Elements of Linear Discriminant Functions

Battista Biggio

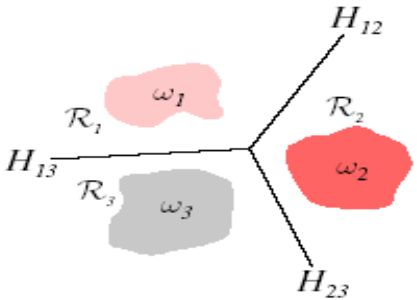
Department of Electrical and Electronic Engineering
University of Cagliari, Italy

Introduction

- We assume here that the form of the discriminant functions $f_k(\boldsymbol{x};\boldsymbol{\theta}),\ k = 1, \ldots, K$ is given, and that we can use the training data to estimate their parameters $\boldsymbol{\theta}$
  - In Part 4, instead, we assumed that the underlying probability densities were known

- These methods are known as *nonparametric*
  - No assumption on the form of the underlying data probability distributions is made

- We focus here on functions that are *linear* in $\boldsymbol{x}$, i.e., $f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{w}^T\boldsymbol{x} + b$, with $\boldsymbol{\theta} = (\boldsymbol{w}, b)$

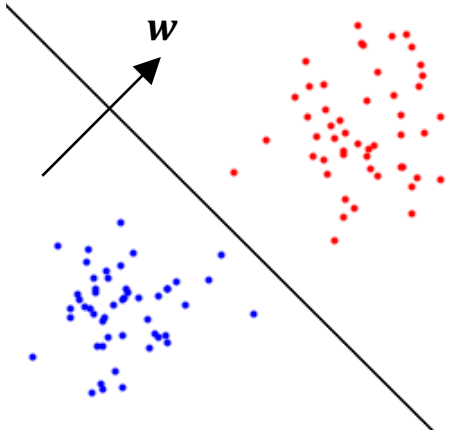


Example of linear discriminant functions on a 3-class classification problem

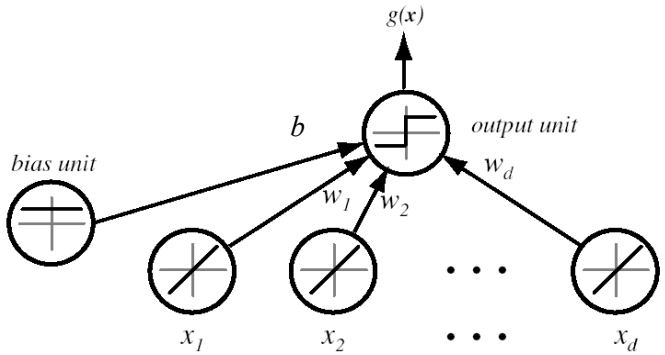
Linear Discriminant Functions

- **Linear function**: $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b = \sum_{j=1}^{d} w_j x_j + b$
  - $\boldsymbol{w}$ is the weight vector, and b the bias

- Two-class classification
  - Positive ($y = +1$) vs negative ($y = -1$) class
  - Decision rule: $y = \begin{cases} +1 & \text{if } f(\boldsymbol{x}) \geq 0 \\ -1, & \text{otherwise} \end{cases}$

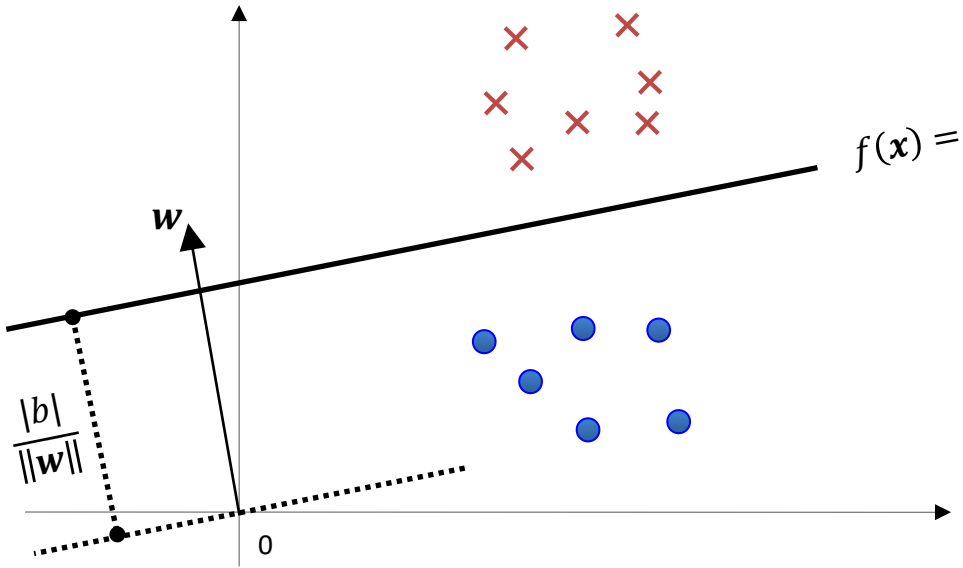


- **Graphical representation**
  - Each input feature value x_j is multiplied by the corresponding weight value w_j
  - Bias is multiplied by 1
  - The output unit sums all its inputs, computing $f(\boldsymbol{x})$ and thresholds it to estimate y (+1 or -1)



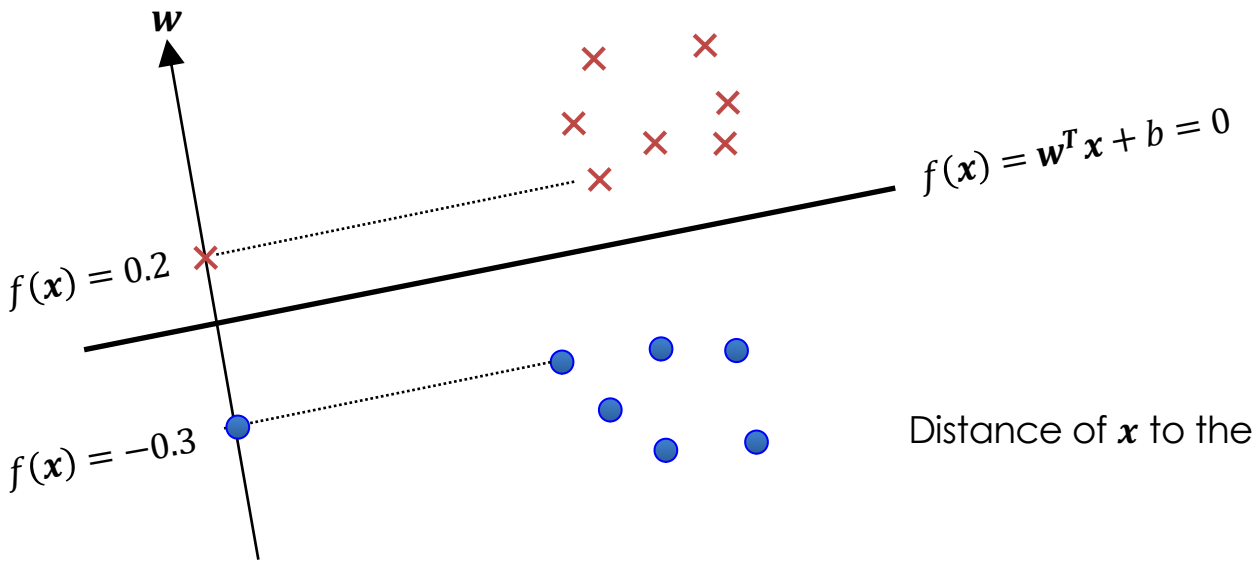
Linear Discriminant Functions

$\mathrm{D} = \{\boldsymbol{x}_i, y_i\}_{i=1}^{n}$
$\boldsymbol{x} \in \mathbb{R}^d$
$y \in \{-1, +1\}$



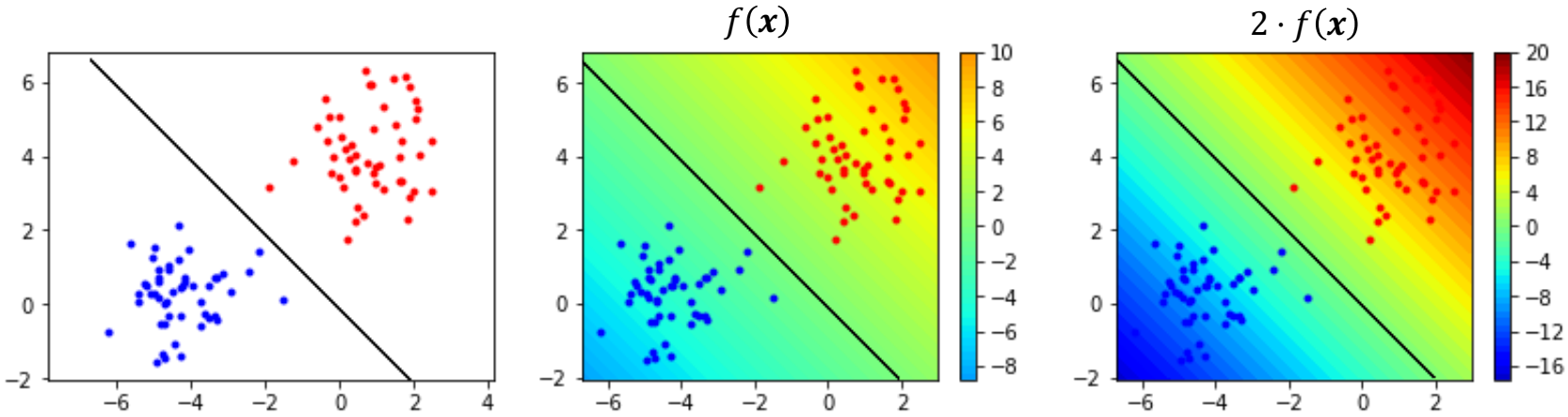
Linear Discriminant Functions

- The function $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$ projects $\boldsymbol{x}$ onto the hyperplane normal
  - Its value is proportional to the distance of $\boldsymbol{x}$ to the hyperplane



Distance of $\boldsymbol{x}$ to the hyperplane: $\frac{|f(\boldsymbol{x})|}{\|\boldsymbol{w}\|}$

Linear Discriminant Functions



- By linearity, it is easy to see that multiplying $f(\boldsymbol{x})$ by a constant factor amounts to multiplying its parameters by the same factor
  - For example: $2 \cdot f(\boldsymbol{x}) = 2\,\boldsymbol{w}^T\boldsymbol{x} + 2b = \hat{\boldsymbol{w}}^T\boldsymbol{x} + \hat{b}$, with $\hat{\boldsymbol{w}} = 2\boldsymbol{w}$ and $\hat{b} = 2b$

- While the boundary at $f(\boldsymbol{x}) = 0$ does not change, the slope of the function changes!

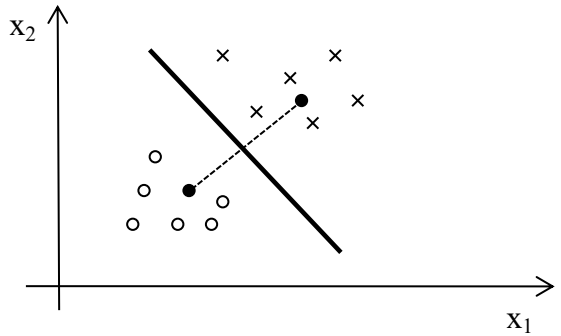
A Simple Example: The Nearest Mean Classifier (NMC)

- This classifier estimates the mean values $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ of the two classes from the training set and assigns unknown samples $\boldsymbol{x}^\star$ to the class with the smallest Euclidean distance:

$$d(\mathbf{x}^*, \boldsymbol{\mu}_2) \underset{\omega_2}{\overset{\omega_1}{\lessgtr}} d(\mathbf{x}^*, \boldsymbol{\mu}_1)$$

- It is easy to see that the decision boundary is the hyperplane perpendicular to the vector $(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$ and passing through the mean point $(\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2)/2$

- Accordingly, the NMC is a linear classifier
  - Try to find its $\boldsymbol{w}$ and b parameter values!



Learning as an Optimization Problem

Learning as an Optimization Problem

- How do we estimate the classifier parameters $\boldsymbol{w}$ and b?

- Modern approaches formulate the learning problem as an **optimization problem**
  - This is generally true also for nonlinear classification functions $f(\boldsymbol{x}; \boldsymbol{\theta})$, including modern deep-learning approaches and neural networks

$$\boldsymbol{w}^\star, b^\star = \underset{\boldsymbol{w},b}{\operatorname{argmin}} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(\boldsymbol{x}_i))}_{\text{loss term } L(D,\boldsymbol{\theta})} + \underbrace{\lambda\, \Omega(\boldsymbol{w})}_{\text{regularization term } \Omega(\boldsymbol{w})}$$

λ: regularization hyperparameter

Learning as an Optimization Problem

- The loss function $\ell(y_i, f(\boldsymbol{x}_i))$ measures how much a prediction is wrong
  - e.g., the zero-one loss is 0 if points are correctly predicted, and 1 if they are not

- The regularization term $\Omega(\boldsymbol{\theta})$ imposes a penalty on the magnitude of the classifier parameters to avoid overfitting and promote smoother functions, i.e., functions that change more gradually as we move across the feature space

- The hyperparameter λ tunes the trade-off between the training loss and regularization
  - Larger values tend to promote more regularized functions but with a larger training error
  - Smaller values tend to reduce the training error but learn more complex functions

Learning as an Optimization Problem

- We start by considering a simplified setting in which we aim to find the best parameters $\boldsymbol{\theta} = (\boldsymbol{w}, b)$ that minimize the loss function $L(D, \boldsymbol{\theta})$, being $D = (\boldsymbol{x}_i, y_i)_{i=1}^{n}$ the training dataset:

$$\boldsymbol{\theta}^* = \text{argmin}_{\boldsymbol{\theta}} \, L(D, \boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta}))$$

- The loss function quantifies the error that the classifier, parameterized by $\boldsymbol{\theta}$, is making on its predictions on the training data D
  - This is also known as the principle of **Empirical Risk Minimization (ERM)**

- How do we select the loss function $L(D, \boldsymbol{\theta})$ and solve the above problem?

Learning as an Optimization Problem

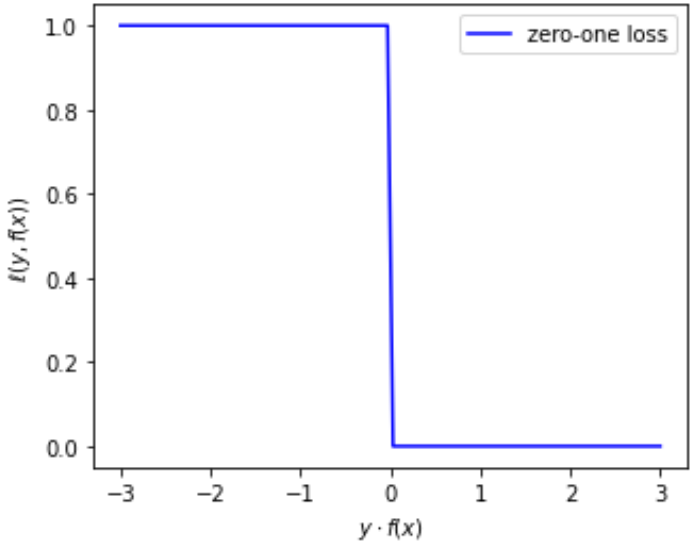
- In principle, we would like to minimize

$$L(D, \boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta}))$$

- where $\ell(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta}))$ is the zero-one loss
  - equal to 0 for correct predictions and 1 otherwise

$$\ell\big(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta})\big) = \begin{cases} 1, & \text{if } \; y \cdot f(\boldsymbol{x}) < 0 \\ 0, & \text{if } \; y \cdot f(\boldsymbol{x}) \geq 0 \end{cases}$$

- However, solving this problem directly is NP-hard and computationally inefficient
  - https://en.wikipedia.org/wiki/NP-hardness

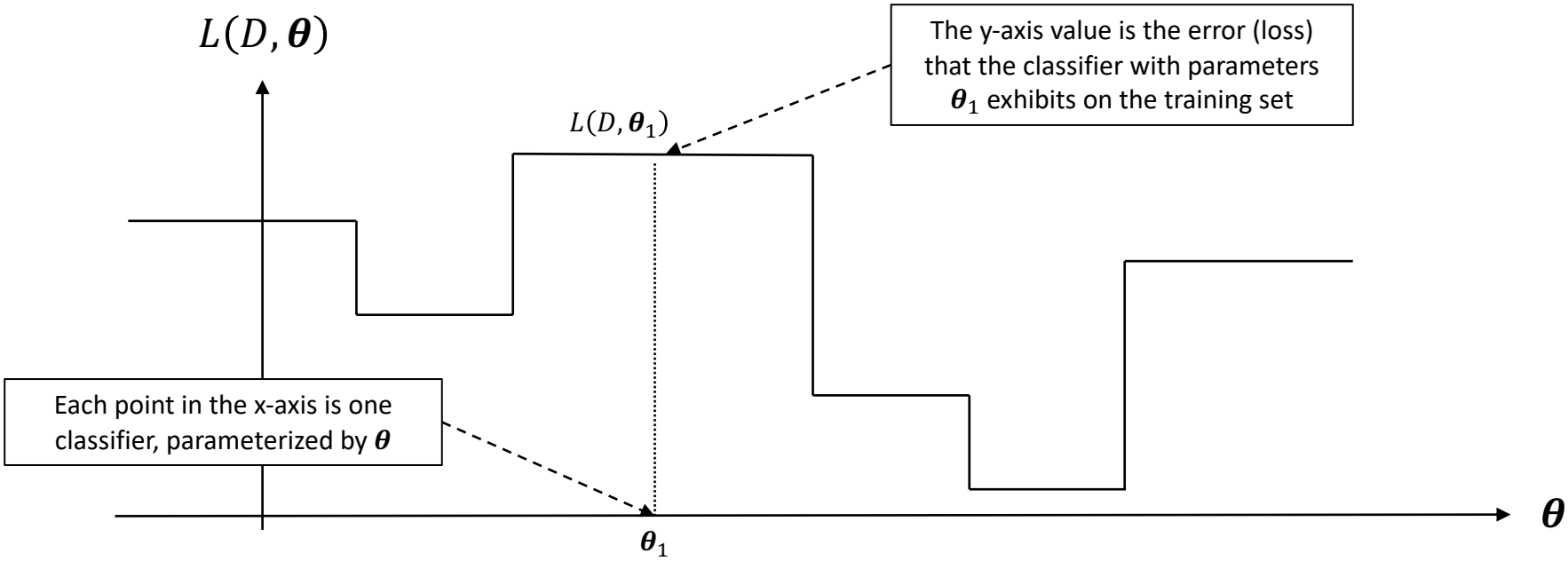


Note that $yf(\boldsymbol{x})$ is positive for correct predictions and negative for wrong ones.

For correct (wrong) predictions, y and $f(\boldsymbol{x})$ agree (disagree) in sign.

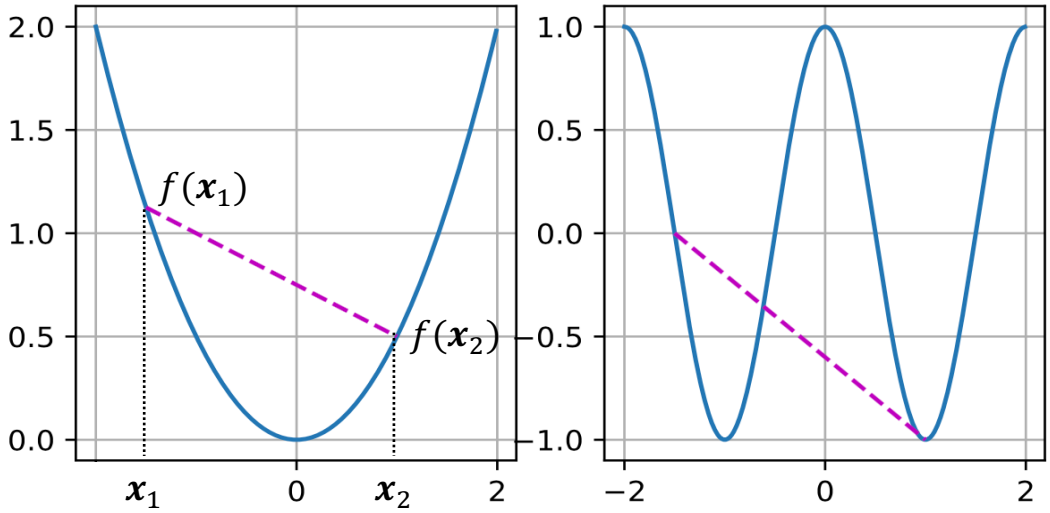
The Zero-One Loss Landscape

- Non convex, hard to optimize (flat regions, bad local minima)



Why Is Convexity Important for Optimization?

- **Convexity**: $f(\lambda \boldsymbol{x}_1 + (1-\lambda)\boldsymbol{x}_2) \leq \lambda f(\boldsymbol{x}_1) + (1-\lambda) f(\boldsymbol{x}_2), \quad \forall \boldsymbol{x}_1, \boldsymbol{x}_2, \quad \lambda \in [0,1]$



- **Desirable properties for optimization:** no local minima, convergence guarantees, etc.

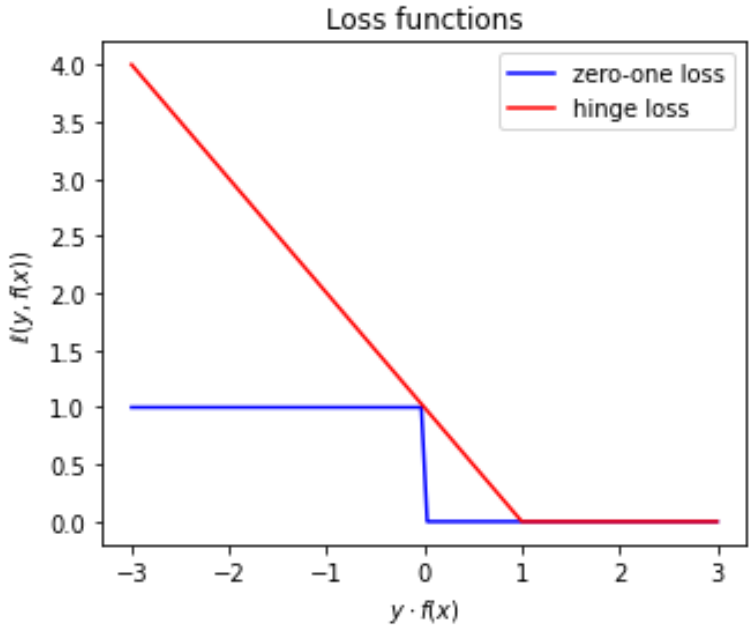
https://d2l.ai/chapter_optimization/convexity.html

Loss Functions

- Now, recall that we aim to minimize:

$$L(D, \boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta}))$$

- being $\ell(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta}))$ the zero-one loss
- However, we know that minimizing this non-convex function is particularly difficult (NP hard)
- For this reason, convex (surrogate) loss functions are typically preferred
  - The tighter convex upper bound on the zero-one loss is called the **hinge loss**

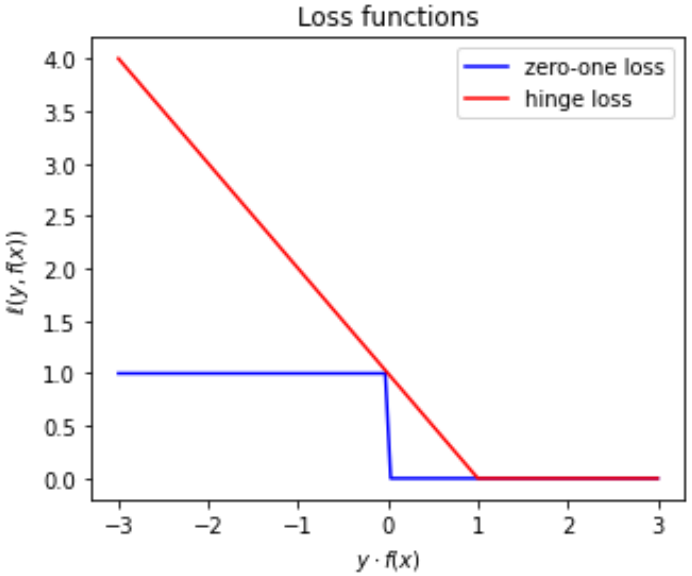


Hinge Loss

- The hinge loss is computed as:

$$\ell\big(y, f(\boldsymbol{x};\theta)\big) = \max(0, 1 - yf)$$

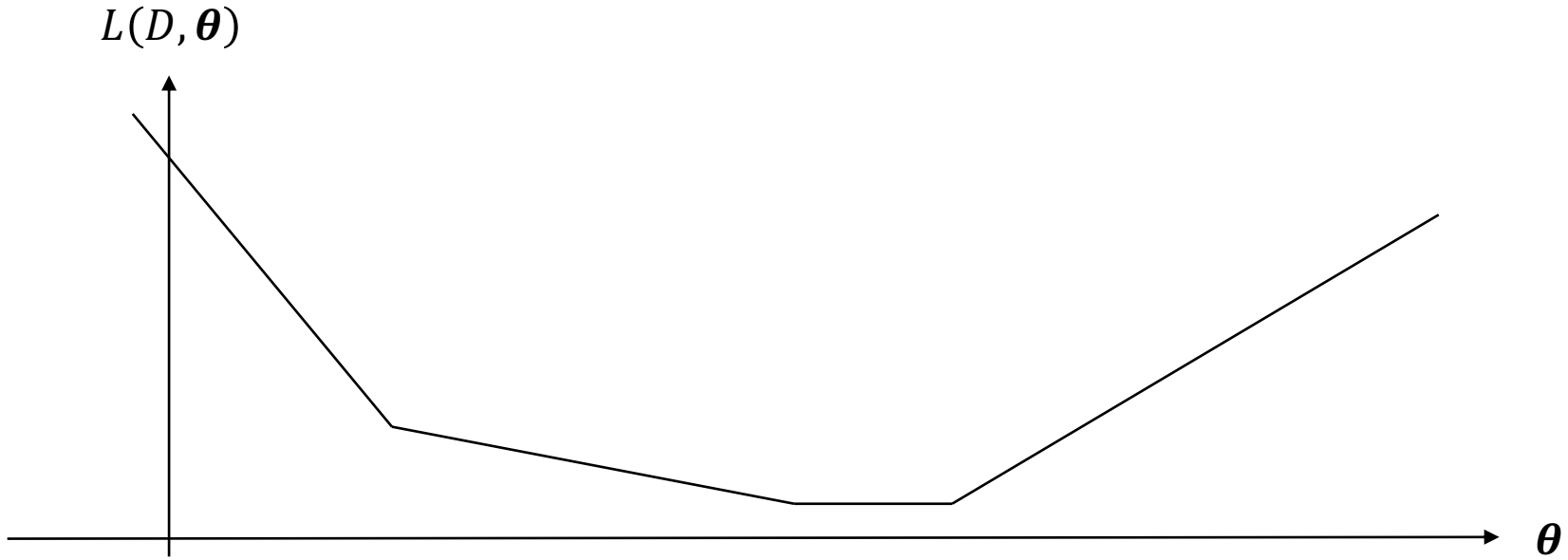
- It is the closest convex upper bound on the 0-1 loss
- Why are we interested in an upper bound?
  - since minimizing it, we also minimize the 0-1 loss
- Convexity helps find solutions efficiently while also providing guarantees on the optimality of the solution (global optima), algorithmic convergence, etc.
  - This is why convex surrogate functions are typically preferred in optimization



Note that $yf(\boldsymbol{x})$ is positive for correct predictions and negative for wrong ones

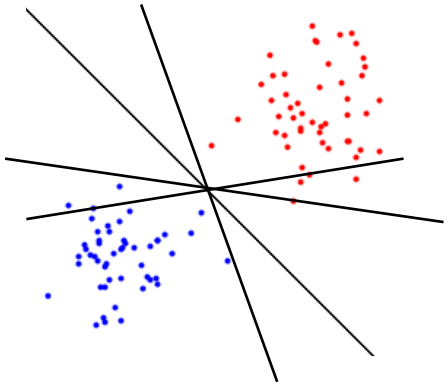
The Hinge Loss Landscape

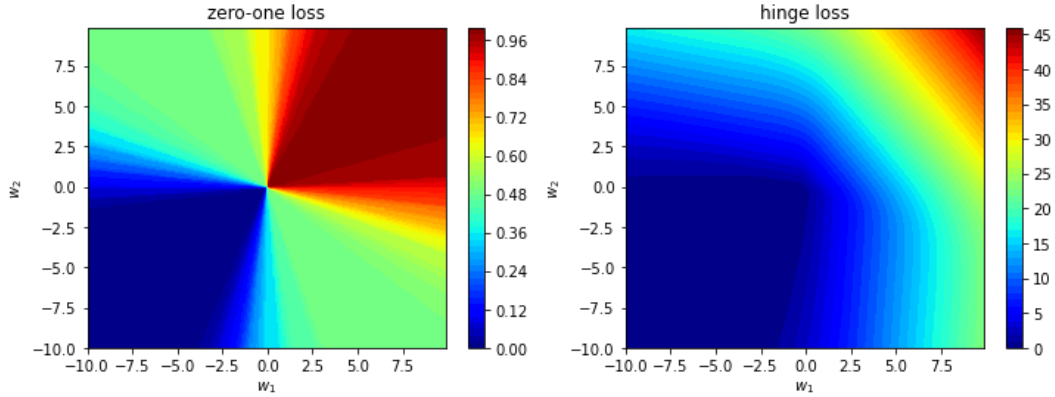
- Piecewise linear and convex, easier to optimize



A Closer Look at the Loss Minimization Problem

- Let's assume we fix $b = 0$ and aim to minimize the training loss only w.r.t. w_1, w_2
- Each pair w_1, w_2 thus represents a different linear classifier (passing through the origin)
- For each of these classifiers, we report the corresponding training loss in a colored plot
  - this will show us the optimization *landscape*, i.e., the surface of the function we aim to minimize

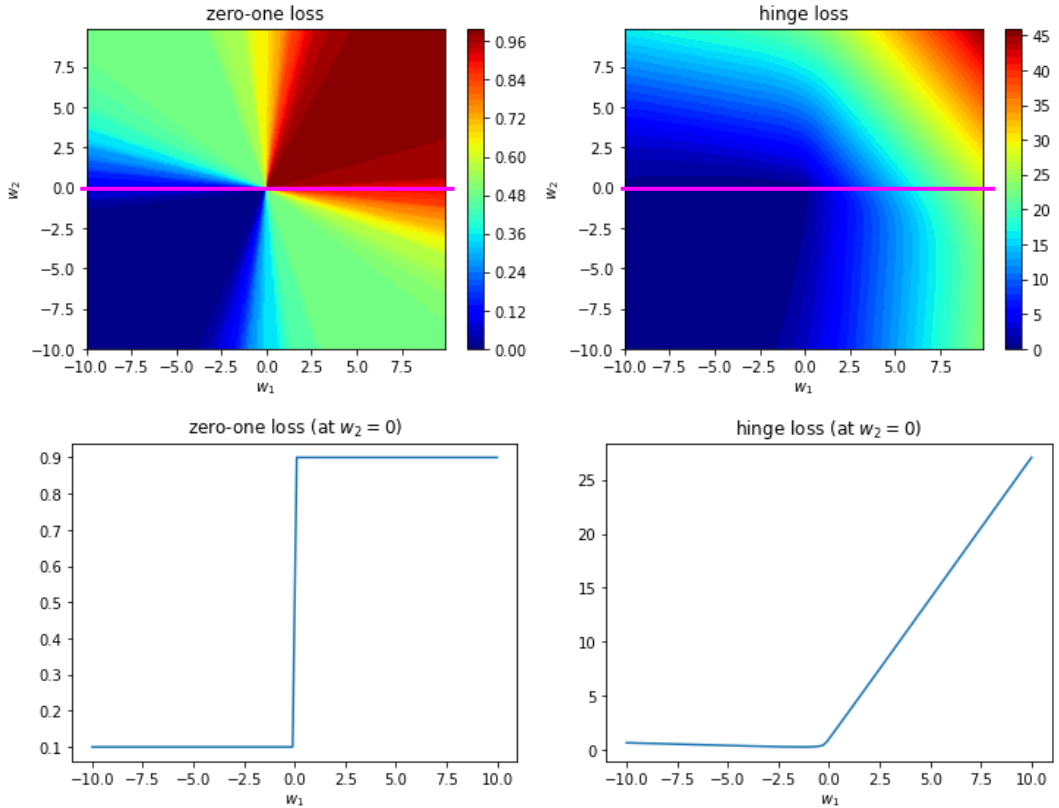




Open in Colab https://colab.research.google.com/github/unica-ml/ml/blob/master/notebooks/ml06.ipynb https://unica-ml.github.io

A Closer Look at the Loss Minimization Problem

- If we also fix $w_2 = 0$, and consider only optimizing w_1, we can also look at the profile of the loss along the line $w_2 = 0$
- Note how the zero-one loss again reports a sharp profile (behaving like a step function)
- The hinge loss instead decreases more gracefully towards one direction
- We can thus say that the hinge loss provides a **smooth** approximation to the 0-1 loss



Open in Colab https://colab.research.google.com/github/unica-ml/ml/blob/master/notebooks/ml06.ipynb

Loss Minimization with Gradient Descent

Gradient-based Optimization

- Optimizing *smooth* functions is much easier and efficient, as we can exploit **gradients**
  - This is not possible for the 0-1 loss (it is flat almost everywhere with gradients equal to zero)

- The key idea of gradient-based optimization is to start from a random point in the parameter space (*random initialization*) and then iteratively update the parameters along the gradient direction

- The gradient is the derivative of the loss function w.r.t. the classifier parameters:

$$L(D,\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(\boldsymbol{x}_i;\theta)), \qquad \nabla_\theta L = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \ell(y_i, f(\boldsymbol{x}_i;\theta))$$

- It is the direction in the parameter space along which the objective maximally increases
  - Following the negative gradient will thus minimize our training loss!

Gradient Descent (a.k.a. Steepest Descent)

- The simplest gradient-based optimizer is the **steepest-descent** method

1. initialize θ, η, K, ε
2. for k in {0, 1, ... , $K-1$}:
3. $\quad \boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla L(\boldsymbol{\theta}_k)$
4. $\quad$ if $|L(\boldsymbol{\theta}_k) - L(\boldsymbol{\theta}_{k+1})| < \varepsilon$:
5. $\quad\quad$ break

- The parameters $\boldsymbol{\theta}$ are updated at each iteration, until
  - a maximum of K iterations are reached, or
  - the convergence/stop condition is met (lines 4-5 above)

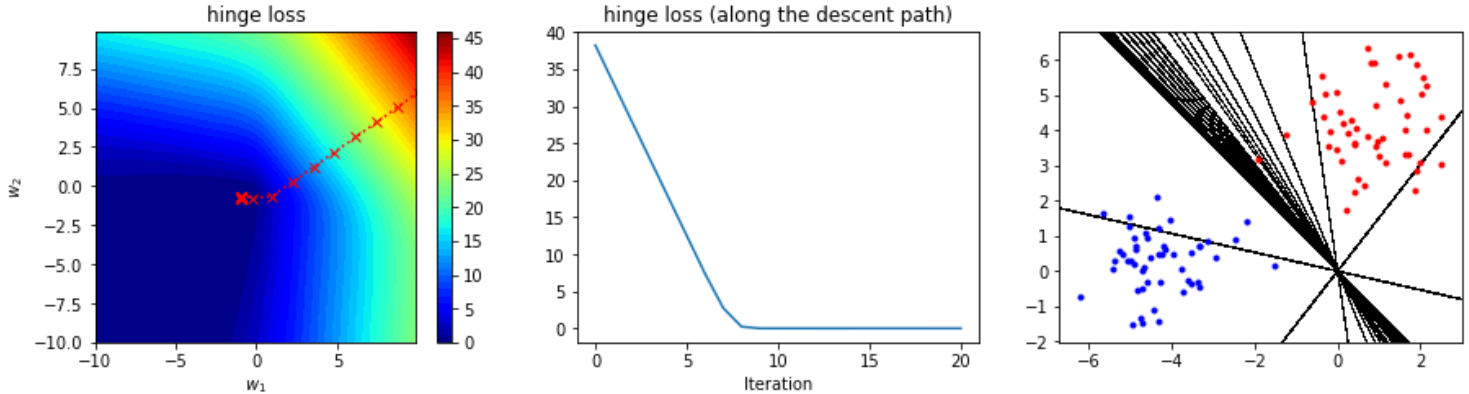
- The stop condition checks that the last update has not significantly modified the objective function (i.e., the training loss is almost constant, as ε is a small number)

- The learning rate (or gradient step size) η affects convergence. If it is too small, convergence is too slow; if it is too large, the gradient algorithm may not even converge at all. Usually η is reduced across iterations to ensure convergence

Steepest Descent on the Hinge Loss

- Consider again our running example. We start optimization from $w = (10, 6)$ and fix $b = 0$
  - See the notebook for details on the computation of the gradient of the hinge loss w.r.t. w, b

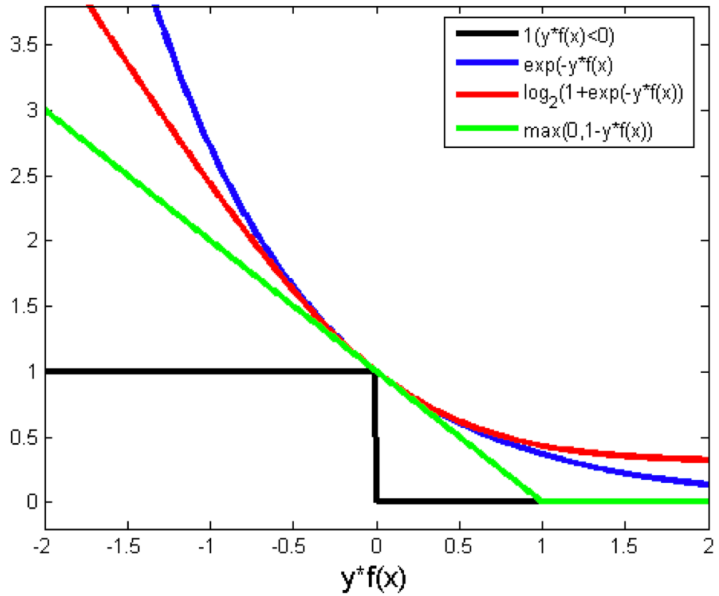
- Here's what the plots show:
  1. how weights change along the path until the local minimum (blue region) is reached
  2. how the hinge loss decreases along the path (convergence is reached when it becomes flat)
  3. how the linear classifier changes across iterations, converging to a solution that splits the training points achieving zero loss



Open in Colab https://colab.research.google.com/github/unica-ml/ml/blob/master/notebooks/ml06.ipynb https://unica-ml.github.io

Other Convex Losses

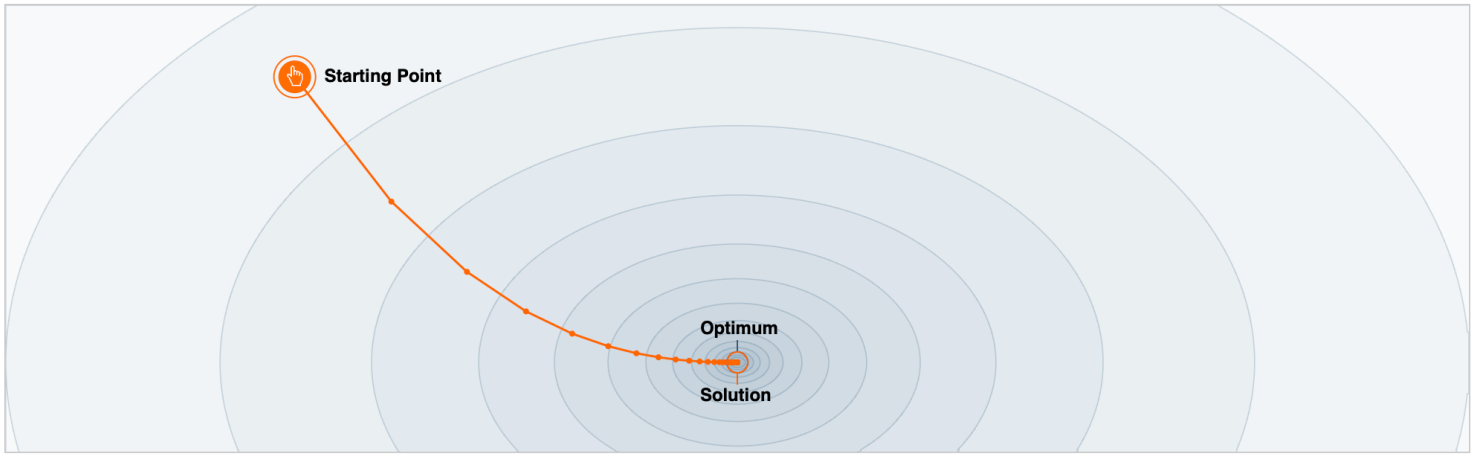
- Other convex upper bounds on the zero-one loss:
  - Hinge loss: $\ell\big(y, f(\boldsymbol{x})\big) = \max(0, 1 - yf)$
  - Exponential loss: $\ell\big(y, f(\boldsymbol{x})\big) = e^{-yf}$
  - Logistic loss: $\ell\big(y, f(\boldsymbol{x})\big) = \log_2(1 + e^{-yf})$



Gradient Descent: Step Size and Convergence

Example: Steepest Descent on Quadratic Objective

- Well-conditioned quadratic objective, small step size



Step-size α = 0.22

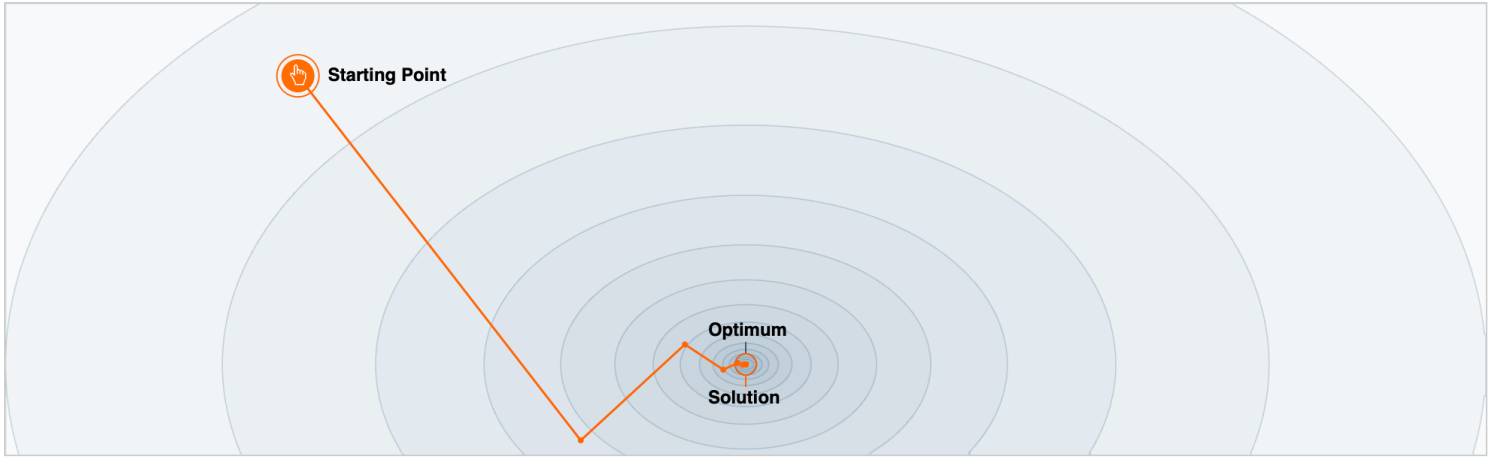
0 0.5 1.0

On a well-conditioned quadratic function, the gradient descent converges on few iterations to the optimum.

Examples from: http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

Example: Steepest Descent on Quadratic Objective

- Well-conditioned quadratic objective, large step size



Step-size α = 0.63

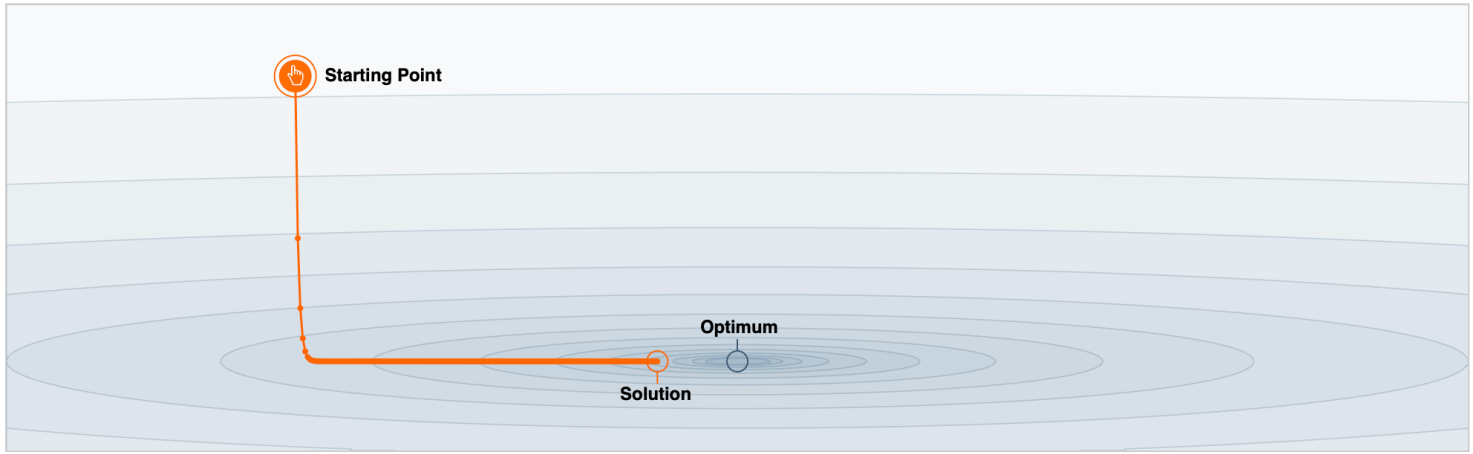
0 0.5 1.0

On a well-conditioned quadratic function, the gradient descent converges on few iterations to the optimum.

Examples from: http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

Example: Steepest Descent on Quadratic Objective

- Badly-conditioned quadratic objective, slow convergence



Step-size α = 0.63

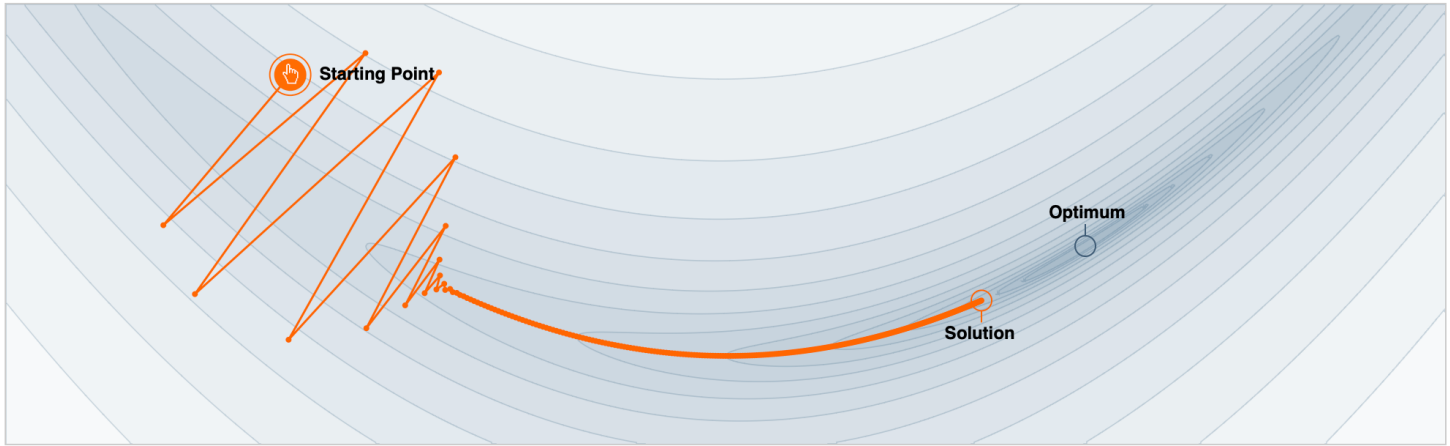
0 0.01 0.02

On a badly-conditioned quadratic function, the gradient descent converges takes many more iterations to converge than on the above well-conditioned problem. This is because gradient descent requires a much smaller step size on this problem to converge.

Examples from: http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

Example: Steepest Descent on Non-convex Objective

- Badly-conditioned non-convex objective, slow convergence and initial instability



Step-size α = 0.63

0 0.01 0.02

Gradient descent also converges on a badly-conditioned non-convex problem. Convergence is slow in this case.

Examples from: http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

Gradient Descent on Quadratic Objectives

Gradient Descent on Quadratic Objectives

- The steepest descent algorithm linearizes the function $L(D, \boldsymbol{\theta})$ around $\boldsymbol{\theta}$
  - It's a first-order Taylor expansion!
- If the function is quadratic or even non-convex, convergence may be too slow
- Alternatively, we can use a quadratic expansion for $L(D, \boldsymbol{\theta})$ around $\boldsymbol{\theta}$:

$$L(\boldsymbol{\theta}_{k+1}) \cong L(\boldsymbol{\theta}_k) + \nabla L(\boldsymbol{\theta}_k)(\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k) + \frac{1}{2}(\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k)^T \mathbf{H}_{\mathrm{k}}(\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k)$$

- Note that $\mathbf{H}_{\mathrm{k}} = \nabla^2_{\boldsymbol{\theta},\boldsymbol{\theta}} L(\boldsymbol{\theta}_k)$ is the Hessian matrix (containing the second derivatives)

Steepest Descent with Exact Step

- The quadratic expansion can be used to find the optimal step size η_k at each iteration
- If we plug-in the update rule: $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \nabla L(\boldsymbol{\theta}_k)$ in the previous expression, we obtain:

$$L(\boldsymbol{\theta}_{k+1}) \cong L(\boldsymbol{\theta}_k) - \eta_k \|\nabla L(\boldsymbol{\theta}_k)\|^2 + \frac{1}{2}\eta_k^2 \nabla L(\boldsymbol{\theta}_k)^T \mathbf{H}_\mathrm{k} \nabla L(\boldsymbol{\theta}_k)$$

- The minimization of the quadratic form of $L(\boldsymbol{\theta}_{k+1})$ derived before can be solved in closed form, by setting the derivate with respect to η_k equal to zero, and solving for η_k
- The solution provides the optimal step size to be set at each iteration:

$$\eta_k = \frac{\|\nabla L(\boldsymbol{\theta}_k)\|^2}{\nabla L(\boldsymbol{\theta}_k)^T \mathbf{H}_\mathrm{k} \nabla L(\boldsymbol{\theta}_k)}$$

Newton-Raphson Method

- The quadratic expansion below can be directly minimized via a closed-form solution

$$L(\boldsymbol{\theta}_{k+1}) \cong L(\boldsymbol{\theta}_k) + \nabla L(\boldsymbol{\theta}_k)(\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k) + \frac{1}{2}(\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k)^T \mathbf{H}_{\mathrm{k}}(\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k)$$

- By setting the derivative of L w.r.t. $\boldsymbol{\theta}_k$ equal to 0, we obtain: $\nabla L(\boldsymbol{\theta}_k) + \mathbf{H}_k(\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k) = \mathbf{0}$
- The solution is the update rule for Newton-Raphson: $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{H}_k^{-1}\nabla L(\boldsymbol{\theta}_k)$

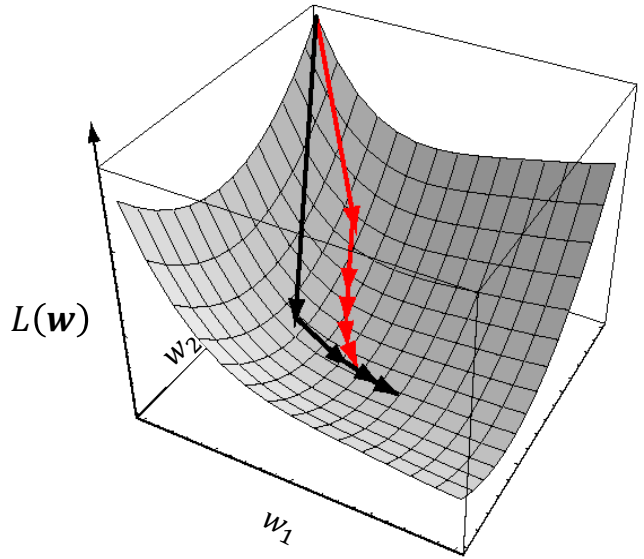
Newton-Raphson method

1. initialize θ, K, ε
2. for k in {0, 1, ... , $K-1$}:
3. $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{H}_k^{-1}\nabla L(\boldsymbol{\theta}_k)$
4. if $|L(\boldsymbol{\theta}_k) - L(\boldsymbol{\theta}_{k+1})| < \varepsilon$:
5. break

Example: Newton-Raphson vs Steepest Descent

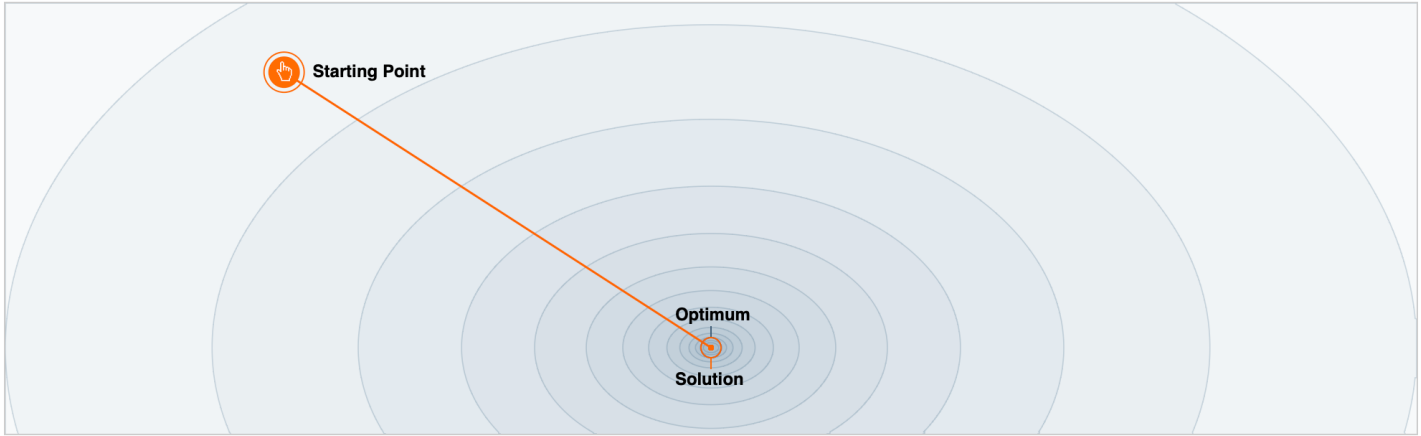
- **Steepest descent** in red
- **Newton-Raphson** in black

- **Newton-Raphson**
  - **Pros.** Faster convergence
  - **Cons.** Hessian inversion is computationally expensive. If Hessian is badly conditioned, convergence may be problematic



Example: Newton-Raphson on Quadratic Objective

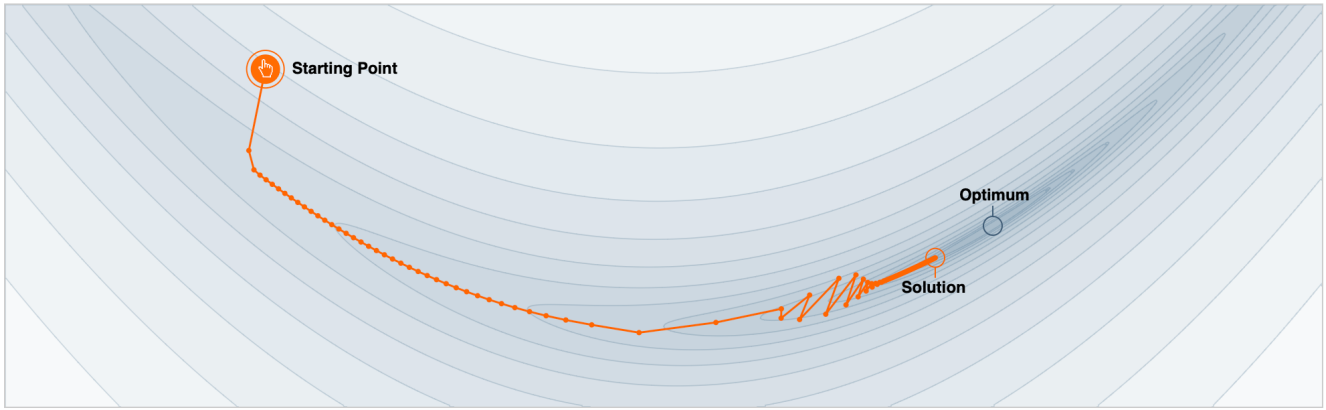
- Exact solution in one iteration



Examples from: http://fa.bianp.net/teaching/2018/eecs227at/newton.html

Example: Newton-Raphson on Non-convex Objective

- Badly-conditioned non-convex objective, numerical instabilities in Hessian inversion



Step-size α = 1

0 1 2

When the Hessian is close to singular there might be some numerical instabilities.

Examples from: http://fa.bianp.net/teaching/2018/eecs227at/newton.html

Tuning the Step Size

- In general, these quadratic approximations are computationally expensive, as they involve Hessian computation/inversion

- Lightweight **line-search methods** are thus preferred to find an approximate good step size at each iteration. See, e.g., *backtracking* line search:
  - http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

- Other strategies instead just decrease η_k at each iteration, with different decaying rates. See, e.g., *cosine annealing*:
  - https://towardsdatascience.com/https-medium-com-reina-wang-tw-stochastic-gradient-descent-with-restarts-5f511975163

Summary

- Linear classification functions for binary (2-class) classification
- Loss minimization principle (*learning the classifier parameters*)
  - Convex optimization / loss functions
- Gradient-based optimizers
  - Steepest descent
  - Newton-Raphson method
- Tuning the step size (*line-search and decay strategies*)

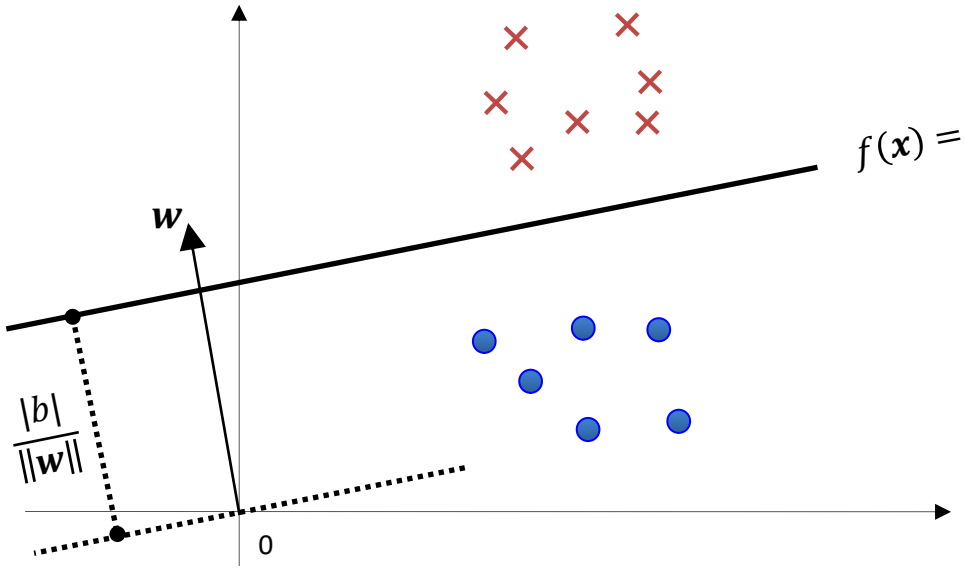
Support Vector Machines

Support Vector Machines (SVMs)

- State-of-the-art model with strong mathematical interpretation and significant practical relevance in many applications
- Invented by Russian mathematician Vladmir Vapnik in mid 1960s, and ideas much extended in 1990s
- Uses a specific type of loss function and learning algorithm
  - convex optimization / quadratic programming
- Can solve non-linearly separable problems

A Short Recap on Linear Discriminant Functions

$\mathrm{D} = \{\boldsymbol{x}_i, y_i\}_{i=1}^{n}$
$\boldsymbol{x} \in \mathbb{R}^d$
$y \in \{-1, +1\}$



$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + b = 0$

Distance of $\boldsymbol{x}$ to the hyperplane: $\frac{|f(\boldsymbol{x})|}{\|\boldsymbol{w}\|}$

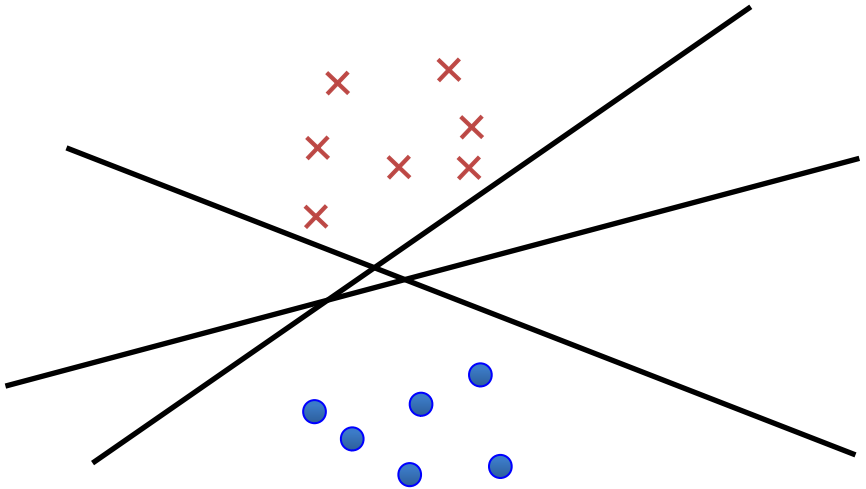
Support Vector Machines

The *“no maths”* version

Underlying Idea

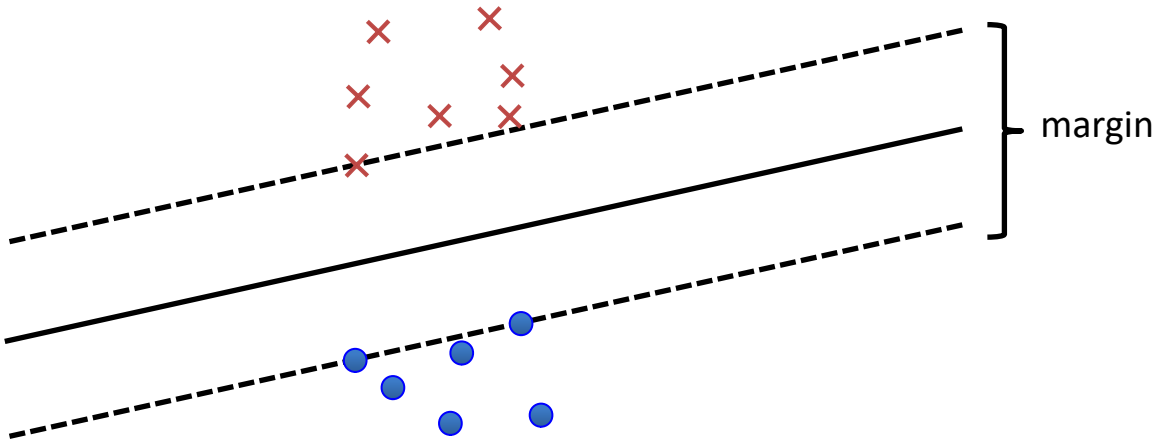
- Several possible decision boundaries
  - All get 100% accuracy on this training data

- Which one would you pick? Why?



Underlying Idea: Margin Maximization

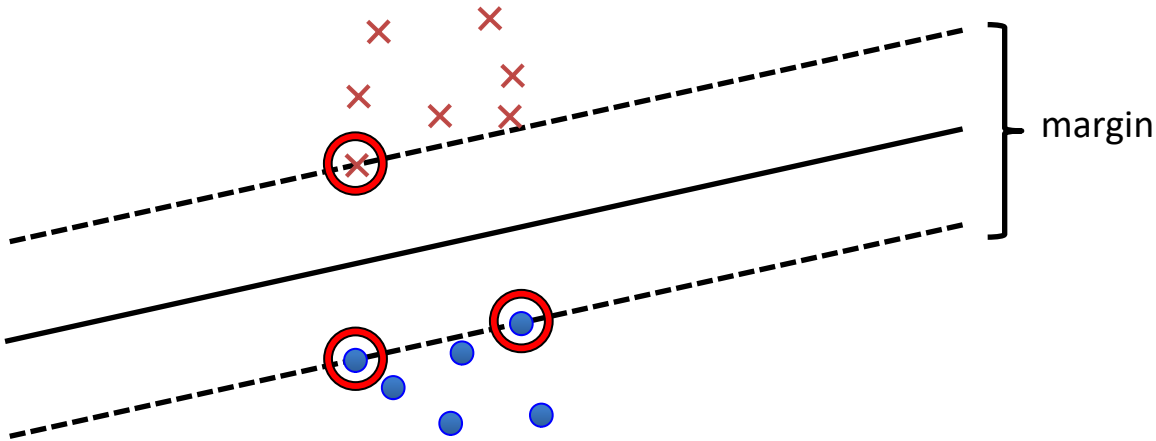
- The SVM finds this one – the furthest boundary from the two clusters



- Distance to the closest training points is called "margin"
  - equal on both sides of the boundary

Margin Maximization and Support Vectors

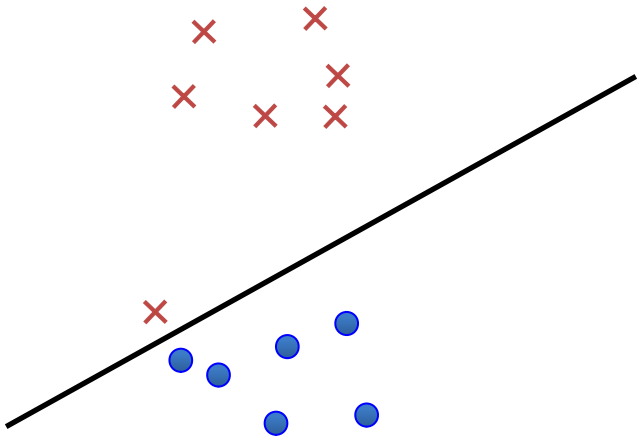
- The circled points are called ***support vectors***



- The decision hyperplane only depends on the SVs
  - the other points can move freely without violating the margin

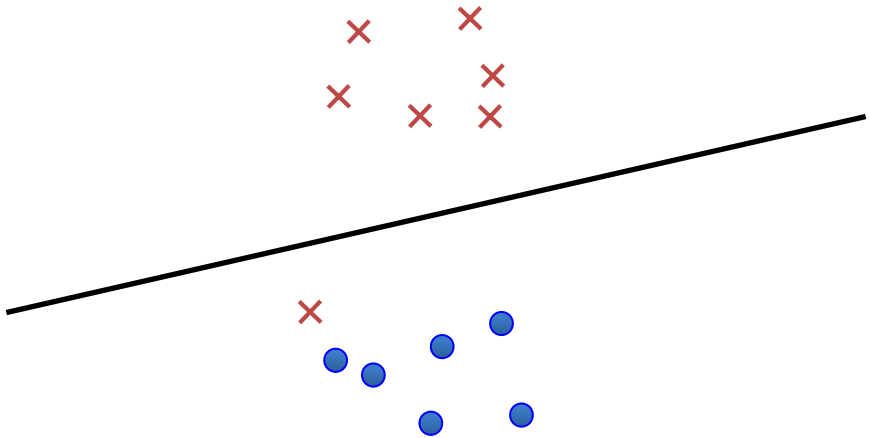
What About Outliers?

- This is the "optimal" boundary – doesn't seem so clever though

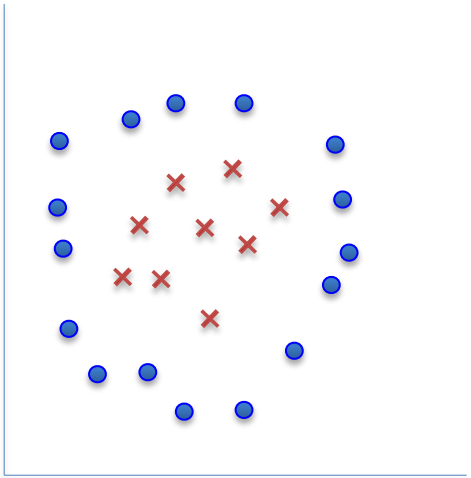


What About Outliers?

- SVM can selectively ignore certain data points (e.g., outliers)
  - soft-margin maximization

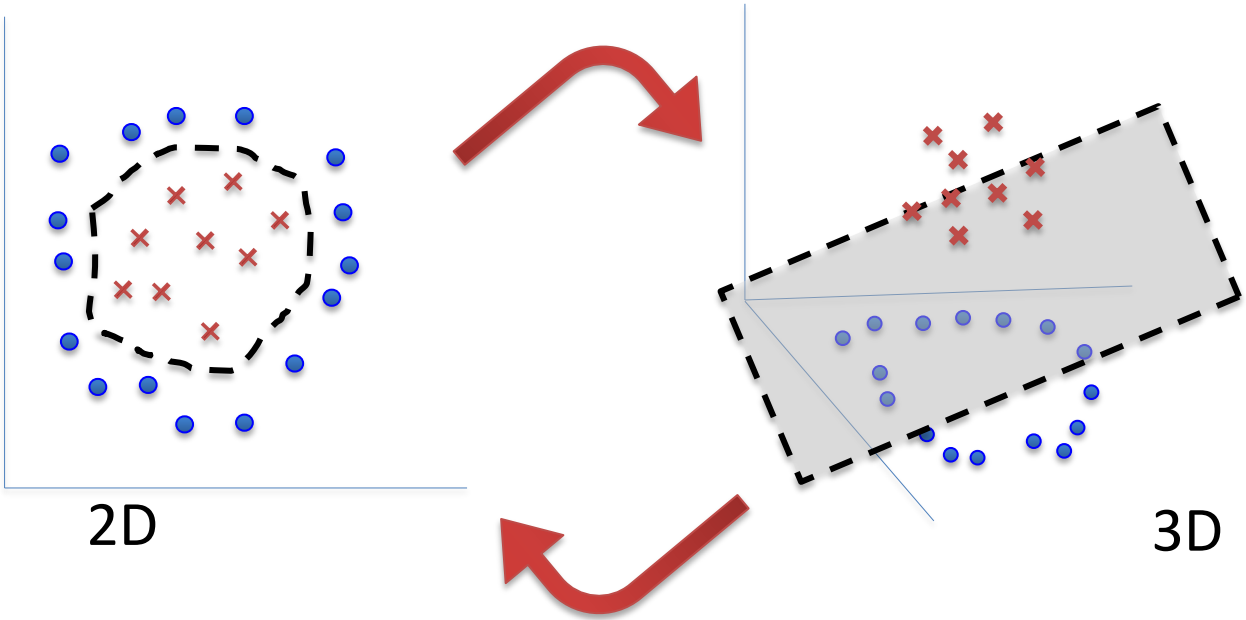


What If Data Is More Complex?



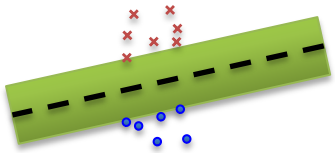
What If Data Is More Complex? Use Kernels!

- Project it onto a higher-dimensional space, and optimize a linear model
  - This amounts to learning a non-linear model in the input space

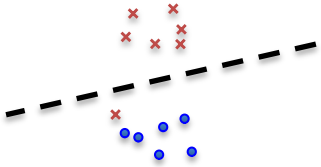


SVMs – Slightly More Technical Terms

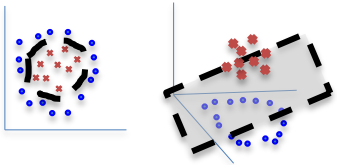
- Find the boundary with **maximum margin**



- Allow **soft-margin** violations to deal with outliers



- Use **kernels**, and the **kernel trick,** to solve nonlinear problems



Support Vector Machines

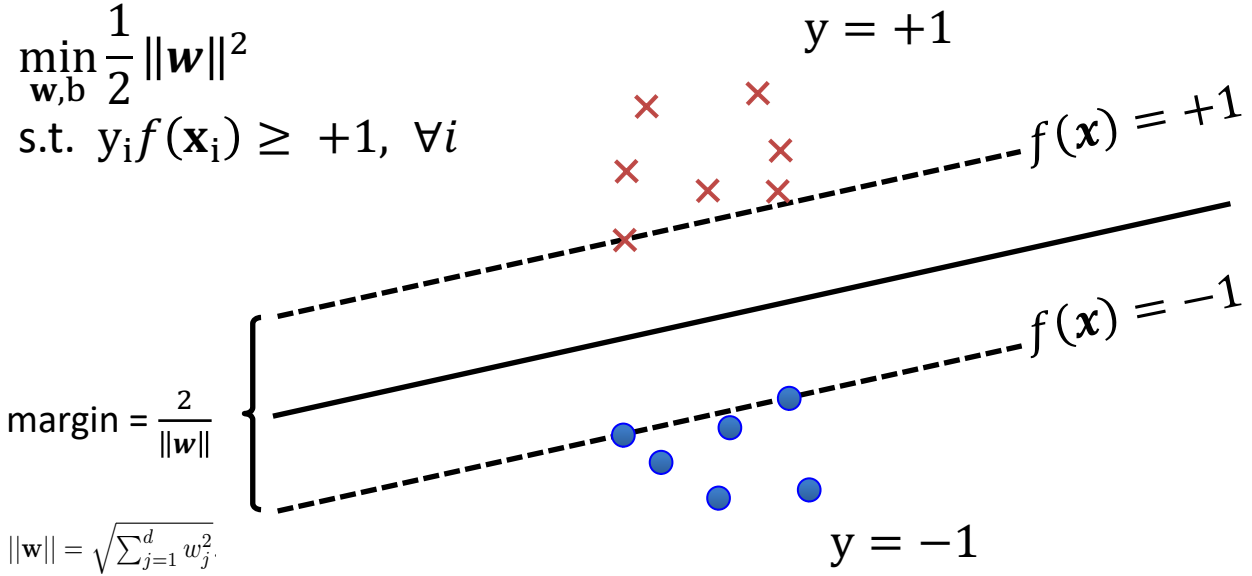
The *“some maths”* version

Hard-margin SVM (linearly-separable data)

- All points should be correctly classified (within the margin)

$$\min_{\mathbf{w},\mathrm{b}} \frac{1}{2}\|\boldsymbol{w}\|^2$$

$$\text{s.t. } \mathrm{y_i} f(\mathbf{x_i}) \geq +1, \ \forall i$$



The Problem of Margin Maximization

$$\text{minimise} \quad \frac{1}{2}\|\mathbf{w}\|^2$$
$$\text{subject to} \quad y_i f(\mathbf{x}_i) \geq 1, \quad i = 1, \ldots, N$$

- The above problem can be solved with the classical Lagrange optimization technique

$$L = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{\ell} \alpha_i \left[y_i \left(\mathbf{w} \cdot \mathbf{x}_i + b \right) - 1 \right]$$

$$\max_{\alpha_1, \ldots, \alpha_\ell} \min_{\mathbf{w}, b} L\left(\alpha_1, \ldots, \alpha_\ell, \mathbf{w}, b\right)$$
$$\text{s.t.} \quad \alpha_i \geq 0, \quad i = 1, \ldots, N$$

Deriving the Dual Form

- At the optimum, the derivatives of L w.r.t. **w** and b vanish, which gives:

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i \qquad \sum_i \alpha_i y_i = 0.$$

- Substituting into L, we obtain the **dual problem for the hard-margin SVM**:

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \ & \textstyle\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} y_i \alpha_i \, \boldsymbol{x}_i^T \boldsymbol{x}_j \alpha_j y_j \\ \text{s.t.} \ & \alpha_i \geq 0, \ \forall i \\ & \textstyle\sum_i \alpha_i y_i = 0 \, , \forall i \end{aligned}$$

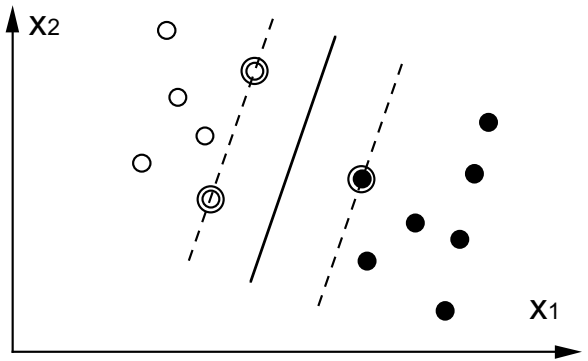
Support Vectors

- Lagrangian of the dual problem: $L_D(\alpha) = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} y_i \alpha_i \, \boldsymbol{x}_i^T \boldsymbol{x}_j \alpha_j y_j$

- The optimal solution is the hyperplane:

$$\sum_{i=1}^{\ell} y_i \alpha_i \left(\mathbf{x}_i \cdot \mathbf{x}\right) + b = 0$$

- The samples for which $\alpha_i \neq 0$ are called the **support vectors**



Soft-margin SVM

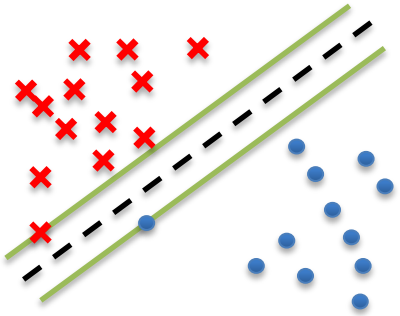
- What if data is not linearly separable / outliers? We allow points to violate the margin
- Trade-off between margin and loss on training data, tuned via the hyperparameter C

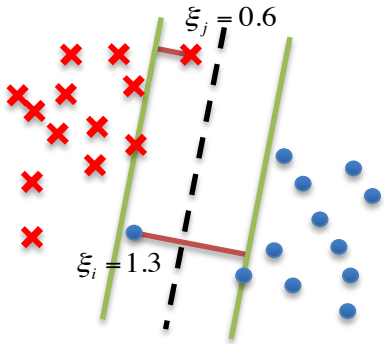
$$\min_{\mathbf{w},\mathrm{b},\xi_i} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_i \xi_i$$

$$\text{s.t. } \mathrm{y_i} f(\mathbf{x_i}) \geq 1 - \xi_i, \ \forall i$$

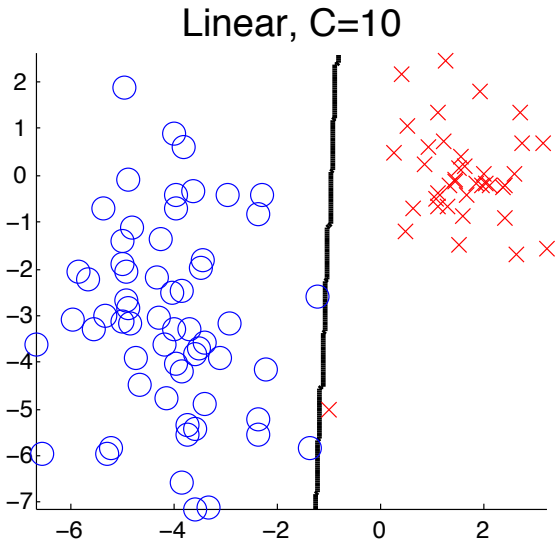
$$\xi_i \geq 0, \ \forall i$$

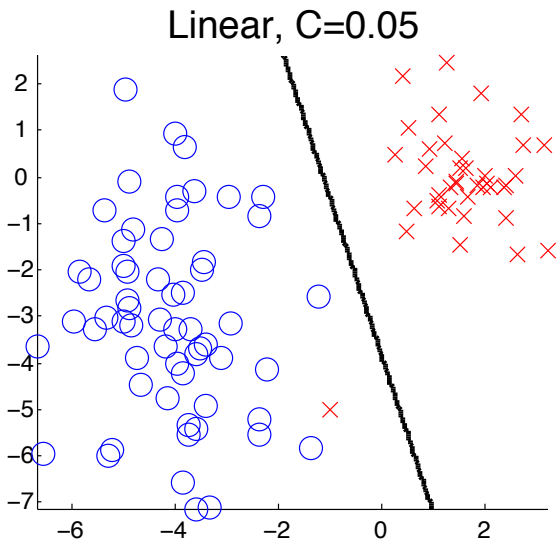
ξ_i: slack variables





Effect of the Hyperparameter C





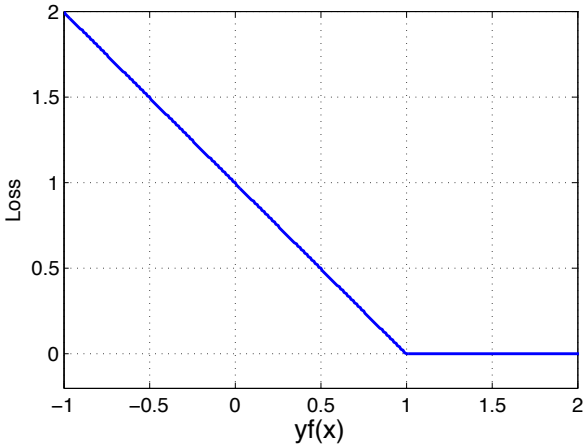
Slack Variables and the Hinge Loss

- The constraints $\xi_i \geq 1 - \mathrm{y_i} f(\mathbf{x_i})$, $\xi_i \geq 0$ are equivalent to $\xi_i = \max\left(0, 1 - \mathrm{y_i} f(\mathbf{x_i})\right)$
- That's exactly the **hinge loss**!
- We can use it to derive an unconstrained version of the primal problem:

$$E = C \sum_{i=1}^{N} \max\left\{0, 1 - y_i f(\mathbf{x}_i)\right\} + \frac{1}{2} \sum_{j=1}^{d} w_j^2$$

hinge loss (summed over training points)

margin term



Dual form of the Soft-margin SVM

- Replacing the optimality conditions

$$\frac{\partial L_P}{\partial w_\nu} = w_\nu - \sum_i \alpha_i y_i x_{i\nu} = 0 \qquad \frac{\partial L_P}{\partial b} = -\sum_i \alpha_i y_i = 0 \qquad \frac{\partial L_P}{\partial \xi_i} = C - \alpha_i - \mu_i = 0$$

- ... into the Lagrangian of the primal problem,

$$L_P = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i \xi_i - \sum_i \alpha_i \{y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 + \xi_i\} - \sum_i \mu_i \xi_i$$

- we obtain:

$$\max_{\boldsymbol{\alpha}} \ \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} y_i \alpha_i \, \boldsymbol{x}_i^T \boldsymbol{x}_j \alpha_j y_j$$
$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \ \forall i$$
$$\sum_i \alpha_i y_i = 0 \, , \forall i$$

How to Solve Primal and Dual SVM learning?

- These are both Quadratic Programming (QP) problems

$$\min_{\mathbf{w},\mathrm{b},\xi_i} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_i \xi_i$$
$$\text{s.t. } \mathrm{y_i} f(\mathbf{x_i}) \geq 1 - \xi_i, \ \forall i$$
$$\xi_i \geq 0, \ \forall i$$

$$\max_{\boldsymbol{\alpha}} \ \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} y_i \alpha_i \boldsymbol{x}_i^T \boldsymbol{x}_j \alpha_j y_j$$
$$\text{s.t. } 0 \leq \alpha_i \leq C, \ \forall i$$
$$\sum_i \alpha_i y_i = 0 \,, \forall i$$

- State-of-the-art QP solvers are standard and efficient

- However, dedicated (and much more efficient) solvers have been developed specifically for SVMs
  - **Sequential Minimal Optimization (SMO)** is one of them, used also in the popular LibSVM library

How to Solve Primal and Dual SVM learning?

- SMO however does not scale efficiently with the training set size
- Not appropriate to deal with modern **big data** sets
- Modern techniques optimize directly the primal (unconstrained) form using Stochastic Gradient Descent (SGD) and subgradients
- Subgradients overcome the non-differentiability of the hinge loss when $yf(\boldsymbol{x}) = 1$

$$E = \sum_{i=1}^{N} \max\left\{0, 1 - y_i f(\mathbf{x}_i)\right\} + \frac{1}{2}\sum_{j=1}^{d} w_j^2$$

Stochastic Gradient Descent (SGD)

- Set the learning rate η (using decay if needed)

- Repeat until an approximate minimum is obtained:
  - Randomly select K samples from the training data (i.e., one batch)
  - Update parameters $w'=w-\eta\nabla L$ using the K samples

- **Pros**: several variants of this algorithm have been proposed
  - They are very efficient – and incremental
  - Data loading in batches avoids keeping the full dataset in memory
  - Convergence is well studied and often satisfied in practice

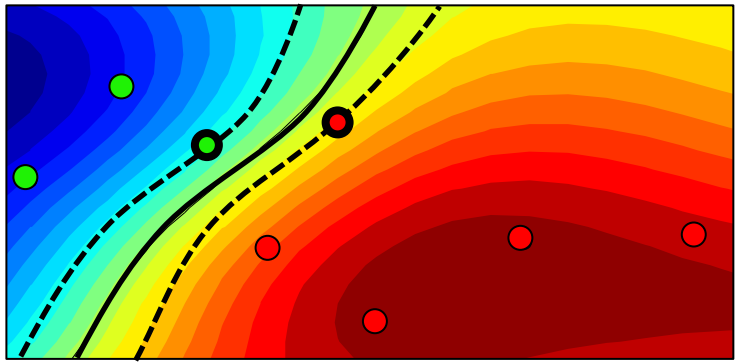
- **Cons**: parameters (e.g., learning rate) can be difficult to tune

The Kernel Trick

- Soft-margin SVM deals with non-linearly separable data but it is still a linear classifier
  - It exhibits poor performance if data is not properly shaped or "mostly" linearly separable

- The *kernel trick* overcomes this limitation by allowing us to learn an **SVM with a nonlinear decision function in input space!**

How?



The Kernel Trick

- The dual SVM formulation reveals that both the learning problem and classification can be expressed only in terms of **scalar products** between samples!

$$\max_{\boldsymbol{\alpha}} \ \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} y_i \alpha_i \, \boldsymbol{x}_i^T \boldsymbol{x}_j \alpha_j y_j$$
$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \ \forall i$$
$$\sum_i \alpha_i y_i = 0 \, , \forall i$$

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + b = \sum_i y_i \alpha_i \, \boldsymbol{x}_i^T \boldsymbol{x}$$

- It also holds for the primal form, thanks to this property:
  - see the *Representer Theorem* for further details

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$$

The Kernel Trick

- We are not required to compute the scalar product in the input space
- We can use any function $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$given that it implicitly corresponds to a scalar product in some other space (Mercer's condition)

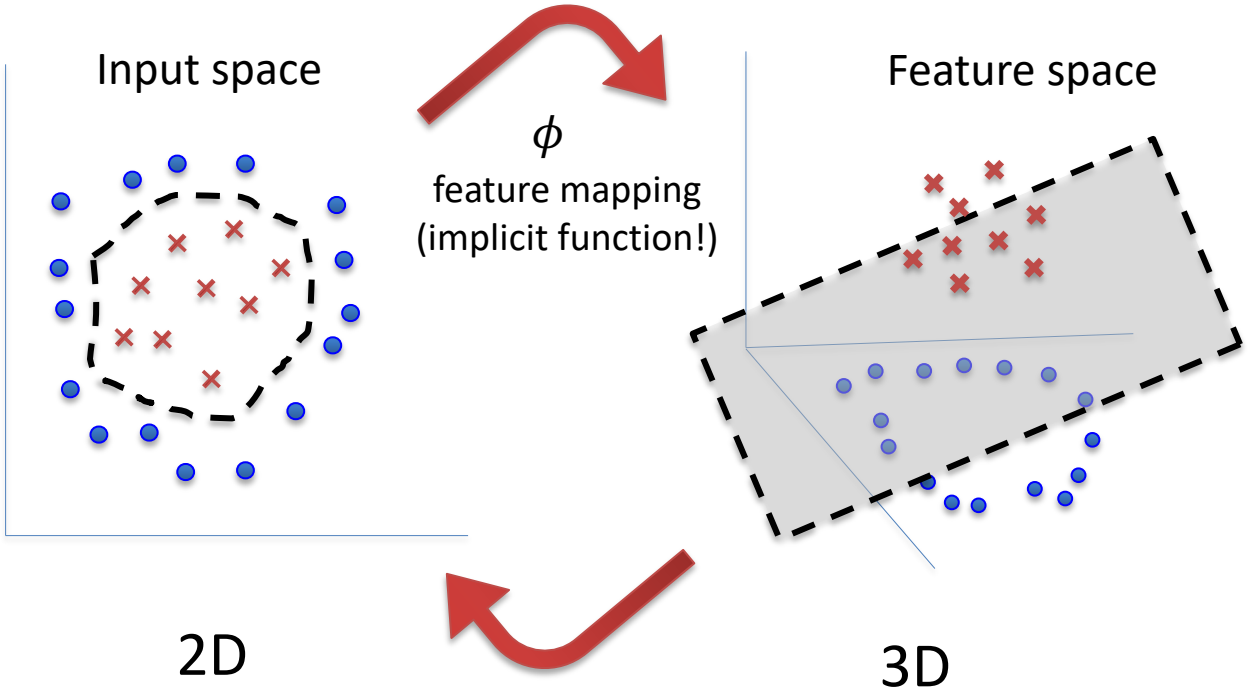
- Kernel functions are symmetric and positive semi-definite (PSD) if

$$\sum_{i,j=1}^{n} c_i c_j K(x_i, x_j) \geq 0 \qquad (1.1)$$

holds for any $n \in \mathbb{N}, x_1, \ldots, x_n \in \mathcal{X}, c_1, \ldots, c_n \in \mathbb{R}$.

- **Note**: SVMs also converge when using symmetric non-PSD kernels (although the learning problem is no longer convex)

The Kernel Trick



The Cover's theorem

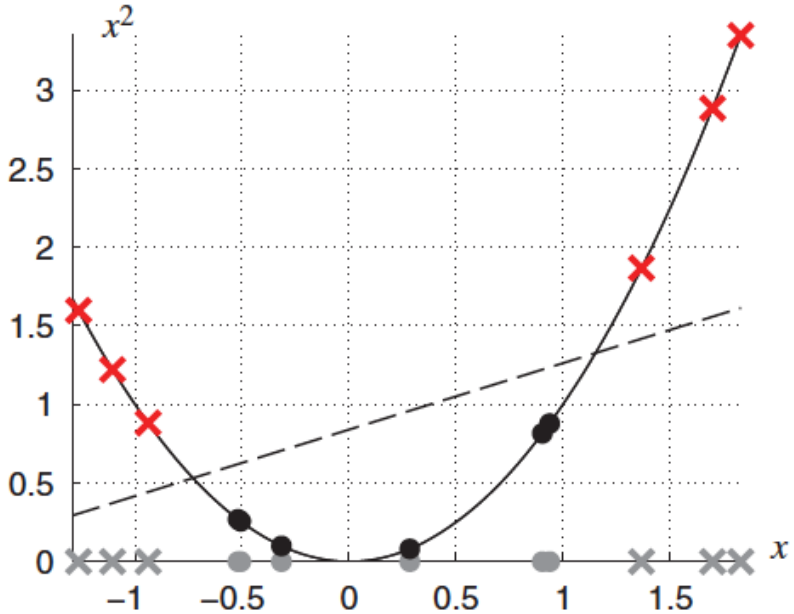
- "*A complex pattern-classification problem cast in a high-dimensional space non-linearly is more likely to be linearly separable than in a low-dimensional space*"

- The power of SVMs resides in the fact that they represent a robust and efficient implementation of Cover's theorem

- SVMs operate in two stages
  - Perform a non-linear mapping of the input vector x onto a high-dimensional space that is hidden from the inputs or the outputs
  - Construct an optimal separating hyperplane in the high-dimensional feature space

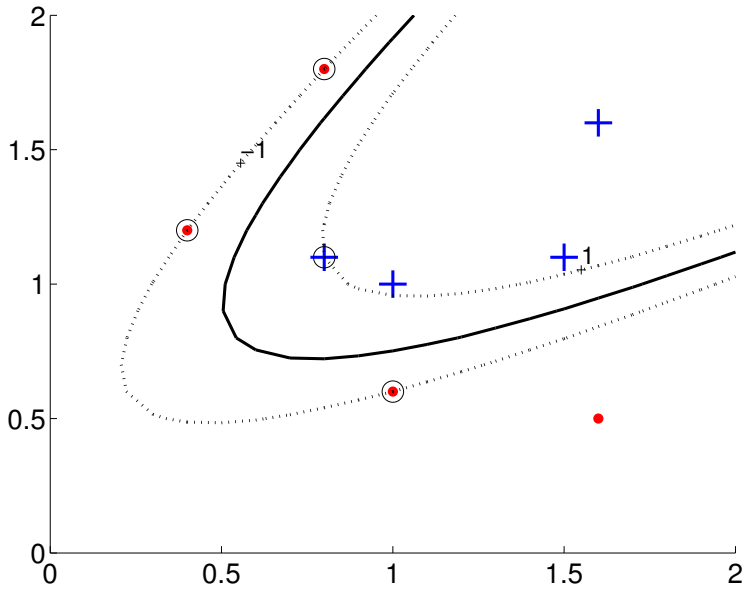
The Cover's theorem

- "*A complex pattern-classification problem cast in a high-dimensional space non-linearly is more likely to be linearly separable than in a low-dimensional space*"



Polynomial Kernel, with d=2

$$K(\mathbf{x}_i, \mathbf{x}') = (1 + \mathbf{x}_i^T \mathbf{x}')^d$$



The Polynomial Kernel

$$K(\mathbf{x}_i, \mathbf{x}') = (1 + \mathbf{x}_i^T \mathbf{x}')^d$$

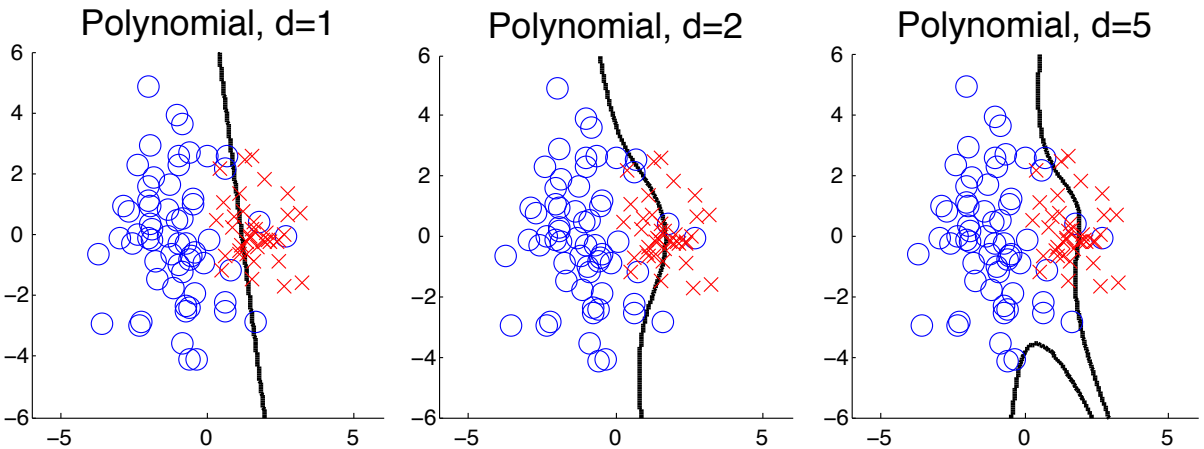
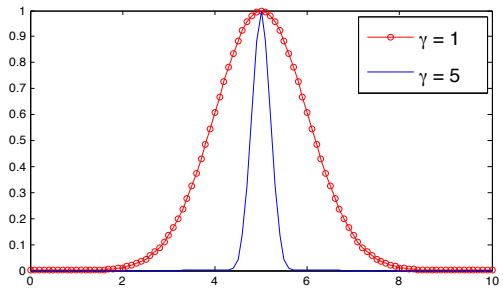
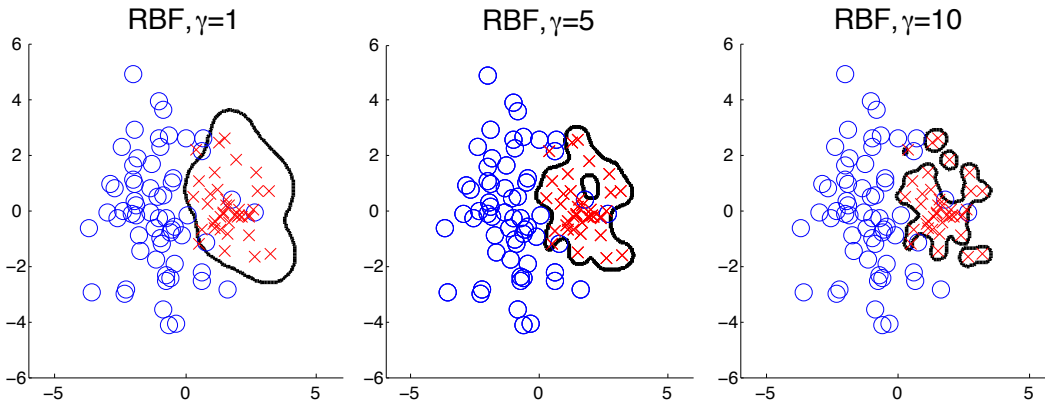


Figure 4.6: The effect of the degree parameter when using a polynomial kernel.

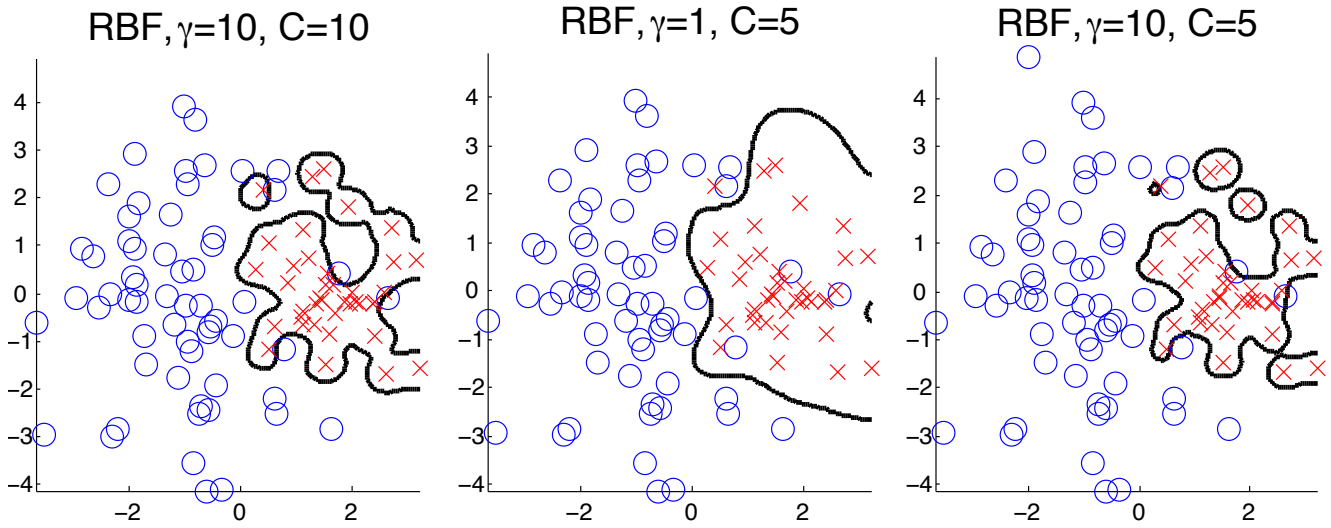
The Gaussian Kernel (Radial Basis Function, RBF)

$$K(\mathbf{x_i}, \mathbf{x}') = e^{-\gamma(\mathbf{x}_i - \mathbf{x}')^2}$$





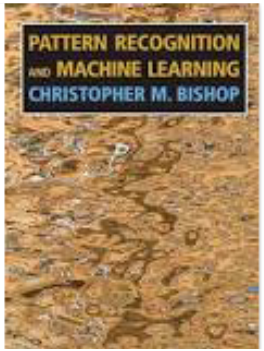
Playing with the Hypeparameters



Suggested Readings

A Tutorial on Support Vector Machines for Pattern Recognition

CHRISTOPHER J.C. BURGES burges@lucent.com
Bell Laboratories, Lucent Technologies



Pattern Recognition and Machine Learning
Book by Christopher Bishop

Chapter 7

Convex Optimization
Stephen Boyd and Lieven Vandenberghe

Cambridge University Press