phi(\mathbf{x})$.

In the bottom equation, we replace the dot product of the transformed vectors with the kernel function.

Top highlight

Remember, our data is only linearly separable as the vectors $\phi(\mathbf{x})$ in the higher dimensional space, and we are finding the optimal separating hyperplane in this higher dimensional space *without having to calculate or in reality even know anything about $\phi(x)$.*

Machine Learning Data Science Statistics Mathematics Data