Machine Learning, Spring 2020

Support Vector Machine

Reading Assignment: Chapter 5 & 6

Python tutorial: http://learnpython.org/

TensorFlow tutorial: https://www.tensorflow.org/tutorials/

PyTorch tutorial: https://pytorch.org/tutorials/

Acknowledge: The slides are partially referred to coursera online machine learning course by Prof. Andrew Ng, and NYU machine learning course.

Prof. Yi Fang (yfang@nyu.edu)

Support Vector Machines (SVMs)

Support vector machines are an optimization based prediction approach used primarily for **binary classification**, and are able to achieve state-of-the-art prediction accuracy on many real-world tasks.

Key idea 1: Learn a **decision boundary** that optimally separates positive and negative training examples. (But what does it mean to be optimal?)

Key idea 2: Learn a **linear** decision boundary in high dimensional space corresponding to a **non-linear** decision boundary for the original problem.

SVM assumes **real-valued** attributes on the **same scale**. Thus it is very important to pre-process your data before training the model:

- Normalize real-valued attributes (scale either to [0,1] or to mean = 0 and variance = 1). Make sure to use same scaling for training and test data.
- Replace discrete-valued attributes with **dummy variables**.

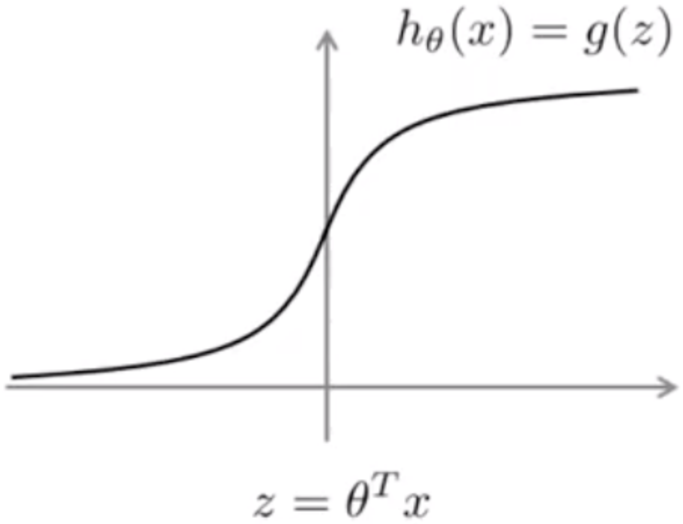
Car	Weight
1	Low
2	Medium
3	Heavy

⇨

Car	Weight=Medium	Weight=Heavy
1	0	0
2	1	0
3	0	1

From Logistic Regression to SVM

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$



If y =1, we wish our predicted hypothesis value is close to 1, then $\theta^T x \gg 0$

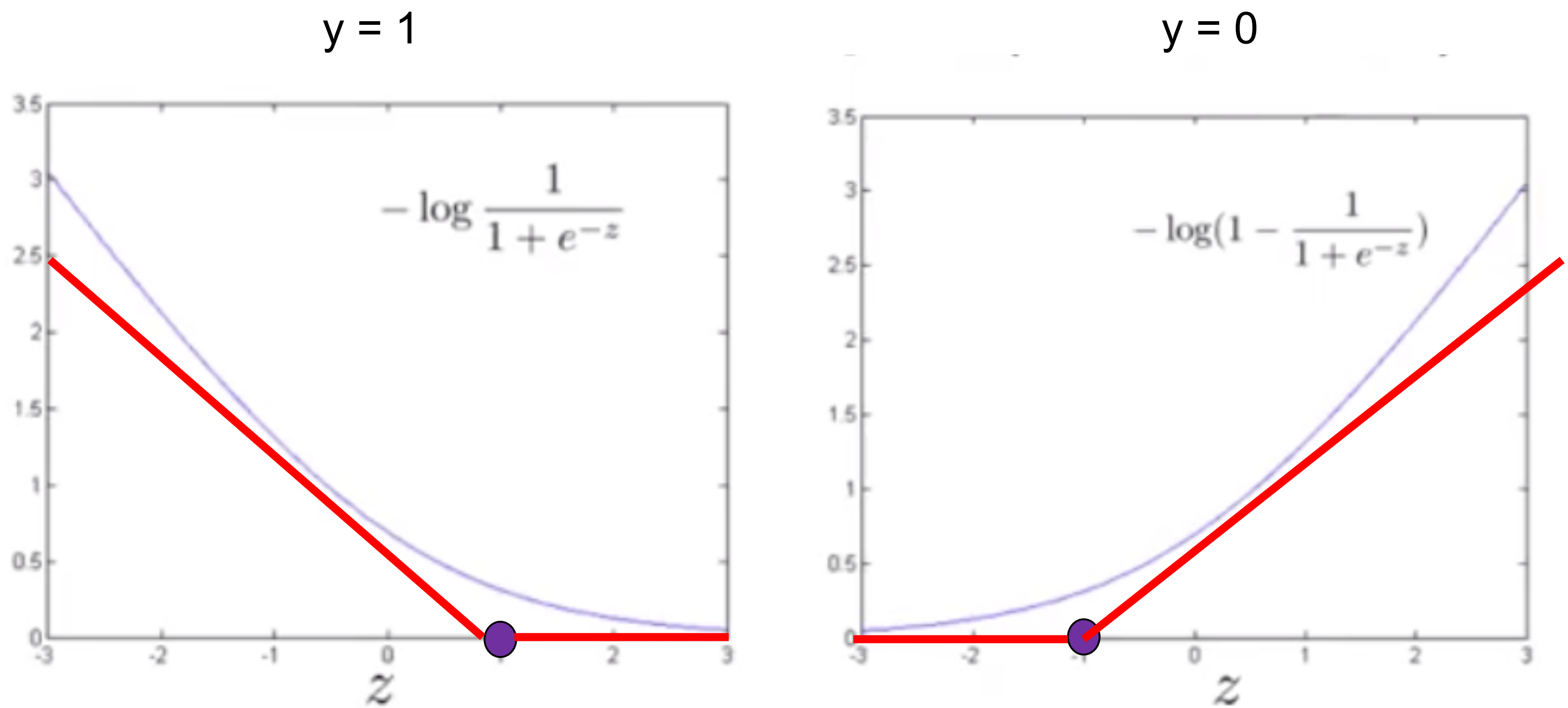
If y =0, we wish our predicted hypothesis value is close to 1, then $\theta^T x \ll 0$

Cost function of Logistic Regression

Cost Function:

$$-(y \log h_\theta(x) + (1-y)\log(1-h_\theta(x)))$$

$$= -y \log \frac{1}{1+e^{-\theta^T x}} - (1-y)\log(1 - \frac{1}{1+e^{-\theta^T x}})$$


y = 1
y = 0
$-\log \frac{1}{1+e^{-z}}$
$-\log(1 - \frac{1}{1+e^{-z}})$
3.5
3
2.5
2
1.5
1
0.5
0
-3
-2
-1
0
1
2
3
z
z

Logistic regression:

$$\min_{\theta} \frac{1}{m} \left[\sum_{i=1}^{m} y^{(i)} \left(-\log h_\theta(x^{(i)}) \right) + (1 - y^{(i)}) \left((-\log(1 - h_\theta(x^{(i)})) \right) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

Support vector machine:

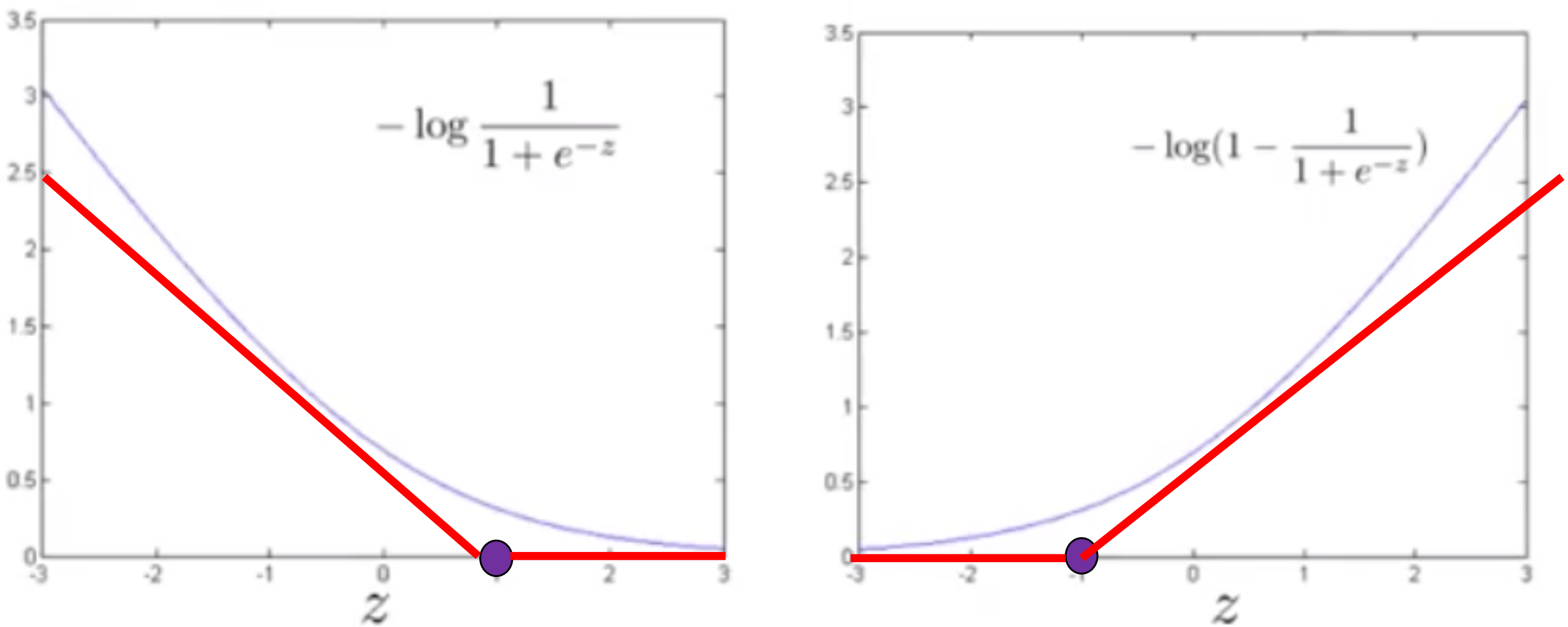
$$\min_{\theta} C \sum_{i=1}^{m} \left[y^{(i)} cost_1(\theta^T x^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{i=1}^{n} \theta_j^2$$

SVM Hypothesis

Support Vector Machine

$$\min_{\theta} C \sum_{i=1}^{m} \left[y^{(i)} cost_1(\theta^T x^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{i=1}^{n} \theta_j^2$$

y = 1 y = 0

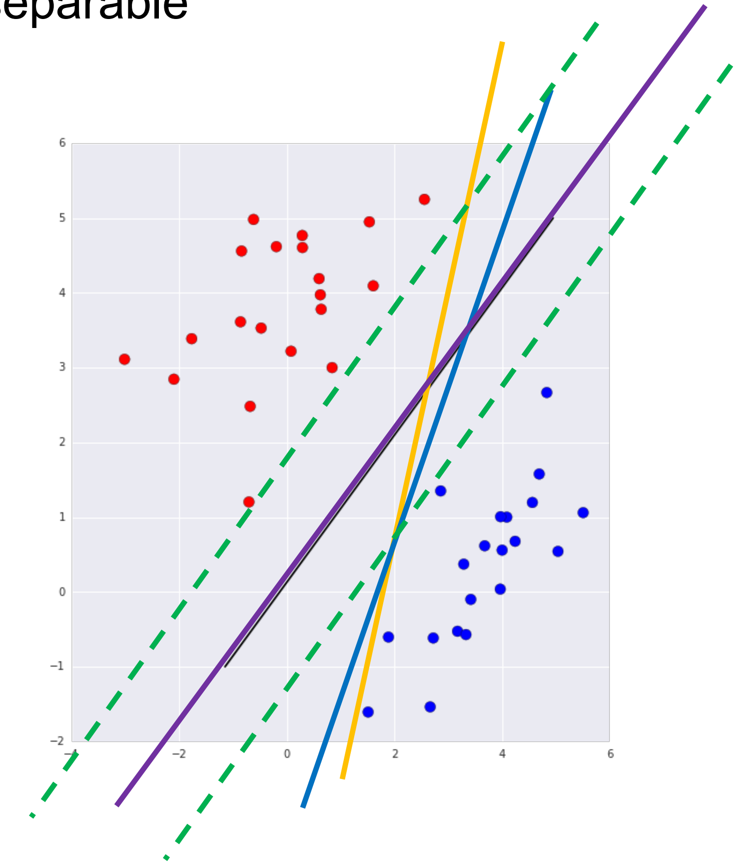


SVM Decision Boundary

$$\min_{\theta} \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

$$\begin{aligned} \text{s.t.} \quad & \theta^T x^{(i)} \geq 1 && \text{if } y^{(i)} = 1 \\ & \theta^T x^{(i)} \leq -1 && \text{if } y^{(i)} = 0 \end{aligned}$$

Example of linearly separable



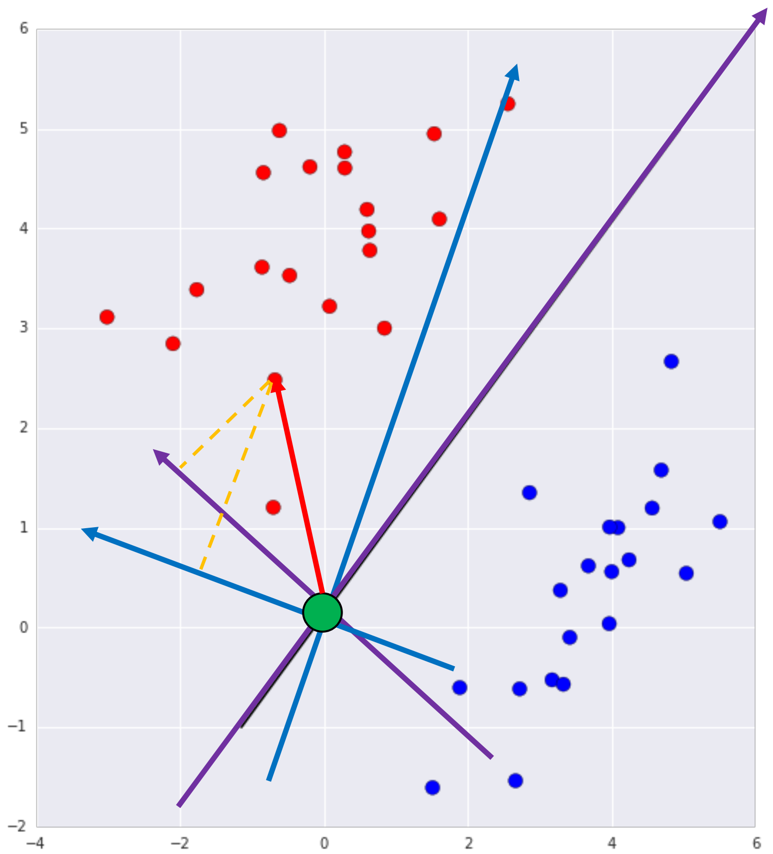
Large Margin Classifier

Why large margin works?

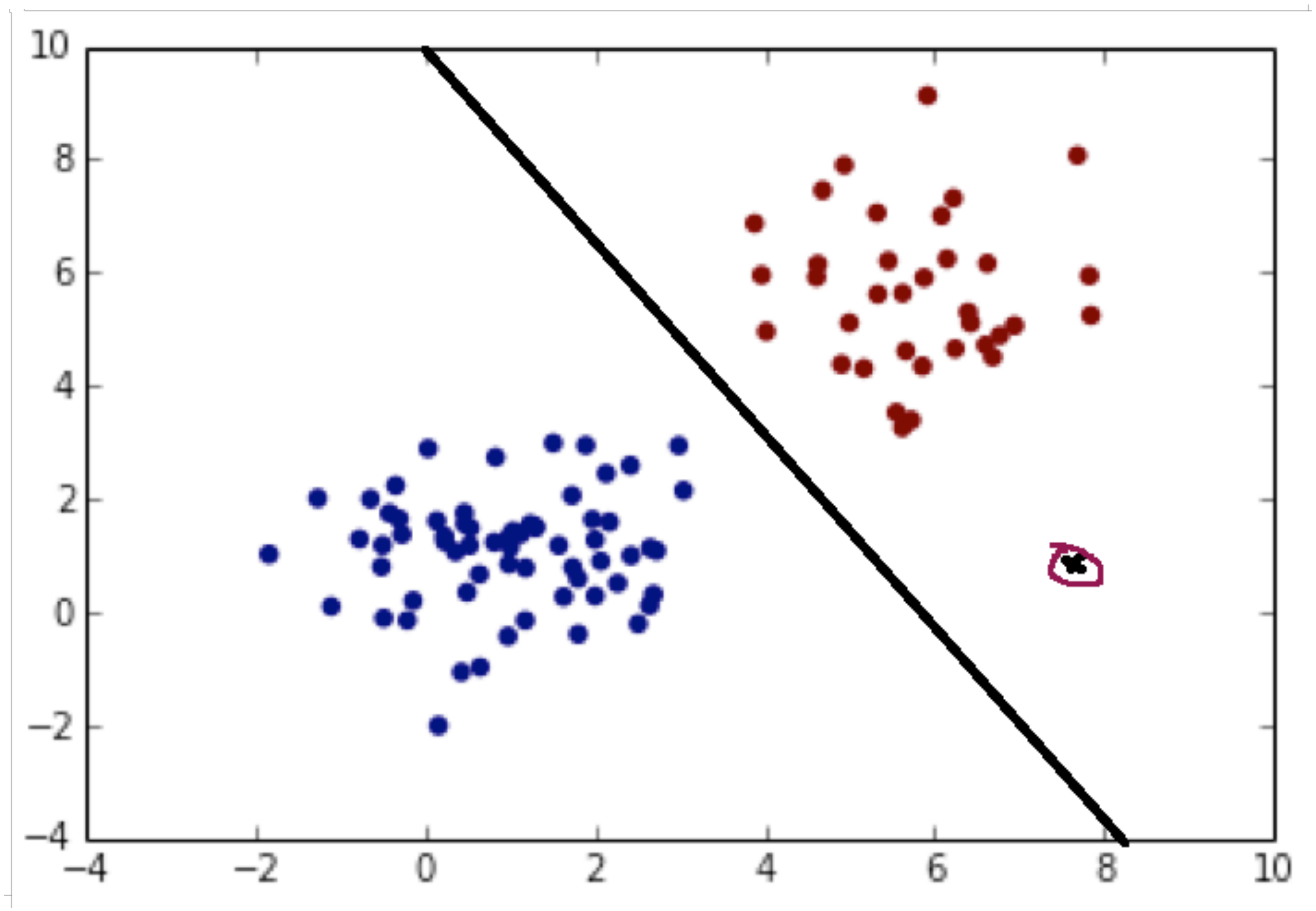
$$\min_{\theta} \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

$$\text{s.t.} \quad \theta^T x^{(i)} \geq 1 \quad \text{if } y^{(i)} = 1$$

$$\theta^T x^{(i)} \leq -1 \quad \text{if } y^{(i)} = 0$$

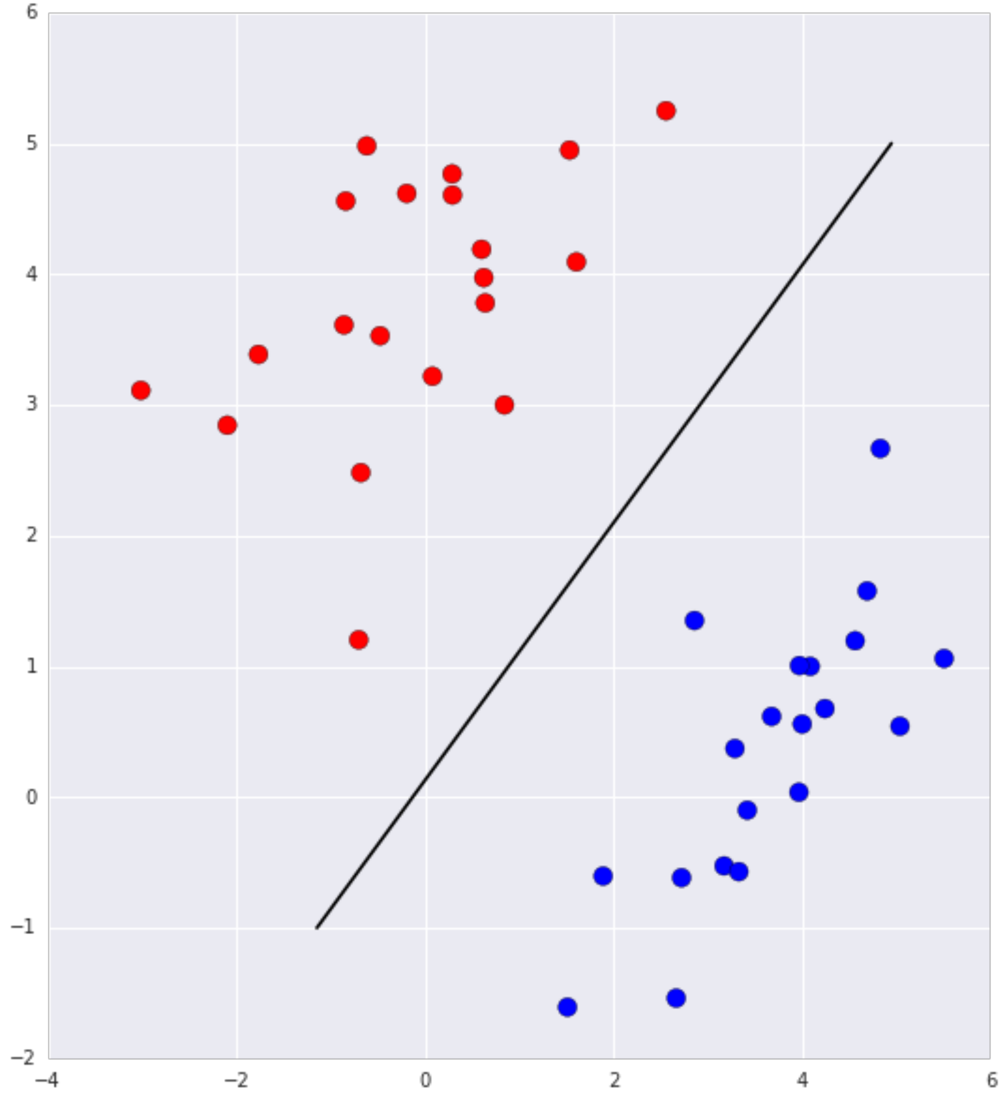


SVMs: the basic idea (linearly separable case)



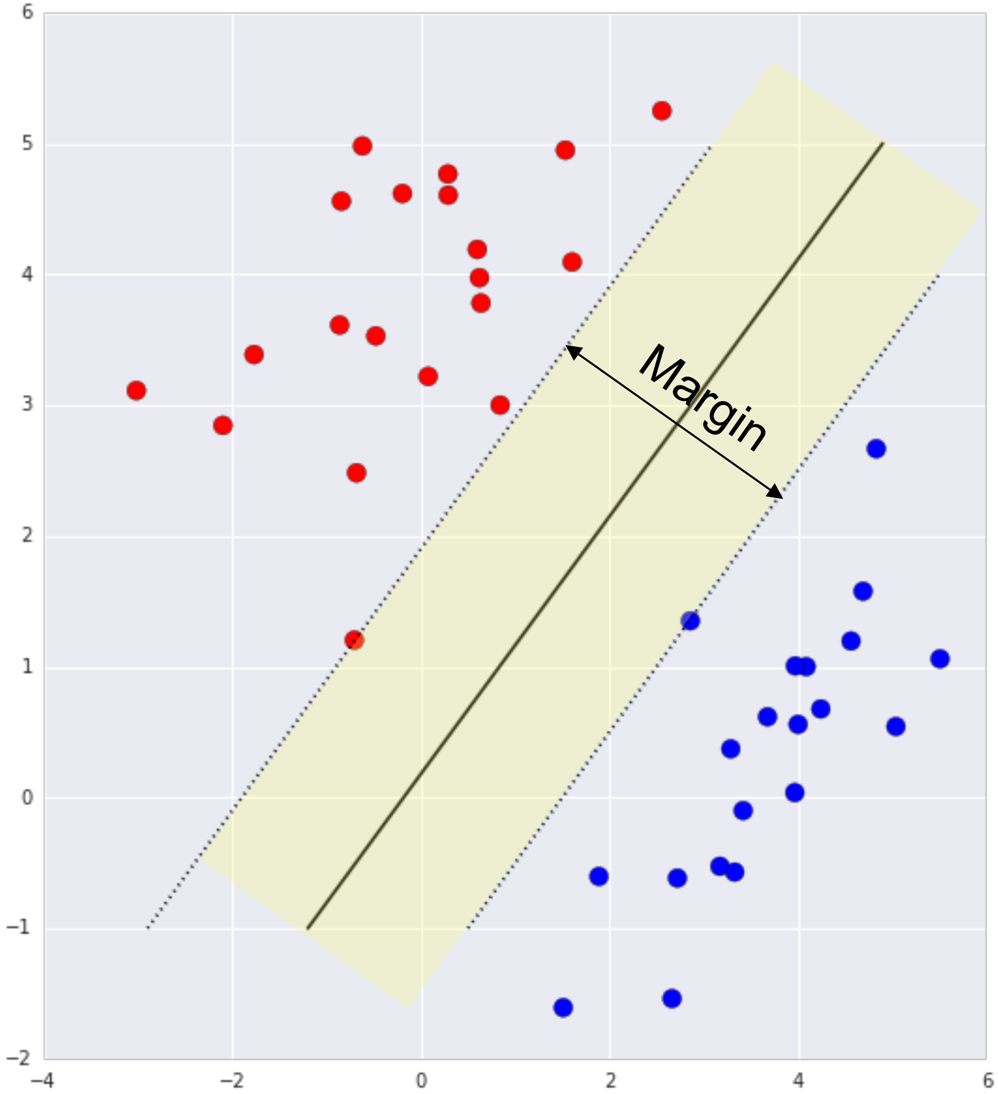
Which line to choose?

This is an optimization problem.



Which line to choose?

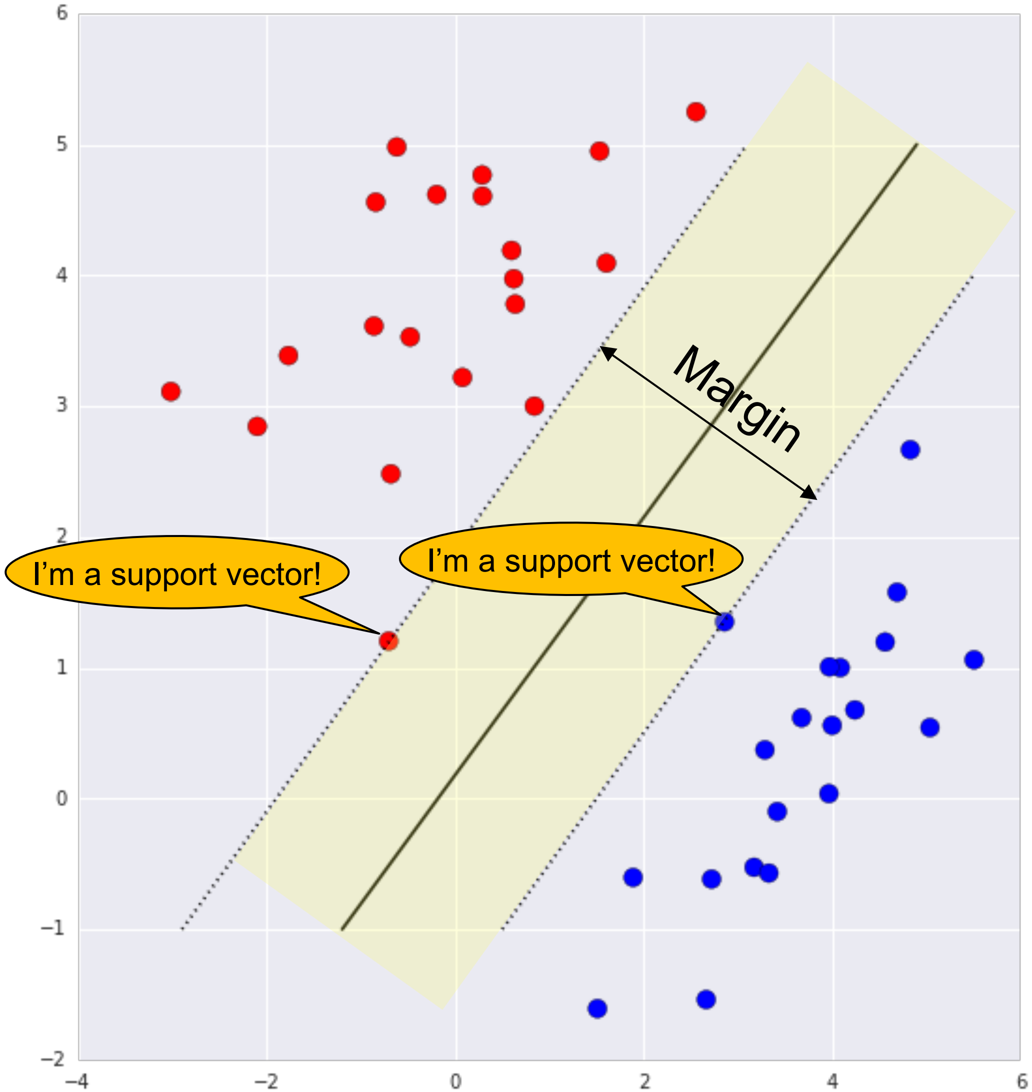
Choose the line that maximizes the **margin** between classes.



Margin = how **wide** we could make the linear decision boundary before it contacts points from either class.

Which line to choose?

Choose the line that maximizes the **margin** between classes.



Points on the margin are called **support vectors**.

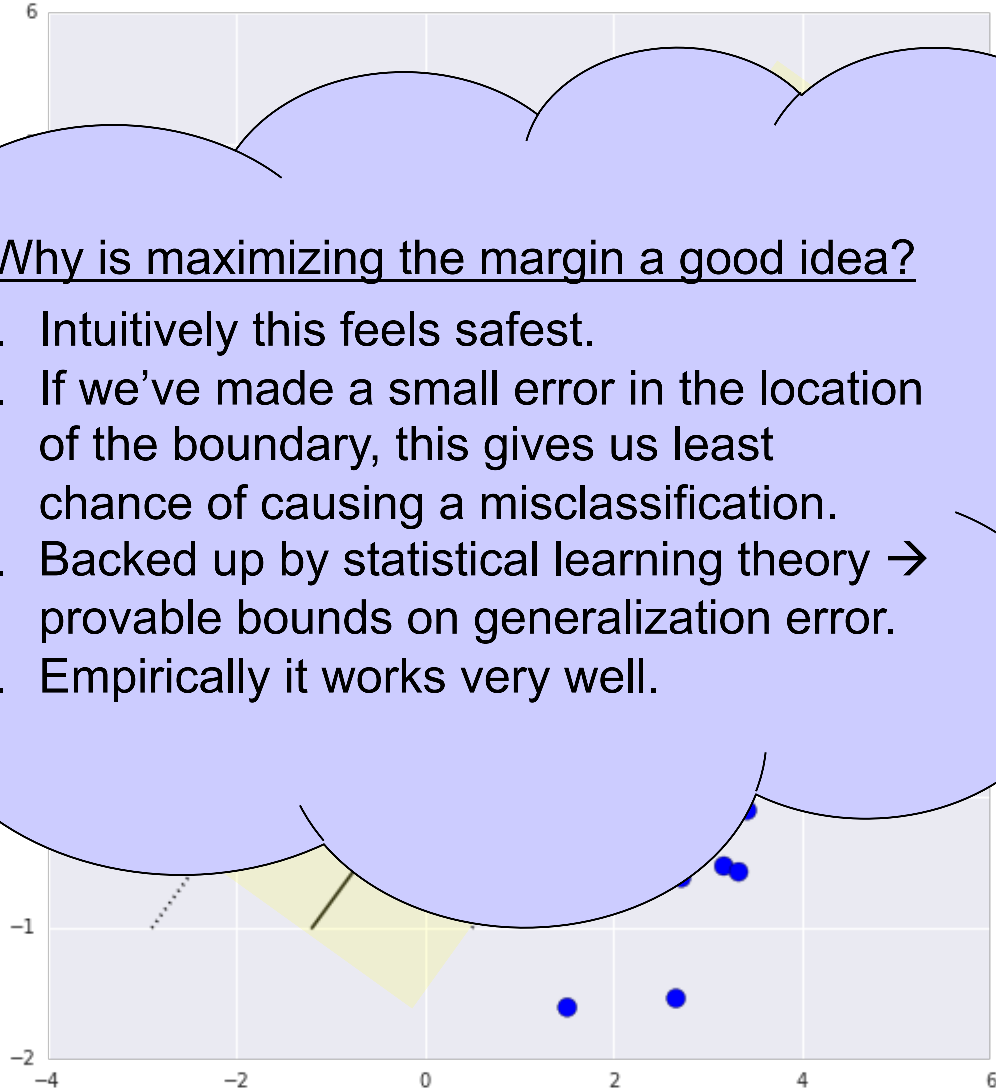
The classifier can be defined entirely by the set of support vectors.

This fact has lots of useful implications:

- Fast classification of test points.
- Fast leave-one-out cross-validation.
- Faster, but still expensive, training.

Which line to choose?

Choose the line that maximizes the **margin** between classes.



Why is maximizing the margin a good idea?

1. Intuitively this feels safest.
2. If we've made a small error in the location of the boundary, this gives us least chance of causing a misclassification.
3. Backed up by statistical learning theory → provable bounds on generalization error.
4. Empirically it works very well.

Points on the
argin are called
pport vectors.

ier can be
ely by the
rt vectors.

lots of
ations:

ification
points.
ast leave-one-out
cross-validation.
Faster, but still
expensive, training.

How to maximize the margin?

To separate, for all points j, we must have:

$$y_j(x_j^T w + b) > 0$$

For the margin, define:

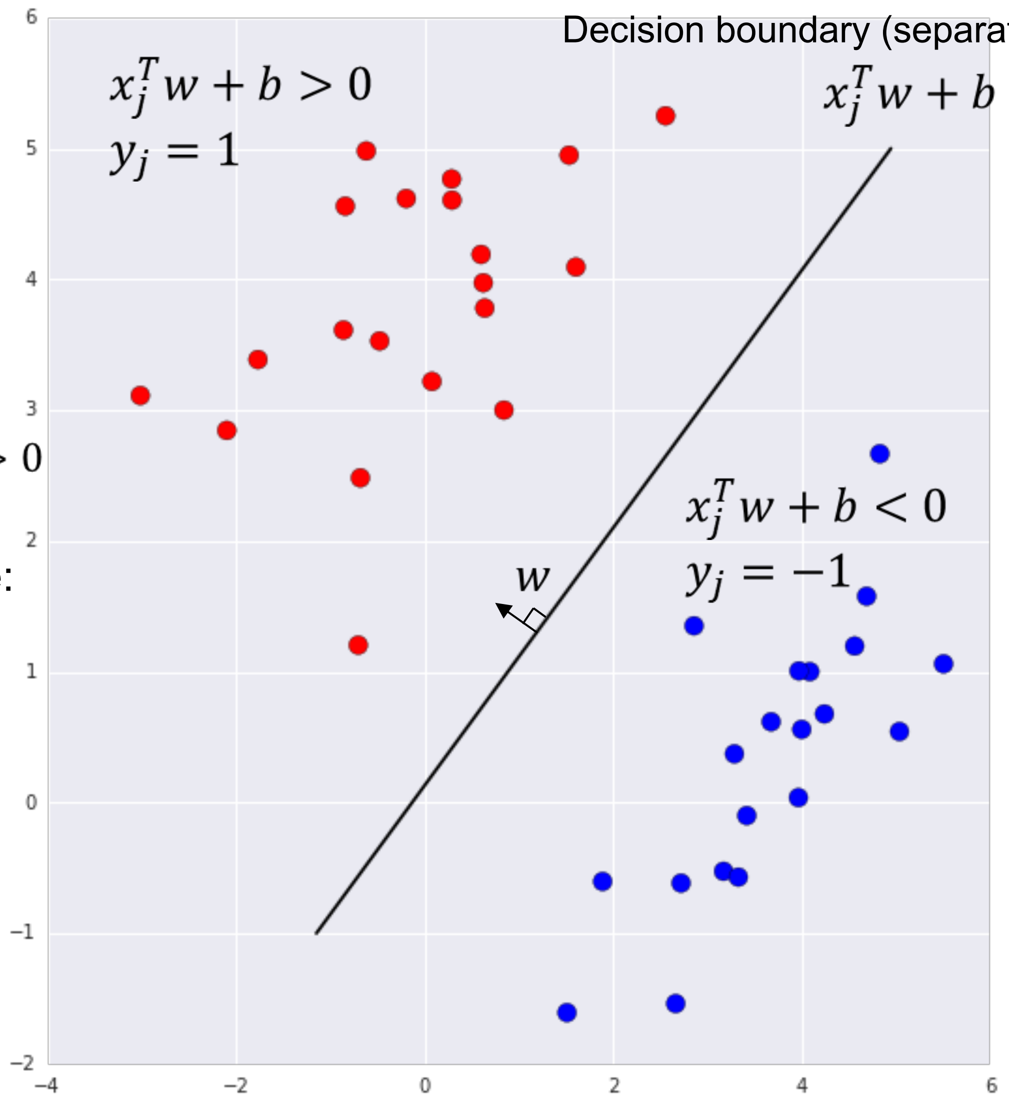
$$M = \min_j y_j\left(x_j^T w + b\right) > 0$$

Then for $y_j = 1$, we have:

$$x_j^T w + b \geq M$$

For $y_j = -1$, we have:

$$x_j^T w + b \leq -M$$



Represent each point as (x_j, y_j), where y_j, the class value we are trying to predict, is +1 or -1. Note that x_j is a vector of length 2 in this example.

How to maximize the margin?

To separate, for all points j, we must have:

$$y_j(x_j^T w + b) > 0$$

For the margin, define:

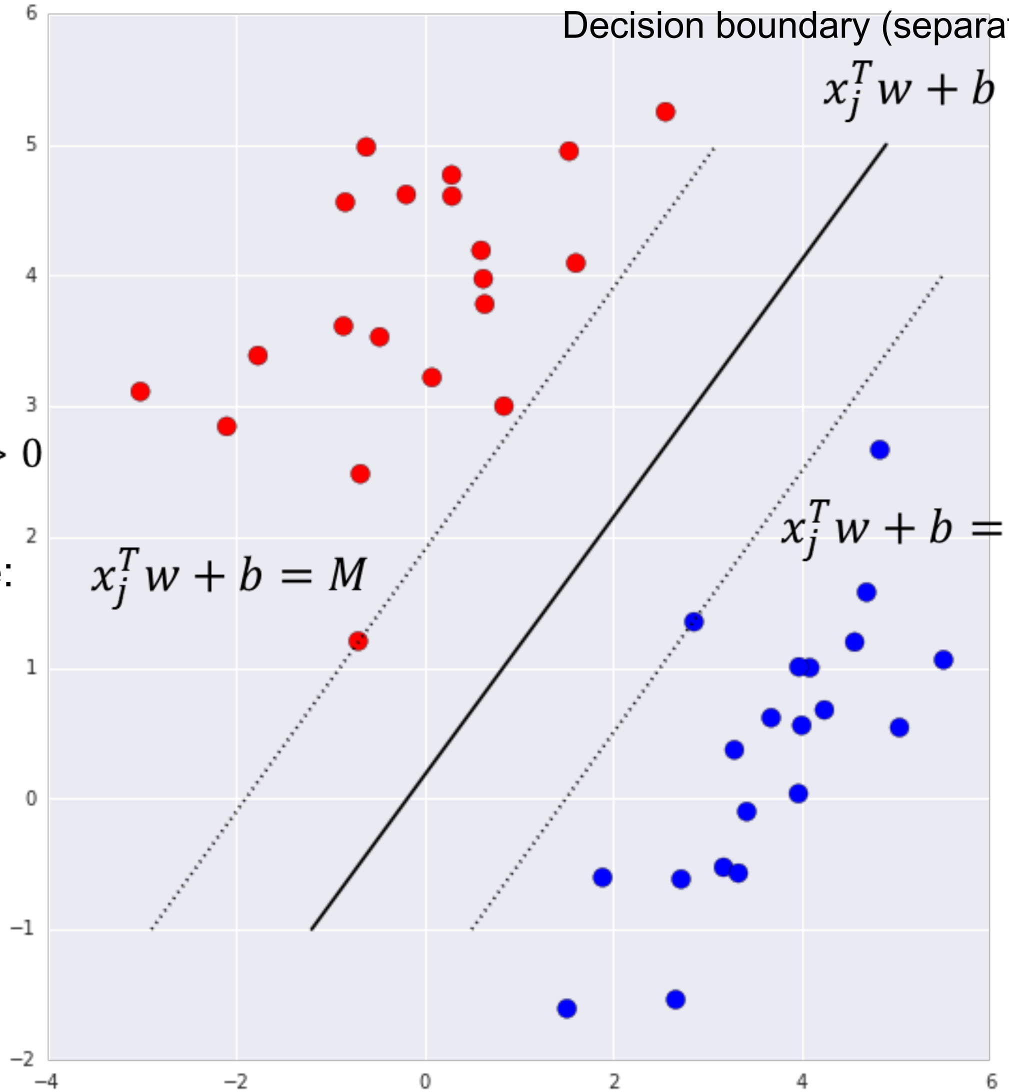
$$M = \min_j y_j\left(x_j^T w + b\right) > 0$$

Then for y_j = 1, we have:

$$x_j^T w + b \geq M$$

For y_j = -1, we have:

$$x_j^T w + b \leq -M$$



Represent each point as (x_j, y_j), where y_j, the class value we are trying to predict, is +1 or -1. Note that x_j is a vector of length 2 in this example.

How to maximize the margin?

Margin = 2M / ||w||.

This follows from computing distance between parallel lines.

Goal: maximize 2M / ||w|| subject to constraints, for all j:

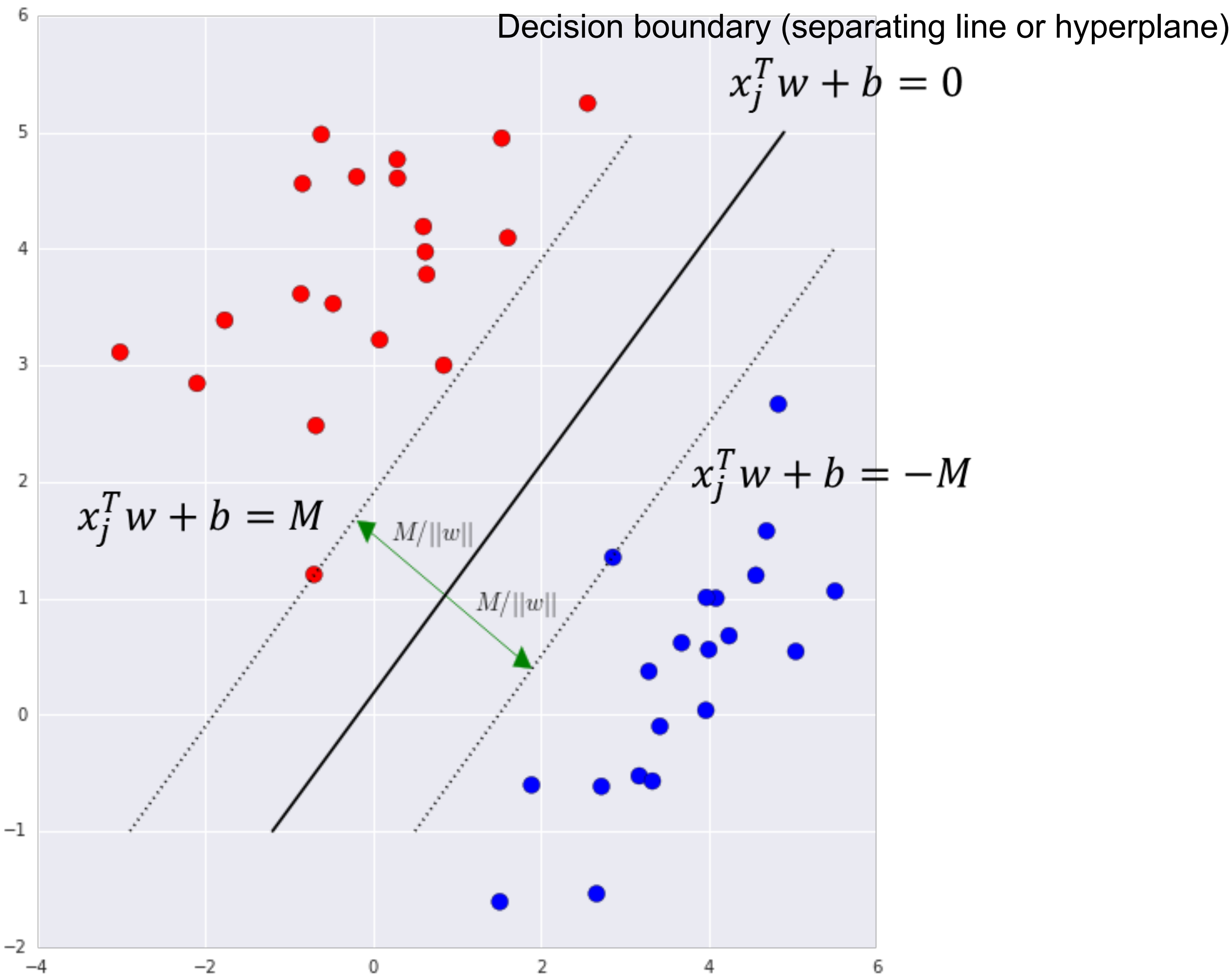
$$y_j(x_j^T w + b) \geq M$$

Simplify by change of variables, dividing w and b through by M.

New goal: minimize ||w|| subject to constraints, for all j:

$$y_j(x_j^T w + b) \geq 1$$

New margin: 2 / ||w||



How to maximize the margin?

Margin = 2M / ||w||.

This follows from computing distance between parallel lines.

<u>Goal</u>: maximize 2M / ||w|| subject to constraints, for all j:

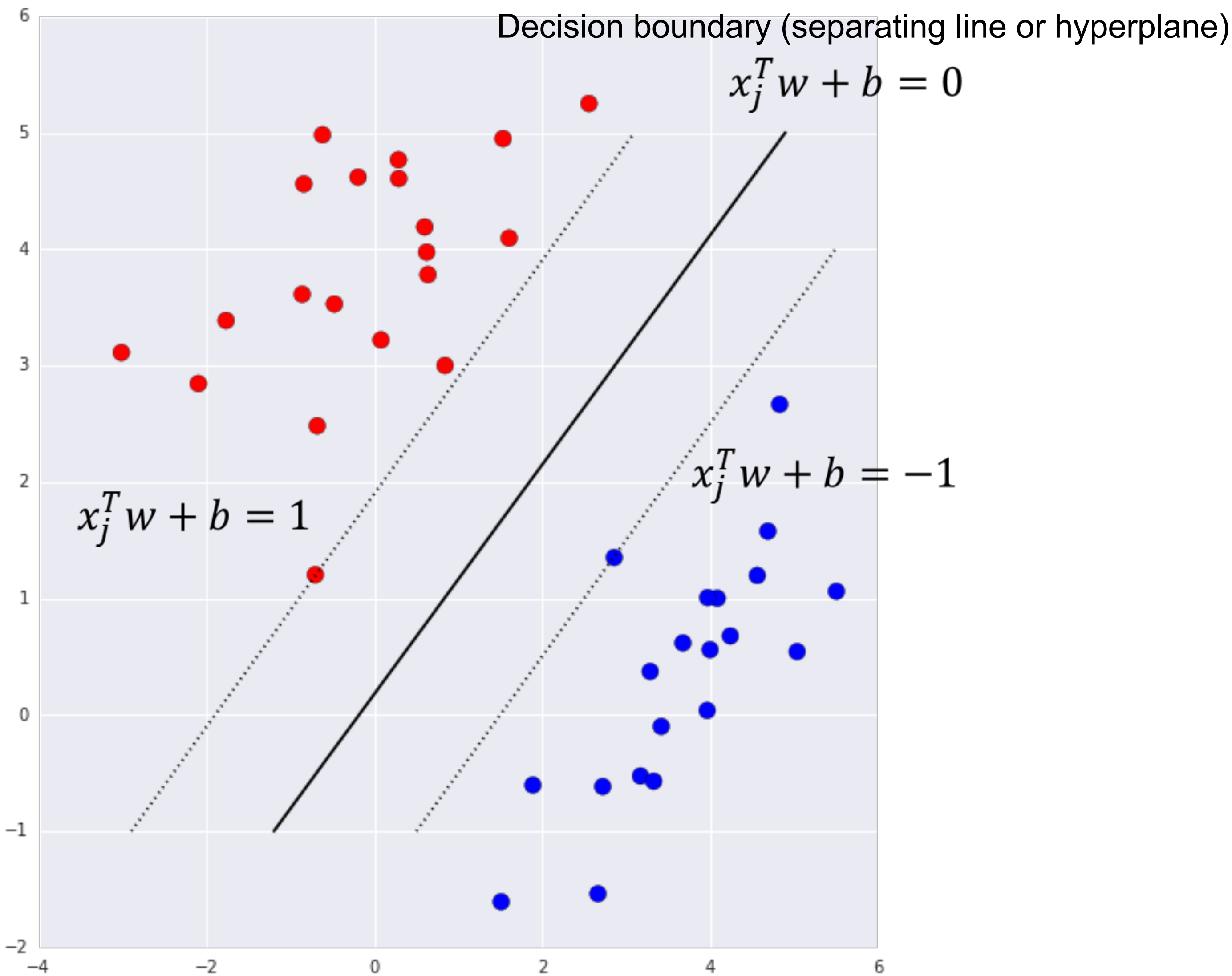
$$y_j(x_j^T w + b) \geq M$$

Simplify by change of variables, dividing w and b through by M.

<u>New goal</u>: minimize ||w|| subject to constraints, for all j:

$$y_j(x_j^T w + b) \geq 1$$

<u>New margin</u>: 2 / ||w||



How to maximize the margin?

Margin = 2M / ||w||.

This follows from computing distance between parallel lines.

<u>Goal</u>: maximize 2M / ||w|| subject to constraints, for all

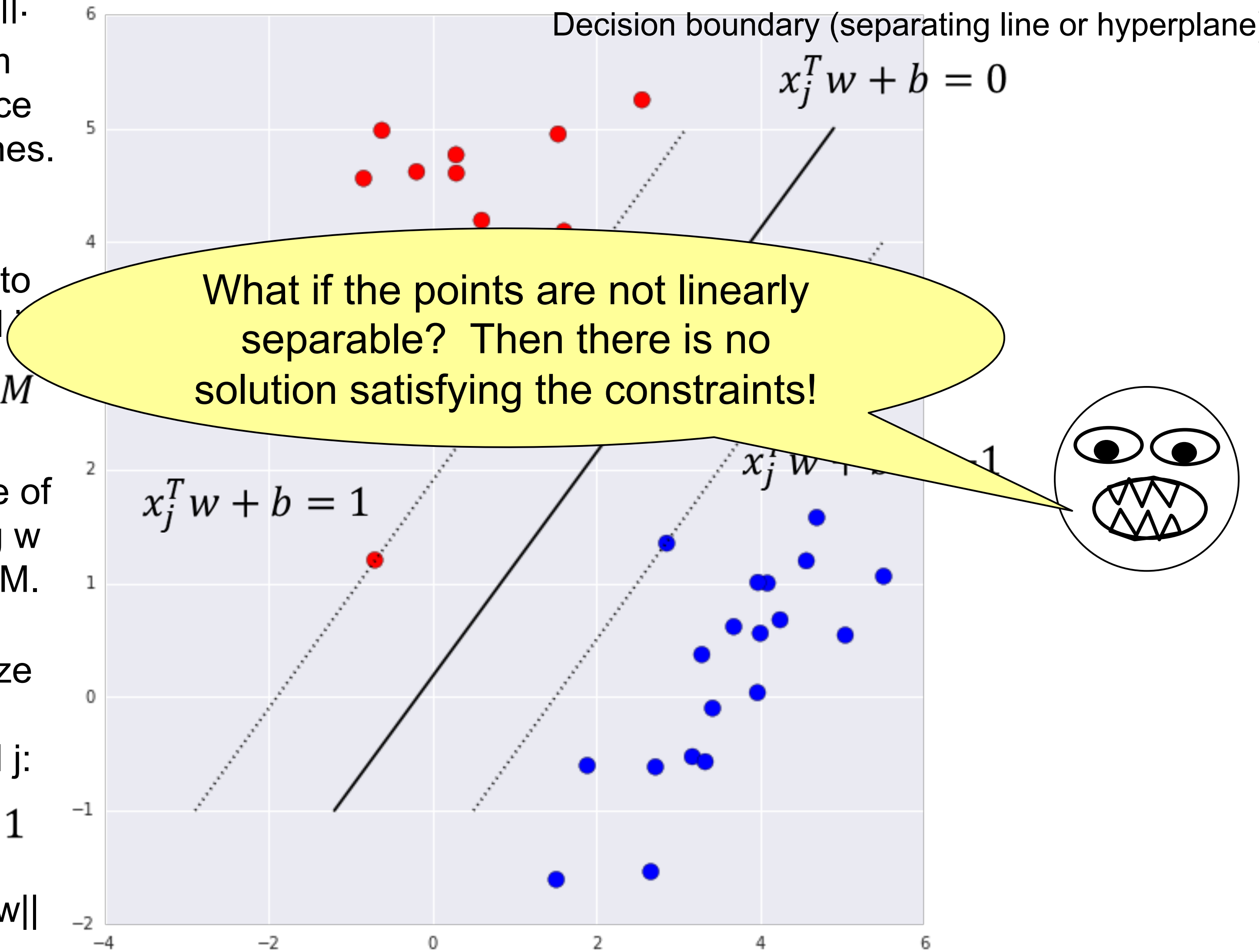
$$y_j(x_j^T w + b) \geq M$$

Simplify by change of variables, dividing w and b through by M.

<u>New goal</u>: minimize ||w|| subject to constraints, for all j:

$$y_j(x_j^T w + b) \geq 1$$

<u>New margin</u>: 2 / ||w||



Non-separable case: soft margins

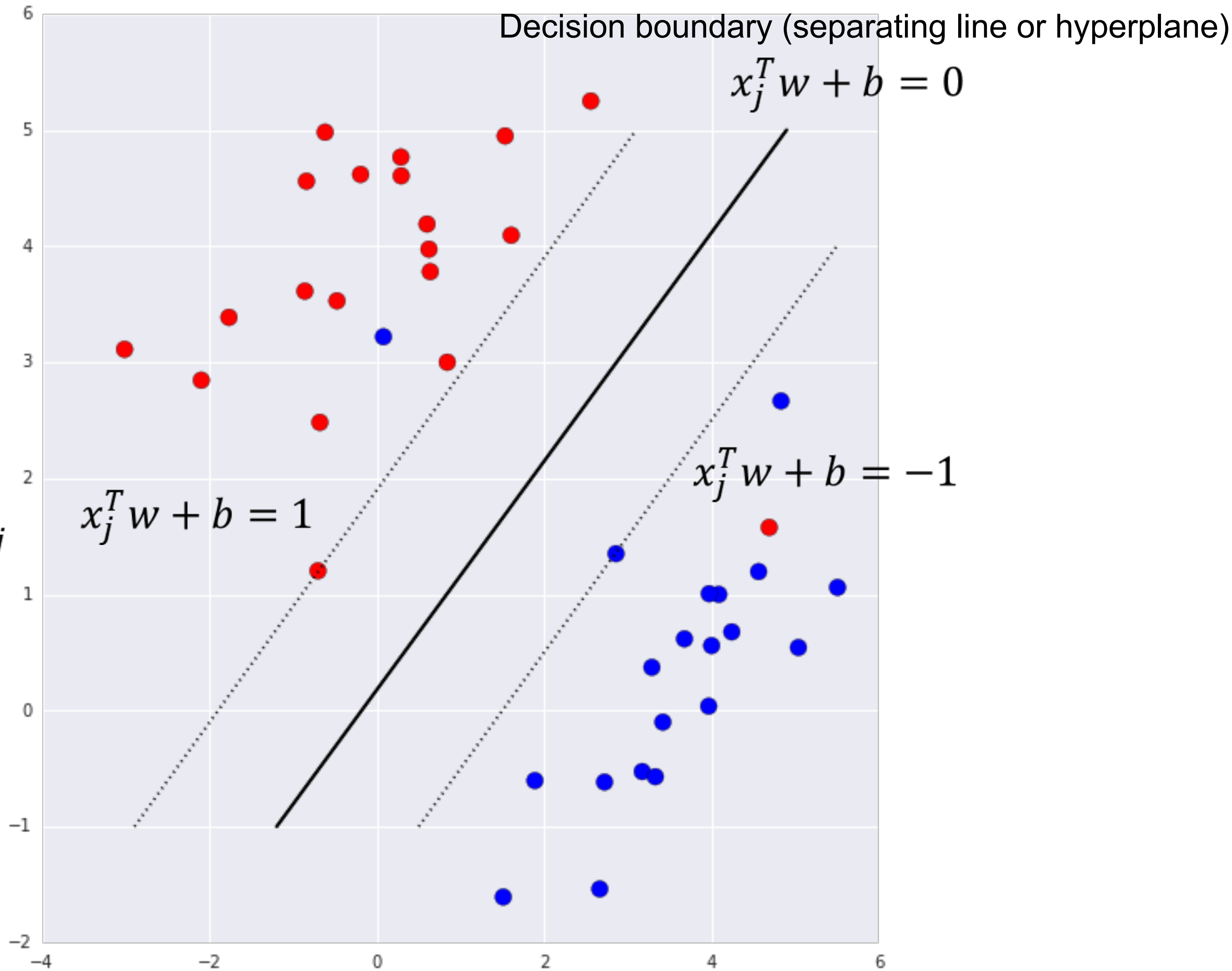
Goal (hard margin): minimize ||w|| subject to constraints, for all j:

$$y_j\left(x_j^T w + b\right) \geq 1$$

Goal (soft margin): minimize subject to constraints, for all j:

$$y_j\left(x_j^T w + b\right) \geq 1 - \xi_j$$

$$\xi_j \geq 0$$



Non-separable case: soft margins

<u>Goal (hard margin)</u>: minimize ||w|| subject to constraints, for all j:

$$y_j\left(x_j^T w + b\right) \geq 1$$

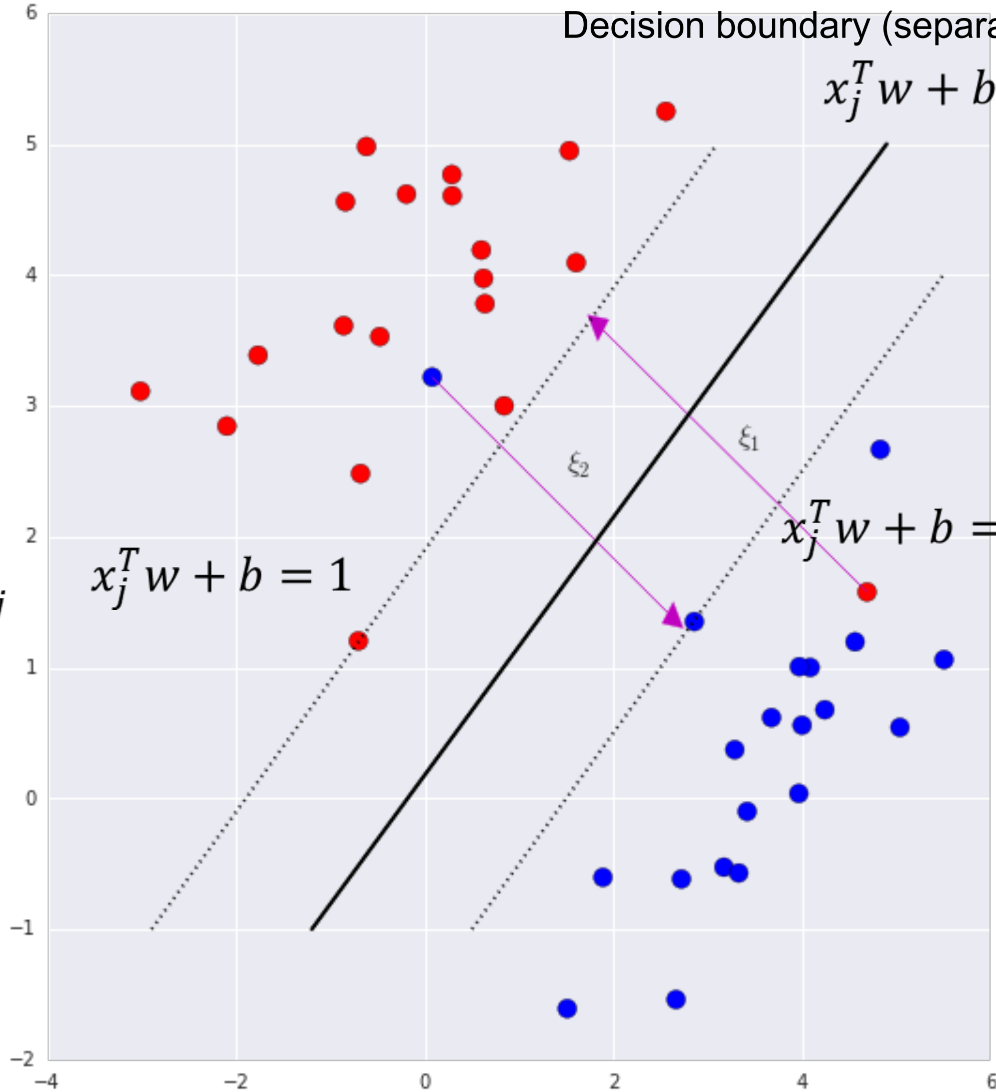
<u>Goal (soft margin)</u>: minimize subject to constraints, for all j:

$$y_j\left(x_j^T w + b\right) \geq 1 - \xi_j$$

$$\xi_j \geq 0$$

But what should we minimize? ||w||?

<u>Answer</u>: minimize $\frac{1}{2}\|w\|^2 + C\sum_j \xi_j$



This is a **quadratic programming** (QP) problem, optimizing a quadratic function with linear constraints.

We can use off-the-shelf QP solvers, or faster methods for large problems, to find the optimal w, b, and ξ.

Classify test points by $sign(x_j^T w + b)$.

Soft margins in practice

<u>Goal (hard margin)</u>: minimize ||w|| subject to constraints, for all j:

$$y_j\left(x_j^T w + b\right) \geq 1$$

<u>Goal (soft margin)</u>: minimize subject to constraints, for all j:

$$y_j\left(x_j^T w + b\right) \geq 1 - \xi_j$$

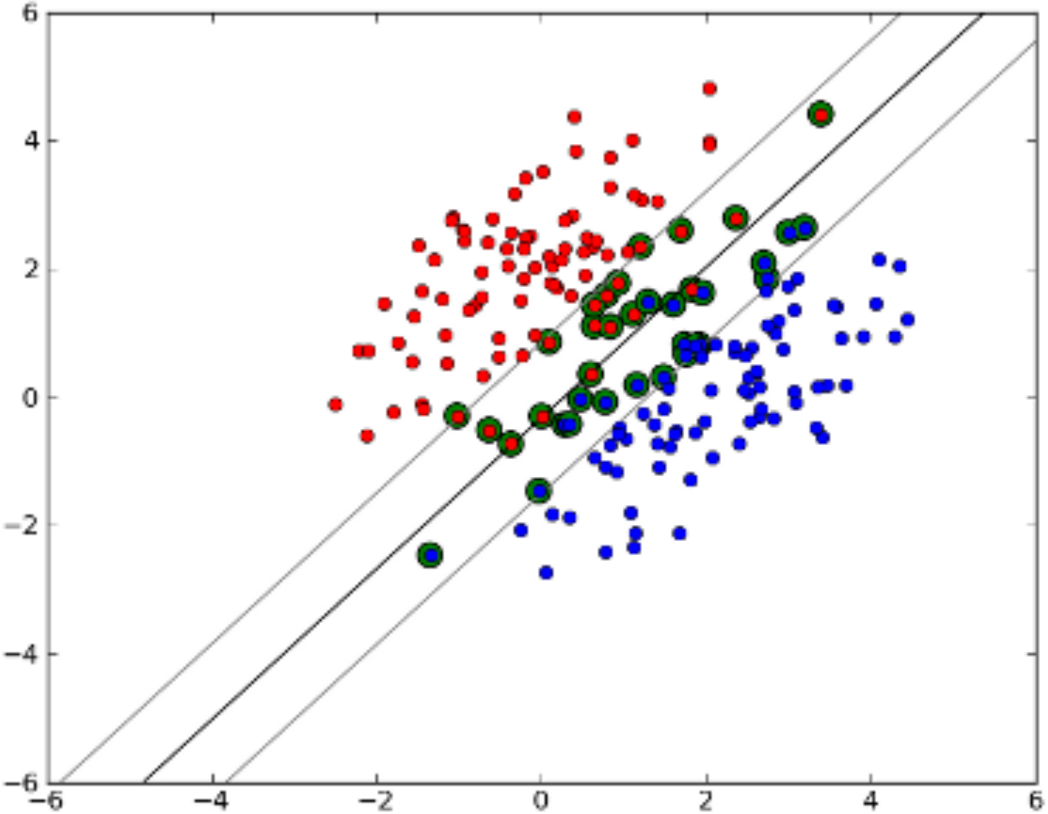
$$\xi_j \geq 0$$

But what should we minimize? ||w||?

<u>Answer</u>: minimize $\frac{1}{2}||w||^2 + C\sum_j \xi_j$

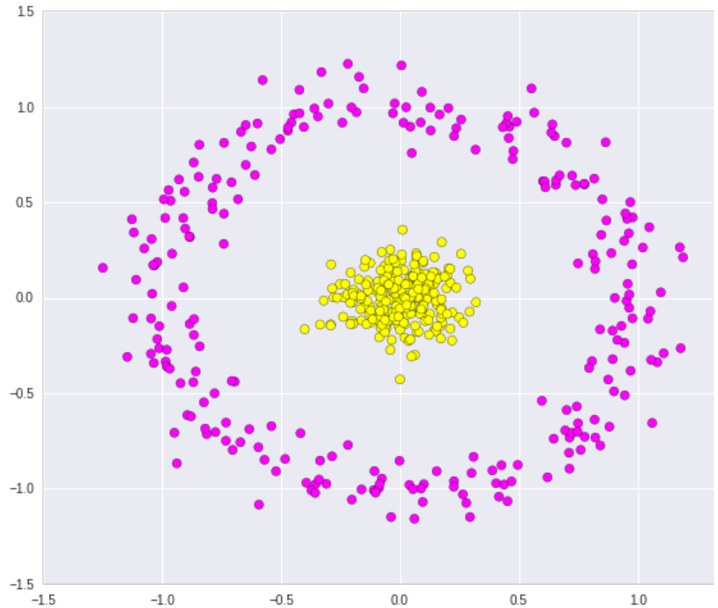
In practice, there may be many training points with $\xi_j > 0$ (all of these are support vectors).

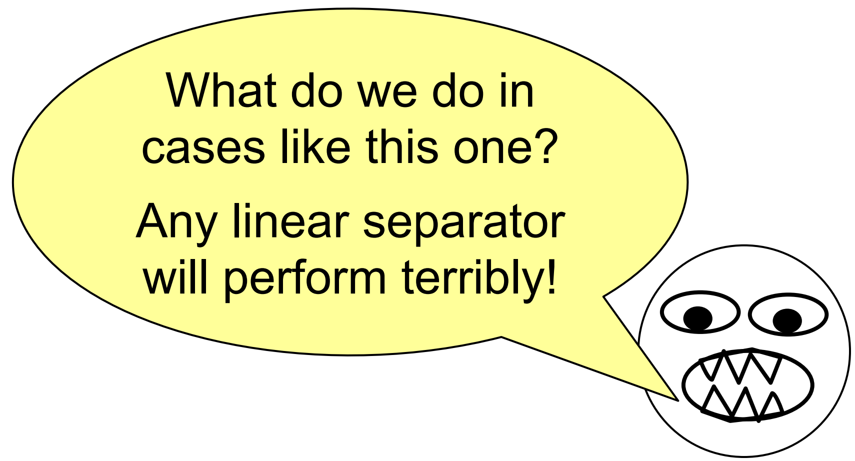
Training points with $\xi_j > 1$ are misclassifications.

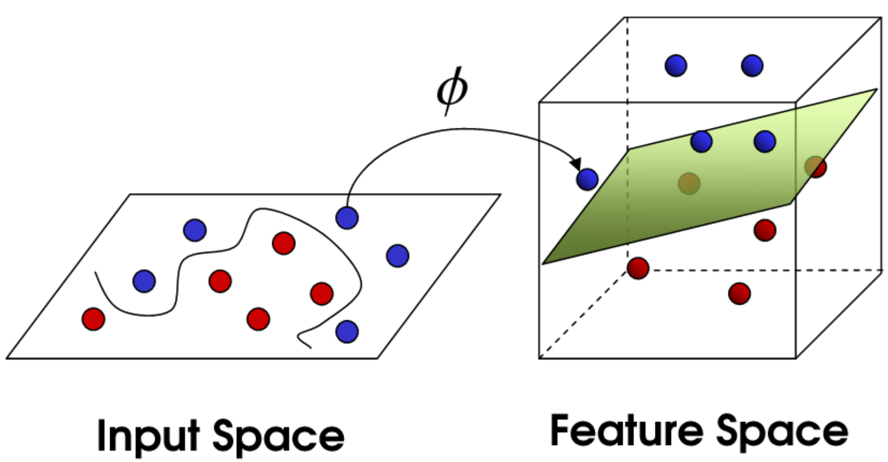


http://www.mblondel.org/journal/2010/09/19/support-vector-machines-in-python/

Non-linear decision boundaries



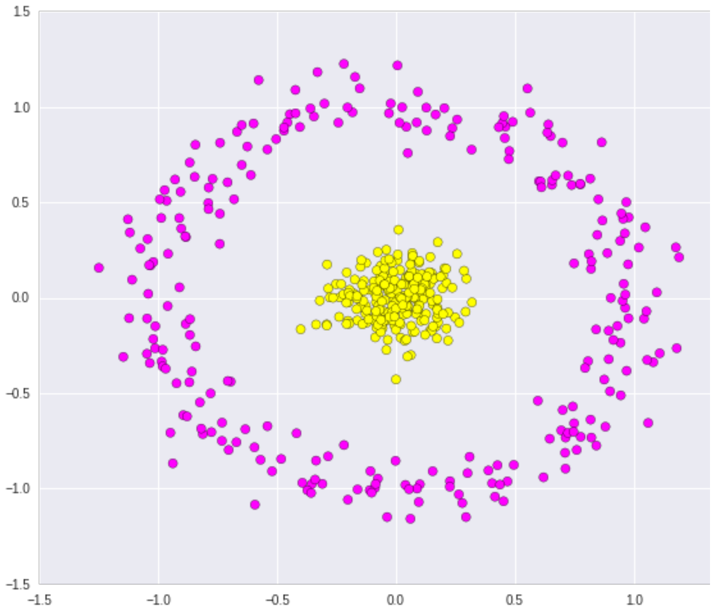




Solution:

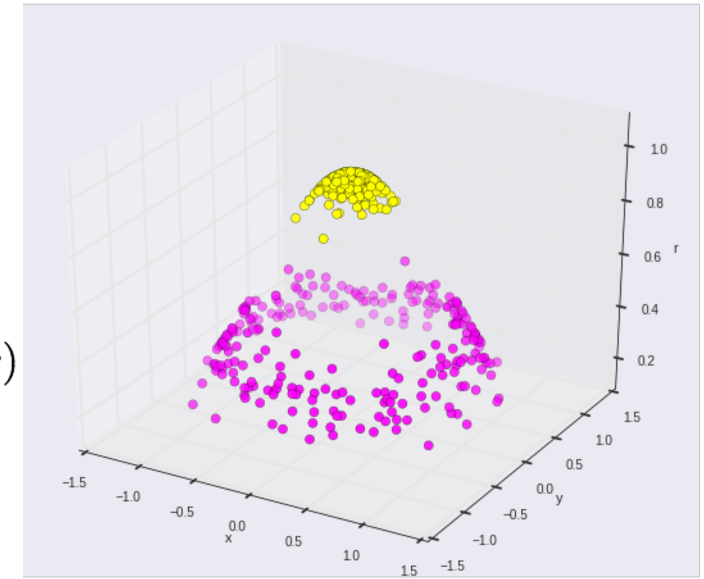
1) Map input space to a high-dimensional feature space.
2) Learn a linear decision boundary (hyperplane) in the high-dimensional space.
3) Map back to lower-dimensional space, giving a non-linear boundary.

Non-linear decision boundaries

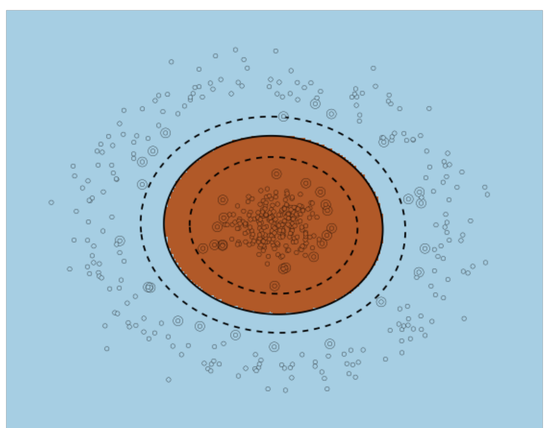


$$\phi : (x, y) \to (x, y, r)$$

$$r = e^{-(x^2+y^2)}$$



The resulting classifier perfectly separates the training data.



Non-linear decision boundaries

Non-linear QP problem: $\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_j \xi_j$ subject to: $y_j\left(w^T\boldsymbol{\Phi}\left(x_j\right) + b\right) \geq 1 - \xi_j$

$\xi_j \geq 0$

Problem: not efficiently computable, since $\boldsymbol{\Phi}\left(x_j\right)$ may be high- or infinite-dimensional!

Solution: transform to equivalent ("dual") QP problem:

$\min_{\alpha} \frac{1}{2}\alpha^T Q\alpha - \sum_j \alpha_j$ subject to: $0 \leq \alpha_j \leq C$ where: $Q_{ij} = y_i y_j(\boldsymbol{\Phi}(x_i)\cdot\boldsymbol{\Phi}\left(x_j\right))$

$\sum_j \alpha_j y_j = 0$

$= y_i y_j K(x_i, x_j)$

Very cool trick (the "kernel trick"): instead of mapping both x_i and x_j into a high-dimensional space and computing the dot product in that space, we can just compute a function $K(x_i, x_j)$ of the original data points.

This makes the QP efficiently solvable. To classify a test point x, we just need to compute $sign(\sum_j \alpha_j y_j K\left(x_j, x\right) + \rho)$.

Sum is just over the support vectors; other points have $\alpha_j = 0$.

Some common kernel functions

Linear kernel: $\phi : x \rightarrow x \qquad K(x_i, x_j) = x_i \cdot x_j$

Non-linear kernels

- Polynomial kernel: $K(x_i, x_j) = (\gamma(x_i \cdot x_j) + r)^d$
- Sigmoid kernel: $K(x_i, x_j) = \tanh(\gamma(x_i \cdot x_j) + r)$
- Gaussian kernel: $K(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right)$

The Gaussian kernel is usually called the "radial basis function", or **RBF**, kernel. It is one of the most widely used choices of kernel and a good default option.

Very cool trick (the "kernel trick"): instead of mapping both x_i and x_j into a high-dimensional space and computing the dot product in that space, we can just compute a function $K(x_i, x_j)$ of the original data points.

This makes the QP efficiently solvable. To classify a test point x, we just need to compute $sign(\sum_j \alpha_j y_j K(x_j, x) + \rho)$.

Sum is just over the support vectors; other points have $\alpha_j = 0$.

Variants and extensions of SVMs

SVMs are mainly used for **non-probabilistic, binary classification**.

To do multi-class classification:

For each class k, learn a binary classifier (class k vs. rest).

To predict the output for a new test example x, predict with each SVM.

Choose whichever one puts the prediction the furthest into the positive region.

To estimate class probabilities:

SVMs are not really the best for this, but can do logistic regression using outputs of k(k-1) pairwise SVMs.

Lots of models + additional cross-validation needed → this approach is very computationally expensive. (See Wu et al., 2004, for details.)

Support vector machines can also be used for **regression** (Smola and Schölkopf, 2003) and for **anomaly detection** (the “one-class SVM”, Schölkopf et al., 2001).

Both are implemented in scikit-learn, but are beyond the scope of this class.

Some advantages of SVMs

- Very good performance: though lately outshined by convolutional neural networks on some benchmarks (e.g., the MNIST digit recognition dataset) they often beat basically everything else.
- Theoretical guarantees about their generalization performance (accuracy for labeling test data) based on statistical learning theory.
- SVMs rely on convex optimization and do not get stuck in suboptimal local minima (neural networks have a big problem with these; similarly, decision trees rely on greedy search).
- Fairly robust to the curse of dimensionality → can effectively solve prediction problems with a large number of features.
- Flexible: can choose kernel to fit very complex decision boundaries.
- Will generally avoid overfitting with well-chosen parameters (but can certainly overfit for poorly chosen values, e.g., if C is too large).
- Classification of test points relies only on the support vectors → fast and memory efficient, especially when # of support vectors is small.

Some disadvantages of SVMs

- Training the model is **computationally expensive** – dependent on # of support vectors, but typically quadratic to cubic in the number of data points.
- Sensitive to choice of **parameters**, particularly the constant C and kernel bandwidth (γ for RBF kernel in sklearn).
    - C trades off misclassification rate against simplicity of the decision surface. Low C → smooth decision surface; High C → more training examples classified correctly.
    - Larger γ = lower bandwidth (increased weight on nearest training examples).
    - Proper choice of C and γ is critical to the SVM's performance.
    - For sklearn, use GridSearchCV with C and γ spaced exponentially far apart.
- Not much **interpretability** for non-linear SVM: can enumerate the support vectors or (in low dimensions) visualize the decision boundary, but actually obtaining these involves calling a black-box optimization routine.

References

- Scikit-learn documentation: http://scikit-learn.org/stable/modules/svm.html
- C.J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining & Knowledge Discovery*, 2: 955-974, 1998. http://research.microsoft.com/en-us/um/people/cburges/papers/svmtutorial.pdf
- A.W. Moore. Support Vector Machines (tutorial slides). https://www.autonlab.org/tutorials/svm.html
- V. Vapnik. *Statistical Learning Theory*. Wiley: 1998.
- T.-F. Wu, C.-J. Lin, and R.C. Weng. Probability estimates for multi-class classification by pairwise coupling. *Journal of Machine Learning Research* 5: 975-1005, 2004.
- A.J. Smola and B. Schölkopf. A tutorial on support vector regression, *Statistics and Computing*, 2003. http://alex.smola.org/papers/2003/SmoSch03b.pdf
- B. Schölkopf et al. Estimating the support of a high-dimensional distribution. *Neural Computation* 13: 1443-1471, 2001.

Up next: a short break, and then Python examples for support vector machines.