Linear Models, Optimization, and SVMs

Battista Biggio

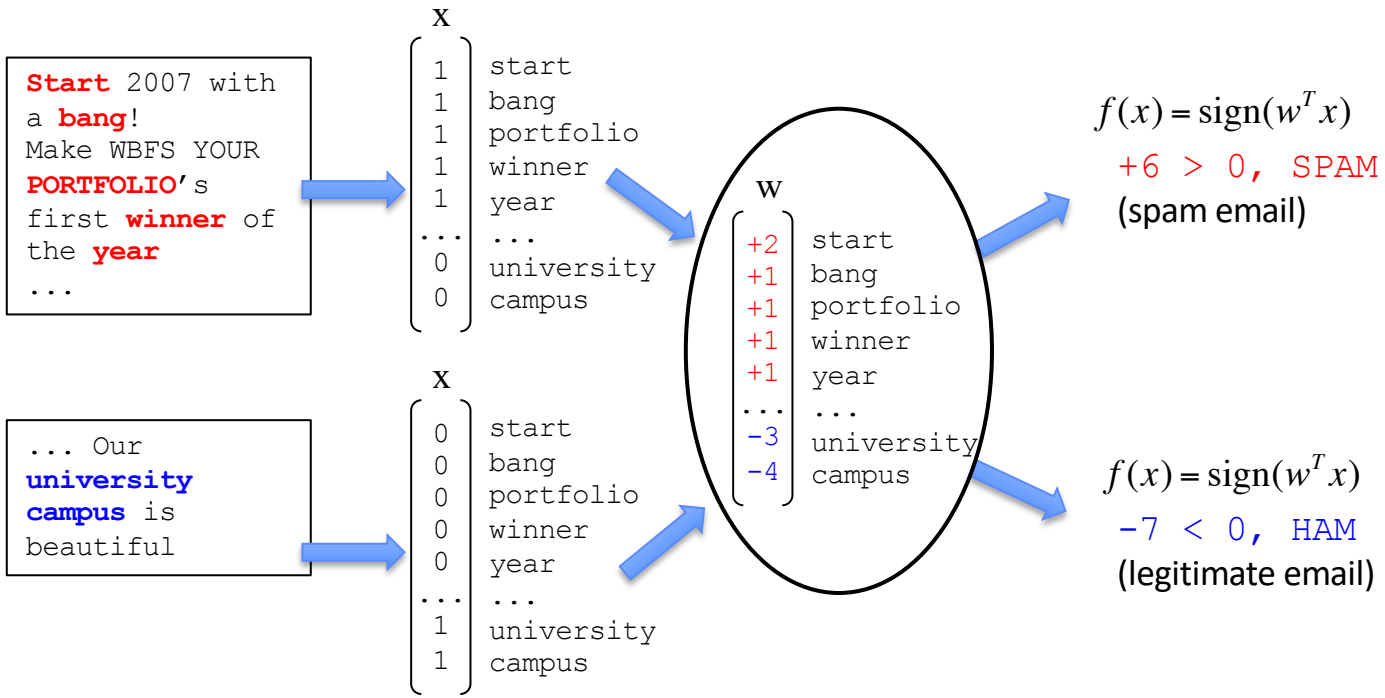
Linear Classifiers

- In general, a classifier can be regarded as a **function** $f(\boldsymbol{x})$ that takes as input the vector $\boldsymbol{x}$ (representing the input) and provides as output the classification (class label)

- Linear classification models have the form:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b = \sum_{j=1}^{d} w_j x_j + b$$

Example: Spam Filtering

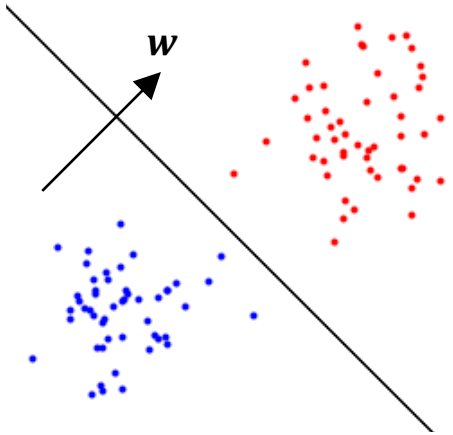


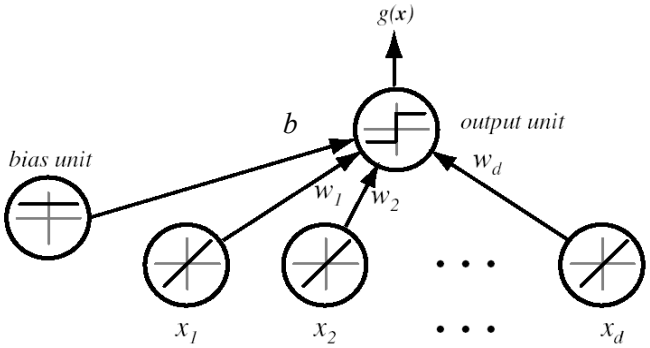
Linear Discriminant Functions

- **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: $y = \begin{cases} +1 \text{ if } f(\boldsymbol{x}) \geq 0 \\ -1, \text{ otherwise} \end{cases}$

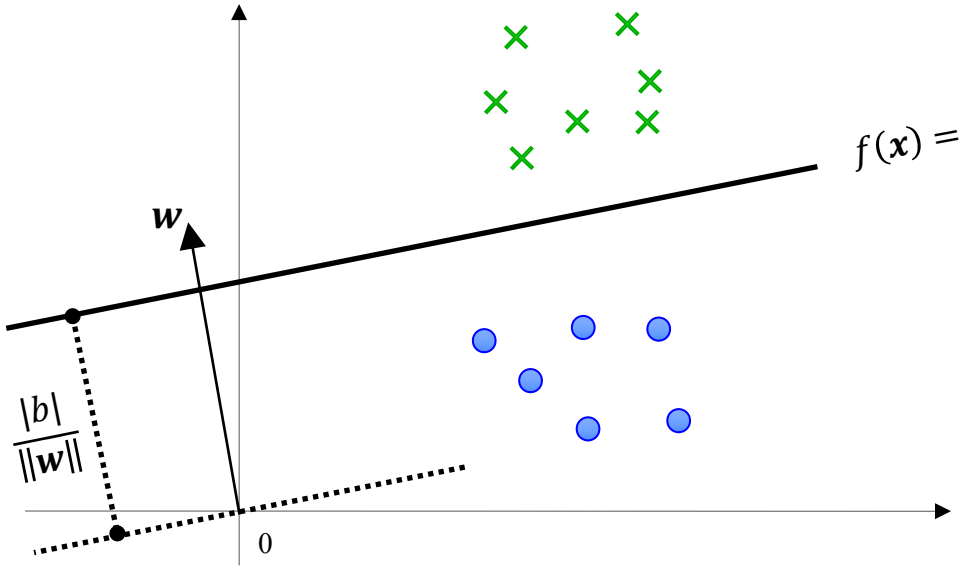
- **Graphical representation**
  - Each input feature value x_j is multiplied by the corresponding weight value w_j
  - Bias is multiplied by 1
  - The output unit sums all its inputs, computing $f(\boldsymbol{x})$ and thresholds it to estimate y (+1 or -1)





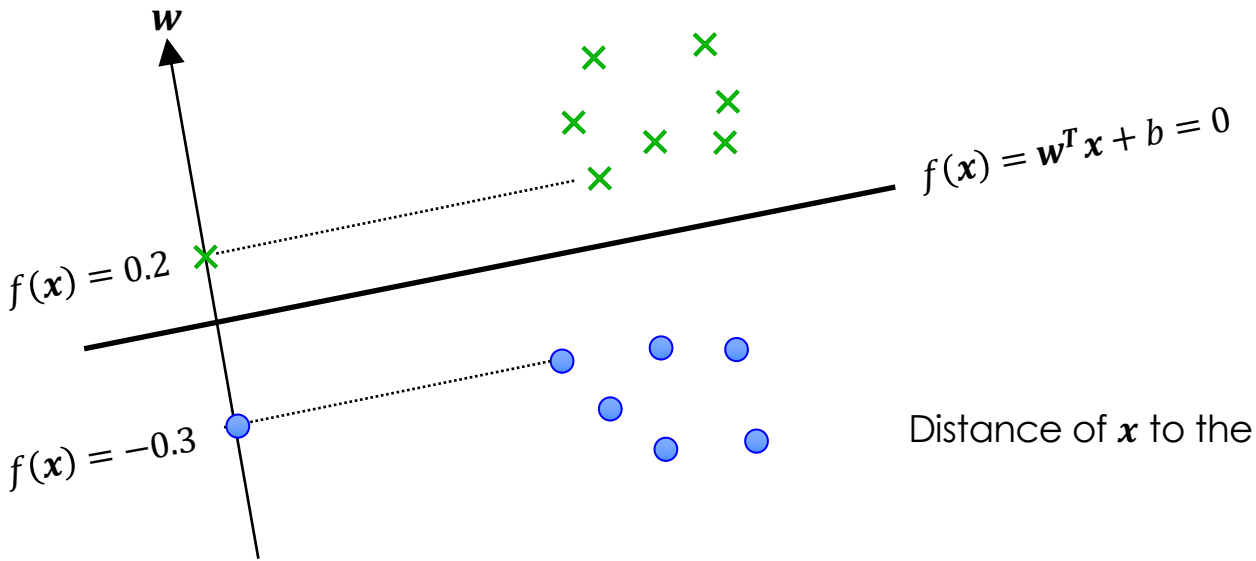
Linear Discriminant Functions

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



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: $\frac{|f(\boldsymbol{x})|}{\|\boldsymbol{w}\|}$

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

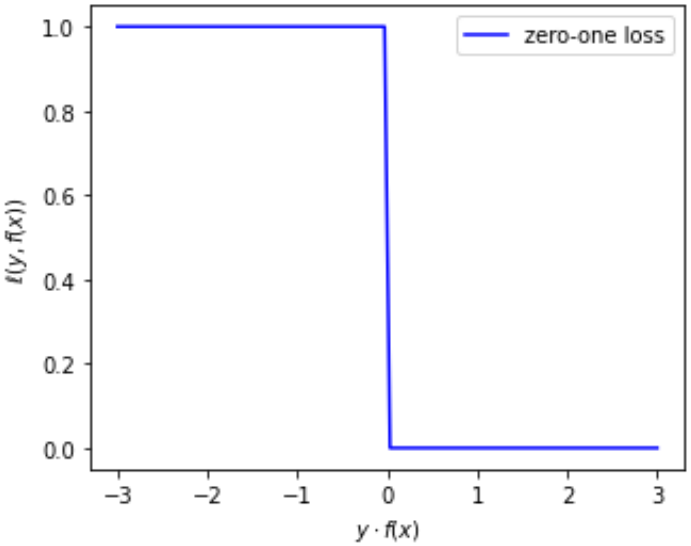
- In principle, we would like to minimize

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

- 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

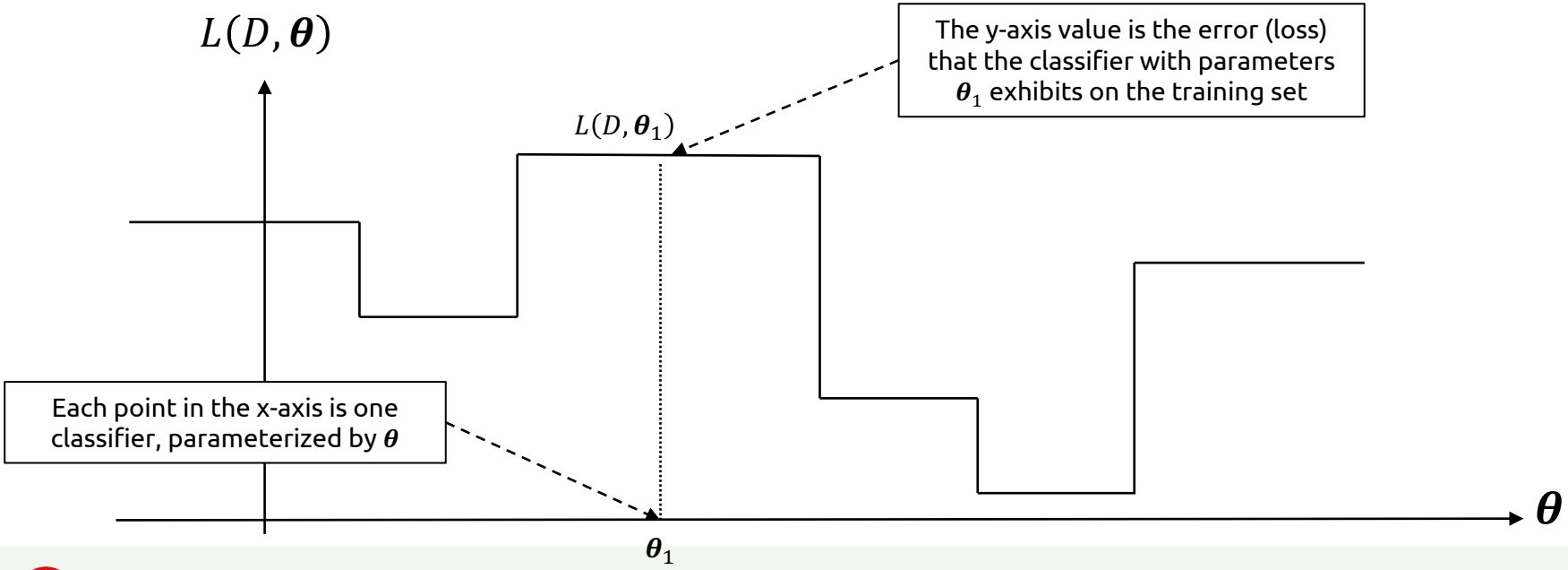


$yf(\boldsymbol{x})$ is positive for correct predictions and negative for wrong ones.

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)

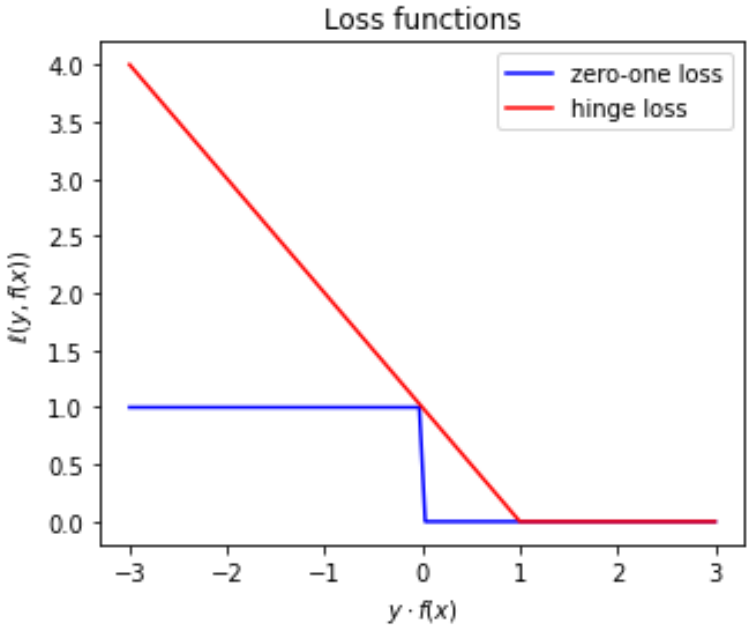


Loss Functions

- 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**

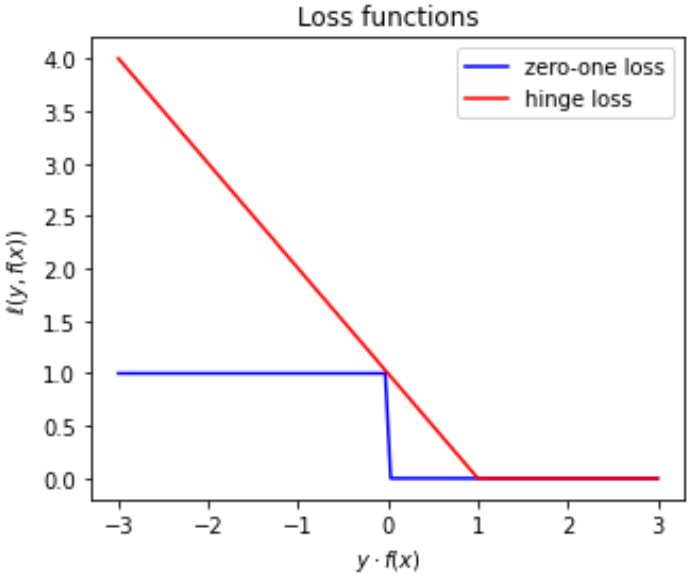


Hinge Loss

- The hinge loss is computed as:

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

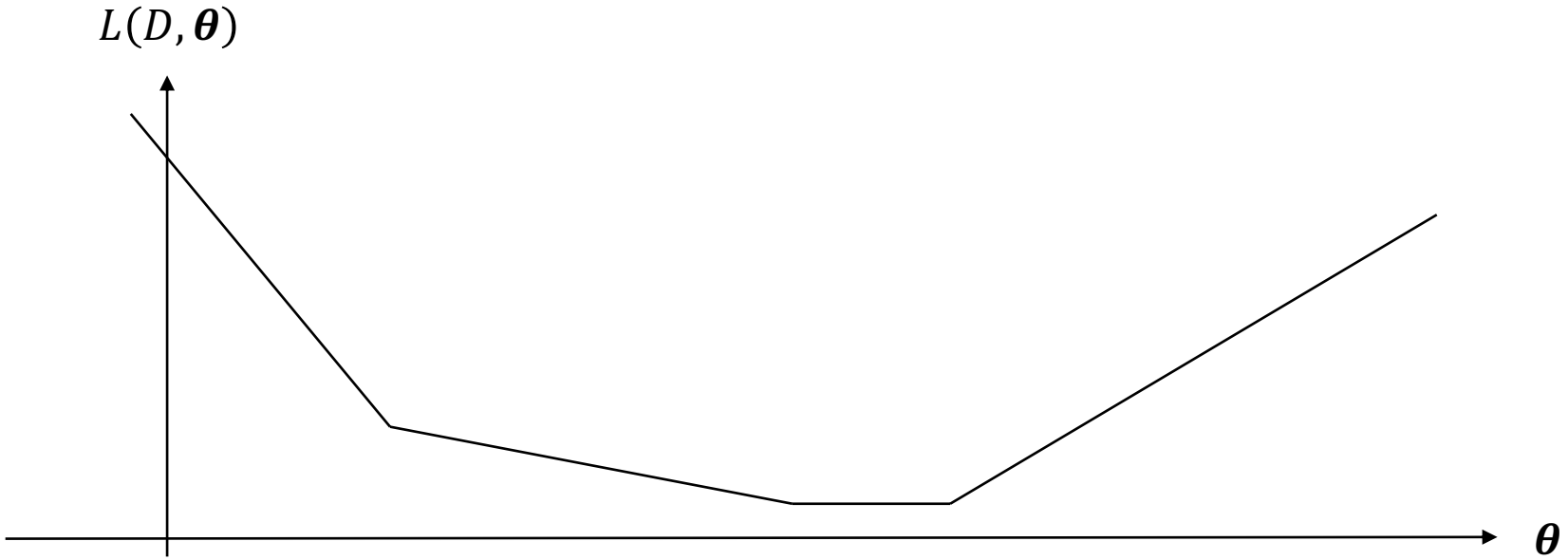
- It is the closest convex upper bound on the 0-1 loss
- Why are we interested in an upper bound?
  - since minimizing it, we also minimize the 0-1 loss
- Convexity helps find solutions efficiently while also providing guarantees on the optimality of the solution (global optima), algorithmic convergence,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

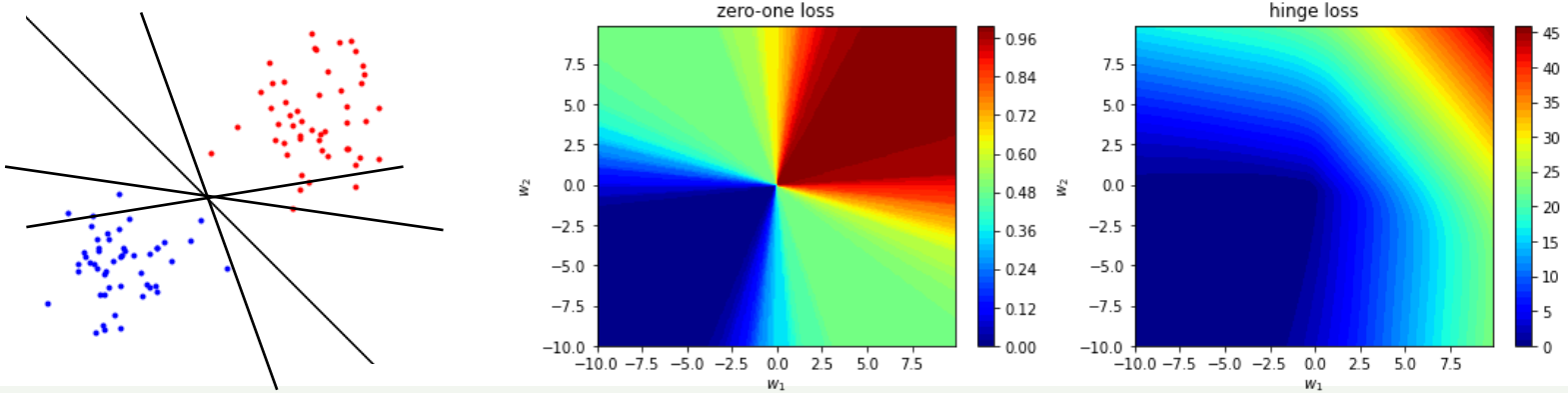
The Hinge Loss Landscape

- 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



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:

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

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

- **Stochastic Gradient Descent (SGD)** uses a random subset of training samples in each iteration

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

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

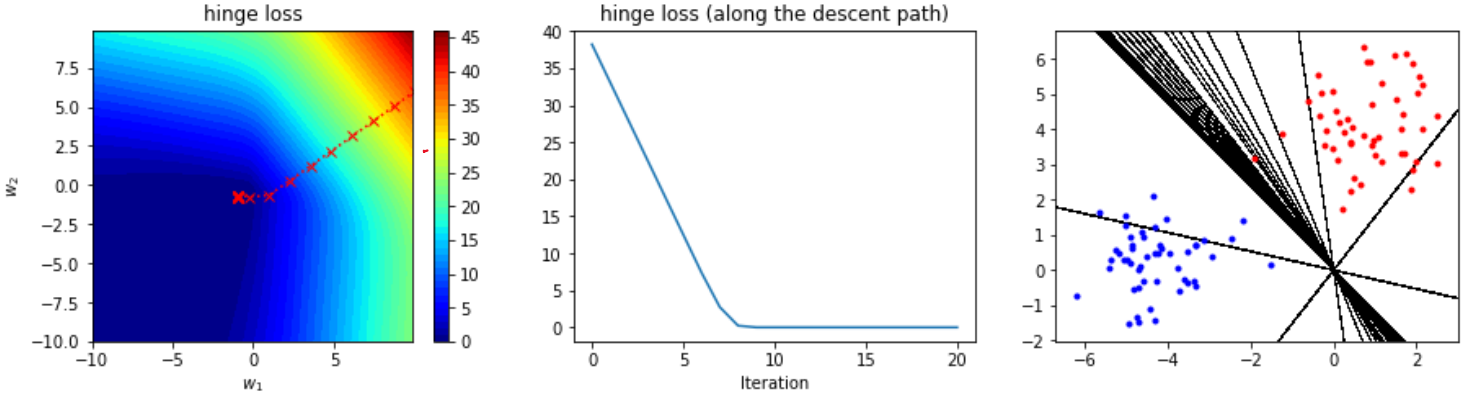
1. initialize θ, η, K, ε
2. for k in {0, 1, ... , $K-1$}:
3. $\quad \boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla L(\boldsymbol{\theta}_k)$
4. $\quad$ if $|L(\boldsymbol{\theta}_k) - L(\boldsymbol{\theta}_{k+1})| < \varepsilon$:
5. $\quad\quad$ break

- 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 algorithm may not even converge at all
  - Usually η is reduced across iterations to ensure convergence

Steepest Descent on the Hinge Loss

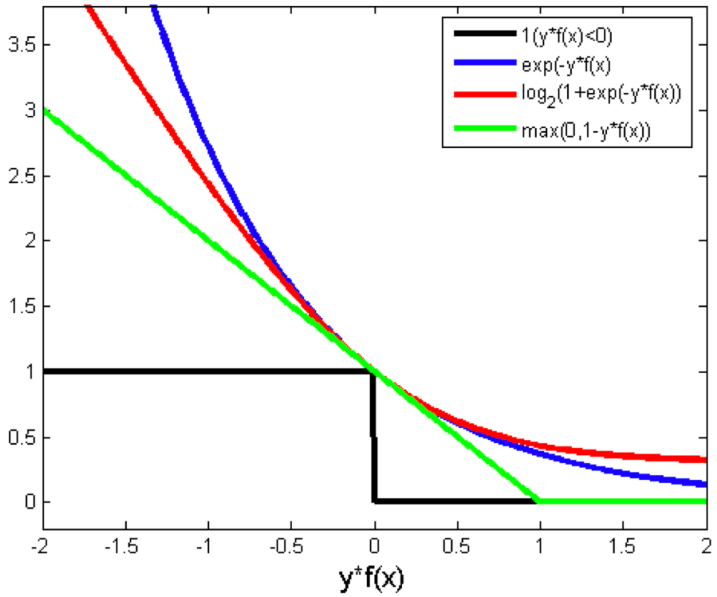
- Consider again our running example. We start optimization from $w = (10, 6)$ and fix $b = 0$
  - See the notebook for details on the computation of the gradient of the hinge loss w.r.t. w, b

- 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 with zero training error



Other Convex Losses

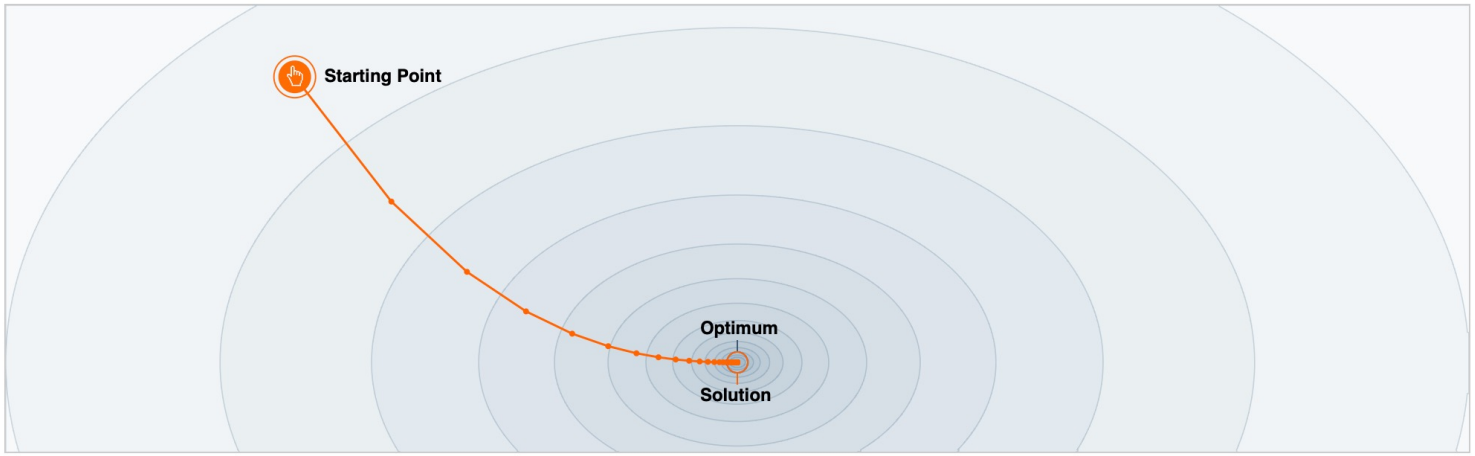
- Other convex upper bounds on the zero-one loss:
  - Hinge loss: $\ell\big(y, f(\boldsymbol{x})\big) = \max(0, 1 - yf)$
  - Exponential loss: $\ell\big(y, f(\boldsymbol{x})\big) = e^{-yf}$
  - Logistic loss: $\ell\big(y, f(\boldsymbol{x})\big) = \log_2(1 + e^{-yf})$



Gradient Descent: Step Size and Convergence

Example: Steepest Descent on Quadratic Objective

- Well-conditioned quadratic objective, small step size



Step-size α = 0.22

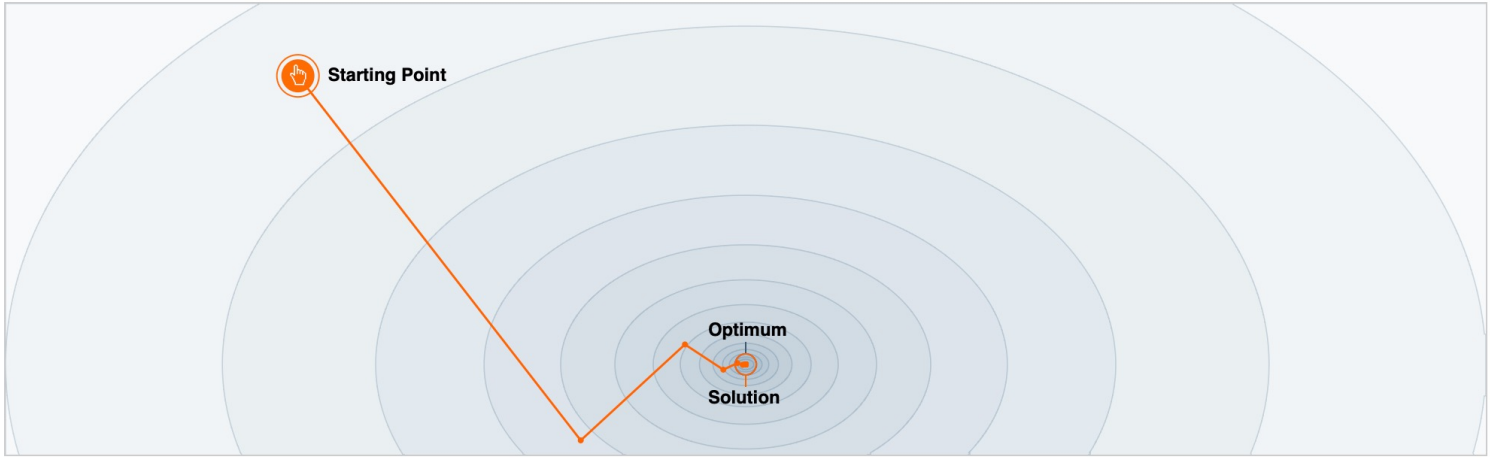
0 0.5 1.0

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

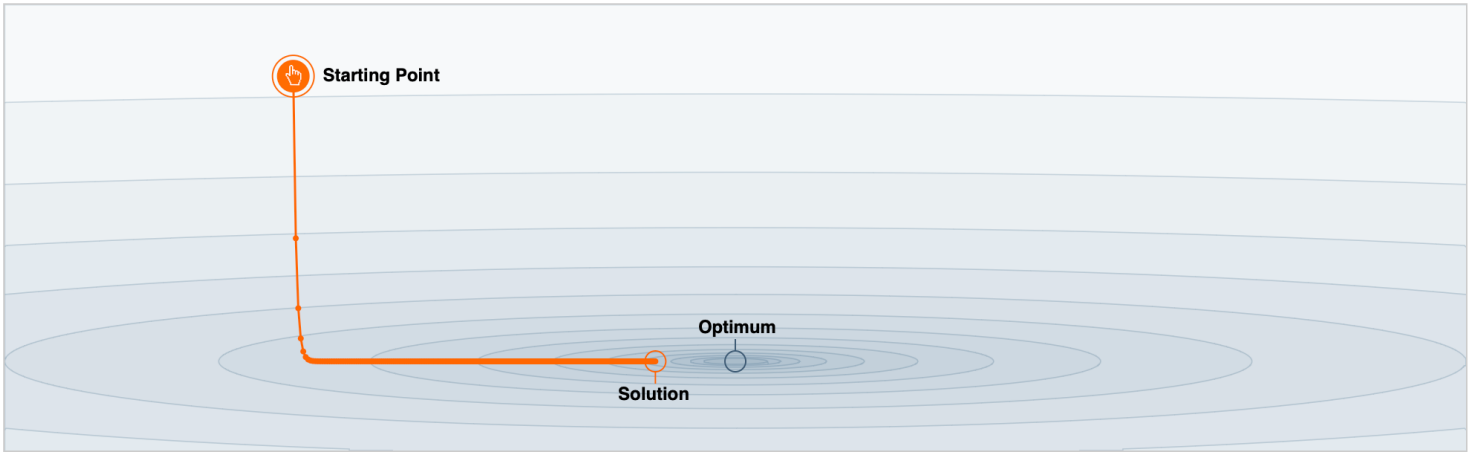
0 0.5 1.0

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

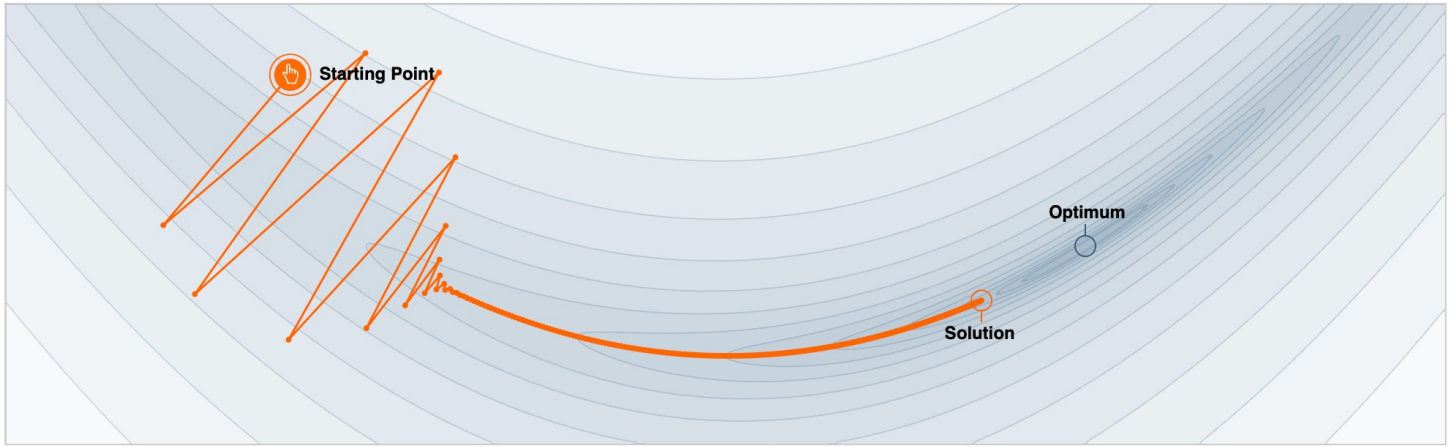
0 0.01 0.02

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.

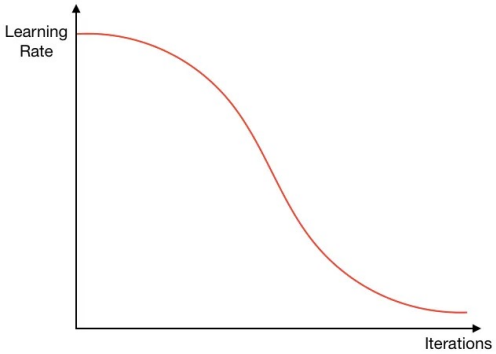
Examples from: http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

Tuning the Step Size

- In general, quadratic approximations (e.g. Newton-Raphson) are computationally expensive, as they involve Hessian computation

- 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 / SGD
- Tuning the step size (*line-search and decay strategies*)

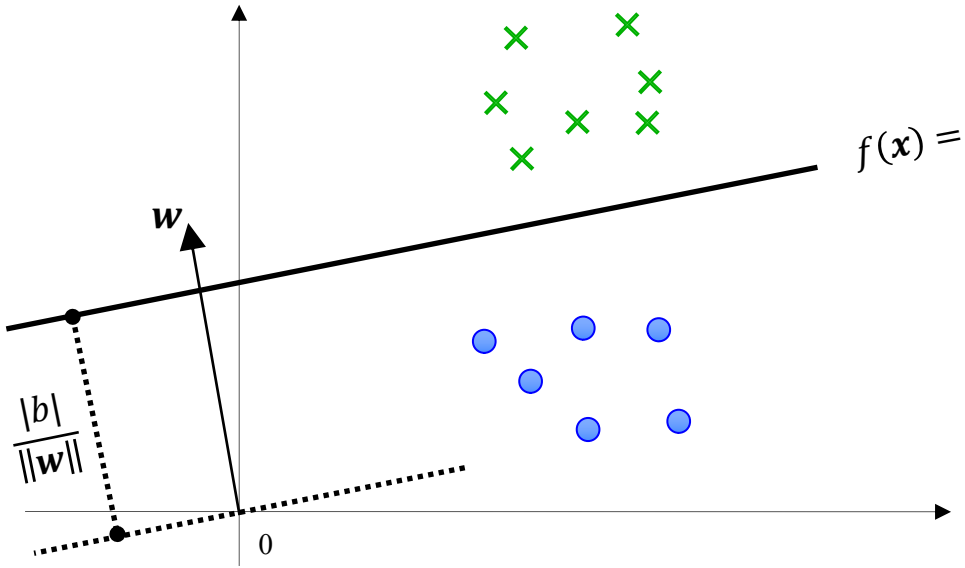
Support Vector Machines

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

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



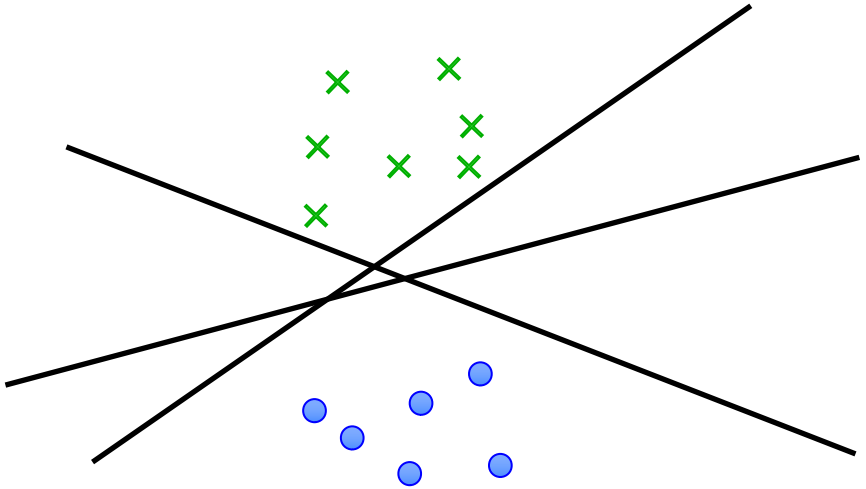
$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + b = 0$

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

Underlying Idea

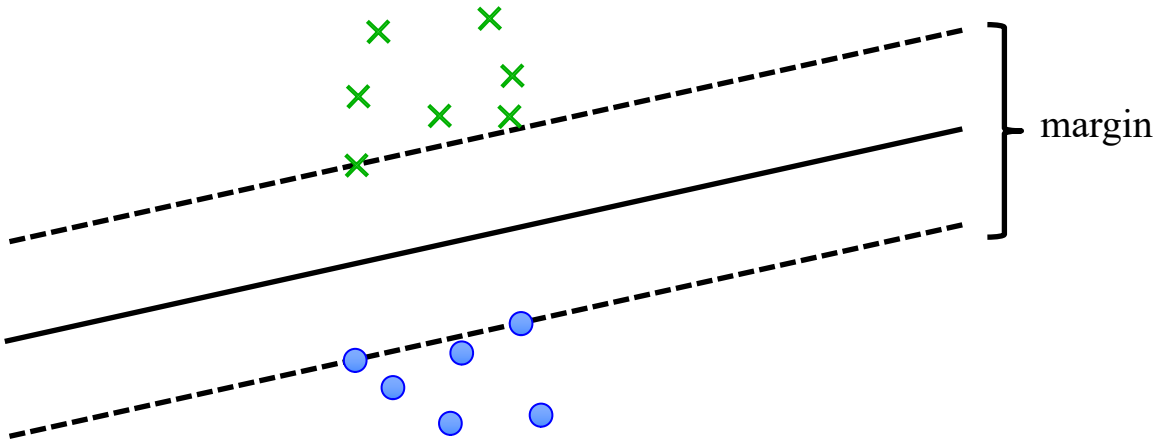
- Several possible decision boundaries
  - All get 100% accuracy on this training data

- Which one would you pick? Why?



Underlying Idea: Margin Maximization

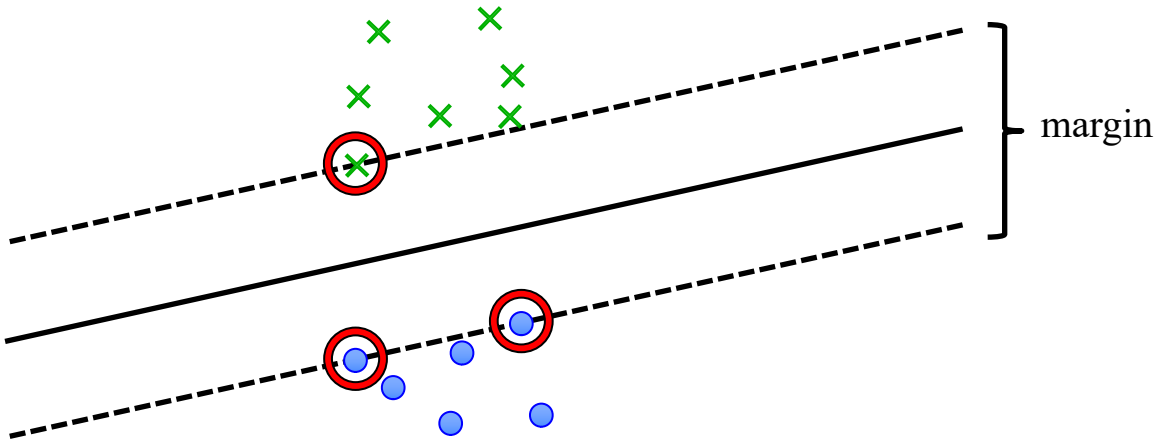
- The SVM finds this one – the furthest boundary from the two clusters



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

Margin Maximization and Support Vectors

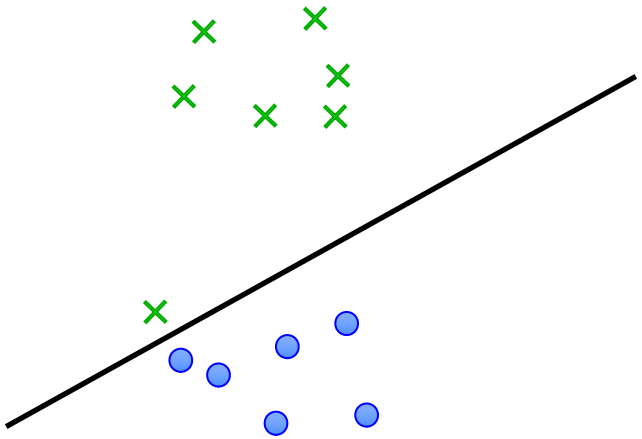
- The circled points are called ***support vectors***



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

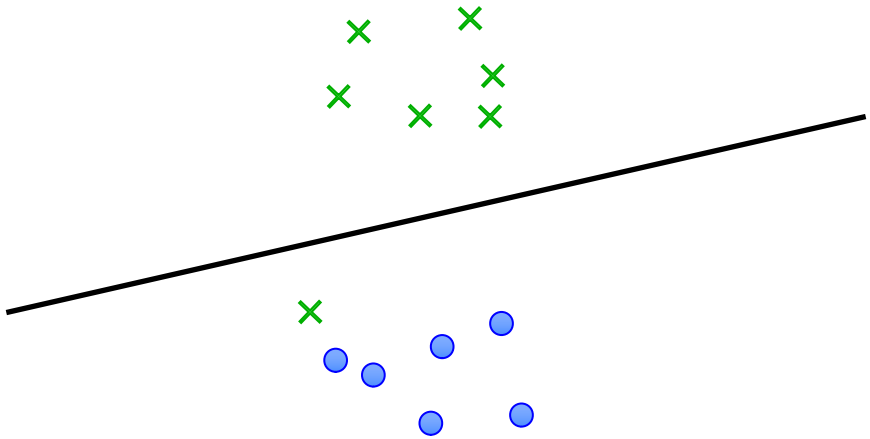
What About Outliers?

- This is the “optimal” boundary – doesn’t seem so clever though

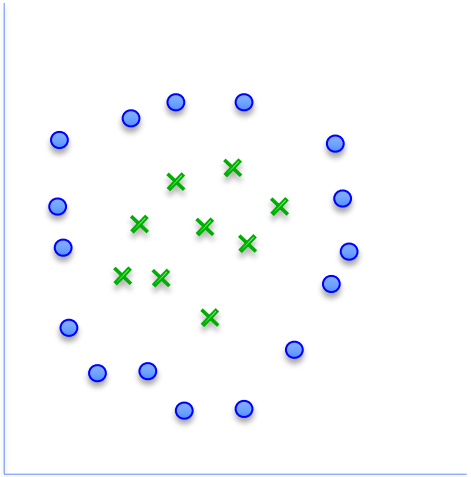


What About Outliers?

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

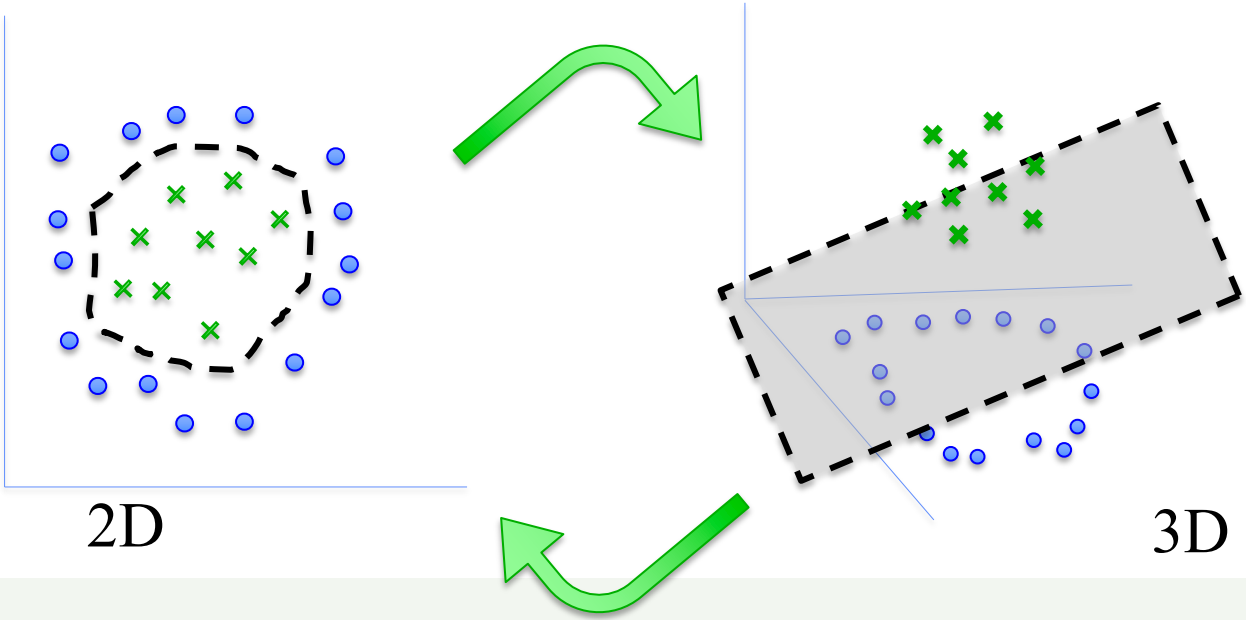


What If Data Is More Complex?



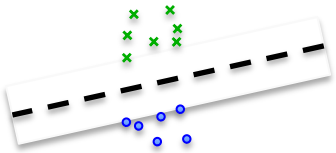
What If Data Is More Complex? Use Kernels!

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

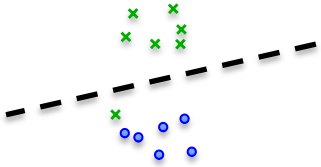


SVMs – Slightly More Technical Terms

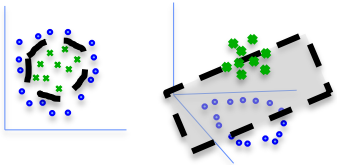
- Find the boundary with **maximum margin**



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



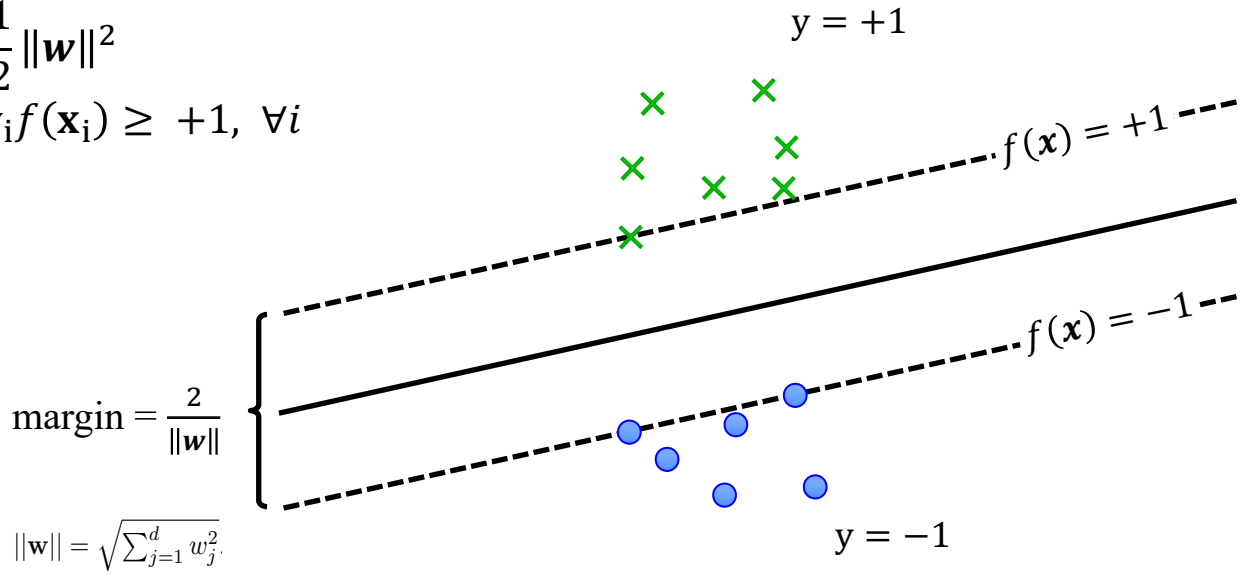
- Use **kernels**, and the **kernel trick,** to solve nonlinear problems



Hard-Margin SVM (Linearly-Separable Data)

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

$$\min_{\mathbf{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2$$
$$\text{s.t. } y_i f(\mathbf{x}_i) \geq +1, \ \forall i$$

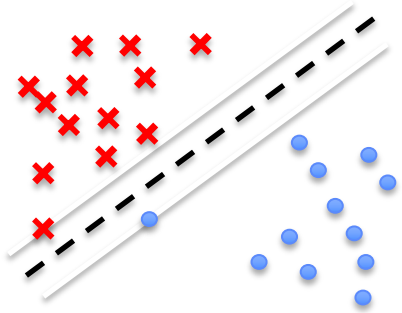


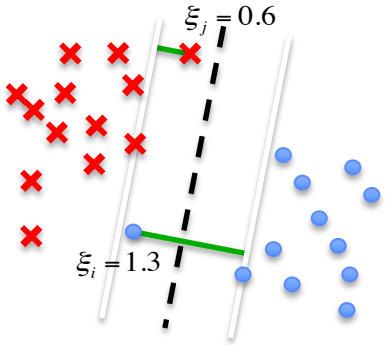
Soft-Margin SVM

- 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

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

ξ_i: slack variables

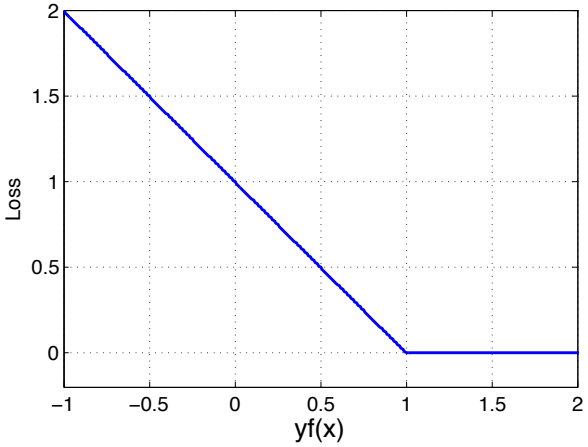




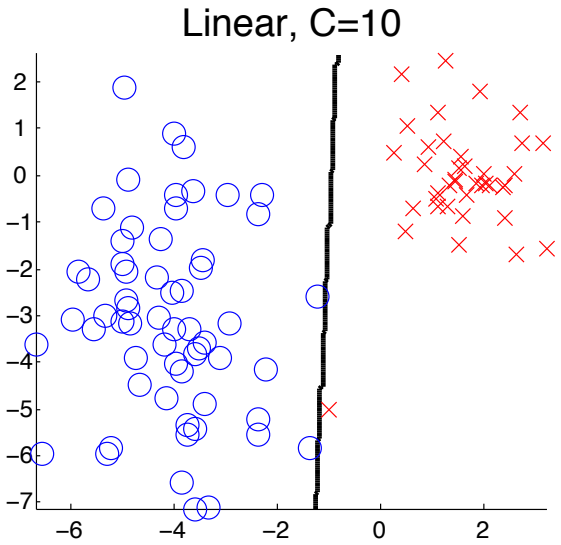
Slack Variables and the Hinge Loss

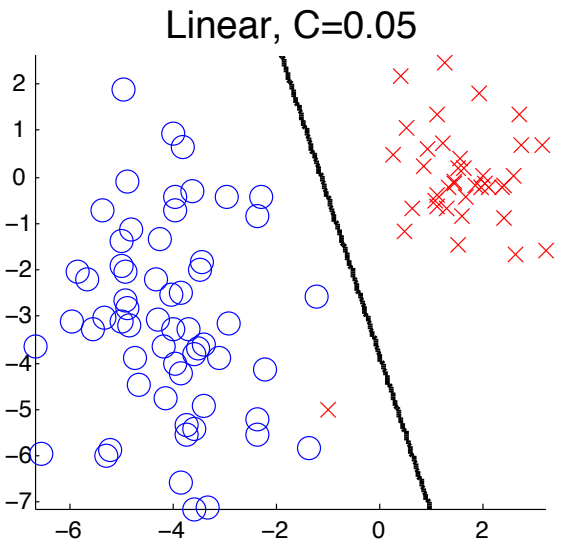
- The constraints $\xi_i \geq 1 - y_i f(\mathbf{x}_i)$, $\xi_i \geq 0$ are equivalent to $\xi_i = \max\left(0, 1 - y_i f(\mathbf{x}_i)\right)$
- That's exactly the **hinge loss**!
- We can use it to derive an unconstrained version of the primal problem:

$$\min_{\boldsymbol{w},b} \underbrace{\frac{1}{2}\|\boldsymbol{w}\|^2}_{\text{margin term}} + C \underbrace{\sum_i \max(0, 1 - y_i f(\boldsymbol{x}_i))}_{\text{hinge loss (summed over training points)}}$$



Effect of the Hyperparameter C



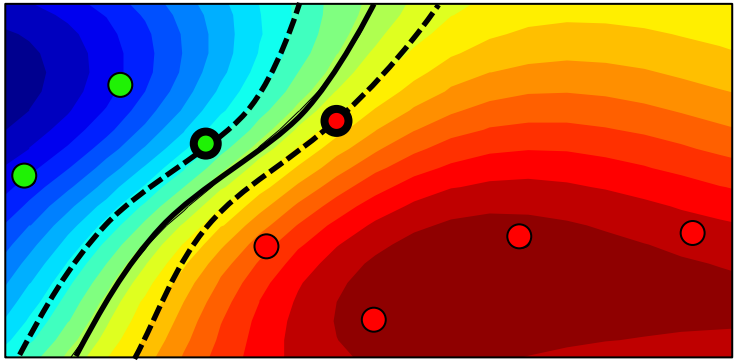


The Kernel Trick

- 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
  - only the SVs have a positive alpha coefficient, i.e., ***w*** only depends on them!

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

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)

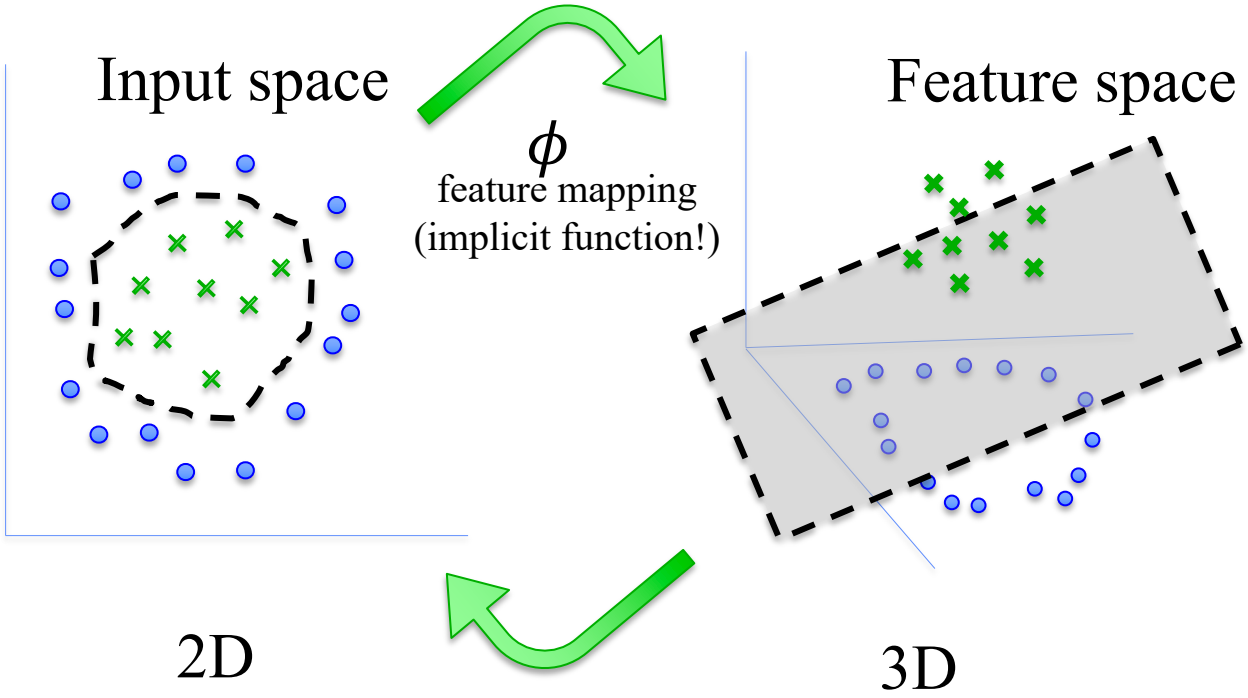
- Kernel functions are symmetric and positive semi-definite (PSD) if

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

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



Cover's Theorem

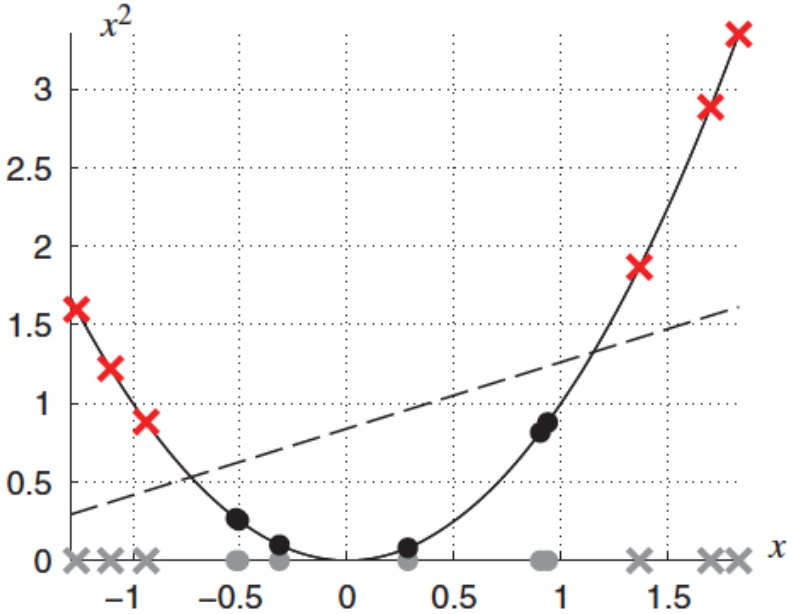
- "*A complex pattern-classification problem cast in a high-dimensional space non-linearly is more likely to be linearly separable than in a low-dimensional space*"

- 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

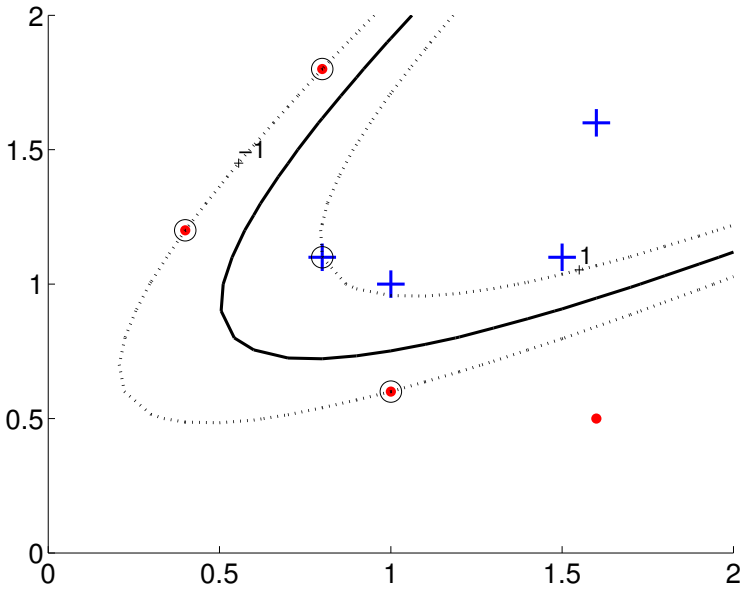
Cover's Theorem

- "*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

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

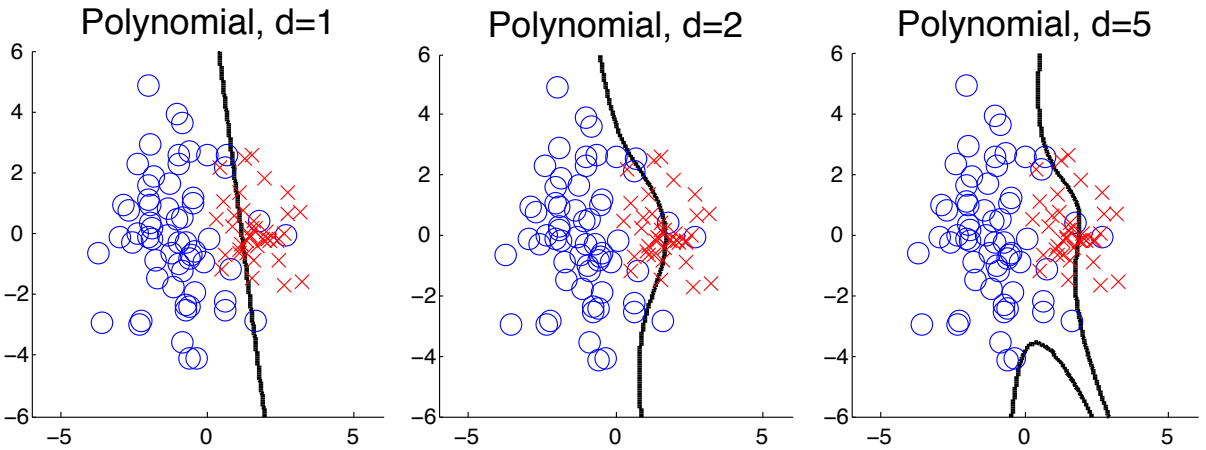
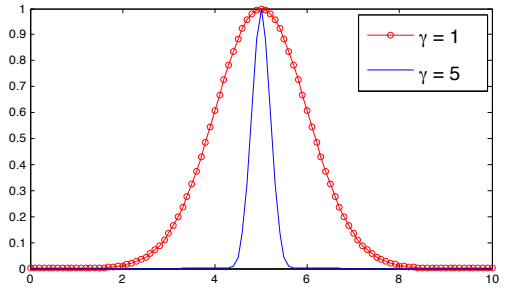
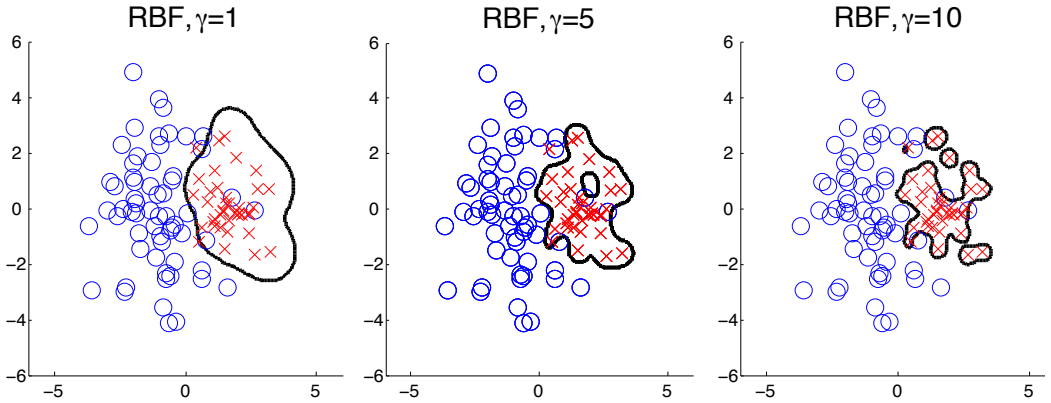


Figure 4.6: The effect of the degree parameter when using a polynomial kernel.

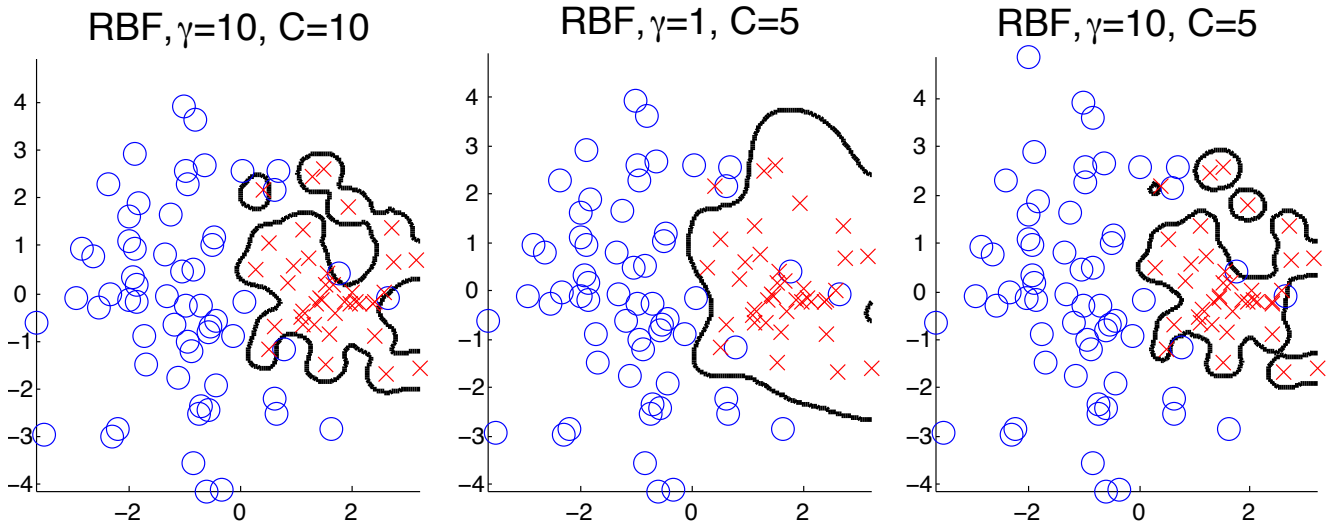
The Gaussian Kernel (Radial Basis Function, RBF)

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





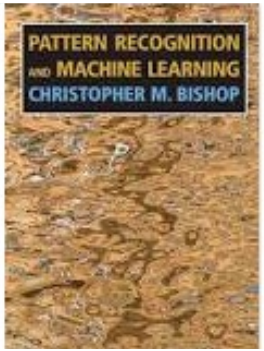
Playing with the Hyperparameters



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

More on Linear Classifiers

More on Linear Classifiers

- The unconstrained version of the SVM primal problem...

$$\min_{\boldsymbol{w},b} C \sum_i \max(0,1 - y_i f(\boldsymbol{x}_i)) + \frac{1}{2}\|\boldsymbol{w}\|^2$$

- ... can be seen as an instance of a more general problem

$$\min_{\boldsymbol{w},b} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(\boldsymbol{x}_i))}_{\text{loss term}} + \underbrace{\lambda\Omega(\boldsymbol{w})}_{\text{regularization term}}$$

λ : trade-off parameter

loss term (empirical loss on training data)

regularization term (controls overfitting / classifier capacity)

More on Linear Classifiers

- This objective function amounts to minimizing the generalization error on test data
  - see C. Burges' tutorial and Statistical Learning Theory by V. Vapnik

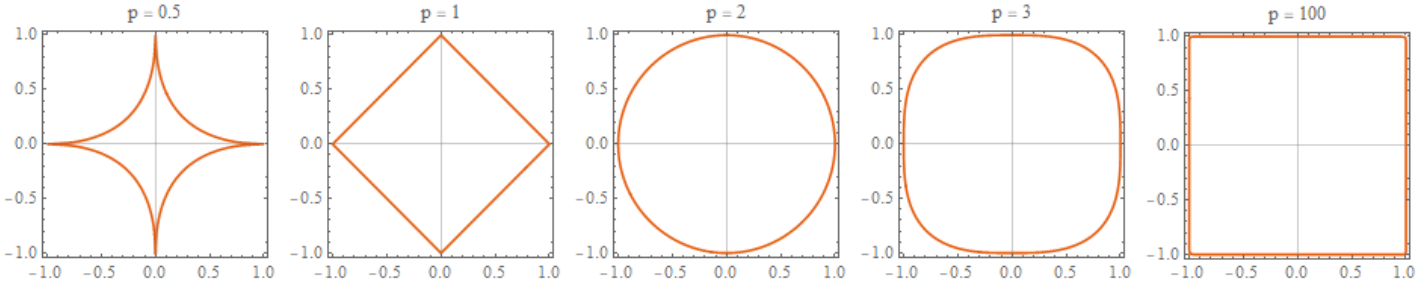
$$\min_{\boldsymbol{w},b} \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(\boldsymbol{x}_i)) + \lambda\Omega(\boldsymbol{w})$$

- Different choices of loss and regularization yield different learning algorithms for linear classifiers – all trainable with SGD (and scikit-learn). Some examples are:
  - Hinge + l2: SVM
  - Hinge + l1: 1-norm SVM
  - Squared loss + l2: Ridge Regression
  - Squared loss + l1: LASSO
  - Logistic loss + l2: Logistic Regression

Regularizers

- $\Omega(\boldsymbol{w})$: typically convex penalties on the classifier parameters are used
  - ℓ_p norms with $p \geq 1$, $\ell_p(\boldsymbol{w}) = \left(\sum_j |w_j|^p\right)^{1/p}$

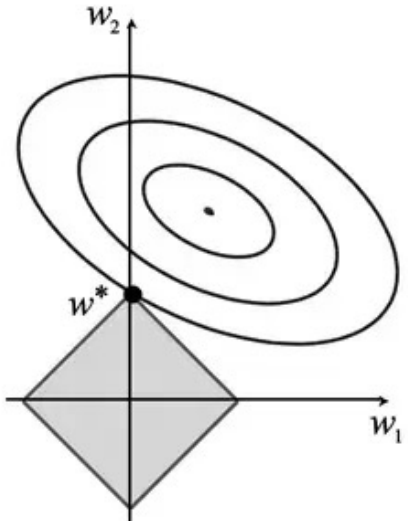
- Most popular examples
  - ℓ_0 is not convex, and amounts to counting non-zero elements in $\boldsymbol{w}$
  - $\ell_1 = |w_1| + |w_2| + \cdots + |w_d|$
  - $\ell_2 = w_1^2 + w_2^2 + \ldots + w_d^2$
  - $\ell_\infty = \max_j |w_j|$



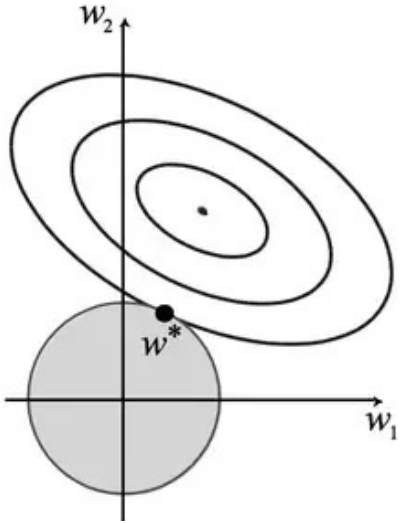
Sparsity

- ℓ_0 and ℓ_1 regularization enforce *sparsity*, i.e., many values in $\boldsymbol{w}$ will be set to zero
  - Why? The optimum is often found at one of the vertices!

- Sparsity
  perform feature selection
- Features
  can be disregarded



ℓ_1



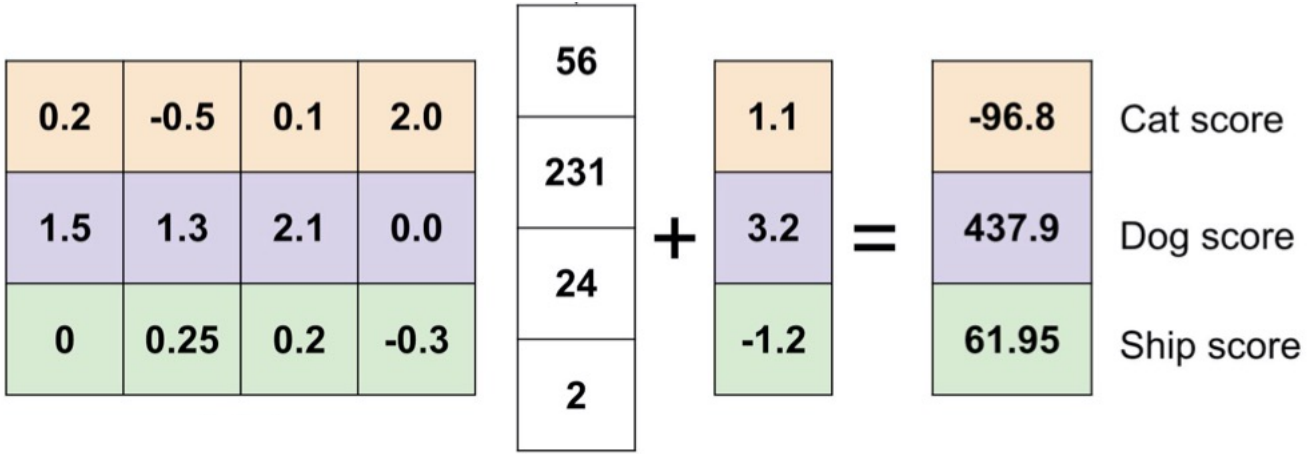
ℓ_2

Multiclass Classification with Linear Classifiers

Multiclass Linear Classifiers

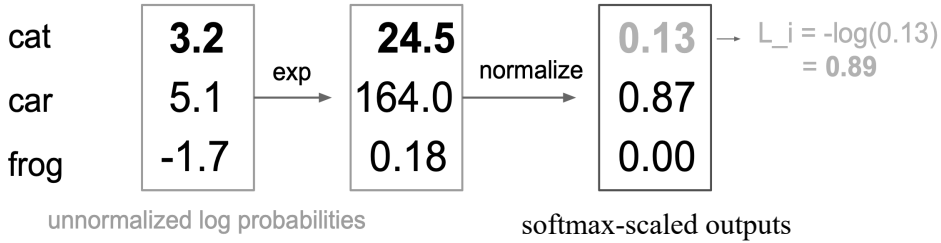
- Linear functions can be also naturally extended to multiclass problems

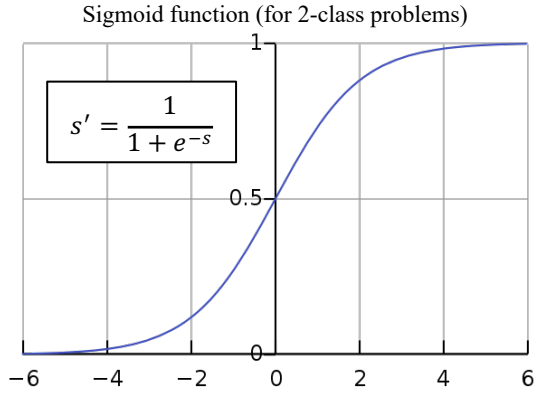
$$f(\boldsymbol{x}) = \mathbf{W}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}$$

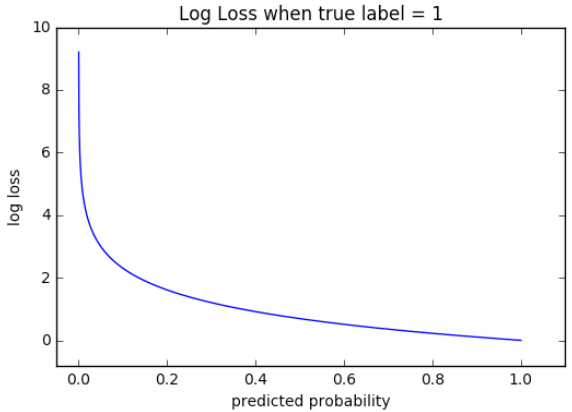


Logistic/Softmax Classifier

- Multiclass loss used to learn $\mathbf{W}, \boldsymbol{b}$
- Softmax scaling and cross-entropy (or log) loss
  - The classifier outputs one score per class $f(\boldsymbol{x}) = (s_1, \ldots, s_k)$
  - Outputs are softmax scaled: $s'_l = \frac{e^{s_l}}{\sum_j e^{s_j}}$
  - Cross-entropy loss: $L(y_i, f(\boldsymbol{x}_i)) = -\log(s'_{y_i})$







More at: https://cs231n.github.io/linear-classify/

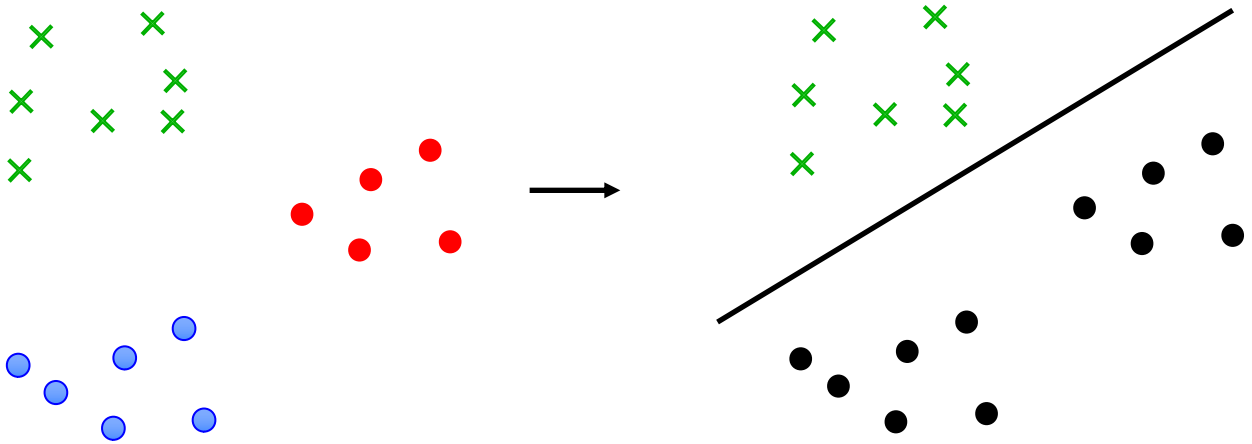
Multiclass Classification with Binary Classifiers

Multiclass Classification with Binary (two-class) Classifiers

- How to use binary classifiers (not necessarily linear) for multi-class classification (c>2)?
- Recall that one object belongs only to one class!
  - y is in the set {0, 1, ..., c-1}
- Two common schemes:
  - One-vs-All (or One-vs-Rest) classification (OVA / OVR)
  - One-vs-One classification (OVO)

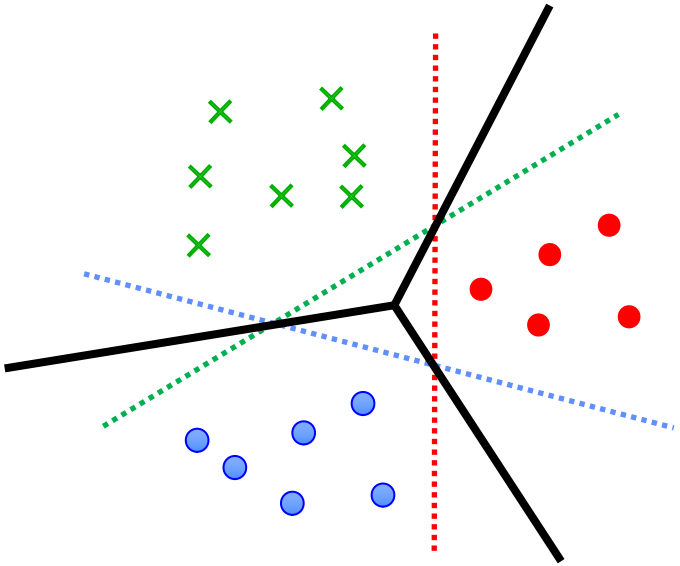
One-vs-All Multiclass Classification

- Train one binary classifier for each class k
- Samples for which y=k are labeled as +1
- The remaining classes are all labeled as -1



One-vs-All Multiclass Classification

- Combine with $y = \text{argmax}_k \, f_k(\mathbf{x})$



One-vs-One Multiclass Classification

- Train one binary classifier for class i vs class j
- All possible pairs are considered
  - c(c-1)/2 total number of binary classifiers
- Combined as:

$$f(\mathbf{x}) = \arg\max_i \left(\sum_j f_{ij}(\mathbf{x}) \right)$$

Note that $f_{ij} = -f_{ji}$ and that the sum is for j=0,...,c-1

Pros and Cons

- One-vs-all classification
  - One classifier per class, trained using all data
- One-vs-one classification
  - combinatorial number of classifiers (all possible pairs, c(c-1)/2), trained using much smaller data subsets
- Trade-off between number of classifiers and complexity of the learning algorithm
- In practice, rather equivalent classification accuracies
  - No free lunch

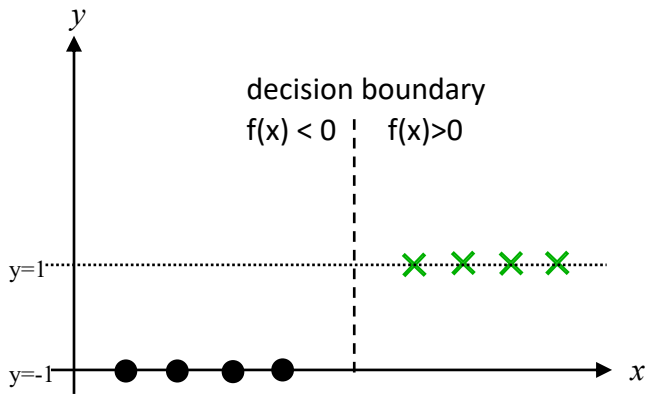
Classification vs Regression (with linear estimators)

Classification vs Regression

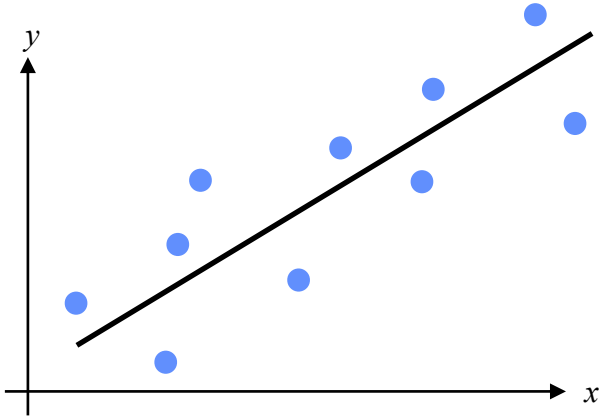
- In both classification and regression, the goal is to estimate a function f(**x**) that estimates the truth value y for each given sample **x**
- **In this lecture, we've discussed linear functions**
- In classification problems, the set of labels is discretized
  - $y \in \{0, \dots, c-1\}$
- In regression problems, the set of labels (or better, target values) is typically a continuous value
  - $y \in \mathrm{R}$ (e.g., the set of real numbers)

Classification vs Regression

Classification (estimates y in a discrete set, e.g., {-1,1})

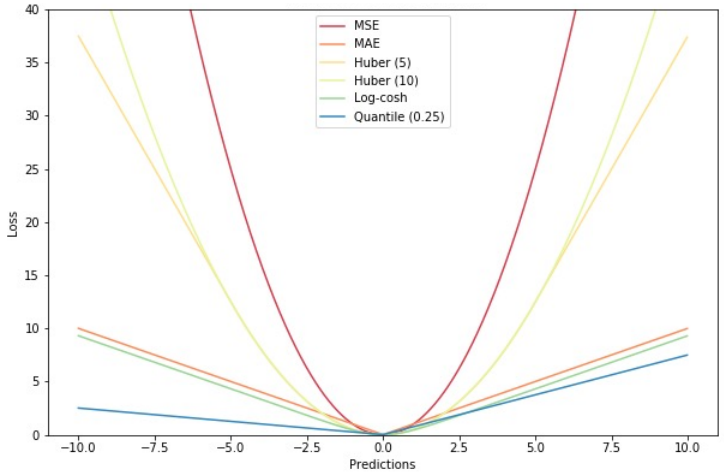


Regression estimates continuous-valued *y*



Classification vs Regression

- The loss functions used in the two problems reflect this behavior
- For classification problems, correctly classified points are assigned a loss equal to zero
  - e.g., the hinge loss gives zero penalty to points for which $yf(\boldsymbol{x}) \geq 1$
- For regression problems, the loss is zero only if $f(\boldsymbol{x})$ is exactly equal to y
  - e.g., the mean squared error (MSE) is given as the average of $\left(\mathrm{y} - f(\boldsymbol{x})\right)^2$ over all points



Ridge Regression

- It uses the mean squared error (MSE) as the error function and l2 regularization on the feature weights:

$$L(\boldsymbol{w}) = \frac{1}{2n}\|\mathbf{X}\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{w}\|^2$$

- Minimizing $L(\boldsymbol{w})$ provides the following closed-form solution:

$$\boldsymbol{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\boldsymbol{y}\,,$$

- being **I** the identity matrix, and $\lambda > 0$ a trade-off parameter.

- In this case, adding a small diagonal (ridge) to the (positive semi-definite) matrix $\mathbf{X}^T\mathbf{X}$ makes it more stable for pseudo-inversion (as it increases its minimum eigenvalue)

Ridge Regression

- Ridge regression can be solved in closed form, through matrix pseudo-inversion
  - Too computationally demanding for large feature sets and datasets

- It is also possible to solve it using gradient-descent procedures, including SGD, which is much faster and better suited to large, high-dimensional training sets

LASSO and Elastic Net

- Least Absolute Shrinkage and Selection Operator (LASSO) uses sparse penalty on $\boldsymbol{w}$ for embedded feature selection

$$L(\boldsymbol{w}) = \frac{1}{2n}\|\mathbf{X}\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{w}\|_1$$

- The Elastic Net overcomes some issues arising when learning LASSO on badly-conditioned problems, by combining l1 and l2 regularization

$$L(\boldsymbol{w}) = \frac{1}{2n}\|\mathbf{X}\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda_1\|\boldsymbol{w}\|_1 + \lambda_2\|\boldsymbol{w}\|_2^2$$