Deep Learning for Computer Vision

Lecture 4: Curse of Dimensionality, High Dimensional Feature Spaces, Linear Classifiers, Linear Regression, Python, and Jupyter Notebooks

Peter Belhumeur

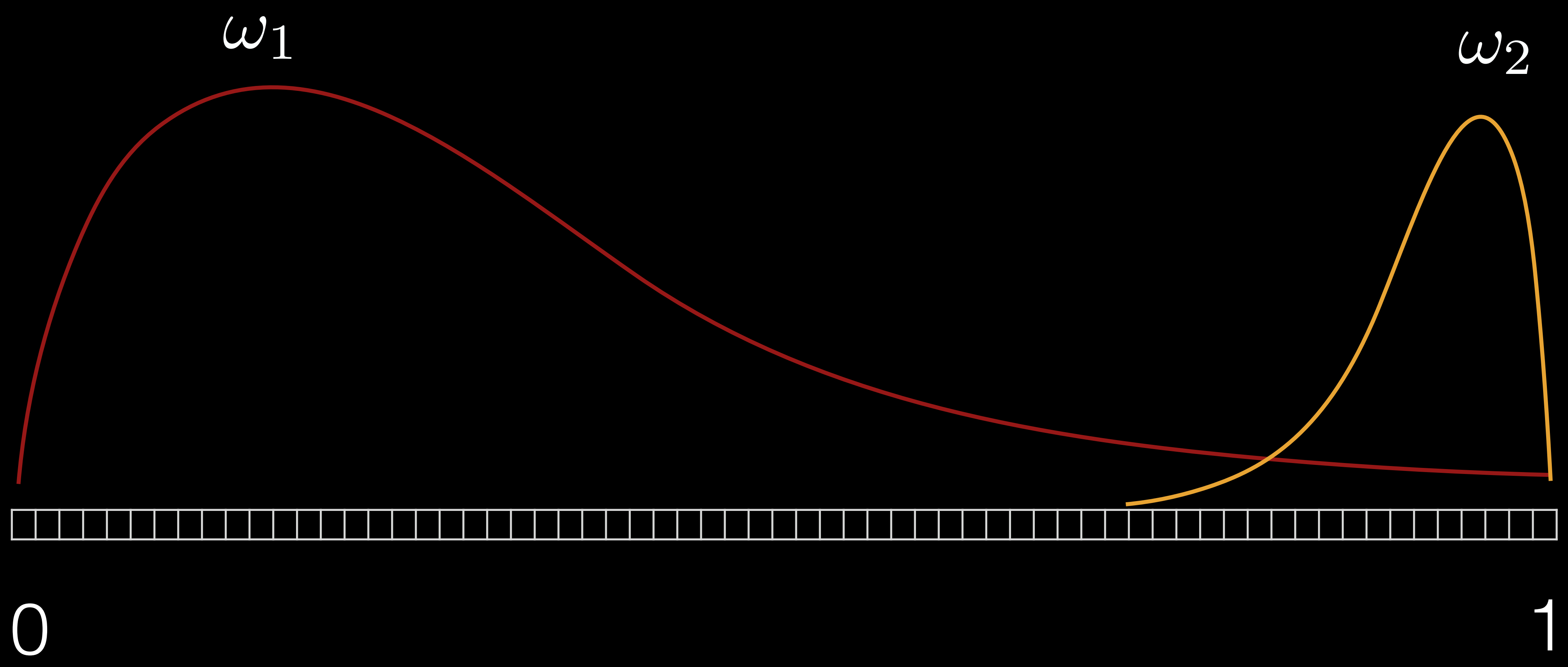
Computer Science
Columbia University

If we want to build a minimum-error rate classifier then we need a very good estimate of $P(\omega_i|\mathbf{x})$.

Let's say our feature space is just 1 dimensional and our feature $x \in [0, 1]$.

And let's say we have **10,000 training samples** from which to estimate our *a posteriori* probabilities.

We could estimate these probabilities using a histogram in which we divided the interval into 100 evenly spaced bins.



On average each bin would have 100 samples.

We could estimate $\rho(x|\omega_i)$ as the number of samples from class i that fall in the same bin that x falls into divided by the total number of samples.

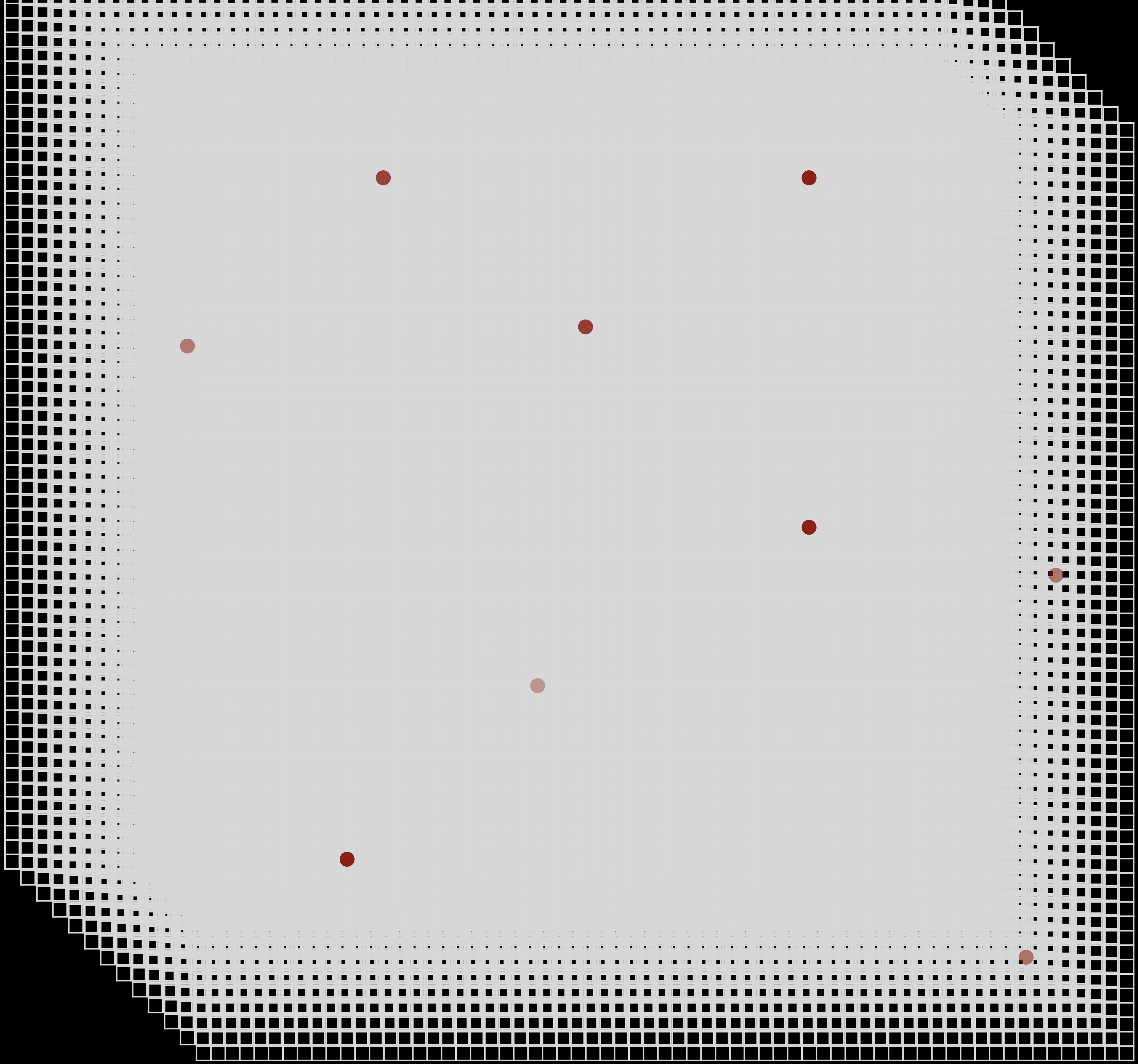
But this plan does not scale as we increase the dimensionality of the feature space!

The Curse of Dimensionality!

Let’s say our feature space is just 3 dimensional and our feature $\mathbf{x} \in [0,1]^3$.

Let’s say we still have **10,000 training samples** from which to estimate our *a posteriori* probabilities.

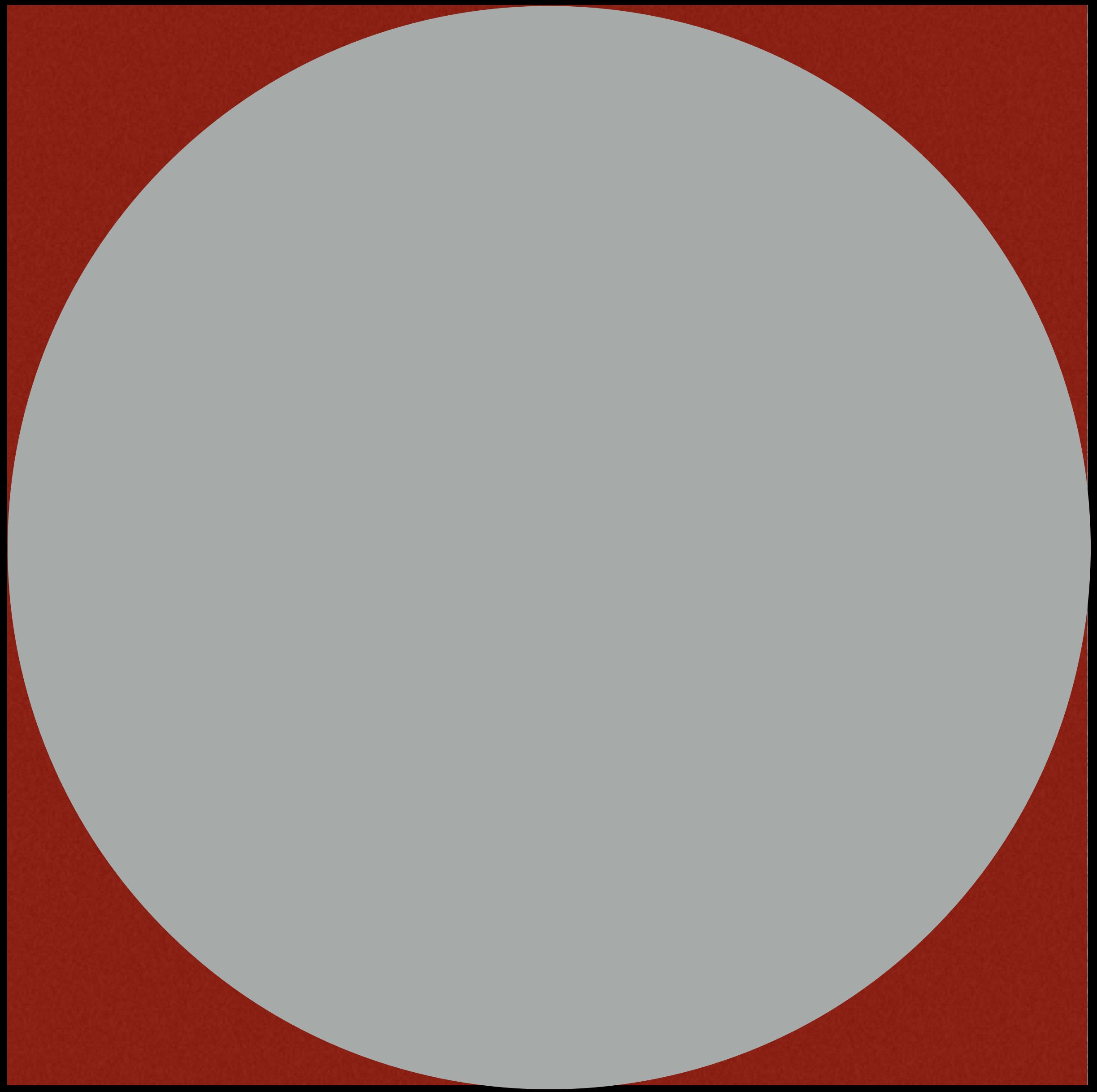
If we estimate these probabilities using a histogram in which we divided the volume into the same width bins as before…



On average each bin would only have 0.01 samples! Ugh.

So we never have enough training samples to densely sample high-dimensional feature spaces!

And to make matters worse, intuition about how things behave in high-dimensional spaces is mostly incorrect.



$$\frac{\pi}{4}$$

What fraction of the volume of the unit hypercube is occupied by the biggest hypersphere that fits within it in n-dimensions? Say if n=20?

Volume of n-d hypersphere $= V_n \, R^n$

$$V_n = \frac{\pi^{\frac{n}{2}}}{\Gamma(\frac{n}{2} + 1)}$$

So if n = 20, the volume $= \dfrac{\pi^{10}}{\Gamma(10+1)} \left(\dfrac{1}{2}\right)^{20}$

$$= \frac{\pi^{10}}{10!} \left(\frac{1}{2}\right)^{20}$$

$$= 0.000000246$$

Another way to think of this is … if we uniformly sampled **100 million** points in the unit hypercube, then we should expect less than 3 of them would lie within its contained hypersphere.

So it’s hard/impossible to accurately estimate joint probabilities for high-dimensional random variables. You just cannot get enough samples!

You can assume the joint density function, in our case the class conditional density functions, are given by a known density of some parametric form, *e.g.*, a multivariate Normal.

Bayes Theorem

$$P(\omega_i|x) = \frac{\rho(x|\omega_i)P(\omega_i)}{P(x)}$$

$$= \frac{\rho(x|\omega_i)P(\omega_i)}{\sum_i \rho(x|\omega_i)P(\omega_i)}$$

$$= \frac{likelihood \;\; \times \;\; prior}{evidence}$$

Bayes Theorem

$$P(\omega_i|x) = \frac{\boxed{\rho(x|\omega_i)}P(\omega_i)}{P(x)}$$

$$= \frac{\rho(x|\omega_i)P(\omega_i)}{\sum_i \rho(x|\omega_i)P(\omega_i)}$$

$$= \frac{likelihood \quad \times \quad prior}{evidence}$$

Let's assume the ccds are multivariate normals.

If $\mathbf{x} \sim N(\mu, \Sigma)$, then

$$\rho(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1}(\mathbf{x}-\mu)}$$

To find the ccds we just need to estimate μ and Σ from our samples $\mathbf{x}_i$ for each class, separately.

$$\mu = \frac{1}{N} \sum_{i}^{N} \mathbf{x}_i$$

$$\Sigma = \frac{1}{N-1} \sum_{i}^{N} (\mathbf{x}_i - \mu)(\mathbf{x}_i - \mu)^T$$

Bayes Theorem

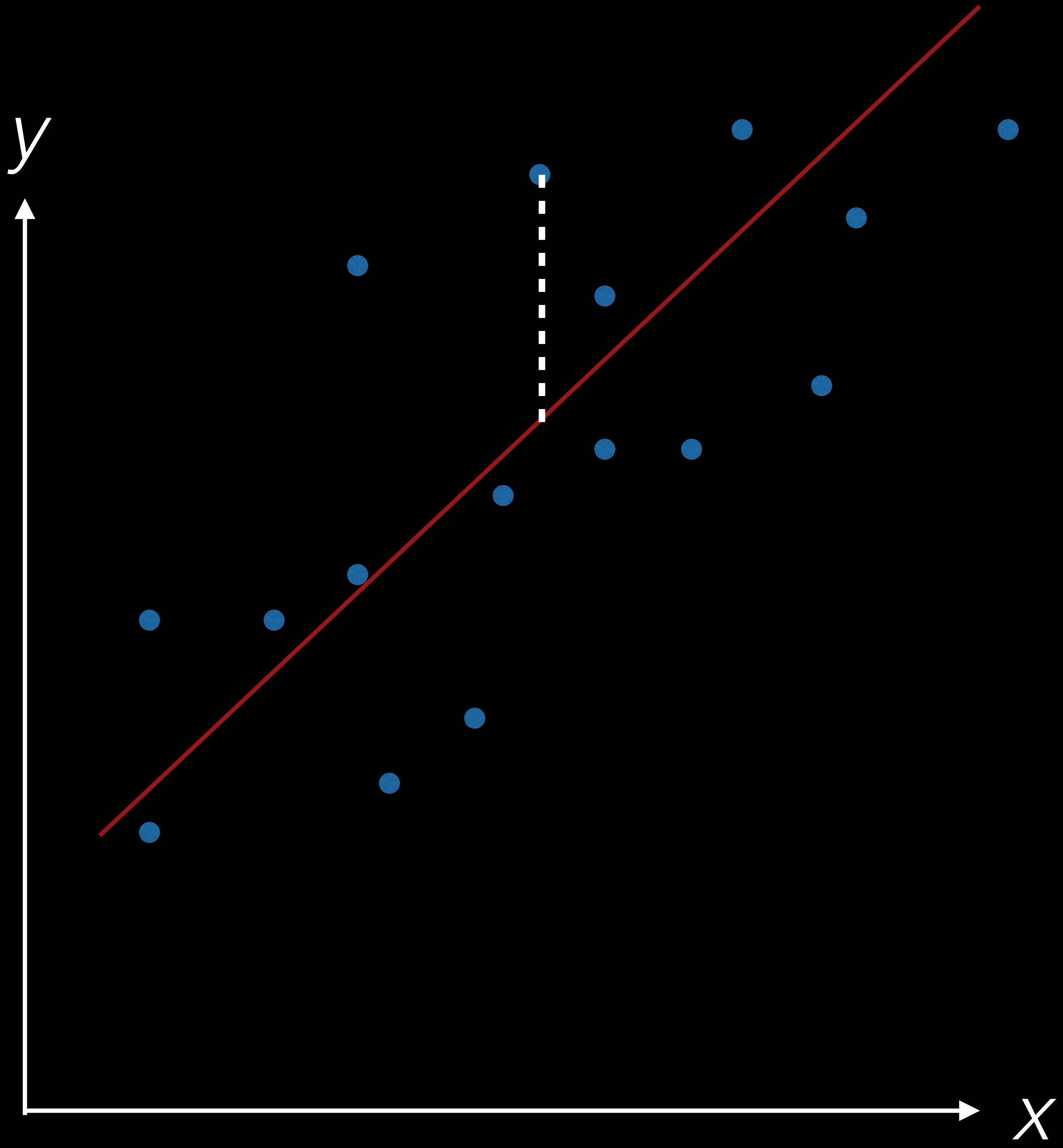
$$P(\omega_i|x) = \frac{\boxed{\rho(x|\omega_i)}P(\omega_i)}{P(x)}$$

$$= \frac{\rho(x|\omega_i)P(\omega_i)}{\sum_i \rho(x|\omega_i)P(\omega_i)}$$

$$= \frac{likelihood \quad \times \quad prior}{evidence}$$

One shortcut around this is “**Naive Bayes**.” Everything is the same as before, but now we assume that our features are independent and we can estimate the class conditionals for each separately. And to get the **joint** class conditional density function, we simply approximate this as the product of the per feature ccds—because of the independence assumption.

Linear Regression


y
x

Find the line that minimizes the sum of the squared vertical distances of the line to the sample points.

Let the line be parameterized by

$$\begin{aligned} y &= \theta_0 + \theta_1 x \\ &= \theta^T \mathbf{x} = h_\theta(\mathbf{x}) \end{aligned}$$

where $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$ and $\mathbf{x} = \begin{bmatrix} 1 \\ x \end{bmatrix}$

Now given sample points $(x^{(i)}, y^{(i)})$,

we can write our loss function to minimize as

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right)^2$$

and its gradient as

$$\nabla_\theta J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right) \mathbf{x}^{(i)}$$

This is a convex optimization problem so we can easily find the solution using gradient descent:

$$\theta(t+1) = \theta(t) - \alpha \nabla_\theta J(\theta(t))$$

Course Programming Language: Python

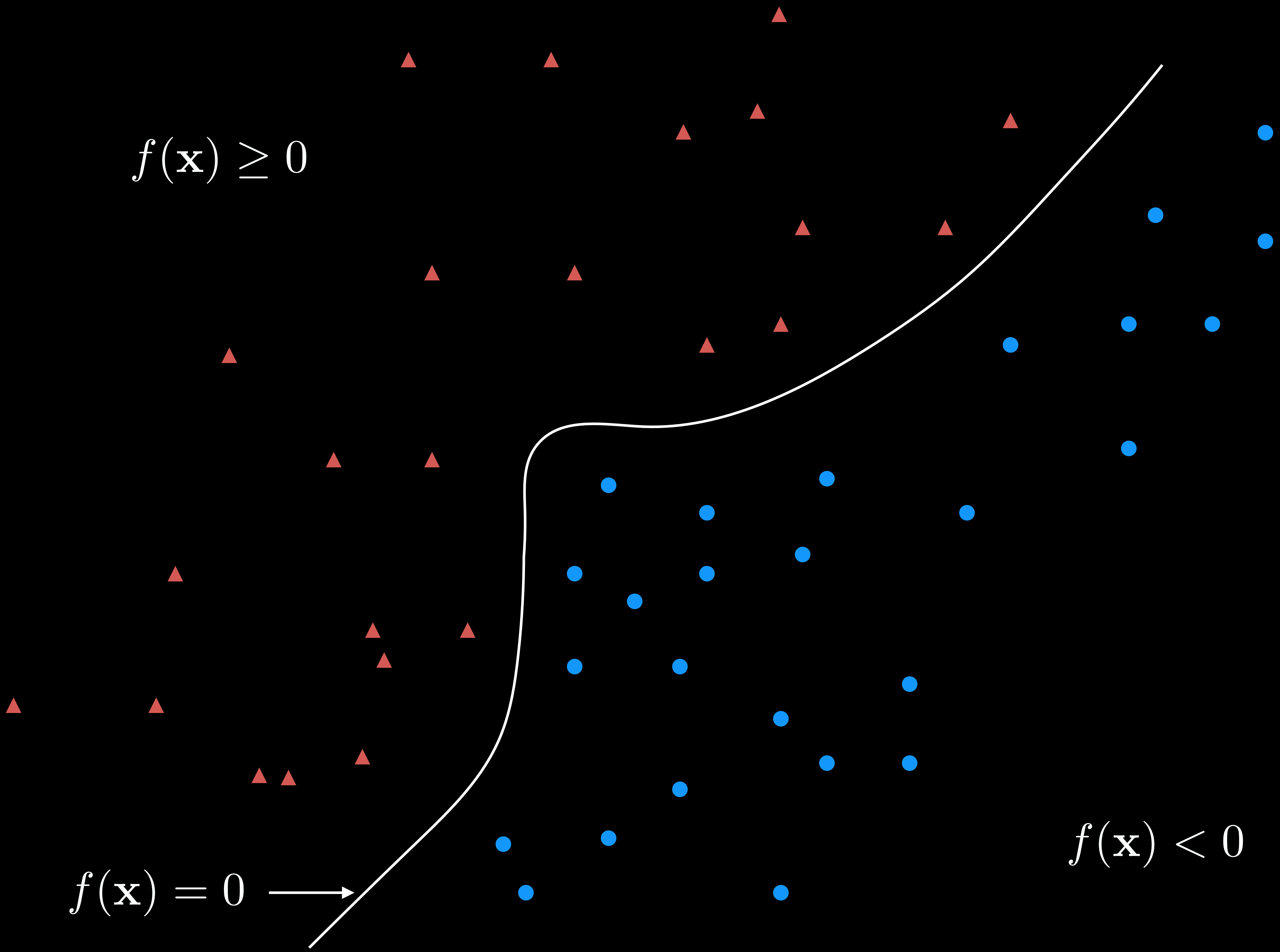
Course Coding Environment: Jupyter Notebooks

Binary Classifier

Let's say we have set of training samples $(\mathbf{x}_i, y_i)$ with $i = 1, \ldots, N$ and $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$.

The goal is to learn a classifier $f(\mathbf{x}_i)$ such that

$$f(\mathbf{x}_i) = \begin{matrix} \geq 0 & y_i = +1 \\ < 0 & y_i = -1 \end{matrix}$$


$f(\mathbf{x}) \geq 0$
$f(\mathbf{x}) < 0$
$f(\mathbf{x}) = 0$

But let’s find a simple parametric separating surface by fitting to the training data using some pre-chosen criterion of optimality.

If we restrict the decision surface to a line, plane, or hyperplane, then we call this a **linear classifier**.

Linear Classifier

Let’s say we have set of training samples $(\mathbf{x}_\mathrm{i}, y_\mathrm{i})$ with $\mathrm{i} = 1, \ldots, \mathrm{N}$ and $\mathbf{x}_\mathrm{i} \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$.

The goal is to learn a classifier $\mathrm{f}(\mathbf{x}_\mathrm{i})$ such that

$$f(\mathbf{x}_i) = \begin{matrix} \geq 0 & y_i = +1 \\ < 0 & y_i = -1 \end{matrix}$$

and where

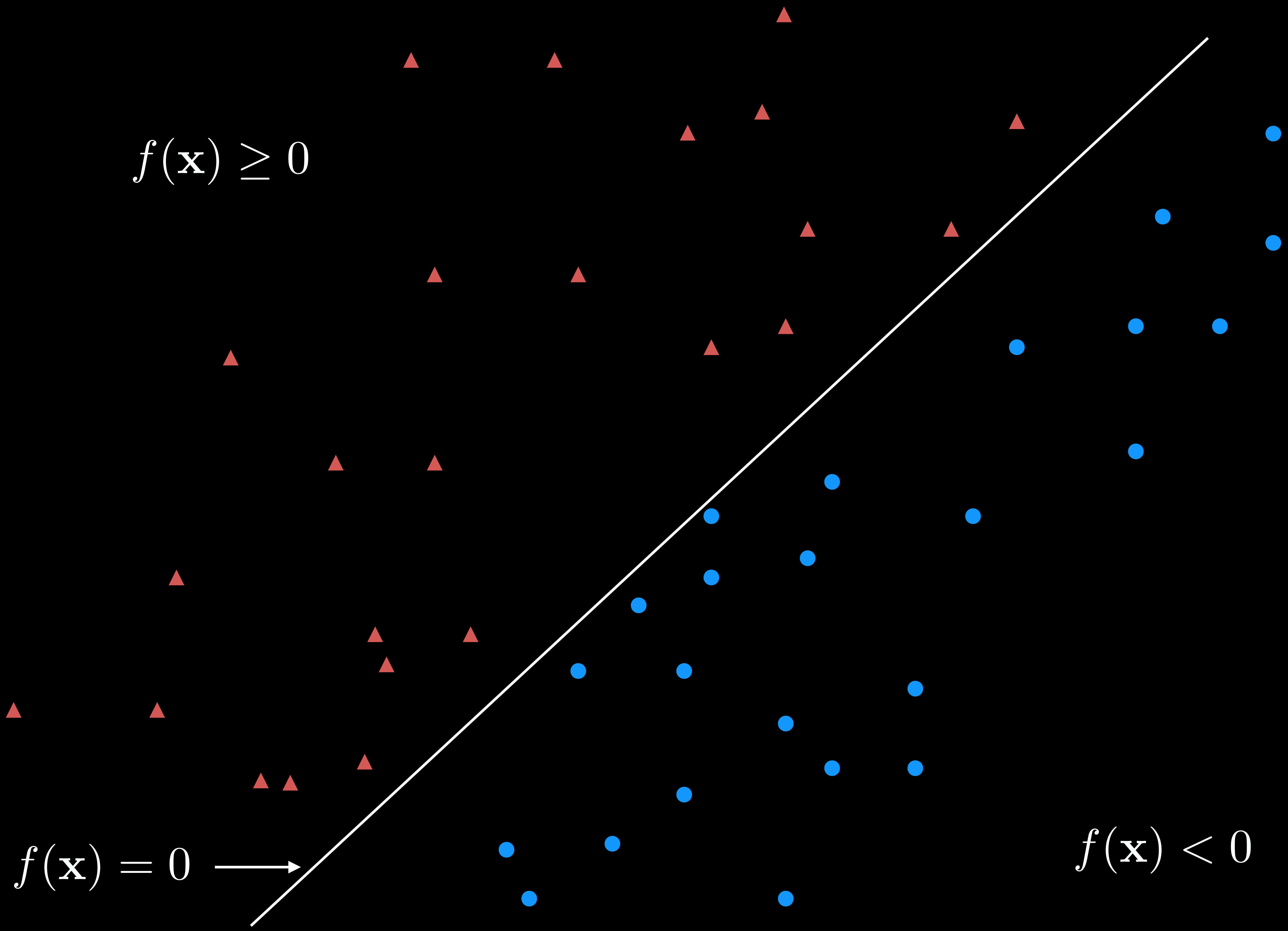
$$f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$$

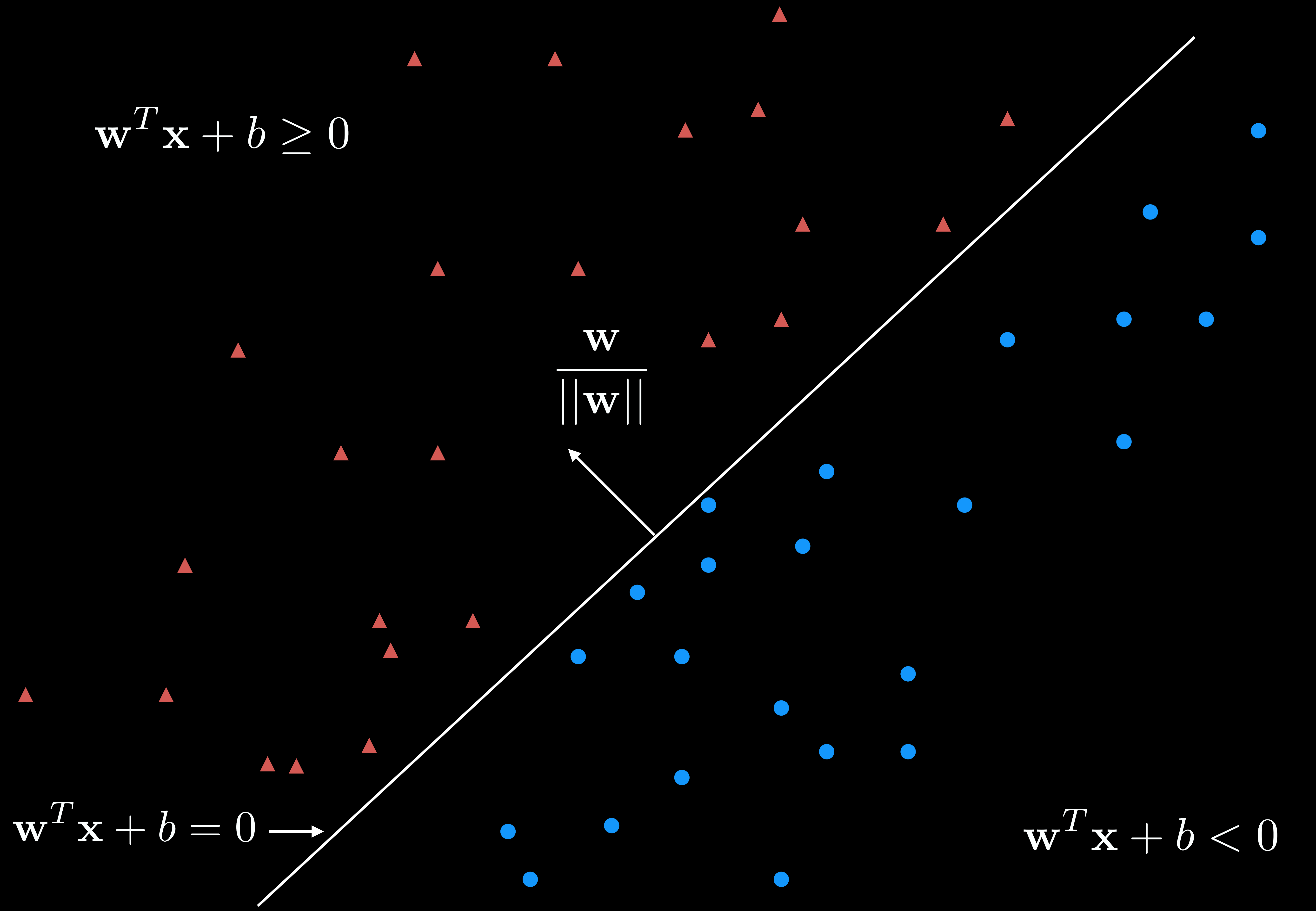
Linear Classifier

Note:

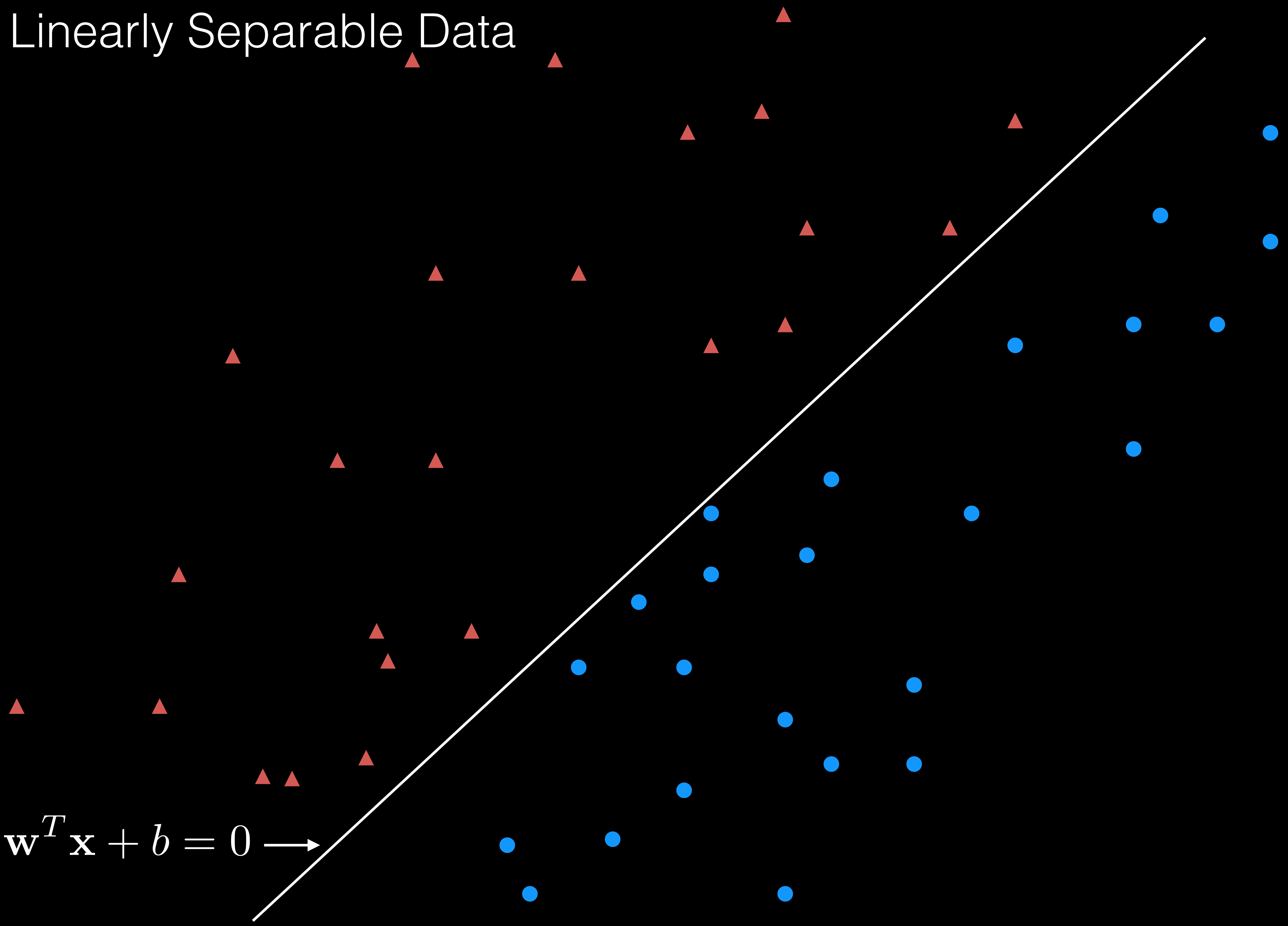
- The classifier is a line, plane, or hyperplane depending of the dimension d
- $\mathbf{w}$ is normal to the line
- b is known as the bias.

$$f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$$

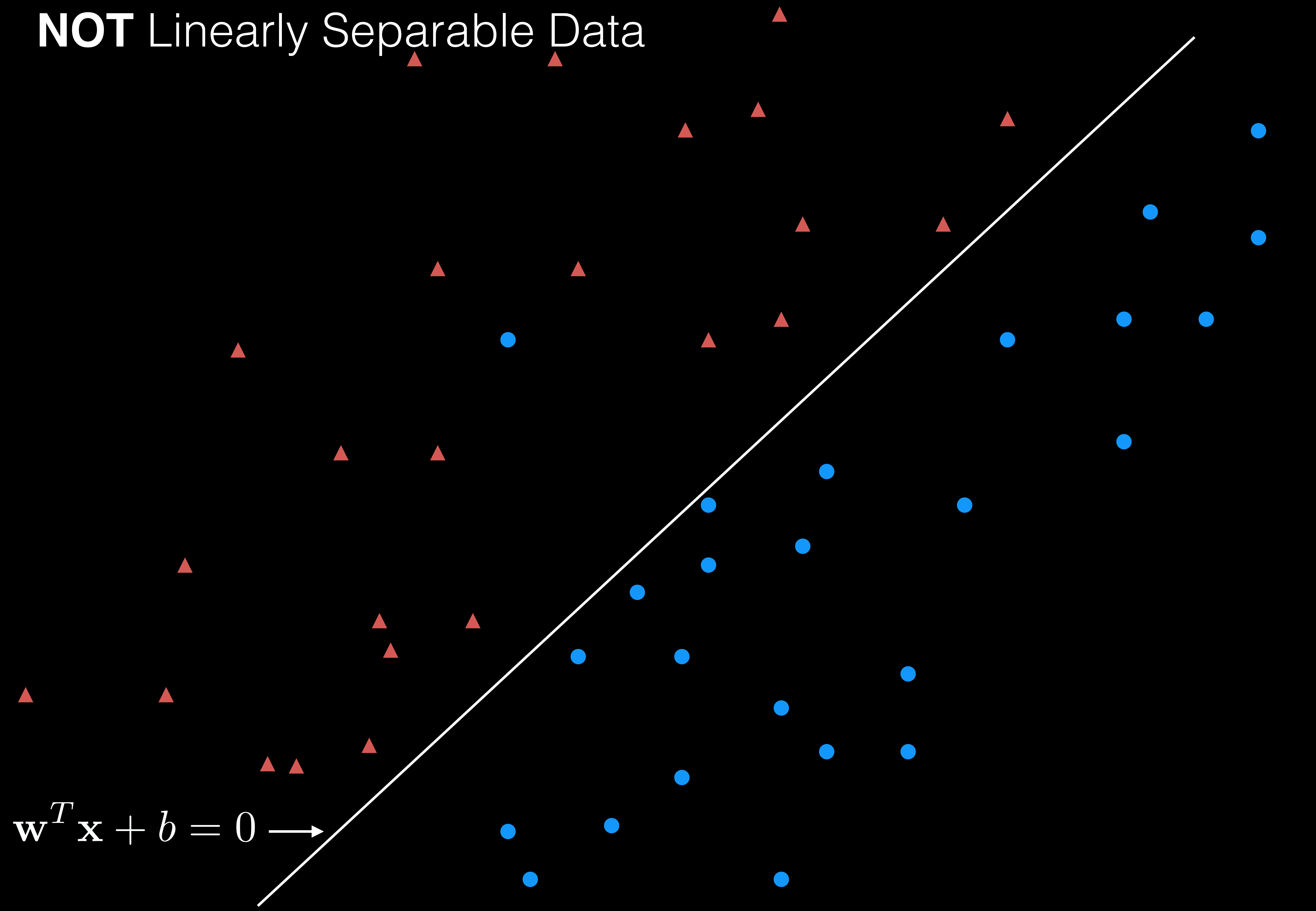

$f(\mathbf{x}) \geq 0$
$f(\mathbf{x}) = 0 \longrightarrow$
$f(\mathbf{x}) < 0$


$\mathbf{w}^T\mathbf{x} + b \geq 0$
$\frac{\mathbf{w}}{||\mathbf{w}||}$
$\mathbf{w}^T\mathbf{x} + b = 0 \longrightarrow$
$\mathbf{w}^T\mathbf{x} + b < 0$

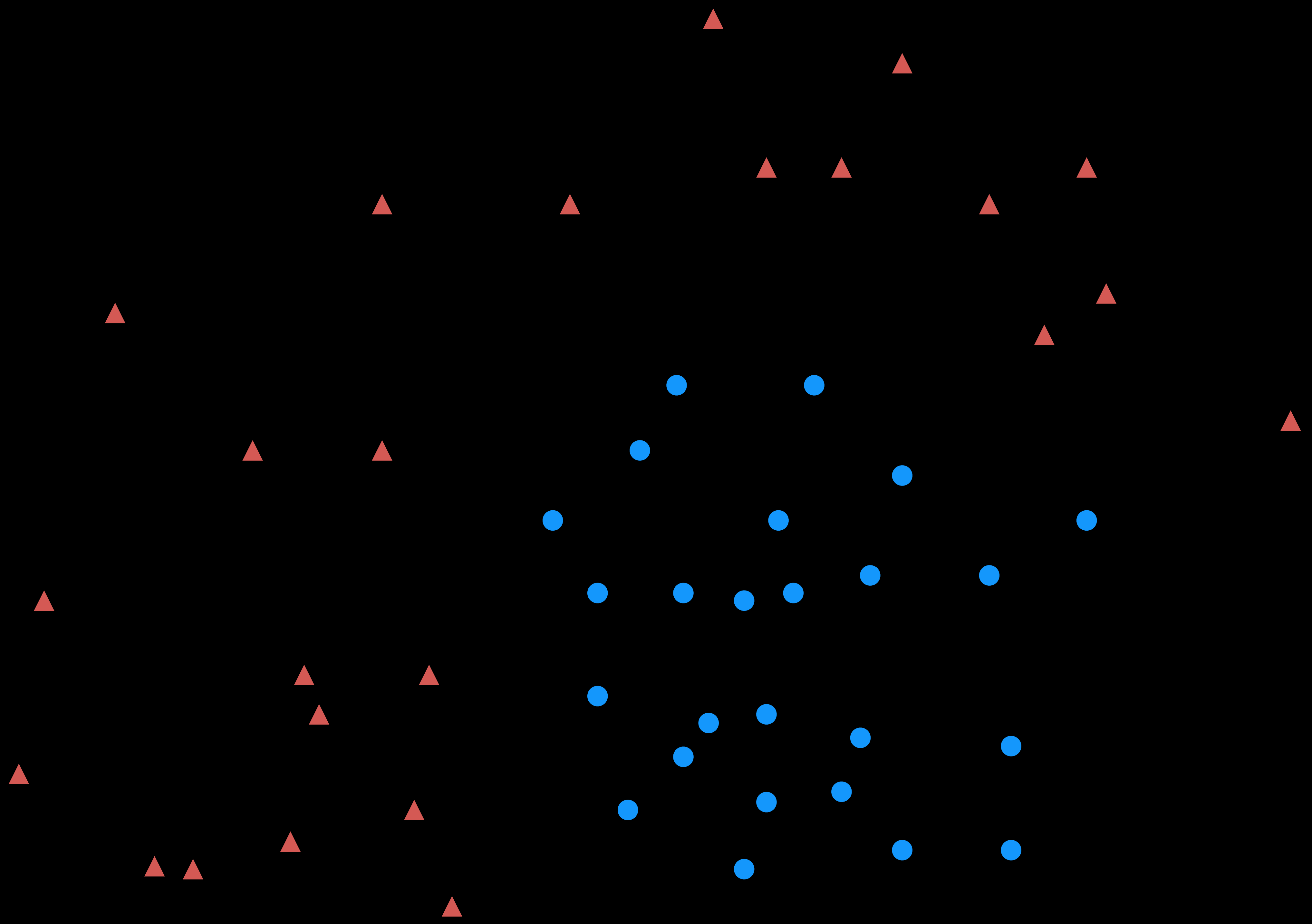
Linearly Separable Data



NOT Linearly Separable Data



NOT Linearly Separable Data



Linear Classifier

How do we find the *weights* given by $\mathbf{w}$ and b?

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$

We need to choose our weights according to some notion of optimality.

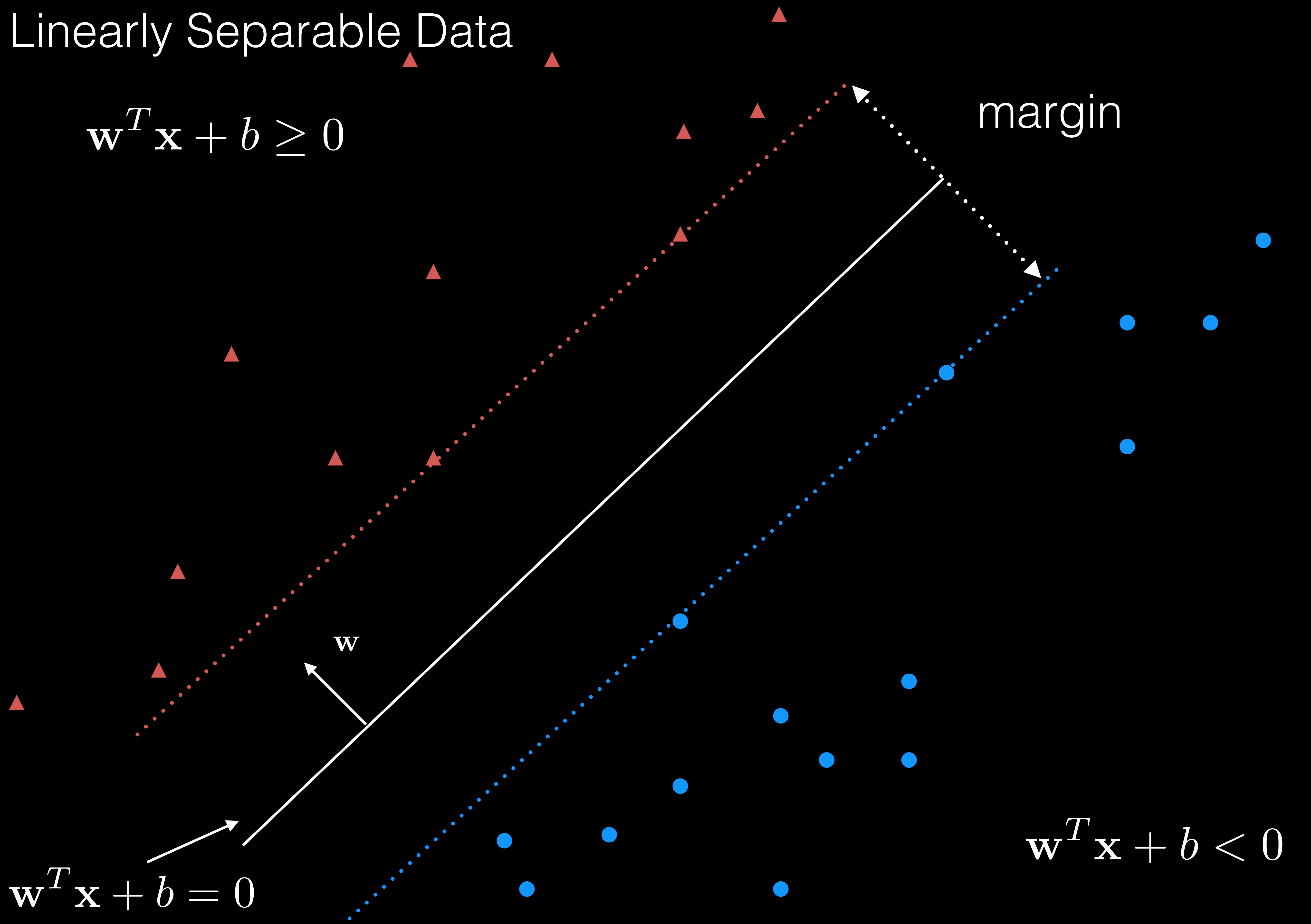
So what is the best $\mathbf{w}$?

Support Vector Machine

Choose *weights* $\mathbf{w}$ and b to maximize the margin between classes.

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$

Linearly Separable Data


$\mathbf{w}^T\mathbf{x} + b \geq 0$
margin
$\mathbf{w}$
$\mathbf{w}^T\mathbf{x} + b < 0$
$\mathbf{w}^T\mathbf{x} + b = 0$

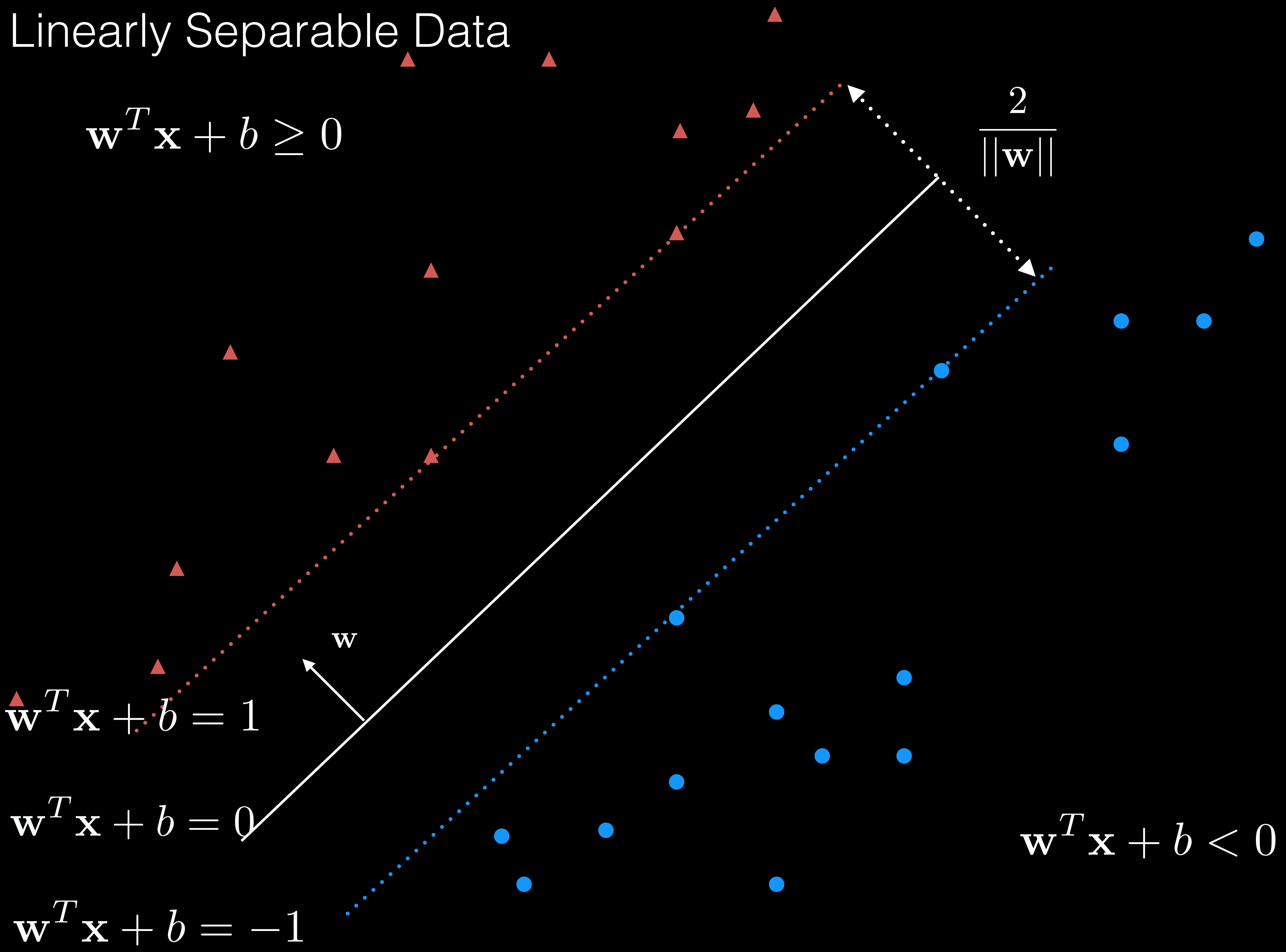
- We can choose the normalization of $\mathbf{w}$ however we please since this is a free parameter.

- Let's choose it so for the positive and negative *support vectors* so we have

$$\mathbf{w}^T\mathbf{x}_+ + b = +1$$
$$\mathbf{w}^T\mathbf{x}_- + b = -1$$

- This makes the *margin* width $\frac{2}{||\mathbf{w}||}$

Linearly Separable Data


$\mathbf{w}^T\mathbf{x} + b \geq 0$
$\frac{2}{||\mathbf{w}||}$
$\mathbf{w}$
$\mathbf{w}^T\mathbf{x} + b = 1$
$\mathbf{w}^T\mathbf{x} + b = 0$
$\mathbf{w}^T\mathbf{x} + b = -1$
$\mathbf{w}^T\mathbf{x} + b < 0$

- We can now set this choice of $\mathbf{w}$ up as an optimization:

$$\max_{\mathbf{w}} \frac{2}{||\mathbf{w}||} \text{ subject to } \begin{array}{l} \mathbf{w}^T\mathbf{x}_i + b \geq +1 \text{ if } y_i = +1 \\ \mathbf{w}^T\mathbf{x}_i + b \leq -1 \text{ if } y_i = -1 \end{array}$$

- Or equivalently,

$$\min_{\mathbf{w}} ||\mathbf{w}||^2 \text{ subject to } y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$$

- This is a quadratic optimization with linear constraints and a unique minimum.

What if the data is not linearly separable or has some bad apples?

Soft Margin and Slack Variables

We *soften* the optimization with “slack” variables:

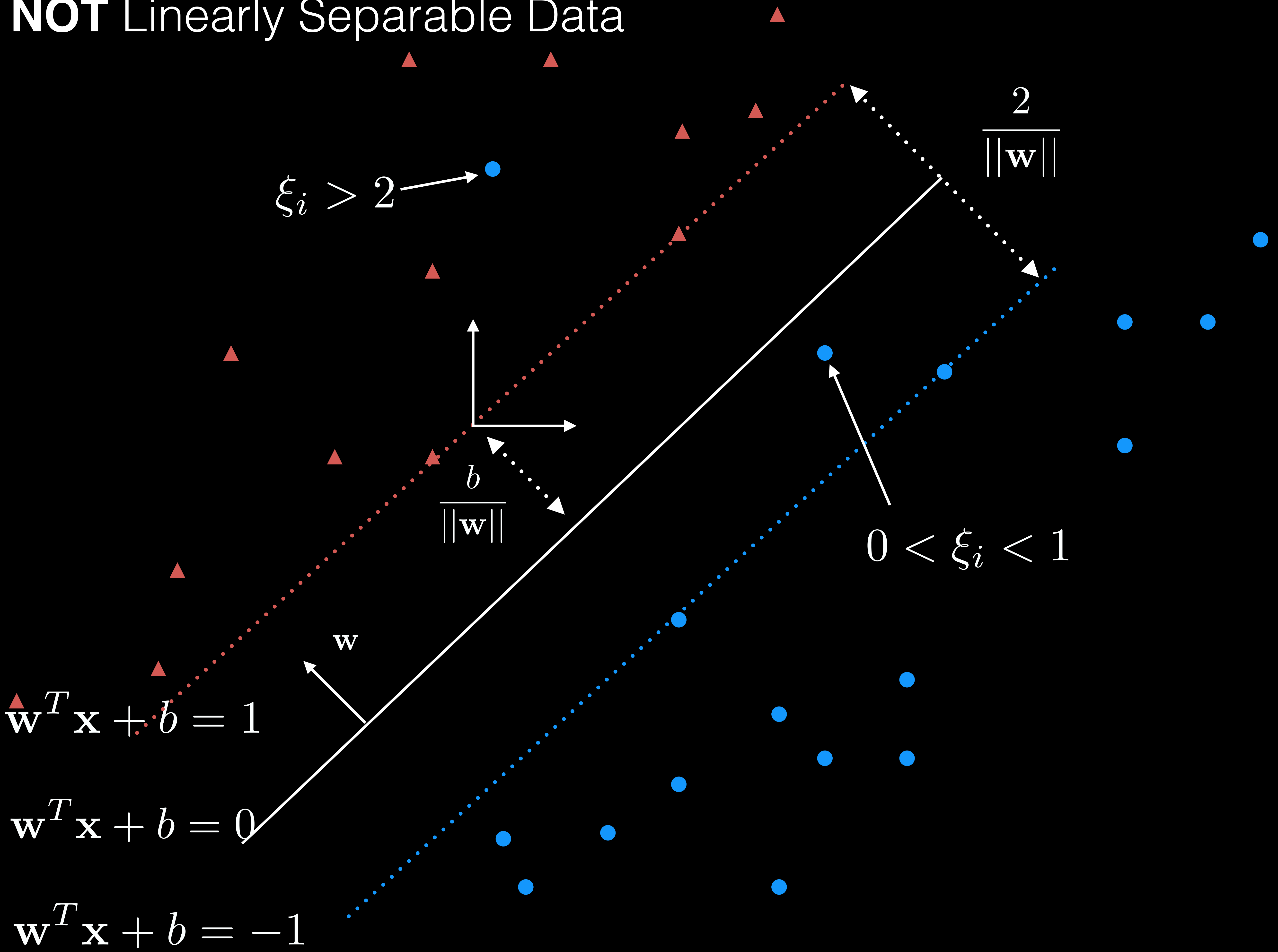
$$\min_{\mathbf{w}} ||\mathbf{w}||^2 + C \sum_{i}^{N} \xi_i$$

subject to

$$y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i \text{ and } \xi_i \geq 0$$

- If we make our slack variables large enough we can satisfy all of the constraints
- C is the regularization parameter
- Large C makes constraints hard to ignore = small margin
- Small C makes constraints easy to ignore = large margin
- Still a unique minimum

NOT Linearly Separable Data



$\xi_i > 2$
$\frac{2}{||\mathbf{w}||}$
$\frac{b}{||\mathbf{w}||}$
$0 < \xi_i < 1$
$\mathbf{w}$
$\mathbf{w}^T\mathbf{x} + b = 1$
$\mathbf{w}^T\mathbf{x} + b = 0$
$\mathbf{w}^T\mathbf{x} + b = -1$

Soft Margin and Slack Variables

And with a little manipulation we get

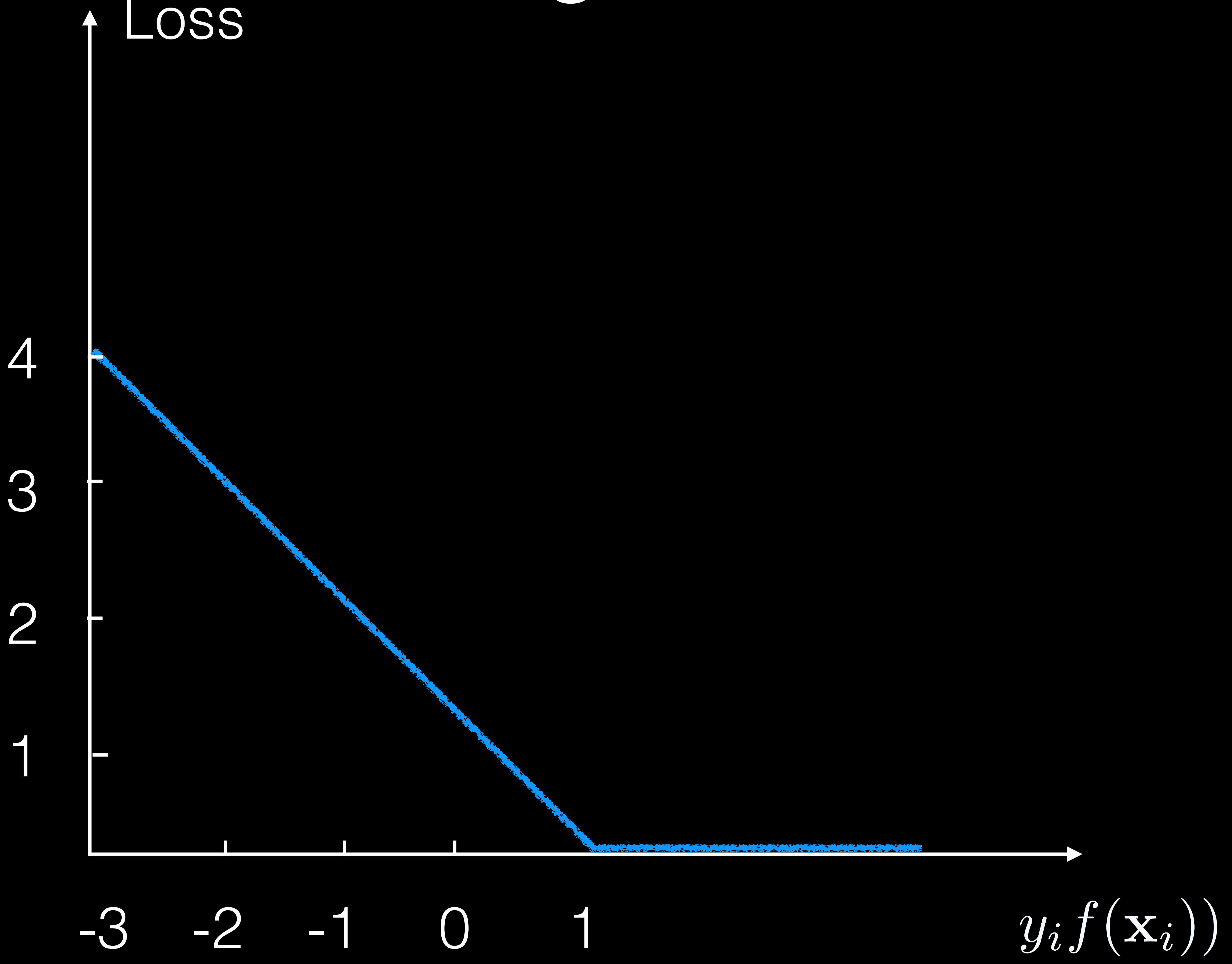
$$\min_{\mathbf{w}} ||\mathbf{w}||^2 + C \sum_{i}^{N} \max(0, 1 - y_i f(\mathbf{x}_i))$$

regularization + loss function

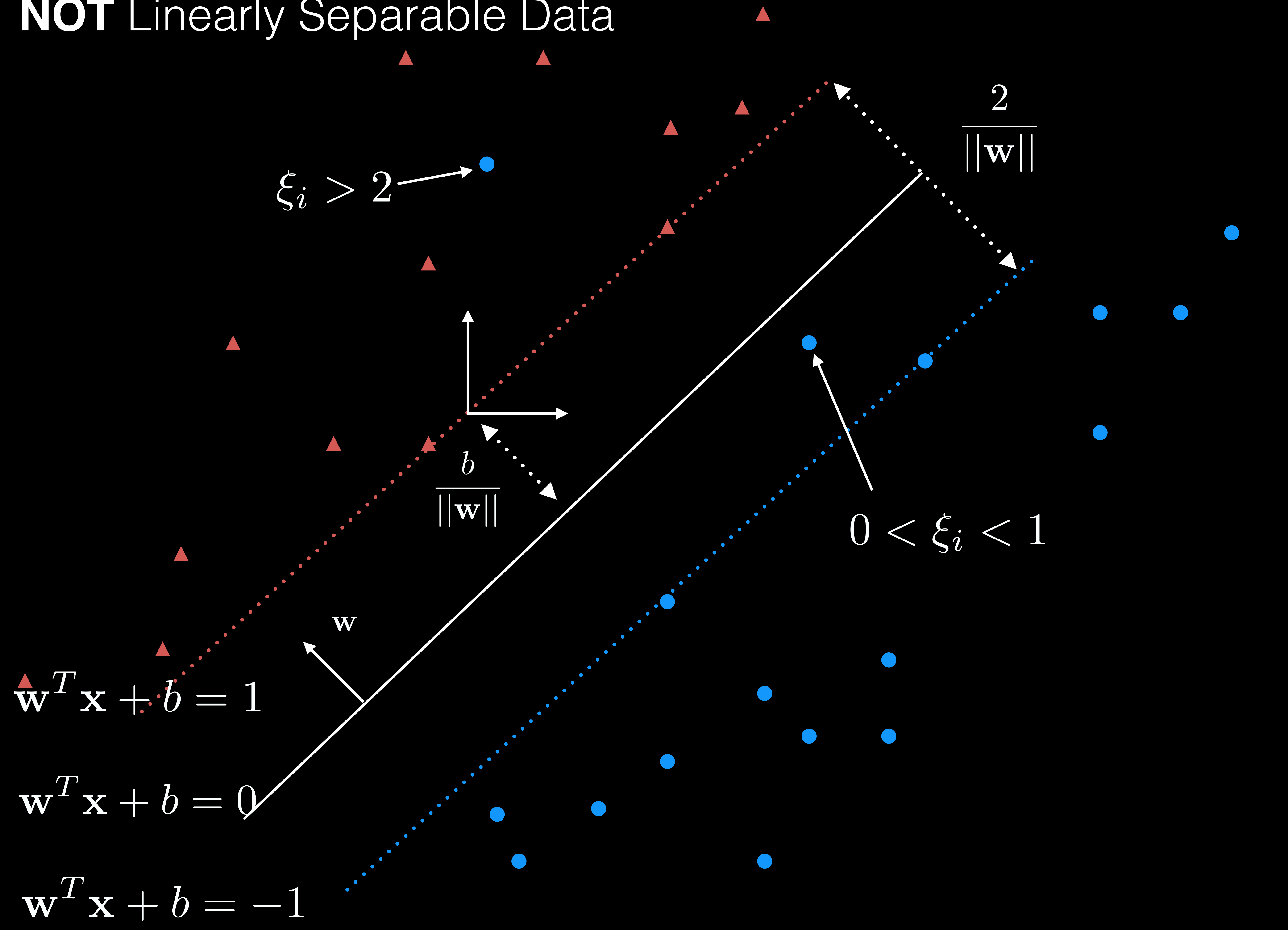
- If a point is on its side of the support vector(s), there is no contribution to the loss as $y_i f(\mathbf{x}_i)) \geq 1$

- If the point is within the margin or on other side of the opposite support vector(s), then there is a contribution to the loss as $y_i f(\mathbf{x}_i)) < 1$

Hinge Loss



NOT Linearly Separable Data


$\xi_i > 2$
$\frac{2}{||\mathbf{w}||}$
$\frac{b}{||\mathbf{w}||}$
$0 < \xi_i < 1$
$\mathbf{w}$
$\mathbf{w}^T\mathbf{x} + b = 1$
$\mathbf{w}^T\mathbf{x} + b = 0$
$\mathbf{w}^T\mathbf{x} + b = -1$

Optimization

There is a closed form solution to optimal $\mathbf{w}$

$$\min_{\mathbf{w}} ||\mathbf{w}||^2 + C \sum_{i}^{N} \max(0, 1 - y_i f(\mathbf{x}_i))$$

regularization + loss function

Alternatively, you use gradient descent!

Gradient Descent for SVM

Our cost function for optimizing can be re-written:

$$\min_{\mathbf{w}} \mathcal{C}(\mathbf{w}) = \frac{\lambda}{2}||\mathbf{w}||^2 + \frac{1}{N}\sum_{i}^{N} \max(0, 1 - y_i\, f(\mathbf{x_i}))$$

with $\lambda = \frac{2}{NC}$ and $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$

Gradient Descent for SVM

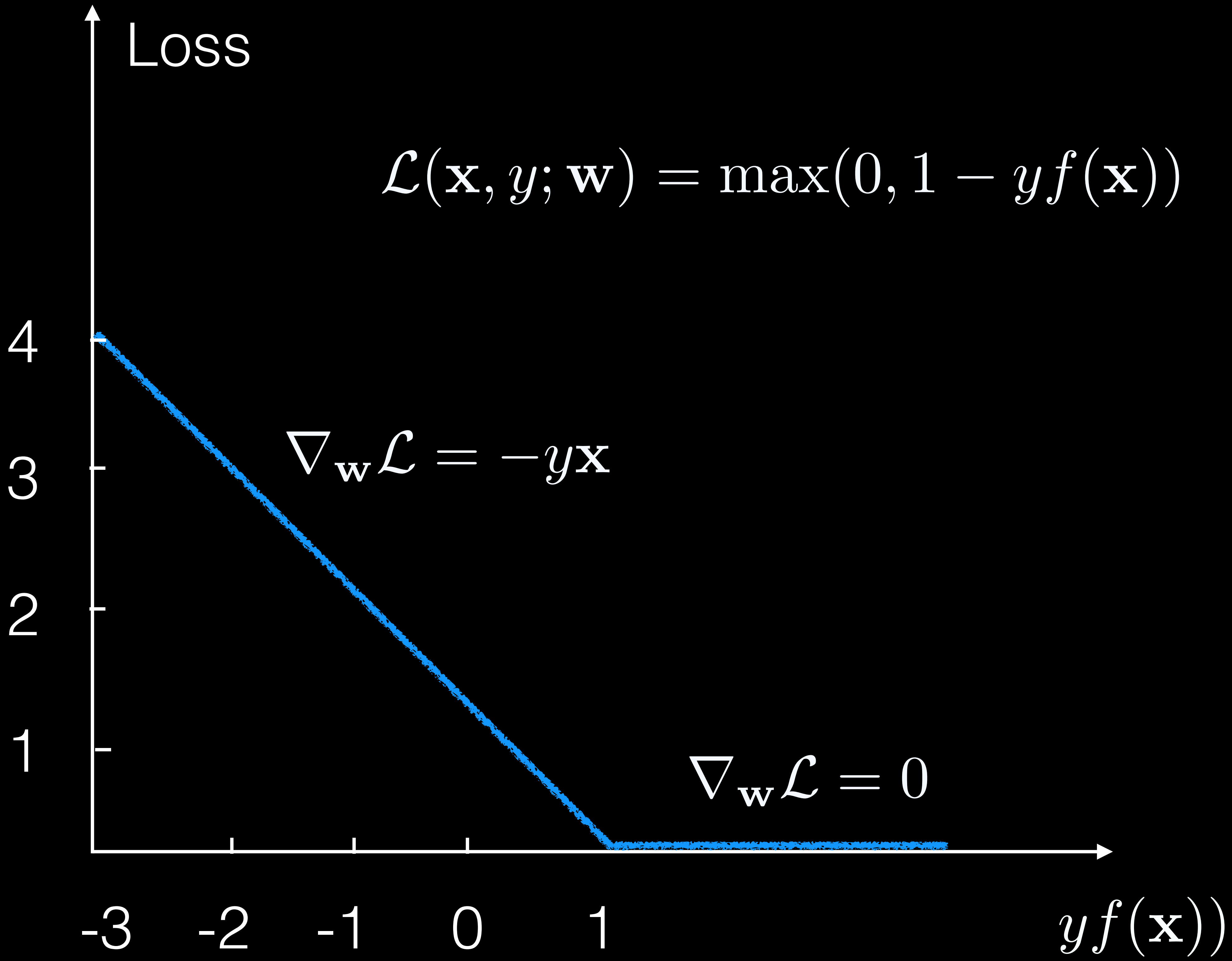
So gradient descent might look like:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \nabla_{\mathbf{w}} \mathcal{C}(\mathbf{w}_t)$$

with the ***learning rate*** η_t

but this gradient is not differentiable $\nabla_{\mathbf{w}} \mathcal{C}(\mathbf{w})$

(Sub-)Gradient of Hinge Loss

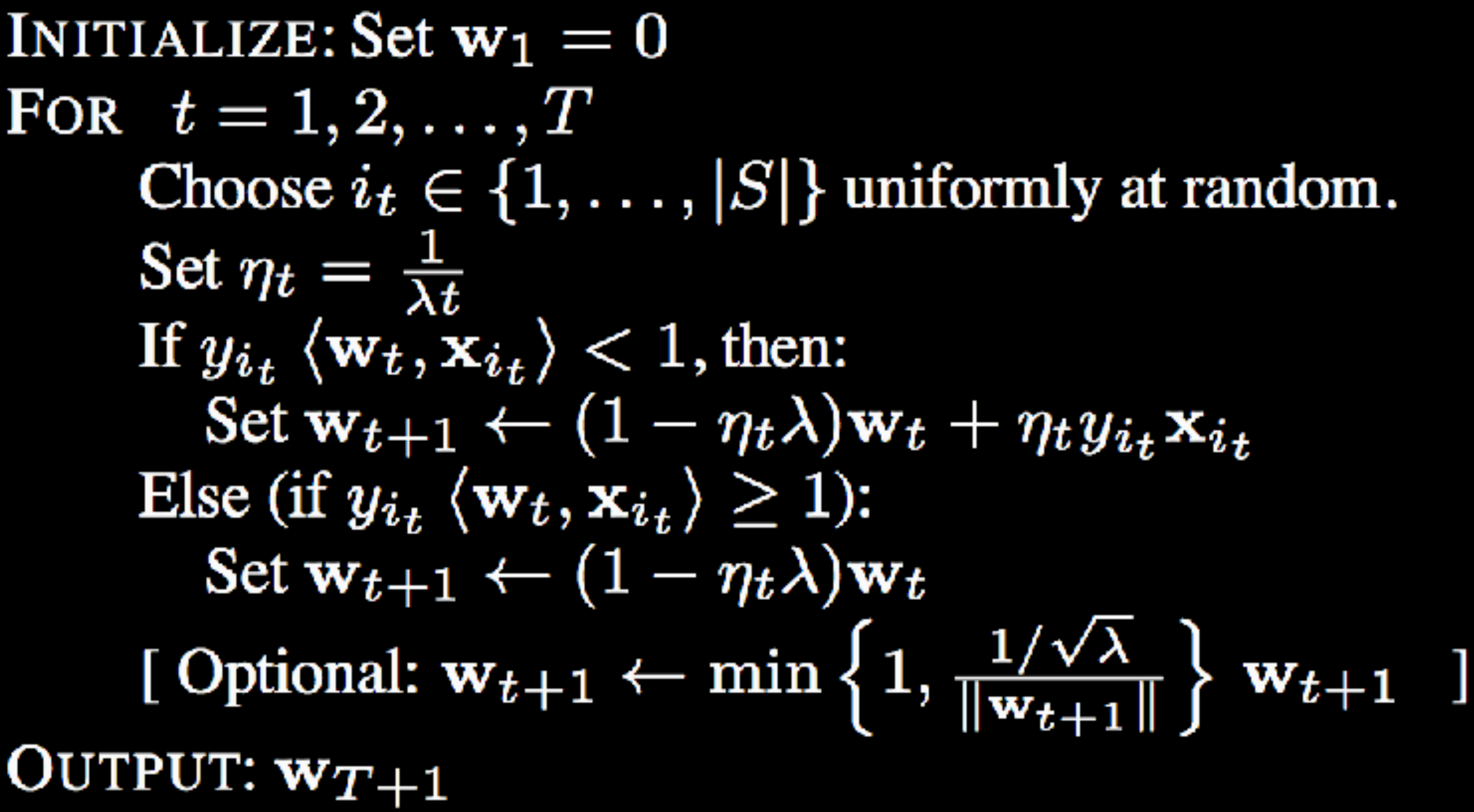


So now the whole gradient becomes:

$$\nabla_{\mathbf{w}}\mathcal{C}(\mathbf{w}) = \lambda\mathbf{w} - \frac{1}{N}\sum_{i}^{N}\mathbf{1}[y\,\mathbf{w}^T\mathbf{x_i} < 1]\,y\,\mathbf{x_i}$$

where **1**[t] is and indicator function = 1 if argument is true and 0 if false.

Pegasos Algorithm



```
INITIALIZE: Set $\mathbf{w}_1 = 0$
FOR  $t = 1, 2, \ldots, T$
    Choose $i_t \in \{1, \ldots, |S|\}$ uniformly at random.
    Set $\eta_t = \frac{1}{\lambda t}$
    If $y_{i_t} \langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle < 1$, then:
        Set $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t + \eta_t y_{i_t} \mathbf{x}_{i_t}$
    Else (if $y_{i_t} \langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle \geq 1$):
        Set $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t$
    [ Optional: $\mathbf{w}_{t+1} \leftarrow \min \left\{ 1, \frac{1/\sqrt{\lambda}}{\|\mathbf{w}_{t+1}\|} \right\} \mathbf{w}_{t+1}$ ]
OUTPUT: $\mathbf{w}_{T+1}$
```

Pegasos is a ***stochastic gradient descent*** algorithm with a ***mini batch*** size = 1!