

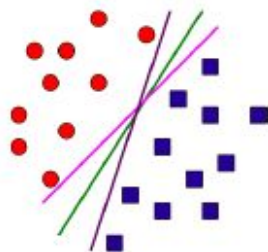
Support Vector Machines

Learning Maximum-Margin Hyperplanes:



Perceptron and (Lack of) Margins

Perceptron learns a hyperplane (of many possible) that separates the classes



Standard Perceptron doesn't guarantee any “margin” around the hyperplane

Note: Possible to “artificially” introduce a margin in the Perceptron

- Simply change the Perceptron mistake condition to

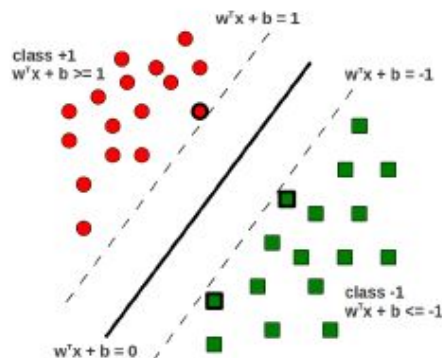
$$y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq \gamma$$

where $\gamma > 0$ is a **pre-specified margin**. For standard Perceptron, $\gamma = 0$

- **Support Vector Machine (SVM)** offers a more principled way of doing this by learning the **maximum margin hyperplane**

Support Vector Machine (SVM)

Learns a hyperplane such that the positive and negative class training examples are as far away as possible from it (ensures good generalization)

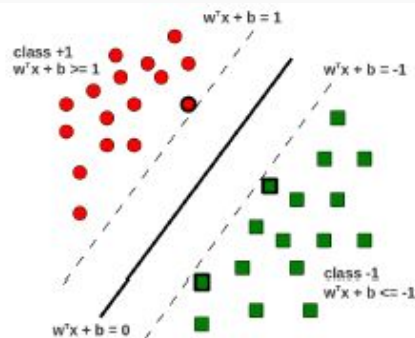


SVMs can also learn nonlinear decision boundaries using **kernels** (though the idea of kernels is not specific to SVMs and is more generally applicable)

Reason behind the name "Support Vector Machine"? SVM finds the most important examples (called "support vectors") in the training data

- These examples also "balance" the margin boundaries (hence called "support"). Also, even if we throw away the remaining training data and re-learn the SVM classifier, we'll get the same hyperplane.

Learning a Maximum Margin Hyperplane



Suppose there exists a hyperplane $\mathbf{w}^T \mathbf{x} + b = 0$ such that

- $\mathbf{w}^T \mathbf{x}_n + b \geq 1$ for $y_n = +1$
- $\mathbf{w}^T \mathbf{x}_n + b \leq -1$ for $y_n = -1$
- Equivalently, $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 \quad \forall n$ (the margin condition)
- Also note that $\min_{1 \leq n \leq N} |\mathbf{w}^T \mathbf{x}_n + b| = 1$
- Thus margin on each side: $\gamma = \min_{1 \leq n \leq N} \frac{|\mathbf{w}^T \mathbf{x}_n + b|}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}$
- Total margin $= 2\gamma = \frac{2}{\|\mathbf{w}\|}$

Want the hyperplane (\mathbf{w}, b) to have the largest possible margin

Large Margin = Good Generalization

Large margins intuitively mean good generalization

We saw that margin $\gamma \propto \frac{1}{\|\mathbf{w}\|}$

Large margin \Rightarrow small $\|\mathbf{w}\|$, i.e., small ℓ_2 norm of \mathbf{w}

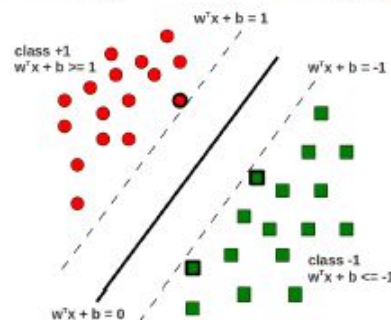
Small $\|\mathbf{w}\| \Rightarrow$ regularized/simple solutions (w_i 's don't become too large)

- Recall our discussion of regularization..

Simple solutions \Rightarrow good generalization on test data

Hard-Margin SVM

Every training example has to fulfil the margin condition $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1$



Also want to maximize the margin $\gamma \propto \frac{1}{\|\mathbf{w}\|}$

- Equivalent to minimizing $\|\mathbf{w}\|^2$ or $\frac{\|\mathbf{w}\|^2}{2}$

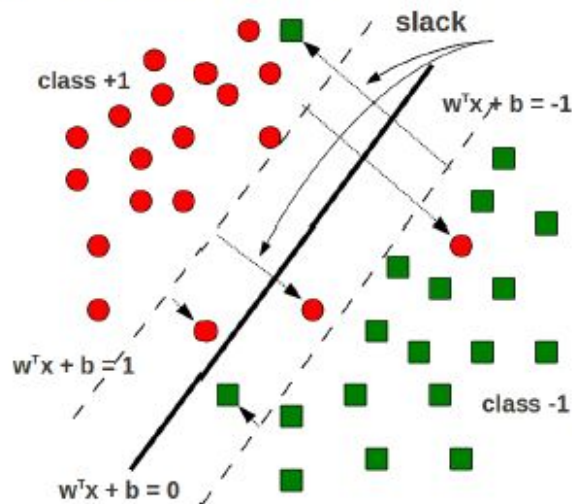
The objective for hard-margin SVM

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & f(\mathbf{w}, b) = \frac{\|\mathbf{w}\|^2}{2} \\ \text{subject to} \quad & y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1, \quad n = 1, \dots, N \end{aligned}$$

Thus the hard-margin SVM minimizes a **convex objective function** which is a **Quadratic Program** (QP) with N linear inequality constraints

Soft-Margin SVM (More Commonly Used)

Allow some training examples to fall **within the margin region**, or be even **misclassified** (i.e., fall on the wrong side). Preferable if training data is noisy

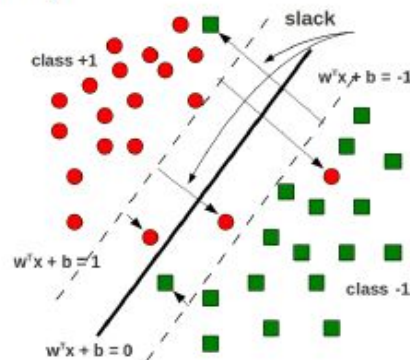


Each training example (\mathbf{x}_n, y_n) given a “slack” $\xi_n \geq 0$ (distance by which it “violates” the margin). If $\xi_n > 1$ then \mathbf{x}_n is totally on the wrong side

- Basically, we want a **soft-margin condition**: $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0$

Soft-Margin SVM (More Commonly Used)

Goal: Maximize the margin, while also minimizing the **sum of slacks** (don't want too many training examples violating the margin condition)



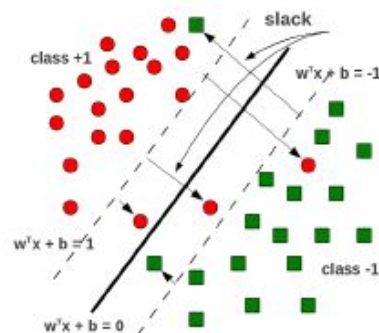
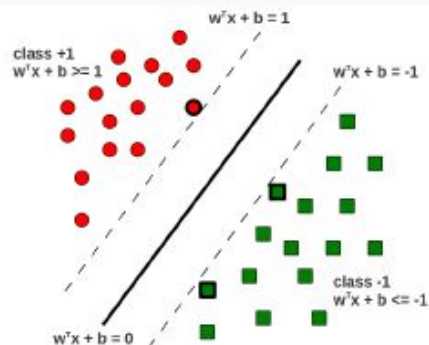
The primal objective for soft-margin SVM can thus be written as

$$\begin{aligned} \min_{\mathbf{w}, b, \xi} \quad & f(\mathbf{w}, b, \xi) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n \\ \text{subject to constraints} \quad & y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0 \quad n = 1, \dots, N \end{aligned}$$

Thus the soft-margin SVM also minimizes a **convex objective function** which is a **Quadratic Program (QP)** with $2N$ linear inequality constraints

Param. C controls the trade-off between large margin vs small training error

Summary: Hard-Margin SVM vs Soft-Margin SVM



- Objective for the hard-margin SVM (unknowns are \mathbf{w} and b)

$$\min_{\mathbf{w}, b} f(\mathbf{w}, b) = \frac{\|\mathbf{w}\|^2}{2}$$

$$\text{subject to constraints } y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1, \quad n = 1, \dots, N$$

- Objective for the soft-margin SVM (unknowns are \mathbf{w} , b , and $\{\xi_n\}_{n=1}^N$)

$$\min_{\mathbf{w}, b, \xi} f(\mathbf{w}, b, \xi) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n$$

$$\text{subject to } y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0 \quad n = 1, \dots, N$$

- In either case, we have to solve constrained, convex optimization problem

SVM: Some Notes

A hugely (perhaps the most!) popular classification algorithm

Reasonably mature, highly optimized SVM softwares freely available (perhaps the reason why it is more popular than various other competing algorithms)

- Some popular ones: libSVM, LIBLINEAR, SVMStruct, Vowpal Wabbit, etc.

Lots of work on scaling up SVMs[†] (both large N and large D)

Extensions beyond binary classification (e.g., multiclass, structured outputs)

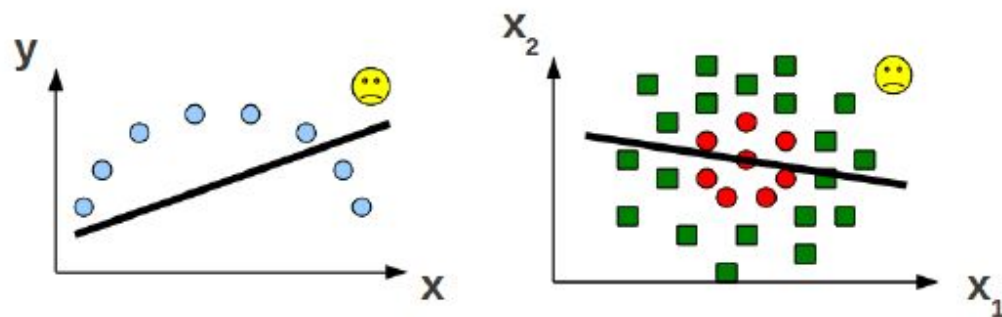
Can even be used for regression problems (Support Vector Regression)

Nonlinear extensions possible via kernels

Nonlinear SVM (Kernels)

Linear Models

Linear models (e.g., linear regression, linear SVM) are nice and interpretable but have limitations. Can't learn "difficult" nonlinear patterns.




Reason: Linear models rely on "linear" notions of similarity/distance

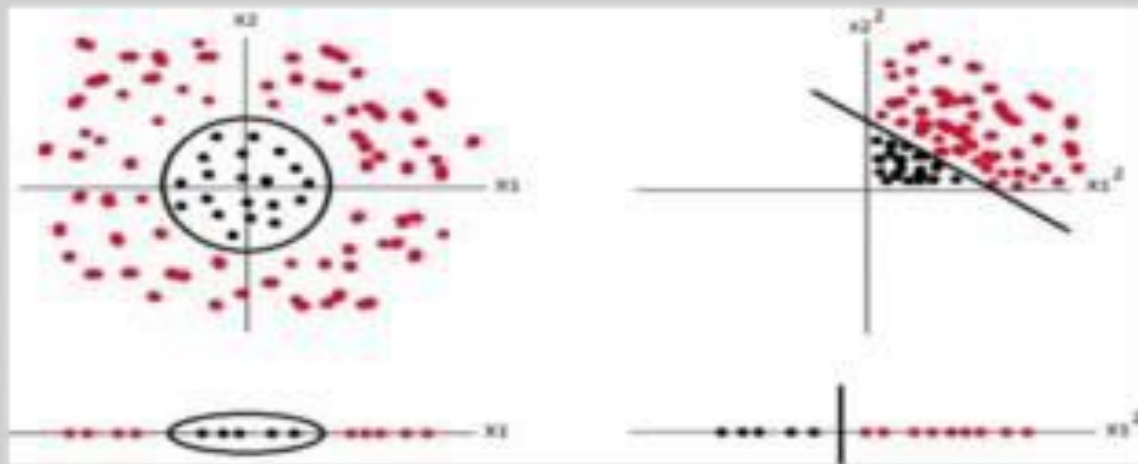
$$Sim(\mathbf{x}_n, \mathbf{x}_m) = \mathbf{x}_n^\top \mathbf{x}_m$$

$$Dist(\mathbf{x}_n, \mathbf{x}_m) = (\mathbf{x}_n - \mathbf{x}_m)^\top (\mathbf{x}_n - \mathbf{x}_m)$$

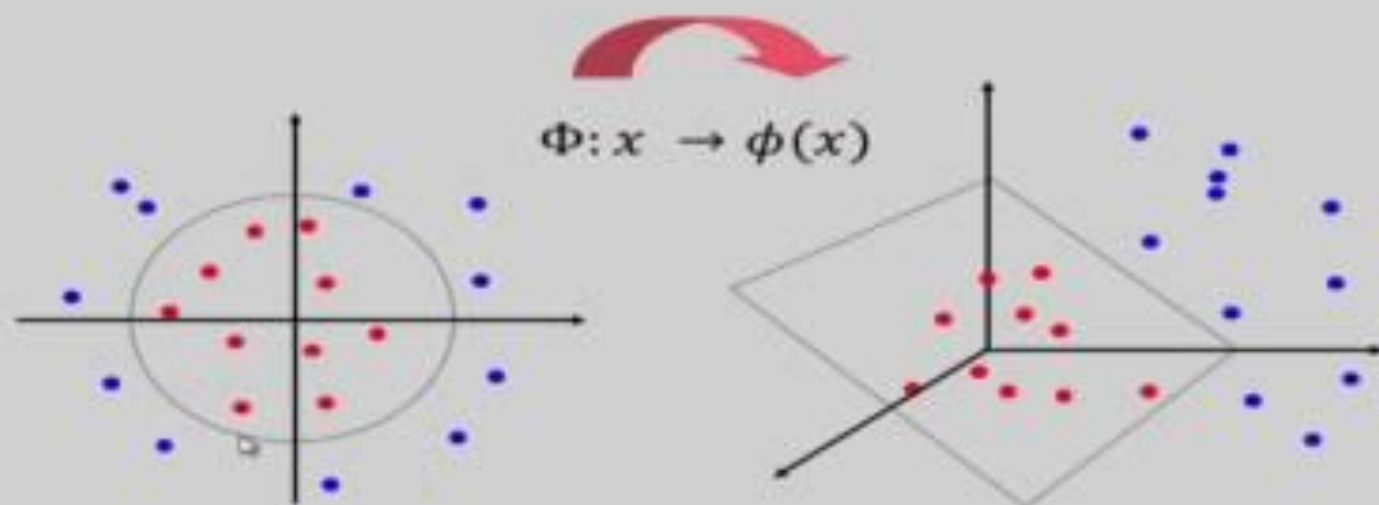
.. which wouldn't work well if the patterns we want to learn are nonlinear

Non-linear SVMs: Feature Space


$$\Phi: x \rightarrow \phi(x)$$



Non-linear SVMs: Feature Space



Kernel

- Original input attributes is mapped to a new set of input features via feature mapping Φ .
- Since the algorithm can be written in terms of the scalar product, we replace x_a, x_b with $\phi(x_a), \phi(x_b)$
- For certain Φ 's there is a simple operation on two vectors in the low-dim space that can be used to compute the scalar product of their two images in the high-dim space

$$K(x_a, x_b) = \phi(x_a) \cdot \phi(x_b)$$

Let the kernel do the work rather than do the scalar product in the high dimensional space.

Nonlinear SVMs: The Kernel Trick

- With this mapping, our discriminant function is now:

$$g(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b = \sum_{i \in SV} \alpha \boxed{\phi(\mathbf{x}_i)^T \phi(\mathbf{x})} + b$$

- We only use the dot product of feature vectors in both the training and test.
- A *kernel function* is defined as a function that corresponds to a dot product of two feature vectors in some expanded feature space:

$$K(x_a, x_b) = \phi(x_a) \cdot \phi(x_b)$$

The kernel trick

$$K(x_a, x_b) = \phi(x_a) \cdot \phi(x_b)$$

Often $K(x_a, x_b)$ may be very inexpensive to compute even if $\phi(x_a)$ may be extremely high dimensional.

Kernel Example

2-dimensional vectors $\bar{x} = [x_1 x_2]$

let $K(x_i, x_j) = (1 + x_i \cdot x_j)^2$

We need to show that $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$

$$K(x_i, x_j) = (1 + x_i \cdot x_j)^2$$

$$= 1 + x_{i1}^2 x_{j1}^2 + 2 x_{i1} x_{j1} x_{i2} x_{j2} + x_{i2}^2 x_{j2}^2 + 2 x_{i1} x_{j1} + 2 x_{i2} x_{j2}$$

$$= [1 \ x_{i1}^2 \ \sqrt{2} \ x_{i1} x_{i2} \ x_{i2}^2 \ \sqrt{2} x_{i1} \ \sqrt{2} x_{i2}] \cdot [1 \ x_{j1}^2 \ \sqrt{2} \ x_{j1} x_{j2} \ x_{j2}^2 \ \sqrt{2} x_{j1} \ \sqrt{2} x_{j2}]$$

$$= \phi(x_i) \cdot \phi(x_j),$$

$$\text{where } \phi(x) = [1 \ x_1^2 \ \sqrt{2} \ x_1 x_2 \ x_2^2 \ \sqrt{2} x_1 \ \sqrt{2} x_2]$$

Commonly-used kernel functions

- Linear kernel: $K(x_i, x_j) = x_i \cdot x_j$

- Polynomial of power p :

$$K(x_i, x_j) = (1 + x_i \cdot x_j)^p$$

- Gaussian (radial-basis function):

$$K(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$$

- Sigmoid

$$K(x_i, x_j) = \tanh(\beta_0 x_i \cdot x_j + \beta_1)$$

In general, functions that satisfy *Mercer's condition* can be kernel functions.

Performance

- Support Vector Machines work very well in practice.
 - The user must choose the kernel function and its parameters
- They can be expensive in time and space for big datasets
 - The computation of the maximum-margin hyper-plane depends on the *square of the number of training cases*.
 - We need to store all the support vectors.
- The kernel trick can also be used to do PCA in a much higher-dimensional space, thus giving a non-linear version of PCA in the original space.

Multi-class classification

- SVMs can only handle two-class outputs
- Learn N SVM's
 - SVM 1 learns Class1 vs REST
 - SVM 2 learns Class2 vs REST
 - :
 - SVM N learns ClassN vs REST
- Then to predict the output for a new input, just predict with each SVM and find out which one puts the prediction the furthest into the positive region.