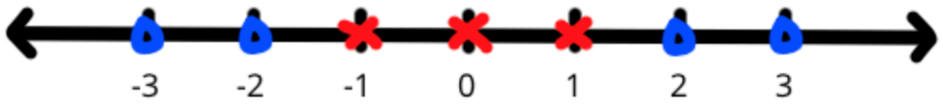
Kernelized Support Vector Machines (SVM)

Feature Mapping

Linear models are great because they are easy to understand and easy to optimize. They suffer because they can only learn very simple decision boundaries. One way of getting a linear model to "behave" non-linearly is to transform the input to higher dimensions, leading to a richer feature space. This technique is known as **feature mapping**.

This can be demonstrated by a simple example:

x	y
-1	1
0	1
1	1
-3	0
-2	0
2	0
3	0



It's clear that **there exists no line that can perfectly separate the data** (in 1D, a boundary is a vertical line given by the equation $x = x_b$, where x_b is the boundary).

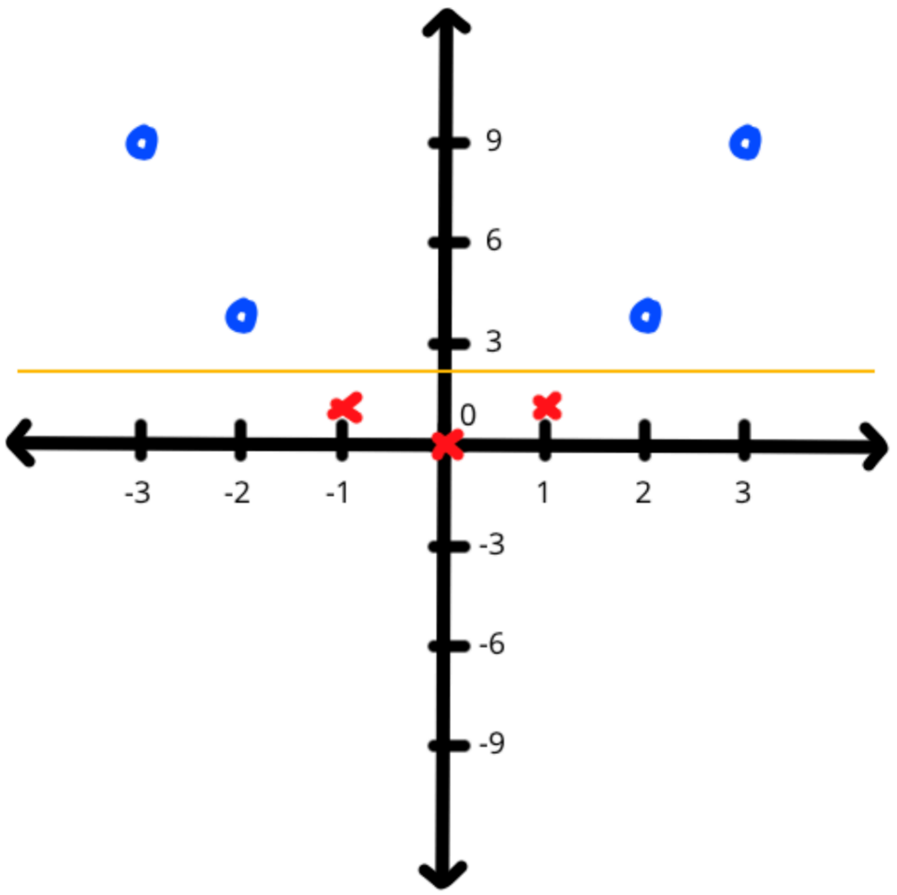
Let's try to use feature mapping to solve this. We introduce a function $\phi(x) : \mathbb{R} \rightarrow \mathbb{R}^2$:

$$\phi(x) = \begin{bmatrix} x \\ x^2 \end{bmatrix}$$

$\phi(x)$ is a function that maps from the first dimension to the second dimension. In this case, the first component of $\phi(x)$ is coordinate for the "x-axis" or the absicissa, and the second component of $\phi(x)$ is the coordinate for the "y-axis" or the ordinate.

Graphically:

x	x^2	y
-1	1	1
0	0	1
1	1	1
-3	9	0
-2	4	0
2	4	0
3	9	0

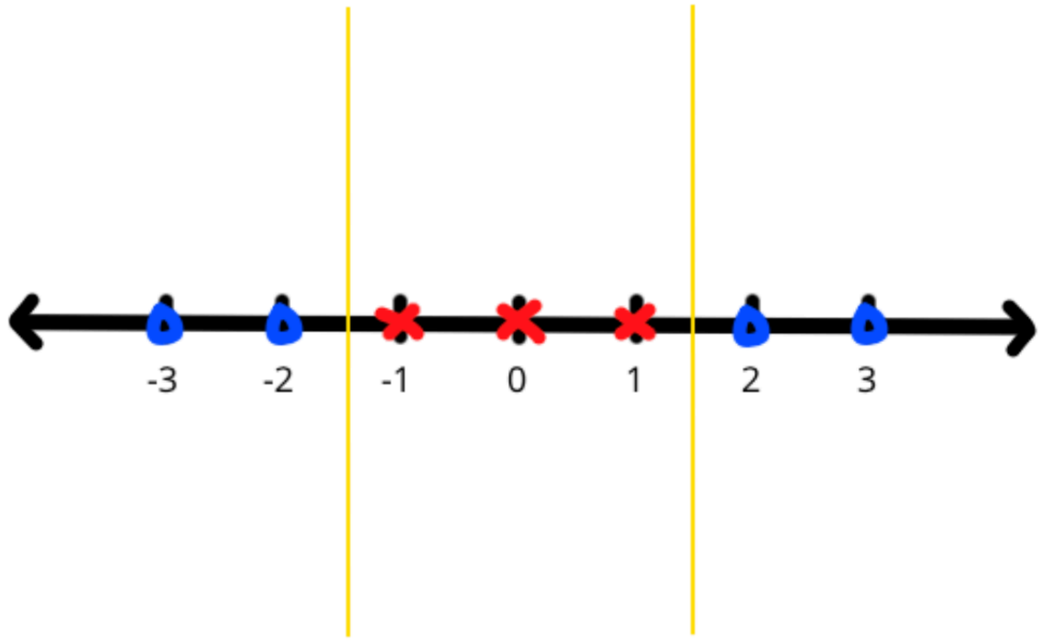


By mapping our data to a higher dimension, we find that is is now **linearly separable**, meaning an algorithm like perceptron learning would now be able to converge (whereas it would never converge in the 1D case).

How does the hyperplane "translate" down to our original feature space, $\mathbb{R}$?

- In traditional xy-coordinate systems, the equation of a horizontal line is given by $y = y_0$, where y_0 is the y-intercept. In our case, the y variable or ordinate is really x^2.
- If we allow our hyperplane to intercept the "x^2"-axis at 2 (which is just one of many acceptable hyperplanes), we have the equation $x^2 = 2$. This equation has two solutions, or $x = \pm\sqrt{2}$.

Therefore, graphically, in $\mathbb{R}$ we have:



The Problem with Feature Mapping

The most notable problem when it comes to feature mapping is **computational complexity**. As you map to higher and higher dimensions, the time (and space) it takes to compute all of the higher-dimensional training examples increases exponentially.

To solve, this we introduce an elegant solution known as Kernels.

Kernels

The key insight in **kernel-based learning** is that you can *rewrite* many linear models in a way that doesn't ever require you to explicitly compute the feature map $\phi(x)$.

- Many machine learning algorithms involve a product of the form $\vec{\theta} \cdot \phi(x)$ (where in this case, we're using feature mapping).
- The goal is to rewrite these algorithms such that they only ever depend on dot products between two examples. If we name these examples $\vec{x}, \vec{z}$, we want to rewrite these algorithms in terms of $\phi(\vec{x}) \cdot \phi(\vec{z})$.
- **Therefore, the point of kernel-based learning is to try and compute $\phi(\vec{x}) \cdot \phi(\vec{z})$ without ever having to compute $\phi(\vec{x})$ or $\phi(\vec{z})$.**
    - Rather, the only computation we would like to have to do is $\vec{x} \cdot \vec{z}$.

It's easiest to see how this works with an example. Let's revisit the 1-D example we used above. We have the feature map:

$$\phi(x) = \begin{bmatrix} x \\ x^2 \end{bmatrix}$$

Remembering that $x \in \mathbb{R}$ and $\phi(x) \in \mathbb{R}^2$, **our goal is to try and express $\phi(x) \cdot \phi(z)$ in terms of $x \cdot z$.** Since we are in the first dimension, we simply have $x \cdot z = xz$.

We start by calculating $\phi(x) \cdot \phi(z)$, and try to see if it is possible to simplify it **solely** in terms of xz.

$$\begin{aligned} \phi(x) \cdot \phi(z) &= \begin{bmatrix} x \\ x^2 \end{bmatrix} \cdot \begin{bmatrix} z \\ z^2 \end{bmatrix} \\ &= xz + x^2 z^2 \\ &= xz(1 + xz) \end{aligned}$$

And there we have it. Although it seems rather arbitrary, what this example shows is that **it is possible to compute the two-dimensional dot product $\phi(x) \cdot \phi(z)$ by only explicitly computing the one-dimensional dot product xz.**

Often, we notate $\phi(\vec{x}) \cdot \phi(\vec{z})$ more simply as $K(\vec{x}, \vec{z})$. We call K the kernel function, and it outputs a scalar.

Let's look at one more example where we start from a kernel function and try to understand the higher-dimensional dot product it represents.

Consider a kernel function for vectors in $\mathbb{R}^2$:

$$K(\vec{x}, \vec{z}) = (1 + \vec{x} \cdot \vec{z})^2$$

Given that $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ and $\vec{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$,

$$\begin{aligned} K(\vec{x}, \vec{z}) &= (1 + x_1 z_1 + x_2 z_2)^2 \\ &= (1 + x_1 z_1 + x_2 z_2)(1 + x_1 z_1 + x_2 z_2) \\ &= 1 + x_1 z_1 + x_2 z_2 + x_1 z_1 + x_1^2 z_1^2 + x_1 x_2 z_1 z_2 + x_2 z_2 + x_1 x_2 z_1 z_2 + x_2^2 z_2^2 \\ &= 1 + 2x_1 z_1 + 2x_2 z_2 + 2x_1 x_2 z_1 z_2 + x_1^2 z_1^2 + x_2^2 z_2^2 \end{aligned}$$

Again, our goal is to try to express this in terms of $\phi(\vec{x}) \cdot \phi(\vec{z})$. To do this, we look at each summand and we find:

$$K(\vec{x}, \vec{z}) = \begin{bmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \sqrt{2}x_1 x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \sqrt{2}z_1 \\ \sqrt{2}z_2 \\ \sqrt{2}z_1 z_2 \\ z_1^2 \\ z_2^2 \end{bmatrix}$$

Now, our underlying feature map is obvious:

$$\phi(\vec{x}) = \begin{bmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \sqrt{2}x_1x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix}$$

Again, the crux is that our original function in 2-D, $K(\vec{x}, \vec{z}) = (1 + \vec{x} \cdot \vec{z})^2$, allows us to compute a dot product in a 6-dimensional space without ever computing $\phi(\vec{x})$ and $\phi(\vec{z})$.

Polynomial Kernel

The kernel seen in the example above is known as the **polynomial kernel** and has the more general form:

$$K_d(\vec{x}, \vec{z}) = (1 + \vec{x} \cdot \vec{z})^d$$

where d is a hyperparameter.

Gaussian Kernel

The **Gaussian kernel** is a kernel that actually maps the input vector to a feature space of infinite dimensions! It has the equation:

$$K(\vec{x}, \vec{z}) = e^{-||\vec{x}-\vec{z}||^2/2\sigma^2}$$

where σ is a hyperparameter determining the "width" of the kernel. It's interpreted as the desired standard deviation of the kernel.

Kernel Criteria

What makes a valid kernel? We've already established one solution; that is, K is valid if and only if there exists a function ϕ such that $K(\vec{x}, \vec{z}) = \phi(\vec{x}) \cdot \phi(\vec{z})$.

An alternative criterion is more rooted in mathematics. This property is called **Mercer's condition**, and states that a function K is a valid kernel if K is **positive semi-definite**. This boils down to proving:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} K(\vec{x}_i, \vec{x}_j) c_i c_j \geq 0 \, \forall \, c_i, c_j \in \mathbb{R}.$$

This allows us to come up with a number of rules for composing kernels out of other kernels. Let K_1, K_2 be valid kernels on X. Then, the following are valid kernels:

- $K(\vec{x}, \vec{z}) = \alpha K_1(\vec{x}, \vec{z}) + \beta K_2(\vec{x}, \vec{z})$, for $\alpha, \beta \geq 0$.

- $K(\vec{x}, \vec{z}) = K_1(\vec{x}, \vec{z})K_2(\vec{x}, \vec{z})$.
- $K(\vec{x}, \vec{z}) = K_1(f(\vec{x}), f(\vec{z}))$, where $f : X \to X$.
- $K(\vec{x}, \vec{z}) = g(\vec{x})g(\vec{z})$, where $g : X \to \mathbb{R}$.
- $K(\vec{x}, \vec{z}) = f(K_1(\vec{x}, \vec{z}))$ where f is a polynomial with positive coefficients.

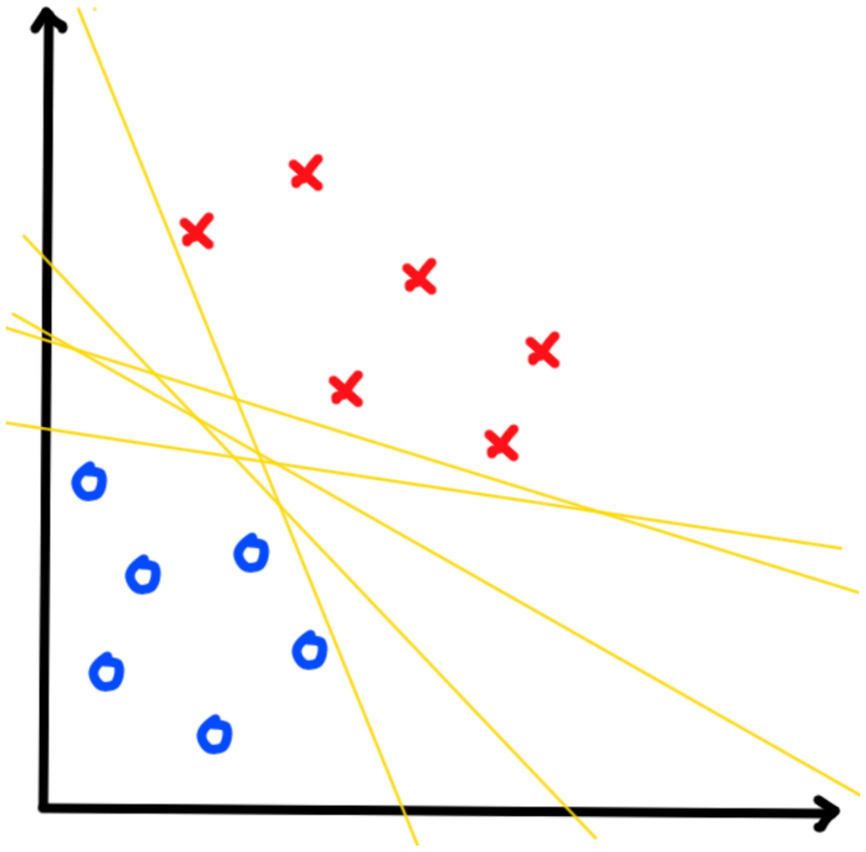
TODO: kernelized knn example

Support Vector Machines

SVMs are among the best (and many believe are indeed the best) “off-the-shelf” supervised learning algorithms. To tell the SVM story, we’ll need to first talk about margins and the idea of separating data with a large “gap.”

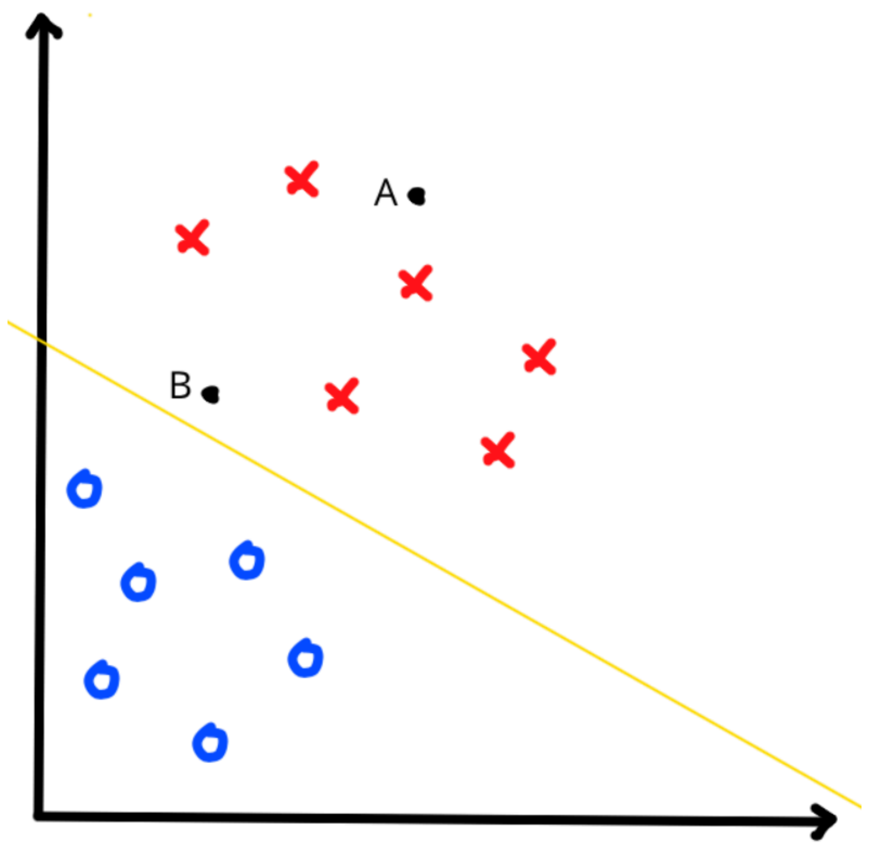
Margins

Consider the following plot of linearly-separable data:



All of the decision boundaries (i.e. yellow lines) in this picture are **valid** for this training set; so which one should we pick? Is this decision arbitrary, or is there some criteria for what makes a "better" boundary?

Here, it's important to remember the point of classifiers in the first place: we would like to classify **new, incoming data** whose labels we **don't** know. Assume we select one of the margins in the photo above, and consider a few new data points, labeled A and B:



We note that:

- **A is far from the decision boundary.** If we were asked to predict what A's class was, we would be very **confident** in saying that A belongs to the class of red Xs.
- Conversely, **B is close to the decision boundary**. While we would indeed predict that B also belongs to the class of red Xs given this boundary, it is very likely that **a small change to the boundary could change our prediction**.

What we can glean from this is that **the further a point is from the decision boundary, the more confident we are in our prediction for that point**.

Given this, we can informally think that it would be nice if, given a training set, we manage to find a decision boundary that allows us to make all **correct** and **confident** (i.e. far from the decision boundary) predictions on the **training examples**.

Geometric Margins

Classifier Equation

In order to formalize this inductive bias, we must first derive some equations that will allow use to talk about distance and our decision boundary in mathematical terms.

First, we notate the **equation of our classifier** as:

$$h(\vec{x}) = \text{sign}(\vec{\theta}^T \vec{x} + b)$$

where the sign function is simply defined as:

$$\text{sign}(z) = \begin{cases} +1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$$

Note that this classifier has the exact same form as the **perceptron**. Again, what this function says is that given a new data point $\vec{x}$, we compute the quantity $a = \vec{\theta}^T \vec{x} + b$, where $\vec{\theta}$ and b are the learned weights and bias respectively.

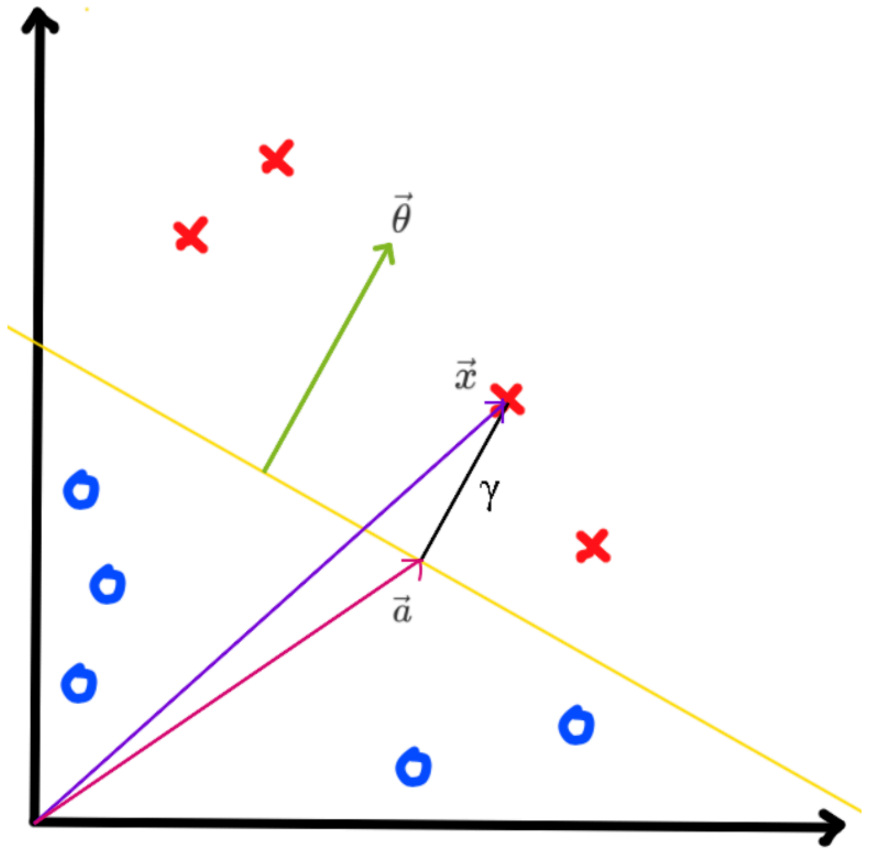
- If a is non-negative, then we predict that $\vec{x}$ belongs to the class $y = +1$.
- Otherwise, if a is negative, then we predict that $\vec{x}$ belongs to the class $y = -1$.

Therefore, our decision boundary is given by the set of points $\vec{x}$ where the sign of our hypothesis, $h(\vec{x})$, changes from -1 to $+1$. In other words, it is the set of points where $\vec{\theta}^T \vec{x} + b = 0$. We now have an equation for our decision boundary:

$$\mathcal{W}: \ \vec{\theta}^T \vec{x} + b = 0$$

Distance from the Decision Boundary

We define the **distance** of a training sample/point from the decision boundary as the **shortest distance from that point to the boundary**. This is equivalent to the length of the perpendicular line from the decision boundary to the point. Consider the following picture:



Here, we have:

- $\vec{x}$, an arbitrary training sample (purple)
- γ, the distance of $\vec{x}$ from the decision boundary (black)
- $\vec{\theta}$, the weight vector defining the decision boundary (green)
    - $\vec{\theta}$ is perpendicular to the decision boundary, as proved in the notes on perceptrons.
- $\vec{a}$, a vector from the origin to the point on the decision boundary (pink)
    - A line from this point to $\vec{x}$ is perpendicular to the boundary.

Our goal is to solve for γ, the distance of $\vec{x}$ from the boundary. We begin by noting that $\frac{\vec{\theta}}{||\vec{\theta}||}$ is a unit vector perpendicular to the decision boundary. Then, $-\gamma\frac{\vec{\theta}}{||\vec{\theta}||}$ is the vector from the head of $\vec{x}$ to the head of $\vec{a}$, meaning we have:

$$\vec{a} = \vec{x} - \gamma\frac{\vec{\theta}}{||\vec{\theta}||}$$

However, since $\vec{a}$ lies on the decision boundary, we know it must satisfy the equation:

$$\vec{\theta}^T\vec{a} + b = 0$$

Plugging in the value of $\vec{a}$ above, we have:

$$\begin{aligned}
\vec{\theta}^T\left(\vec{x} - \gamma\frac{\vec{\theta}}{||\vec{\theta}||}\right) + b &= 0 \\
\vec{\theta}^T\vec{x} - \gamma\frac{\vec{\theta}^T\vec{\theta}}{||\vec{\theta}||} + b &= 0 \\
\vec{\theta}^T\vec{x} - \gamma\frac{||\vec{\theta}||^2}{||\vec{\theta}||} + b &= 0 \\
\gamma||\vec{\theta}|| &= \vec{\theta}^T\vec{x} + b \\
\therefore \quad \gamma &= \frac{\vec{\theta}^T\vec{x}}{||\vec{\theta}||} + \frac{b}{||\vec{\theta}||}.
\end{aligned}$$

The Geometric Margin

One issue with this definition is that γ can be negative (remember we classify a point as $y = -1$ when $h(\vec{x})$ is negative). To fix this, we use the fact that $y \in \{-1, 1\}$.

Thus, we define the **geometric margin** of $(\vec{\theta}, b)$ with respect to a **training example** $(\vec{x}, y)$ as:

$$\gamma = y\left(\left(\frac{\vec{\theta}}{||\vec{\theta}||}\right)^T\vec{x} + \frac{b}{||\vec{\theta}||}\right).$$

Finally, the define the **geometric margin** of $(\vec{\theta}, b)$ with respect to the **entire training set** S as:

$$\gamma = \min_{i=1,\ldots,m} \gamma^{(i)}.$$

- This just means that γ with respect to S is defined as the **smallest** distance from the decision boundary to one of the points in S.

The Optimal Margin Classifier

Given a training set and our results above, our goal now is to find a decision boundary that **maximizes the geometric margin**, since this would reflect a very confident set of predictions. Specifically, we want to maximize the "distance" between the positive and negative examples.

With this in mind, we now propose our first **constrained** optimization problem:

$$\begin{aligned} \max_{\gamma, \vec{\theta}, b} \quad & \gamma \\ \text{s.t.} \quad & y_i(\vec{\theta}^T \vec{x}_i + b) \geq \gamma,\ i = 1, \ldots, m, \\ & ||\vec{\theta}|| = 1. \end{aligned}$$

- Here, we have used the property that scaling the weight vector does not change the decision boundary it defines (as proved in the notes on perceptrons) to set $||\vec{\theta}|| = 1$. This allows us to simplify the equation of γ.
- Intuitively, this problem is trying to **maximize γ such that the geometric margin of each point is at least γ.**

Unfortunately, the constraint $||\vec{\theta}|| = 1$ is **non-convex**, which essentially means it makes our question incredibly difficult to solve. We need to try to transform it into something nicer.

Is there another way we can get rid of the $||\vec{\theta}||$ in the denominator? Consider setting $\gamma = \frac{\hat{\gamma}}{||\vec{\theta}||}$:

$$\begin{aligned} y\left(\left(\frac{\vec{\theta}}{||\vec{\theta}||}\right)^T \vec{x} + \frac{b}{||\vec{\theta}||}\right) &= \frac{\hat{\gamma}}{||\vec{\theta}||} \\ \frac{y}{||\vec{\theta}||}(\vec{\theta}^T \vec{x} + b) &= \frac{\hat{\gamma}}{||\vec{\theta}||} \\ y(\vec{\theta}^T \vec{x} + b) &= \hat{\gamma} \end{aligned}$$

Therefore, our optimization problem above is also equivalent to:

$$\begin{aligned} \max_{\hat{\gamma}, \vec{\theta}, b} \quad & \frac{\hat{\gamma}}{||\vec{\theta}||} \\ \text{s.t.} \quad & y_i(\vec{\theta}^T \vec{x}_i + b) \geq \hat{\gamma},\ i = 1, \ldots, m. \end{aligned}$$

- Note again that this means the geometric margin is $\gamma = \frac{\hat{\gamma}}{||\vec{\theta}||}$.

This, however, is *still* not quite good enough as the equation $\frac{\hat{\gamma}}{||\vec{\theta}||}$ is also non-convex! To do better, we will need an additional property of the geometric margin:

Proposition: If we scale $\vec{\theta}$ and b by a constant k, the geometric margin γ does not change.

Proof:

$$\begin{aligned}\gamma &= y\left(\left(\frac{k\vec{\theta}}{k||\vec{\theta}||}\right)^T \vec{x} + \frac{kb}{k||\vec{\theta}||}\right)\\ &= y\left(\left(\frac{\vec{\theta}}{||\vec{\theta}||}\right)^T \vec{x} + \frac{b}{||\vec{\theta}||}\right).\end{aligned}$$

Although seemingly arbitrary, recall the equation we have when $\gamma = \frac{\hat{\gamma}}{||\vec{\theta}||}$:

$$y(\vec{\theta}^T \vec{x} + b) = \hat{\gamma}.$$

The property we just proved **allows us to set $\hat{\gamma}$ to whatever constant we want**, since we can accomplish that by simply scaling $\vec{\theta}$ and b! So, setting $\hat{\gamma} = 1$, we arrive at the problem:

$$\begin{aligned}&\max_{\vec{\theta},b} \ \frac{1}{||\vec{\theta}||}\\ &\text{s.t. } \ y_i(\vec{\theta}^T \vec{x}_i + b) \geq 1, \ i = 1, \dots, m.\end{aligned}$$

Finally (to make the math a little nicer), we use the fact that maximizing $\frac{1}{||\vec{\theta}||}$ is the exact same thing as minimizing $||\vec{\theta}||^2$. We now have:

$$\begin{aligned}&\min_{\vec{\theta},b} \ \frac{1}{2}||\vec{\theta}||^2\\ &\text{s.t. } \ y_i(\vec{\theta}^T \vec{x}_i + b) \geq 1, \ i = 1, \dots, m.\end{aligned}$$

This problem has a **convex** quadratic objective function and **linear** constraints, and can now be efficiently solved to give us the **optimal margin classifier**.

Constrained Optimization Problems

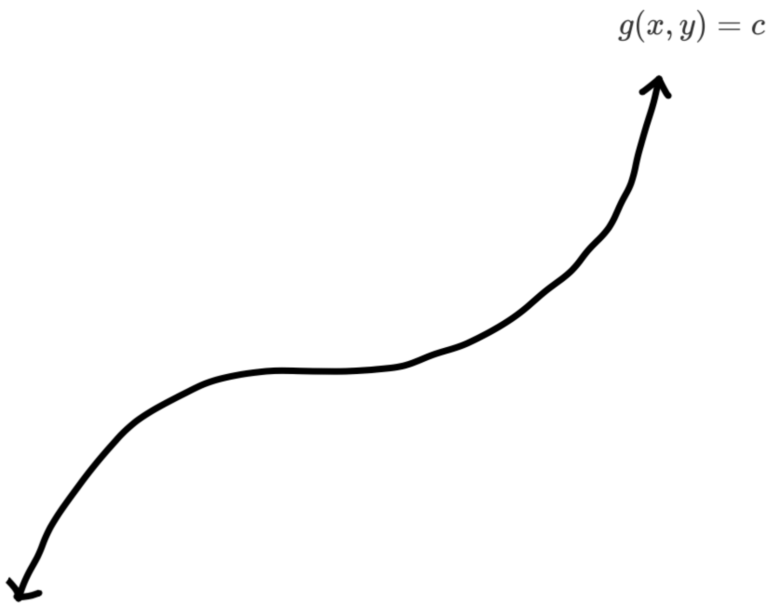
Lagrange Multipliers

In order to solve the problem we've posed above, we need to talk about the method of **Lagrange multipliers**. Specifically, we can pose the following problem:

$$\max_{x,y,z} \; f(x,y,z)$$
$$\text{s.t.} \; g(x,y,z) = c.$$

- In words, maximize (or minimize!) $f(x, y, z)$ **subject to the constraint** $g(x, y, z) = c$, where c is a constant.

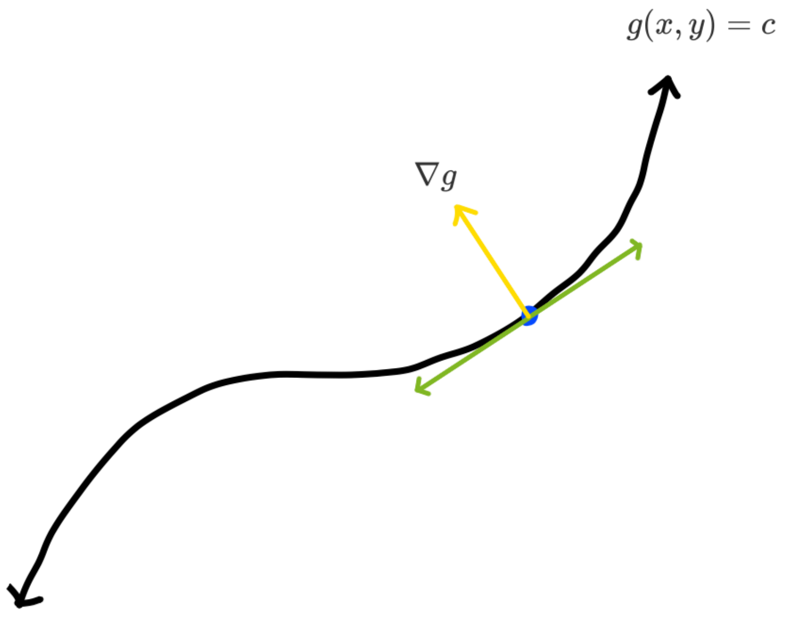
If we didn't have the constraint, we'd already know how to maximize f; that is, to find f's critical points we would calculate the **gradient** of f and calculate where it equals zero. Unfortunately, with the constraint g the prcoess isn't so simple. To come up with a solution, we'll consider this example in 2D.



In the plain optimization case, when we're just trying to maximize/minimize f without constraints, we can "walk" in any direction in the xy-plane we want in search of these extrema. Since the gradient of f always points in the direction of greatest increase, we simply follow it until it is $\vec{0}$; in other words, until we reach a point where f is decreasing in all directions.

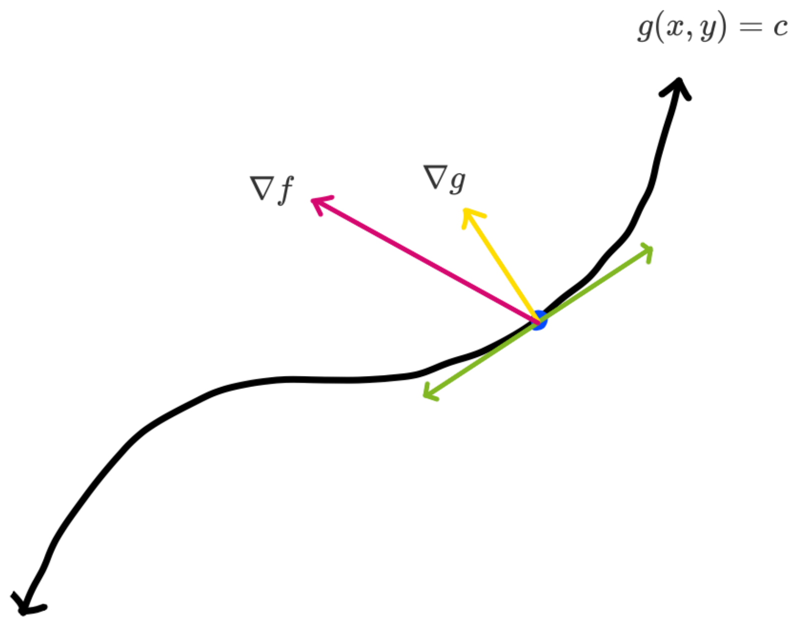
Returning to constrained optimization, we can think of g (pictured above) as a "path" that we are constrained to walk on, and we want to find the maximum value of f along this path.

First, we note that the gradient of g is always perpendicular to the path $g(x, y) = c$. A well-known property of the gradient is that it is always perpendicular to the level curves of its function. Intuitively, you can think of it like this: As you walk along the path of g, its value is not changing (after all, it's equal to c). Therefore, it must be perpendicular to the gradient, which points in the direction of greatest increase. This can be easily visualized:

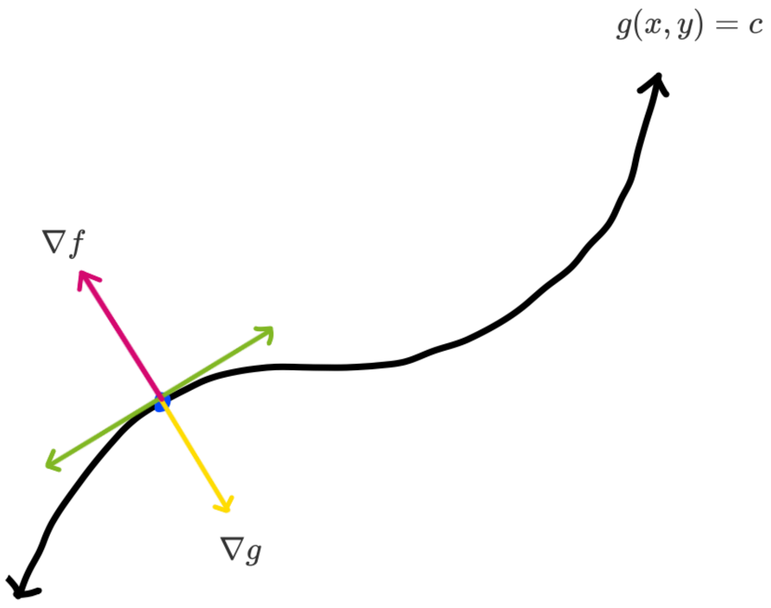


At the (arbitrary) point in blue, the only two direction we can walk in to stay on g are **tangent** to g. These directions are perpendicular to ∇g.

The final piece of the puzzle is to look at what is happening to ∇f as we walk along g. Consider the following picture:



∇f is pointing in the direction of greatest increase in f, but we are constrained to walk along g. Is it possible to increase f is some direction? The answer is yes; from the blue point, we can walk left. Although this isn't in the direction of **greatest** increase, since ∇f has a component in this direction we **will still be increasing** f. Now consider the following:



At this point, the gradient of f **has no component along either direction we can walk**. In other words, ∇f and ∇g are **parallel**. Since we cannot increase f in either direction, we must have found a local maximum. This is the key intuition behind Lagrange multipliers, which allows us to find solutions to constrained optimization problems by solving:

$$\begin{aligned} \nabla f(x, y) &= \alpha \nabla g(x, y) \\ g(x, y) &= c \end{aligned}$$

The first equation captures our intuition above that the gradients of f and g are parallel. α is called a **Lagrange multiplier**.

It is important to note that the method of Lagrange multipliers usually yields multiple solutions. These solutions must be verified by plugging them back into the original equation of f and determining which one is the true maximum or minimum.

The Lagrangian

An alternative form of the method of Lagrange multiples is as follows. We define a new function called **the Lagrangian**:

$$\mathcal{L}(x, y, \alpha) = f(x, y) - \alpha g(x, y).$$

Note that while $f, g \in \mathbb{R}^2$, $\mathcal{L} \in \mathbb{R}^3$. Intuitively, this approach is solving the constrained optimization problem by moving to a higher dimensional space. The constrained local extrema of f become the critical points of $\mathcal{L}$. To solve for them, we simply take the gradient of $\mathcal{L}$ and set it to 0:

$$\nabla \mathcal{L}(x, y, \alpha) = \nabla f(x, y) - \alpha \nabla g(x, y) = 0$$

$$\begin{cases} \nabla f(x, y) = \alpha \nabla g(x, y) \\ g(x, y) = 0 \end{cases}$$

- The first equation arises from taking the partial derivatives of $\mathcal{L}$ with respect to x and y, which are the first and second components of $\mathcal{L}$ respectively.
- The second equation arises from taking the partial derivative of $\mathcal{L}$ with respect to β.

Multiple Constraints

When there are multiple constraints, we no longer seek points where the gradient of f is parallel to the gradients of one of the constraints, but rather we seek points where the gradient of f is a **linear combination** of the gradients of all the constraints. Mathematically, we now have to solve:

$$\begin{aligned} \nabla f &= \sum_{i=1}^{n} \alpha_i \nabla g_i \\ g_1 &= c_1 \\ g_2 &= c_2 \\ &\dots \\ g_n &= c_n \end{aligned}$$

And our Lagrangian becomes:

$$\mathcal{L} = f - \sum_{i=1}^{n} \alpha_i g_i$$

Lagrange Duality

Consider the following, which we'll call the **primal** optimization problem:

$$\begin{aligned} \min_{x,y} \;\; & f(\vec{x}) \\ \text{s.t.} \;\; & g_i(\vec{x}) \le 0, \; i = 1, \dots, m. \end{aligned}$$

- Here, $\vec{x}$ is simply a vector of input variables $x_1, x_2, \dots x_n$.

Note the use of **inequalities** in the constraints. To solve it, we start by defining the Lagrangian:

$$\mathcal{L}(\vec{x}, \vec{\alpha}) = f(\vec{x}) + \sum_{i=1}^{k} \alpha_i g_i(\vec{x})$$

The Primal Problem

Now, consider the quantity, where $\mathcal{P}$ denotes **primal**:

$$\theta_{\mathcal{P}}(\vec{x}) = \max_{\vec{\alpha}} \mathcal{L}(\vec{x}, \vec{\alpha}).$$

Expanding, we have:

$$\theta_{\mathcal{P}}(\vec{x}) = \max_{\vec{\alpha}} \left(f(\vec{x}) + \sum_{i=1}^{k} \alpha_i g_i(\vec{x}) \right).$$

Now, assume we are given an input vector $\vec{x}_0$. If $\vec{x}_0$ does not satisfy **some** constraint, i.e. we have $g_i(\vec{x}) > 0$ for some i, then we could make α_i as arbitrarily large as we want to blow up the value of $\theta_{\mathcal{P}}(\vec{x})$ to infinity. Therefore, we can say the following about our primal problem:

$$\theta_{\mathcal{P}}(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if } \vec{x} \text{ satisfies primal constraints,} \\ \infty & \text{otherwise.} \end{cases}$$

What this means is that the primal problem is **exactly equal to the objective function** we are trying to minimize **when $\vec{x}$ satisfies the given constraints**. Therefore, we can reformulate our original primal optimization problem as:

$$\min_{\vec{x}} \theta_{\mathcal{P}}(\vec{x}) = \min_{\vec{x}} \max_{\vec{\alpha}} \mathcal{L}(\vec{x}, \vec{\alpha}) = p^*$$

Here, for later purposes, we denote the solution to this problem as p^*.

The Dual Problem

We now consider a slightly different quantity, known as the **dual optimization problem**. Although it does not have an intuitive interpretation in the context of the original problem, we will later see this it is related mathematically to the primal problem, and that under certain conditions the solution of the dual optimization problem is equal to the primal optimization problem.

- For now, you can sort of think of it like this:

$$\begin{aligned} \frac{e}{e+2} &= \frac{e+2-2}{e+2} \\ &= 1 - \frac{1}{e+2} \end{aligned}$$

    - Here, we added and subtracted two to the numerator in order to convert the expression to a different (and correct) form. What is the intuition behind adding and subtracting two? There isn't one really; **it's just a mathematical trick that gets the job done**.

Back to optimization, consider the quantity where $\mathcal{D}$ denotes **dual**:

$$\theta_{\mathcal{D}}(\vec{x}) = \min_{\vec{x}} \mathcal{L}(\vec{x}, \vec{\alpha})$$

We can now pose the dual optimization problem:

$$\max_{\vec{\alpha}} \theta_{\mathcal{D}}(\vec{x}) = \max_{\vec{\alpha}} \min_{\vec{x}} \mathcal{L}(\vec{x}, \vec{\alpha}) = d^*$$

As above, we denote the solution to this problem as d^*. Most notably, this equation is the exact same as the primal optimization problem, but the order of the max and min is switched. So what is the relationship between the two? We need one more thereom to see:

Proposition: $\max_x \min_y f(x, y) \leq \min_y \max_x f(x, y)$.

Proof:

Since this proof is rather difficult to think of intuitively, annotations have been provided at each step.

1. Let $g(x) = \min_y f(x, y)$.
2. $g(x) \leq f(x, y) \; \forall \; x, y$.
   - This follows from how we defined $g(x)$.
3. $\max_x g(x) \leq \max_x f(x, y) \; \forall \; y$.
   - We already know that $g(x) \leq f(x, y)$ everywhere. So, even if we maximize $g(x)$, it will of course still be less than or equal to the maximum of $f(x, y)$ over x.
4. $\max_x \min_y f(x, y) \leq \max_x f(x, y) \; \forall \; y$.
   - Substitute the value of $g(x)$ back into the inequality.
5. $\max_x \min_y f(x, y) \leq \min_y \max_x f(x, y)$.
   - Again, we already know that $\max_x \min_y f(x, y) \leq \max_x f(x, y)$ **for all** y. So, even at the value of y that minimizes the quantity on the right-hand side, the inequality still holds. QED.

This is known as the **min-max inequality** and allows us to arrive at the astonishingly simple conclusion:

$$d^* \leq p^* .$$

There are a set of conditions that govern when $d^* = p^*$, but proving them is outside the scope of these notes. Most importantly, these conditions basically say that there exists $\vec{x}^*, \vec{a}^*$ such that $p^* = d^*$. Moreover, $\vec{x}^*, \vec{a}^*$ satisfy the **Karush-Kuhn-Tucker (KKT) conditions**, which are as follows:

$$\begin{aligned} \frac{\partial}{\partial x_i} \mathcal{L}(\vec{x}^*, \vec{a}^*) &= 0, \; i = 1, \ldots, n \\ a_i^* g_i(\vec{x}^*) &= 0, \; i = 1, \ldots, m \\ g_i(\vec{x}^*) &\leq 0, \; i = 1, \ldots, m \\ a_i^* &\geq 0, \; i = 1, \ldots, m \end{aligned}$$

The final three conditions give rise to an important insight that is key for SVMs: when $a_i^* > 0$, $g_i(\vec{x}^*) = 0$. The constraint g_i is **active**, meaning it holds by equality rather than by inequality. This property is also called the **KKT dual complimentarity condition**.

Back to the Optimal Margin Classifier

Previously, we derived the following (**primal**) optimization problem for finding the optimal margin classifier:

$$\begin{aligned} \min_{\vec{\theta},b} \quad & \frac{1}{2}||\vec{\theta}||^2 \\ \text{s.t.} \quad & y_i(\vec{\theta}^T\vec{x}_i + b) \geq 1,\ i = 1, \ldots, m. \end{aligned}$$

Let's rewrite the constraints so that they are of the form "≤ 0". This is simple enough:

$$\begin{aligned} y_i(\vec{\theta}^T\vec{x}_i + b) - 1 \geq 0 \\ g_i(\vec{\theta}) = -y_i(\vec{\theta}^T\vec{x}_i + b) + 1 \leq 0. \end{aligned}$$

Note that every training sample in our data set has an associated constraint. Now, we finally have all the tools we need to solve this equation. We begin by forming the Lagrangian:

$$\mathcal{L}(\vec{\theta}, b, \vec{\alpha}) = \frac{1}{2}||\vec{\theta}||^2 - \sum_{i=1}^{m} \alpha_i(y_i(\vec{\theta}^T\vec{x}_i + b) - 1)$$

We could just solve this and call the problem complete, but here is why we discussed Lagrangian duality: it turns out that **the dual optimization problem will allow us to kernelize the SVM algorithm.**

So let's find the dual form of the problem. To do so, we need to first minimize $\mathcal{L}$ with respect to $\vec{\theta}$ and b to get $\theta_{\mathcal{D}}$. We can do this by setting the partial derivatives of $\mathcal{L}$ with respect to $\vec{\theta}$ and b to zero.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \vec{\theta}} = \vec{\theta} - \sum_{i=1}^{m} \alpha_i y_i \overrightarrow{x_i} = 0 \\ \vec{\theta} = \sum_{i=1}^{m} \alpha_i y_i \overrightarrow{x_i}. \end{aligned}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^{m} \alpha_i y_i = 0$$

If we take the definition of $\vec{\theta}$ we found and plug it back into the Lagrangian, we get:

$$\begin{aligned}\mathcal{L}(b,\vec{\alpha}) &= \frac{1}{2}\Big\|\sum_{i=1}^{m}\alpha_i y_i \overrightarrow{x_i}\Big\|^2 - \sum_{i=1}^{m}\alpha_i\left(y_i\left(\left(\sum_{j=1}^{m}\alpha_j y_j \overrightarrow{x_j}\right)^T \vec{x}_i + b\right) - 1\right) \\ &= \frac{1}{2}\left(\sum_{i=1}^{m}\alpha_i y_i \overrightarrow{x_i}\right)^T\left(\sum_{j=1}^{m}\alpha_j y_j \overrightarrow{x_j}\right) - \sum_{i=1}^{m}\left(\alpha_i y_i\left(\sum_{j=1}^{m}\alpha_j y_j \overrightarrow{x_j}\right)^T \vec{x}_i + \alpha_i y_i b - \alpha_i\right) \\ &= \frac{1}{2}\left(\sum_{i=1}^{m}\alpha_i y_i \overrightarrow{x_i}\right)^T\left(\sum_{j=1}^{m}\alpha_j y_j \overrightarrow{x_j}\right) - \left(\sum_{i=1}^{m}\alpha_i y_i \overrightarrow{x_i}\right)^T\left(\sum_{j=1}^{m}\alpha_j y_j \overrightarrow{x_j}\right) - \sum_{i=1}\alpha_i y_i b + \sum_{i=1}\alpha_i \\ &= \sum_{i=1}\alpha_i - \frac{1}{2}\left(\sum_{i=1}^{m}\alpha_i y_i \overrightarrow{x_i}\right)^T\left(\sum_{j=1}^{m}\alpha_j y_j \overrightarrow{x_j}\right) - b\sum_{i=1}\alpha_i y_i \\ &= \sum_{i=1}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \vec{x}_i^T\vec{x}_j - b\sum_{i=1}\alpha_i y_i\end{aligned}$$

We know from the calculation of $\frac{\partial \mathcal{L}}{\partial b}$ however, that the last term must be 0. Therefore, we finally obtain:

$$\mathcal{L}(\vec{\alpha}) = \sum_{i=1}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \overrightarrow{x_i}^T\overrightarrow{x_j}$$

Putting everything together, we arrive at the problem:

$$\begin{aligned}\max_{\vec{\alpha}}\ \ & \sum_{i=1}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \overrightarrow{x_i}^T\overrightarrow{x_j} \\ \text{s.t.}\ \ & \alpha_i \geq 0,\ i = 1,\ldots,m \\ & \sum_{i=1}^{m}\alpha_i y_i = 0.\end{aligned}$$

- The last constraint comes from the calculation of $\frac{\partial \mathcal{L}}{\partial b}$. Together with the first constraint, it ensures that the KKT conditions are satisfied so that the answer to this dual problem is equal to the answer to the primal problem ($d^* = p^*$).
- During this process, we found a solution for $\vec{\theta}^*$, namely:

  $$\vec{\theta}^* = \sum_{i=1}^{m}\alpha_i y_i \overrightarrow{x_i}$$

  The optimal value for b can also be calculated, and is given by:

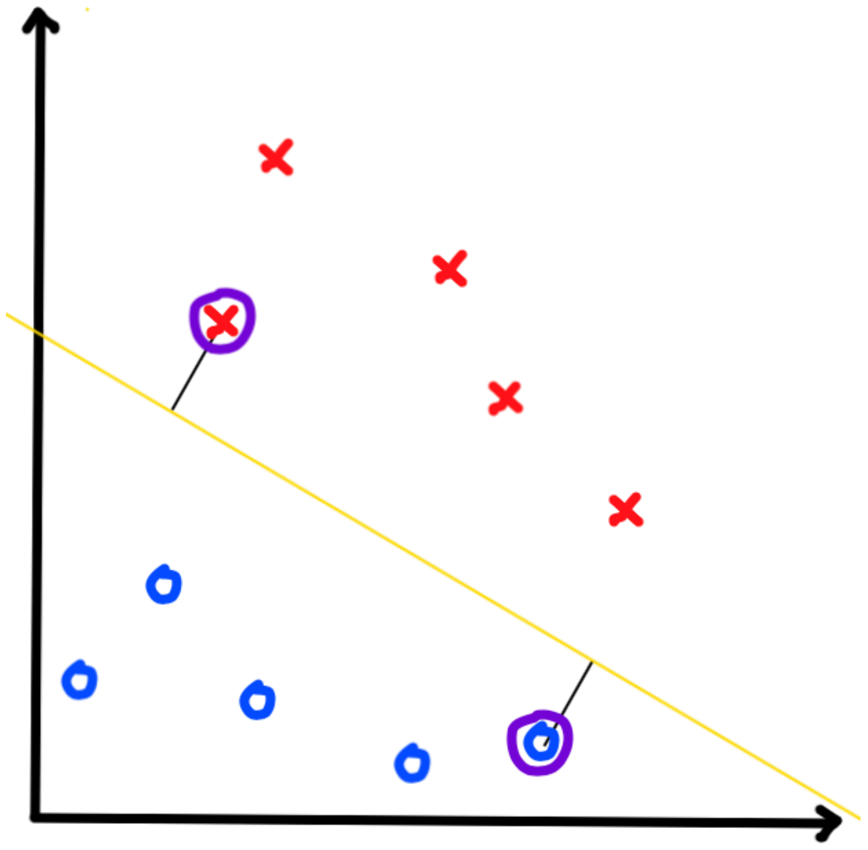
  $$b^* = -\frac{\max_{i:y_i=-1}\vec{\theta}^{*T}x_i + \min_{i:y_i=1}\vec{\theta}^{*T}x_i}{2}$$

  Therefore, given $\vec{a}$, we can compute $\vec{\theta}^*$ and b^*.

Support Vectors

Through all these complex computations, we seem to have gotten very far from our simple goal of maximizing the distance from the decision boundary to all of the points. Where does $\vec{\alpha}$ fit into all of this? Recall from the KKT dual complimentaritY condition that we will have $\alpha_i > 0$ **only** for the training examples that have a margin equal to $\frac{\hat{\gamma}}{||\vec{\theta}||}$ (all other training examples have a margin greater than this quantity).

We can visualize this:



In this picture, the maximum margin separating hyperplane is shown by the yellow line. The points with the smallest margins (circled in purple) are exactly the ones closest to the boundary. Thus, only two of the α_i (i.e. the ones corresponding to these two training examples) will be **non-zero** at the optimal solution.

These points are called **support vectors** (yes, it took us this long to actually see why we call this algorithm the support vector machine!), since in a sense they are "supporting" the decision boundary. In fact, you can argue that the support vectors are the only points which truly define the decision boundary.

Kernelizing the SVM

Recall the optimal value of $\vec{\theta}^*$ that we found:

$$\vec{\theta}^* = \sum_{i=1}^{m} \alpha_i y_i \overrightarrow{x_i}$$

Suppose we've fit our models' parameters to a training set, and now wish to make a prediction at a new input point $\vec{x}$. We calculate:

$$\begin{aligned}\vec{\theta}^T \vec{x} + b &= \left(\sum_{i=1}^{m} \alpha_i y_i \overrightarrow{x_i} \right)^T \vec{x} + b \\ &= \left(\sum_{i=1}^{m} \alpha_i y_i \overrightarrow{x_i}^T \vec{x} \right) + b\end{aligned}$$

In order to make our prediction, **all we need is to compute are the dot products of the support vectors with** $\vec{x}$. Our algorithm is perfectly primed to use the kernel trick, which will allow SVM to efficiently learn in very high dimensional spaces.

TODO: Soft Margin SVM