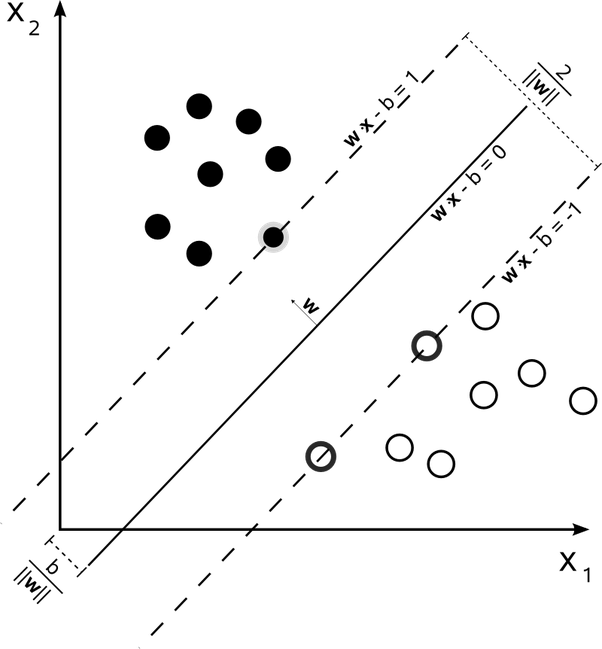
**Intuitive Explanation of Support Vector Machines**

****

A vanilla SVM is a type of linear separator. Suppose we want to split black circles from the white ones above by drawing a line. Notice that there are an infinite number of lines that will accomplish this task. SVMs, in particular, find the "maximum-margin" line - this is the line "in the middle". Intuitively, this works well because it allows for noise and is most tolerant to mistakes on either side.   
  
That's all a (binary) SVM really is. We draw a straight line through our data down the middle to separate it into two classes. But what if we don't want a straight line but a curved one? We achieve this not by drawing curves, but by "lifting" the features we observe into higher dimensions. For example, if we can't draw a line in the space (x1, x2) then we may try adding a third dimension, (x1, x2, x1 \* x2). If we project the "line" (actually called a "hyperplane") in this higher dimension down to our original dimension, it looks like a curve.

By using the kernel trick, we can do these lifts very efficiently (actually, we don't really do them at all). Kernels allow us to draw "straight lines" in very high dimensions, and even infinite dimensions.  
  
EDIT: notice that to draw the maximum margin line, only the data points closest to the line matter (these are the ones with darker borders). These points are known as "support vectors" which gives SVMs its name.

In layman’s terms, support vector machine is a generalization of [Nearest Neighbor](https://en.wikipedia.org/wiki/Nearest_neighbor_search) (NN) algorithm.

So some of the observations from NN algorithm are :

1. There is no training phase in this algorithm. We don’t do any computation alone with training points. Only when a test point comes, we compute similarity from **every** training point. This is a big disadvantage of NN algorithm. Consider having millions of training points, and for every test point, we have to calculate millions of similarities from test point.
2. We calculate similarities from the training points which are very very far from test points, which are not really required.
3. We don’t give any importance to other training points except the nearest one.

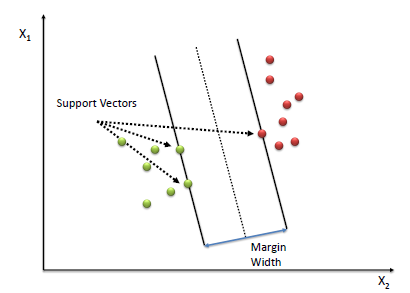
**Support Vector Machine**

In SVM, we remove these disadvantages as :

1. Instead of finding similarity from every training point, we calculate similarity from only a **subset** of training points, which we compute in the **training phase**. So in the training phase, we fix a subset of training points from which we compute the similarity of any test point. These selected training points are called **support vectors,** since only these points will support our decision of selecting the class of a test point. Our hope is that our training phase finds as few as support vectors so that we have to compute fewer number of similarities.
2. Now once we have selected support vectors, we assign a **weight** to each support vector, which basically tells how much importance we want to give to that support vector while making our decision. So unlike NN, we don’t give importance to only a single training point, instead we give each support vector a separate importance.

Now having understood what SVMs are, we have following questions :

1. **How do we choose support vectors ?** It is actually a by-product of finding weights (the alphas). So in training phase, we find weight for each training point, and those points whose weight becomes zero are not support vectors i.e. their importance is zero during test time, and rest are support vectors.
2. **So then how to learn alphas ?** Its at this point, where all the math comes. Since the question asks for layman’s explanation, I will skip the math here (you can read the math [here](http://cs229.stanford.edu/notes/cs229-notes3.pdf)). Intuitively, we try to find that separating hyperplane, from which distance of closest training points is maximum (also known as max-margin classifier), and those closest training points then become support vectors. During the math, while optimizing the function, we get the weights of the support vectors. Pictorially, this looks like :



In the above figure, there are only 3 support vectors, so at the test time, we will compute similarity test point and only these 3 support vectors. Generally, there are very few support vectors out of all training points, and hence at test time, we need to calculate very few similarities.

3. What exactly the similarity function *sim*(*x*,*sj*)

is ? During the math we do in above step, it turns out that the similarity function is **dot product** i.e. *sim*(*x*,*sj*)=⟨*x*,*sj*⟩

(remember dot product measures similarity between two vectors).

**The Fun Part :** Now here comes the fun part, instead of taking dot product between two simple vectors *x*

and *s* (where *s* is one of the support vectors), we can transform each of the vector into some other **higher dimensional** vector, and then take dot product between those two complex, higher dimensional vectors, and whole thing still works, and in fact works better since we have improved the similarity function. In particular, we can apply some transformation function *ϕ* to *x* and *s* to get two big vectors *ϕ*(*x*) and *ϕ*(*s*)

and then we can take their dot product to get a quantity which is called **Kernel** :

*K*(*x*,*s*)=⟨*ϕ*(*x*),*ϕ*(*s*)⟩

So Kernel is nothing but a similarity measure between two transformed vectors.

So our equation for computing *D*

becomes :

*D*=∑*kj*=1*yj*∗*αj*∗*K*(*x*,*sj*)

**The Kernel Trick :**

So now the question arises whether we need to actually transform the vectors into higher dimension to get the Kernel value ? It turns out that the answer is **not necessarily**. See, in above equation of calculating *D*

, we require only the final answer of *K*(*x*,*sj*)

. We actually don’t care, which transformation function was applied to each of the vectors. So if there is a similarity function (a kernel), which can be factorized into dot product of two transformed vectors, then that kernel function can be used in above equation. We don’t even need to compute exact transformation, as long as we can prove above thing. This is called **the kernel trick**.

So earlier, we used a very simple kernel, which was just the dot product between *x*

and *s*

(in their original dimension). Now we may ask the question whether square of that dot product is a kernel ? i.e. whether

*K*(*x*,*s*)=(⟨*x*,*s*⟩)2

is a kernel or not. For simplicity, let both vectors *x*

and *s*

are two dimensional vectors. We can prove that it is in fact a valid kernel by noting that

(⟨*x*,*s*⟩)2=(*x*1*s*1+*x*2*s*2)2=(*x*21*s*21+*x*22*s*22+2*x*1*s*1*x*2*s*2)

=⟨{*x*21,*x*22,2–√*x*1*x*2},{*s*21,*s*22,2–√*s*1*s*2}⟩

So as we see here, this kernel is a dot product between two vectors :

*ϕ*(*x*)={*x*21,*x*22,2–√*x*1*x*2},*ϕ*(*s*)={*s*21,*s*22,2–√*s*1*s*2}

which are nothing but transformation of vectors *x*

and *s*

.

Note that we don’t have to actually calculate transformed vectors, we just require kernel function, which is very easy to compute, for example, here, its just the square of dot product.

4. Which kernel function to use ? We have 3 kinds of common kernel functions :

* **Linear Kernel :** *K*(*x*,*s*)=⟨*x*,*s*⟩

 . This is just the dot product between original vectors.

 **Polynomial Kernel :** *K*(*x*,*s*)=(⟨*x*,*s*⟩)*p*. We saw an example of this above, in which *p*=2

 .

 **Gaussian Kernel :** *K*(*x*,*s*)=exp(−||*x*−*s*||/*σ*)

* . This is just an unnormalized gaussian function (we don’t need to normalize it, since we need just similarity measure, and not a distribution). This is super powerful kernel since it is the dot product between two infinite dimensional vectors. So the amazing part is that we are calculating dot product between two infinite dimensional vectors without actually having those vectors !!!.

So to summarize, SVM is the generalization of NN algorithm with weights assigned to some of the training instances (support vectors) which decide the class of test instance. Moreover, instead of using simple similarity functions, we can apply a variety of complex similarity functions.

There is a lot of math behind it, but I think you get the big picture.