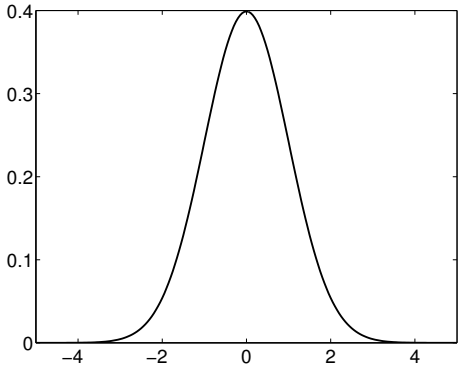
TOOLS: MAXIMUM LIKELIHOOD

Gaussian density in one dimension

$$g(x;\mu,\sigma) := \frac{1}{\sqrt{2\pi}\sigma}\exp\Big(-\frac{(x-\mu)^2}{2\sigma^2}\Big)$$

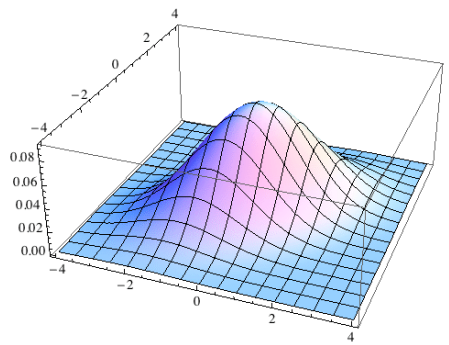


- μ = expected value of x, σ^2 = variance, σ = standard deviation
- The quotient $\frac{x-\mu}{\sigma}$ measures deviation of x from its expected value in units of σ (i.e. σ defines the length scale)

Gaussian density in d dimensions

The quadratric function

$$-\frac{(x-\mu)^2}{2\sigma^2} \quad = \quad -\frac{1}{2}(x-\mu)(\sigma^2)^{-1}(x-\mu)$$



is replaced by a quadratic form:

$$g(\mathbf{x};\boldsymbol{\mu},\Sigma) := \frac{1}{\sqrt{2\pi\det(\Sigma)}}\exp\Big(-\frac{1}{2}\Big\langle(\mathbf{x}-\boldsymbol{\mu}),\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\Big\rangle\Big)$$

Models

A **model** $\mathcal{P}$ is a set of probability distributions. We index each distribution by a parameter value $\theta \in \mathcal{T}$; we can then write the model as

$$\mathcal{P} = \{P_\theta | \theta \in \mathcal{T}\} .$$

The set $\mathcal{T}$ is called the **parameter space** of the model.

Parametric model

The model is called **parametric** if the number of parameters (i.e. the dimension of the vector θ) is (1) finite and (2) independent of the number of data points.
Intuitively, the complexity of a parametric model does not increase with sample size.

Density representation

For parametric models, we can assume that $\mathcal{T} \subset \mathbb{R}^d$ for some fixed dimension d. We usually represent each P_θ be a density function $p(x|\theta)$.

Setting

- Given: Data $x_1, \ldots, x_n$, parametric model $\mathcal{P} = \{p(x|\theta) \mid \theta \in \mathcal{T}\}$.
- Objective: Find the distribution in $\mathcal{P}$ which best explains the data. That means we have to choose a "best" parameter value $\hat{\theta}$.

Maximum Likelihood approach

Maximum Likelihood assumes that the data is best explained by the distribution in $\mathcal{P}$ under which it has the highest probability (or highest density value).

Hence, the **maximum likelihood estimator** is defined as

$$\hat{\theta}_{\text{ML}} := \arg\max_{\theta \in \mathcal{T}} p(x_1, \ldots, x_n|\theta)$$

the parameter which maximizes the joint density of the data.

The i.i.d. assumption

The standard assumption of ML methods is that the data is **independent and identically distributed (i.i.d.)**, that is, generated by independently sampling repeatedly from the same distrubtion P.

If the density of P is $p(x|\theta)$, that means the joint density decomposes as

$$p(x_1, \ldots, x_n) = \prod_{i=1}^{n} p(x_i|\theta)$$

Maximum Likelihood equation

The analytic criterion for a maximum likelihood estimator (under the i.i.d. assumption) is:

$$\nabla_\theta \left(\prod_{i=1}^{n} p(x_i|\theta) \right) = 0$$

We use the "logarithm trick" to avoid a huge product rule computation.

Recall: Logarithms turn products into sums

$$\log\left(\prod_i f_i\right) = \sum_i \log(f_i)$$

Logarithms and maxima

The logarithm is monotonically increasing on $\mathbb{R}_+$.

Consequence: Application of log does not change the *location* of a maximum or minimum:

$$\max_y \log(g(y)) \neq \max_y g(y) \qquad \text{The } \textit{value} \text{ changes.}$$

$$\arg\max_y \log(g(y)) = \arg\max_y g(y) \qquad \text{The } \textit{location} \text{ does not change.}$$

Likelihood and logarithm trick

$$\hat{\theta}_{\mathrm{ML}} = \arg\max_{\theta} \prod_{i=1}^{n} p(x_i|\theta) = \arg\max_{\theta} \log\Big(\prod_{i=1}^{n} p(x_i|\theta)\Big) = \arg\max_{\theta} \sum_{i=1}^{n} \log p(x_i|\theta)$$

Analytic maximality criterion

$$0 = \sum_{i=1}^{n} \nabla_{\theta} \log p(x_i|\theta) = \sum_{i=1}^{n} \frac{\nabla_{\theta} p(x_i|\theta)}{p(x_i|\theta)}$$

Whether or not we can solve this analytically depends on the choice of the model!

Example: Gaussian Mean MLE

Model: Multivariate Gaussians

The model $\mathcal{P}$ is the set of all Gaussian densities on $\mathbb{R}^d$ with *fixed* covariance matrix Σ,

$$\mathcal{P} = \{g(\,.\,|\mu, \Sigma) \,|\, \mu \in \mathbb{R}^d\}\ ,$$

where g is the Gaussian density function. The parameter space is $\mathcal{T} = \mathbb{R}^d$.

MLE equation

We have to solve the maximum equation

$$\sum_{i=1}^{n} \nabla_\mu \log g(x_i|\mu, \Sigma) = 0$$

for μ.

Example: Gaussian Mean MLE

$$
\begin{aligned}
0 &= \sum_{i=1}^{n} \nabla_\mu \log \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\Big(-\frac{1}{2}\left\langle (x_i-\mu), \Sigma^{-1}(x_i-\mu)\right\rangle\Big) \\
&= \sum_{i=1}^{n} \nabla_\mu \Big(\log\Big(\frac{1}{\sqrt{(2\pi)^d |\Sigma|}}\Big) + \log\Big(\exp\Big(-\frac{1}{2}\left\langle (x_i-\mu), \Sigma^{-1}(x_i-\mu)\right\rangle\Big)\Big) \\
&= \sum_{i=1}^{n} \nabla_\mu \Big(-\frac{1}{2}\left\langle (x_i-\mu), \Sigma^{-1}(x_i-\mu)\right\rangle\Big) = -\sum_{i=1}^{n} \Sigma^{-1}(x_i-\mu)
\end{aligned}
$$

Multiplication by $(-\Sigma)$ gives

$$
0 = \sum_{i=1}^{n} (x_i - \mu) \qquad \Rightarrow \qquad \mu = \frac{1}{n}\sum_{i=1}^{n} x_i
$$

Conclusion

The maximum likelihood estimator of the Gaussian expectation parameter for fixed covariance is

$$
\hat{\mu}_{\mathrm{ML}} := \frac{1}{n}\sum_{i=1}^{n} x_i
$$

Model: Multivariate Gaussians

The model $\mathcal{P}$ is now

$$\mathcal{P} = \{g(\,.\,|\mu, \Sigma) \,|\, \mu \in \mathbb{R}^d, \Sigma \in \Delta_d\}\ ,$$

where Δ_d is the set of positive definite $d \times d$-matrices. The parameter space is $\mathcal{T} = \mathbb{R}^d \times \Delta_d$.

ML approach

Since we have just seen that the ML estimator of μ does not depend on Σ, we can compute $\hat{\mu}_{\text{ML}}$ first. We then estimate Σ using the criterion

$$\sum_{i=1}^{n} \nabla_\Sigma \log g(x_i|\hat{\mu}_{\text{ML}}, \Sigma) = 0$$

Solution

The ML estimator of Σ is

$$\hat{\Sigma}_{\text{ML}} := \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu}_{\text{ML}})(x_i - \hat{\mu}_{\text{ML}})^t\ .$$

CLASSIFICATION

Assumptions and Terminology

In a **classification problem**, we record measurements $\mathbf{x}_1, \mathbf{x}_2, \ldots$.

We assume:

1. All measurements can be represented as elements of a Euclidean $\mathbb{R}^d$.
2. Each $\mathbf{x}_i$ belongs to exactly one out of K categories, called **classes**. We express this using variables $y_i \in [K]$, called **class labels**:

$$y_i = k \quad \Leftrightarrow \quad \text{"}\mathbf{x}_i \text{ in class } k\text{"}$$

3. The classes are characterized by the (unknown!) joint distribution of (X, Y), whose density we denote $p(x, y)$. The conditional distribution with density $p(x|y = k)$ is called the **class-conditional distribution** of class k.
4. The only information available on the distribution p is a set of example measurements *with* labels,

$$(\tilde{\mathbf{x}}_1, \tilde{y}_1), \ldots, (\tilde{\mathbf{x}}_n, \tilde{y}_n) \ ,$$

called the **training data**.

Definition

A **classifier** is a function

$$f : \mathbb{R}^d \longrightarrow [K] \ ,$$

i.e. a function whose argument is a measurement and whose output is a class label.

Learning task

Using the training data, we have to estimate a good classifier. This estimation procedure is also called **training**.

A good classifier should generalize well to new data. Ideally, we would like it to perform with high accuracy on data sampled from p, but all we know about p is the training data.

Simplifying assumption

We first develop methods for the two-class case (K=2), which is also called **binary classification**. In this case, we use the notation

$$y \in \{-1, +1\} \qquad \text{instead of} \qquad y \in \{1, 2\}$$

Supervised vs. unsupervised

Fitting a model using labeled data is called **supervised learning**. Fitting a model when only $\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_n$ are available, but no labels, is called **unsupervised learning**.

Types of supervised learning methods

- Classification: Labels are discrete, and we estimate a classifier $f : \mathbb{R}^d \longrightarrow [K]$,
- Regression: Labels are real-valued ($y \in \mathbb{R}$), and we estimate a continuous function $f : \mathbb{R}^d \longrightarrow \mathbb{R}$. This functions is called a **regressor**.

Algorithm

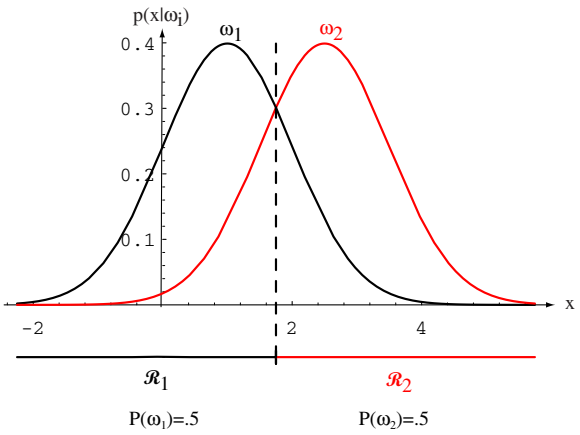
1. On training data, fit a Gaussian into each class (by MLE).
   Result: Densities $g(\mathbf{x}|\mu_\oplus, \Sigma_\oplus)$ and $g(\mathbf{x}|\mu_\ominus, \Sigma_\ominus)$
2. Classify test point according to which density assigns larger value:

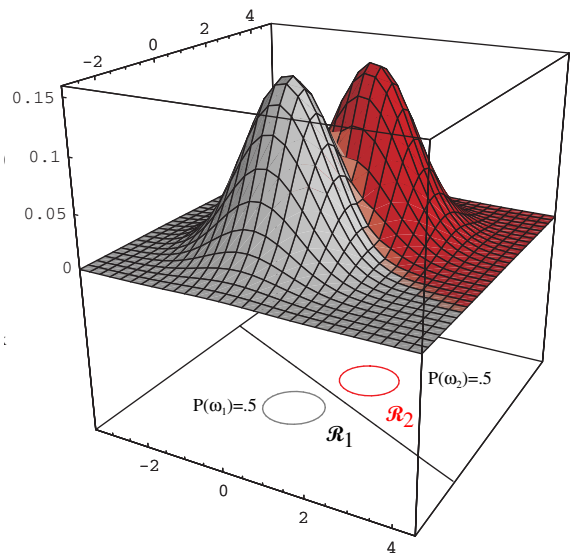
$$y_i := \begin{cases} +1 & \text{if } g(\mathbf{x}_i|\mu_\oplus, \Sigma_\oplus) > g(\mathbf{x}_i|\mu_\ominus, \Sigma_\ominus) \\ -1 & \text{otherwise} \end{cases}$$

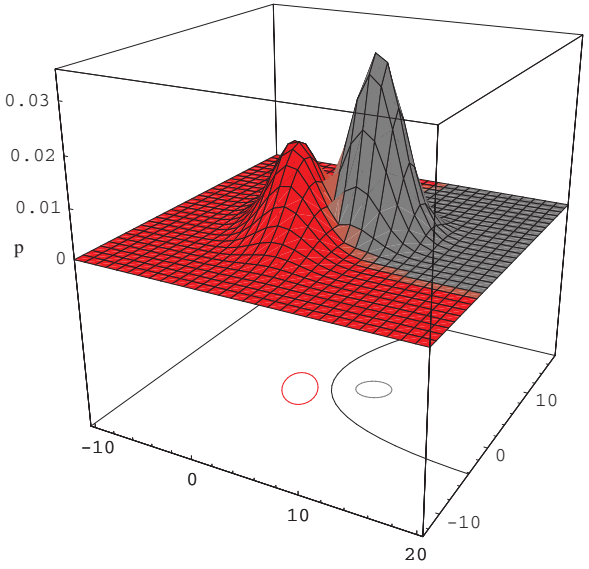
Resulting classifier

- Hyperplane if $\Sigma_\oplus$=$\Sigma_\ominus$ = constant $\cdot$ diag$(1, \ldots, 1)$ (=isotropic Gaussians)
- Quadratic hypersurface otherwise.

A Very Simple Classifier







Possible weakness

1. Distributional assumption.
2. Density estimates emphasize main bulk of data. Critical region for classification is at decision boundary, i.e. region between classes.

Consequence

- Classification algorithms focus on class boundary.
- Technically, this means: We focus on estimating a good decision surface (e.g. a hyperplane) between the classes; we do *not* try to estimate a distribution.

Our program in the following

- First develop methods for the linear case, i.e. separate classes by a hyperplane.
- Then: Consider methods that transform linear classifier into non-linear ones.
- Finally: Discuss a family of classification methods that are non-linear by design.

Definition

A **loss function** is a function

$$L : [K] \times [K] \longrightarrow [0, \infty) \ ,$$

which we read as

$$L : (\text{true class label } y, \ \text{classifier output } f(x)) \longmapsto \ \text{loss value} \ \ .$$

Example: The two most common loss functions

1. The **0-1 loss** is used in classification. It counts mistakes:

$$L^{\text{0-1}}(y, f(\mathbf{x})) = \begin{cases} 0 & f(\mathbf{x}) = y \\ 1 & f(\mathbf{x}) \neq y \end{cases}$$

2. **Squared-error loss** is used in regression:

$$L^{\text{se}}(y, f(\mathbf{x})) := \|y - f(\mathbf{x})\|_2^2$$

   Its value depends on how far off we are: Small errors hardly count, large ones are very expensive.

Motivation

It may be a good strategy to allow (even expensive) errors for values of $\mathbf{x}$ which are very unlikely to occur

Definition

The **risk** $R(f)$ of a classifier f is its expected loss under p, that is,

$$R(f) := \mathbb{E}_p[L(y, f(\mathbf{x})] = \int L(y, f(\mathbf{x}))p(\mathbf{x}, y)d\mathbf{x}dy = \sum_{y=1}^{K} \int L(y, f(\mathbf{x}))p(\mathbf{x}, y)d\mathbf{x} \ .$$

When we train f, we do not know p, and have to approximate R using the data:

The **empirical risk** $\hat{R}_n(f)$ is the plug-in estimate of $R(f)$, evaluated on the training sample $(\tilde{\mathbf{x}}_1, \tilde{y}_1), \ldots, (\tilde{\mathbf{x}}_n, \tilde{y}_n)$:

$$\hat{R}_n(f) := \frac{1}{n} \sum_{i=1}^{n} L(\tilde{y}_i, f(\tilde{\mathbf{x}}_i))$$

NAIVE BAYES CLASSIFIERS

Bayes Equation

Simplest form

- Random variables $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$, where $\mathbf{X}, \mathbf{Y}$ are finite sets.
- Each possible value of X and Y has positive probability.

Then

$$P(X = x, Y = y) = P(y|x)P(x) = P(x|y)P(y)$$

and we obtain

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} = \frac{P(x|y)P(y)}{\sum_{y \in \mathcal{Y}} P(x|y)P(y)}$$

It is customary to name the components,

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}}$$

In terms of densities

For continuous sets $\mathbf{X}$ and $\mathbf{Y}$,

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} = \frac{p(x|y)p(y)}{\int_{\mathbf{Y}} p(x|y)dy}$$

Classification

We define a classifier as

$$f(\mathbf{x}) := \arg\max_{y \in [K]} p(y|\mathbf{x})$$

where $\mathbf{Y} = [K]$ and $\mathbf{X}$ = sample space of data variable.

With the Bayes equation, we obtain

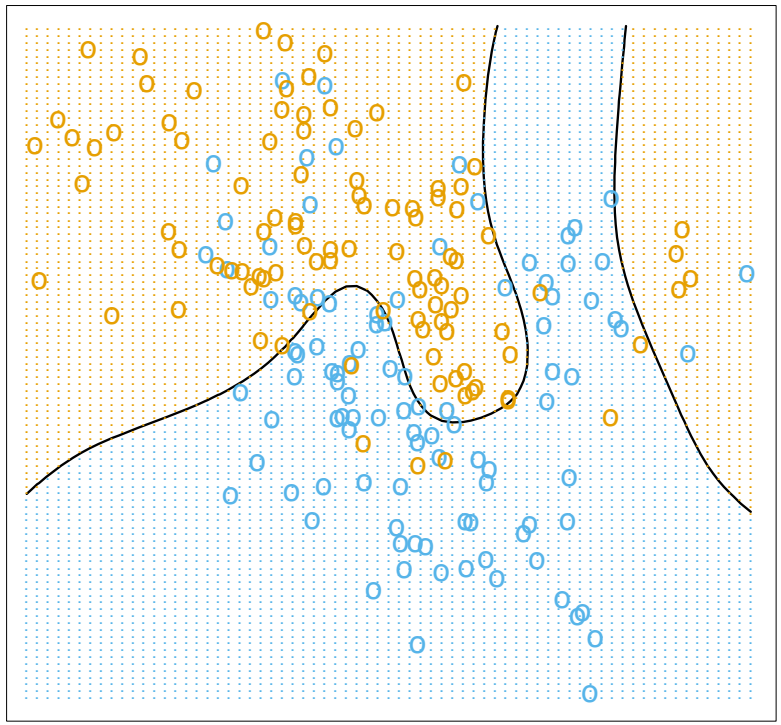
$$f(\mathbf{x}) = \arg\max_{y} \frac{P(x|y)P(y)}{P(x)} = \arg\max_{y} P(x|y)P(y)$$

If the class-conditional distribution is continuous, we use

$$f(\mathbf{x}) = \arg\max_{y} p(x|y)P(y)$$

Optimal classifier

- In the risk framework, the best possible classifier is the one which minimizes the expected risk.
- Which classifier is optimal depends on the chosen cost function.



Zero-one loss

Under zero-one loss, the classifier which minimizes the risk is the classifier

$$f(\mathbf{x}) = \arg\max_y P(x|y)P(y)$$

from the previous slide. When computed from the *true* distribution of (X, Y), this classifier is called the **Bayes-optimal classifier** (or **Bayes classifier** for short).

EXAMPLE: SPAM FILTERING

Representing emails

- $\mathbf{Y} = \{$ spam, email $\}$
- $\mathbf{X} = \mathbb{R}^d$
- Each axis is labelled by one possible word.
- d = number of words in vocabulary
- x_j = number of occurences of word j in email represented by $\mathbf{x}$

For example, if axis j represents the term "the", $x_j = 3$ means that "the" occurs three times in the email $\mathbf{x}$. This representation is called a **vector space model of text**.

Example dimensions

	george	you	your	hp	free	hpl	!	our	re	edu	remove
spam	0.00	2.26	1.38	0.02	0.52	0.01	0.51	0.51	0.13	0.01	0.28
email	1.27	1.27	0.44	0.90	0.07	0.43	0.11	0.18	0.42	0.29	0.01

With Bayes equation

$$f(\mathbf{x}) = \underset{y \in \{\text{spam,email}\}}{\operatorname{argmax}} P(y|\mathbf{x}) = \underset{y \in \{\text{spam,email}\}}{\operatorname{argmax}} p(\mathbf{x}|y)P(y)$$

Simplifying assumption

The classifier is called a **naive Bayes** classifier if it assumes

$$p(\mathbf{x}|y) = \prod_{j=1}^{d} p_j(x_i|y) \ ,$$

i.e. if it treats the individual dimensions of $\mathbf{x}$ as conditionally independent given y.

In spam example

- Corresponds to the assumption that the number of occurrences of a word carries information about y.
- Co-occurrences (how often do given combinations of words occur?) is neglected.

Class prior

The distribution $P(y)$ is easy to estimate from training data:

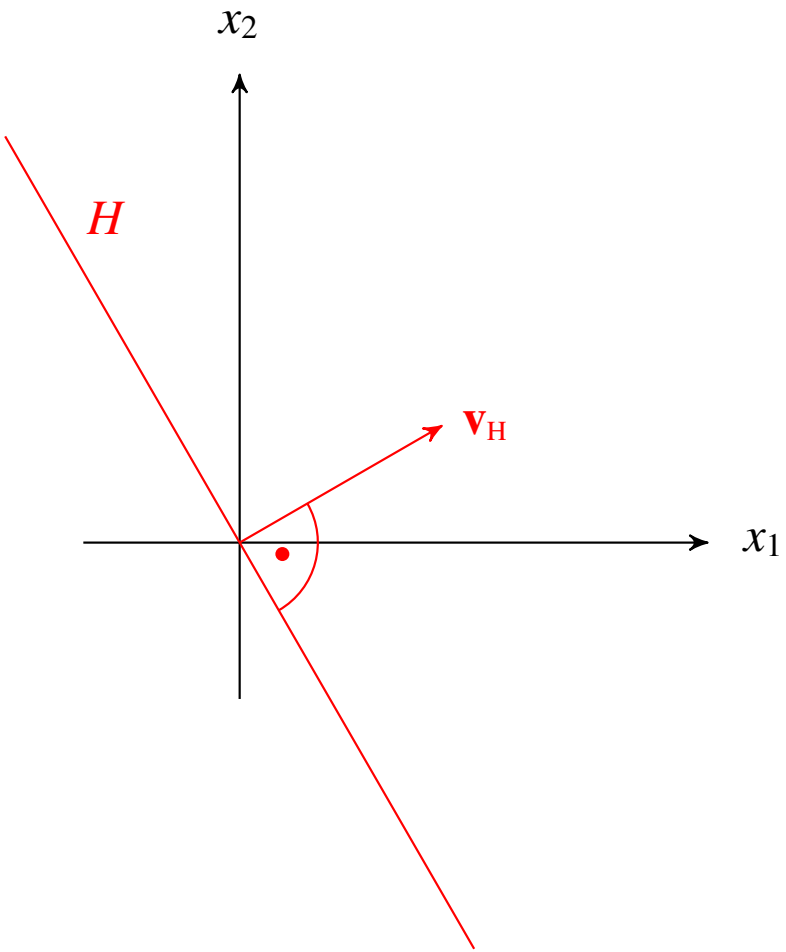
$$P(y) = \frac{\#\text{observations in class } y}{\#\text{observations}}$$

Class-conditional distributions

The class conditionals $p(x|y)$ usually require a modeling assumption. Under a given model:

- Separate the training data into classes.
- Estimate $p(x|y)$ on class y by maximum likelihood.

Linear Classification



Hyperplanes

A **hyperplane** in $\mathbb{R}^d$ is a linear subspace of dimension $(d-1)$.

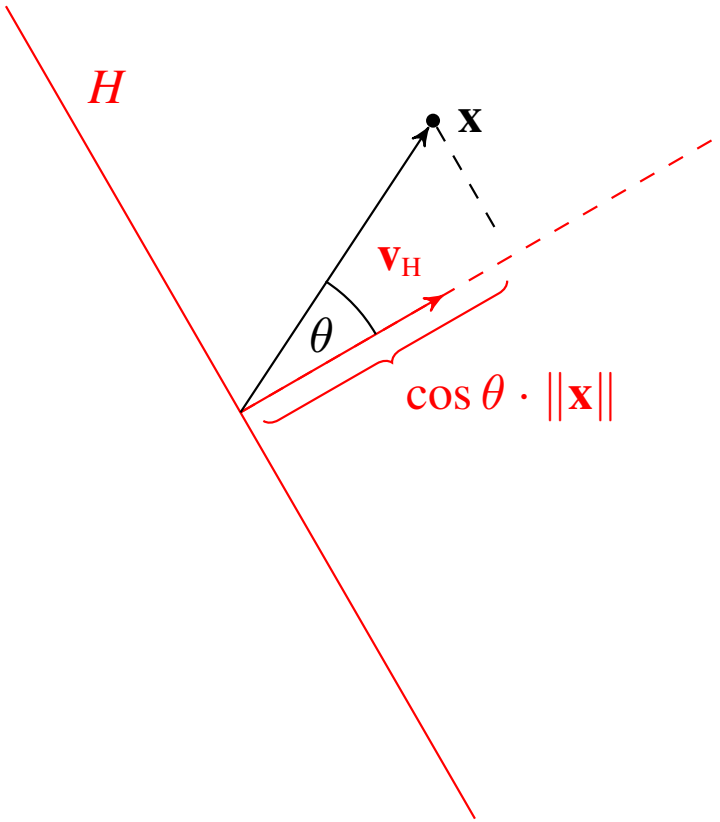
- A $\mathbb{R}^2$-hyperplane is a line, a $\mathbb{R}^3$-hyperplane is a plane.
- As a linear subspace, a hyperplane always contains the origin.

Normal vectors

A hyperplane H can be represented by a **normal vector**. The hyperplane with normal vector $\mathbf{v}_H$ is the set

$$H = \{\mathbf{x} \in \mathbb{R}^d \mid \langle \mathbf{x}, \mathbf{v}_H \rangle = 0\} \ .$$

WHICH SIDE OF THE PLANE ARE WE ON?



Distance from the plane

- The projection of $\mathbf{x}$ onto the direction of $\mathbf{v}_H$ has length $\langle \mathbf{x}, \mathbf{v}_H \rangle$ *measured in units of* $\mathbf{v}_H$, i.e. length $\langle \mathbf{x}, \mathbf{v}_H \rangle / \|\mathbf{v}_H\|$ in the units of the coordinates.
- Recall the cosine rule for the scalar product,

$$\cos\theta = \frac{\langle \mathbf{x}, \mathbf{v}_H \rangle}{\|\mathbf{x}\| \cdot \|\mathbf{v}_H\|} \ .$$

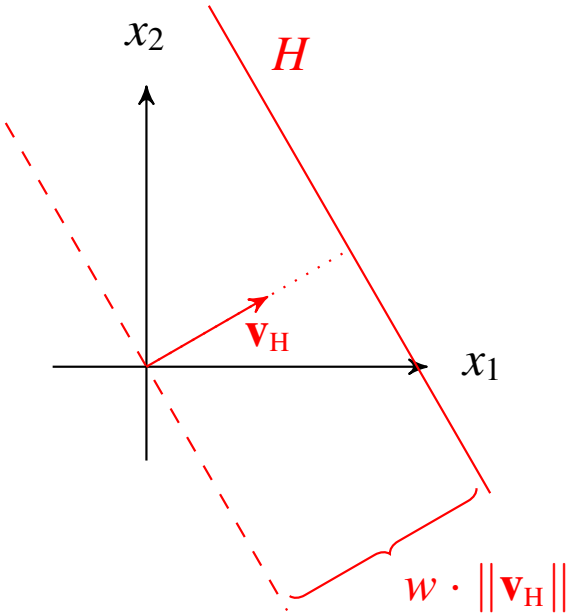
- Consequence: The distance of $\mathbf{x}$ from the plane is given by

$$d(\mathbf{x}, H) = \frac{\langle \mathbf{x}, \mathbf{v}_H \rangle}{\|\mathbf{v}_H\|} = \cos\theta \cdot \|\mathbf{x}\| \ .$$

Which side of the plane?

- The cosine satisfies $\cos\theta > 0$ iff $\theta \in (-\pi, \pi)$.
- We can decide which side of the plane $\mathbf{x}$ is on using

$$\operatorname{sgn}(\cos\theta) = \operatorname{sgn} \langle \mathbf{x}, \mathbf{v}_H \rangle \ .$$



Affine Hyperplanes

- An **affine hyperplane** $H_{\mathbf{w}}$ is a hyperplane translated (shifted) by a vector $\mathbf{w}$, i.e. $H_{\mathbf{w}} = H + \mathbf{w}$.
- We choose $\mathbf{w}$ in the direction of $\mathbf{v}_H$, i.e. $\mathbf{w} = c \cdot \mathbf{v}_H$ for $c > 0$.

Which side of the plane?

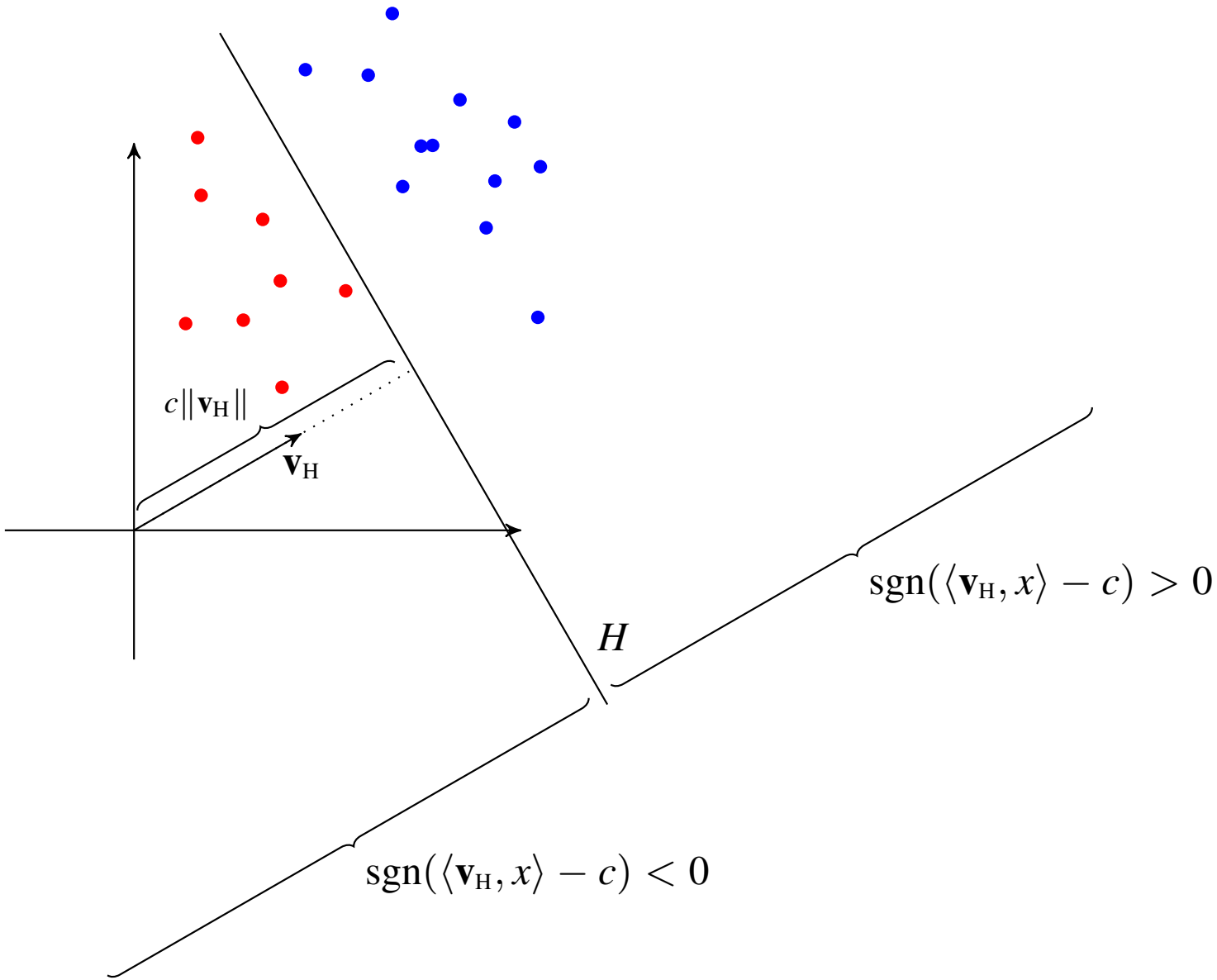
- Which side of $H_{\mathbf{w}}$ a point $\mathbf{x}$ is on is determined by

$$\operatorname{sgn}(\langle \mathbf{x} - \mathbf{w}, \mathbf{v}_H \rangle) = \operatorname{sgn}(\langle \mathbf{x}, \mathbf{v}_H \rangle - c \langle \mathbf{v}_H, \mathbf{v}_H \rangle) = \operatorname{sgn}(\langle \mathbf{x}, \mathbf{v}_H \rangle - c\|\mathbf{v}_H\|^2) \ .$$

- If $\mathbf{v}_H$ is a unit vector, we can use

$$\operatorname{sgn}(\langle \mathbf{x} - \mathbf{w}, \mathbf{v}_H \rangle) = \operatorname{sgn}(\langle \mathbf{x}, \mathbf{v}_H \rangle - c) \ .$$

Classification with Affine Hyperplanes



Definition

A **linear classifier** is a function of the form

$$f_{\mathrm{H}}(\mathbf{x}) := \operatorname{sgn}(\langle \mathbf{x}, \mathbf{v}_{\mathrm{H}} \rangle - c) \ ,$$

where $\mathbf{v}_{\mathrm{H}} \in \mathbb{R}^d$ is a vector and $c \in \mathbb{R}_+$.

Note: We usually assume $\mathbf{v}_{\mathrm{H}}$ to be a unit vector. If it is not, f_{H} still defines a linear classifier, but c describes a shift of a different length.

Definition

Two sets $A, B \in \mathbb{R}^d$ are called **linearly separable** if there is an affine hyperplane H which separates them, i.e. which satisfies

$$\langle \mathbf{x}, \mathbf{v}_{\mathrm{H}} \rangle - c = \begin{cases} < 0 & \text{if } \mathbf{x} \in A \\ > 0 & \text{if } \mathbf{x} \in B \end{cases}$$

The Perceptron Algorithm

Definition

Let $\mathcal{H}$ be the set of all classifiers considered in a given classification problem. The set $\mathcal{H}$ is called a **hypothesis space**.

For linear classifiers, $\mathcal{H} = \{$ all hyperplanes in $\mathbb{R}^d\}$.

Selecting a classifier

Select $f \in \mathcal{H}$ which minimizes risk. With zero-one loss:

$$f \in \operatorname*{argmin}_{f \in \mathcal{H}} R(f) = \operatorname*{argmin}_{f \in \mathcal{H}} \mathbb{E}_p[L(y, f(\mathbf{x}))]$$

We cannot evaluatate this expression, since we do not know p.

Approximation with data: Empirical risk minimization

We approximate the risk criterion by the empirical risk

$$f \in \arg\min f \in \mathcal{H} \hat{R}_n(f) = \operatorname*{argmin}_{f \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(\mathbf{x}_i))$$

If we choose $L = L^{0\text{-}1}$, this minimizes the number of errors on the training data.

Parameterizing the hypothesis space

- Linear classification: Every $f \in \mathcal{H}$ is of the form $f(\mathbf{x}) = \operatorname{sgn}(\langle \mathbf{x}, \mathbf{v}_{\mathrm{H}} \rangle - c)$.
- f can be specified by specifying $\mathbf{v}_{\mathrm{H}} \in \mathbb{R}^d$ and $c \in \mathbb{R}$.
- We collect $\mathbf{v}_{\mathrm{H}}$ and c in a single vector $\mathbf{z} := (-c, \mathbf{v}_{\mathrm{H}}) \in \mathbb{R}^{d+1}$.

We now have

$$\langle \mathbf{x}, \mathbf{v}_{\mathrm{H}} \rangle - c = \left\langle \begin{pmatrix} 1 \\ \mathbf{x} \end{pmatrix}, \mathbf{z} \right\rangle \qquad \text{and} \qquad f(\mathbf{x}) = \operatorname{sgn}\left\langle \begin{pmatrix} 1 \\ \mathbf{x} \end{pmatrix}, \mathbf{z} \right\rangle$$

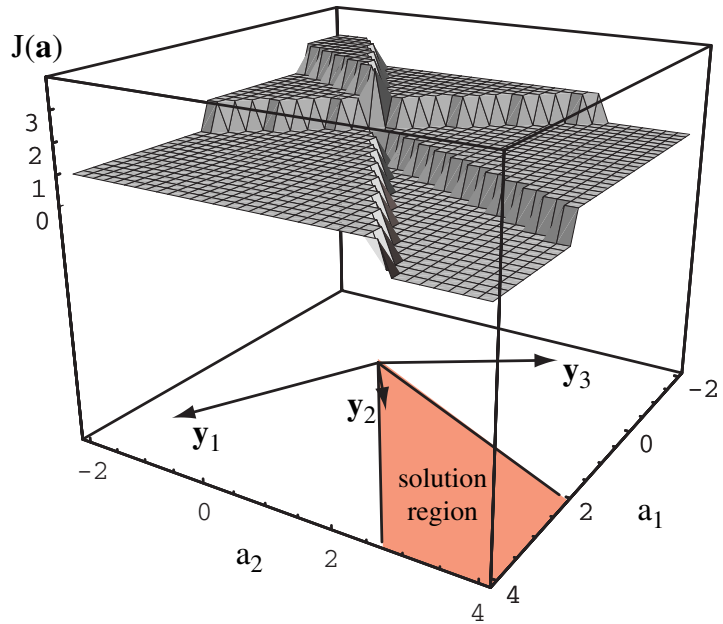
The *affine* plane in $\mathbb{R}^d$ can now be interpreted as a *linear* plane in $\mathbb{R}^{d+1}$. The $d+1$-dimensional coordinates in the representation are called **homogeneous coordinates**.

Numerical minimization of the empirical risk

Naive strategy:

1. Substitute the parametrization of f into $\hat{R}_n(f)$ (evaluated on the training data).
2. Minimize with respect to $\mathbf{z}$ by numerical optimization.

Problem: $\hat{R}_n(f)$ is piece-wise constant.



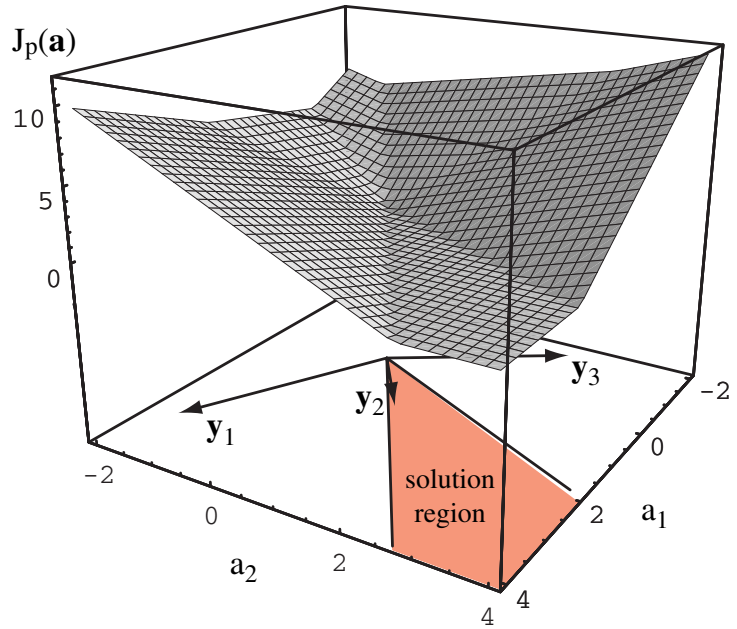
Solution region

The solution region is set of vectors $\mathbf{z}$ which achieve zero training error.

- If the training data is linearly separable, the solution region is a cone in $\mathbb{R}^{d+1}$.
- Otherwise, the solution region is empty.

Perceptron cost function

- Error rate not suited for numerical optimization.
- Strategy: Approximate $\hat{R}_n(f)$ by a piece-wise linear function.



The approximation

$$C_{\mathrm{P}}(f) := \sum_{i=1}^{n} \mathbb{I}\{f(\tilde{\mathbf{x}}_i) \neq \tilde{y}_i\} \left| \left\langle \mathbf{z}, \begin{pmatrix} 1 \\ \tilde{\mathbf{x}}_i \end{pmatrix} \right\rangle \right|$$

is called the **Perceptron cost function**.

Cost functions

The more general theme is that we substitute $\hat{R}_n$ by a **cost function** $C : \mathcal{H} \longrightarrow \mathbb{R}_+$.
A cost function defines a training strategy as

$$\text{training method} \;=\; \text{cost function} \;+\; \text{minimization algorithm}$$

The Perceptron

A linear classifier obtained by minimizing the Perceptron cost function is called a **Perceptron**.

Algorithm

Repeat until $C_{\mathrm{P}}(\mathbf{z}^k) = 0$:

$$\mathbf{z}^{k+1} := \mathbf{z}^k - \alpha(k)\nabla C_{\mathrm{P}}(\mathbf{z}^k)$$

where k enumerates iterations.

Step size

The step size parameter α is called the **learning rate**. Common choices are

$$\alpha(k) = 1 \qquad \text{or} \qquad \alpha(k) = \frac{1}{k}\ .$$

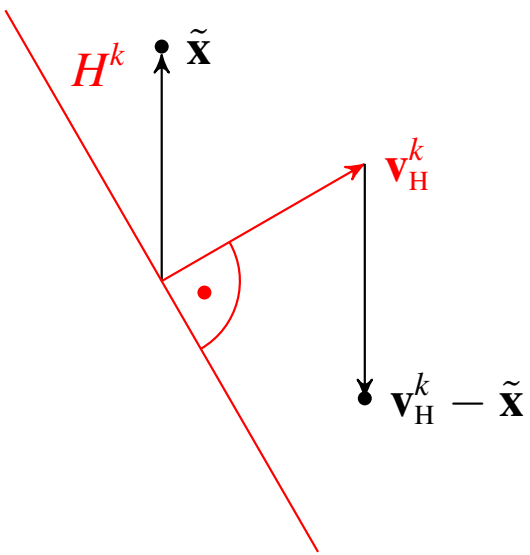
Gradient of the cost function

$$\nabla_{\mathbf{z}} C_{\mathrm{P}}(\mathbf{z}) = \sum_{i=1}^{n} \mathbb{I}\{f_{\mathrm{H}}(\tilde{\mathbf{x}}_i) \neq \tilde{y}_i\} \, \nabla_{\mathbf{z}} \left| \left\langle \mathbf{z}, \begin{pmatrix} 1 \\ \tilde{\mathbf{x}}_i \end{pmatrix} \right\rangle \right| = \sum_{i=1}^{n} \mathbb{I}\{f(\tilde{\mathbf{x}}_i) \neq \tilde{y}_i\} \cdot \operatorname{sgn}\left(\left\langle \mathbf{z}, \begin{pmatrix} 1 \\ \tilde{\mathbf{x}}_i \end{pmatrix} \right\rangle \right) \cdot \begin{pmatrix} 1 \\ \tilde{\mathbf{x}}_i \end{pmatrix}$$

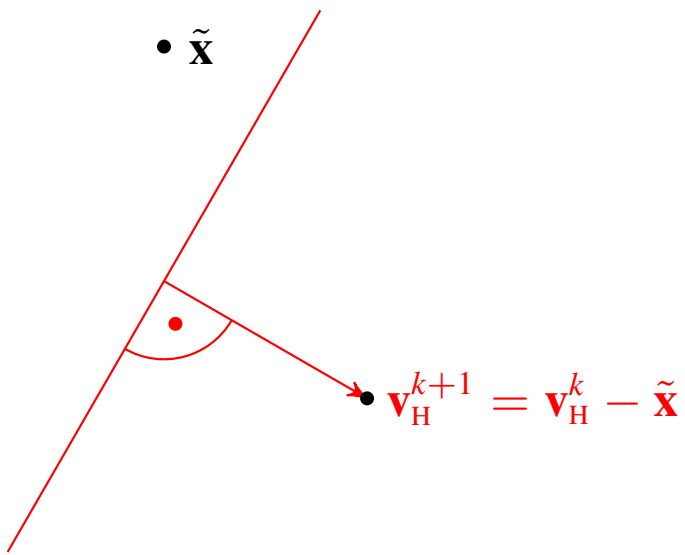
$$= \sum_{i=1}^{n} \mathbb{I}\{f(\tilde{\mathbf{x}}_i) \neq \tilde{y}_i\} \cdot (-\tilde{y}_i) \begin{pmatrix} 1 \\ \tilde{\mathbf{x}}_i \end{pmatrix} \; .$$

Effect for a single training point

Step k: $\tilde{\mathbf{x}}$ (in class -1) classified incorrectly



Step $k+1$



Simplifying assumption: H contains origin

Does the Perceptron Work?

The algorithm we discussed before is called the **batch Perceptron**. For learning rate $\alpha = 1$, we can equivalently add data points one at a time.

Alternative Algorithm

Repeat until $C_{\mathrm{P}}(\mathbf{z}) = 0$:

1. For all $i = 1, \ldots, n$: $\quad \mathbf{z}^k := \mathbf{z}^k + \tilde{y}_i \begin{pmatrix} 1 \\ \tilde{\mathbf{x}}_i \end{pmatrix}$
2. $k := k + 1$

This is called the **fixed-increment single-sample Perceptron**, and is somewhat easier to analyze than the batch Perceptron.

Theorem: Perceptron convergence

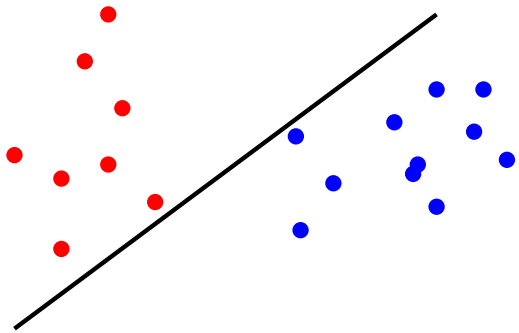
If (and only if) the training data is linearly separable, the fixed-increment single-sample Perceptron terminates after a finite number of steps with a valid solution vector $\mathbf{z}$ (i.e. a vector which classifies all training data points correctly).

Maximum Margin Classifiers

Maximum margin idea

Setting

Linear classification, two linearly separable classes.



Recall Perceptron

- Selects *some* hyperplane between the two classes.
- Choice depends on initialization, step size etc.

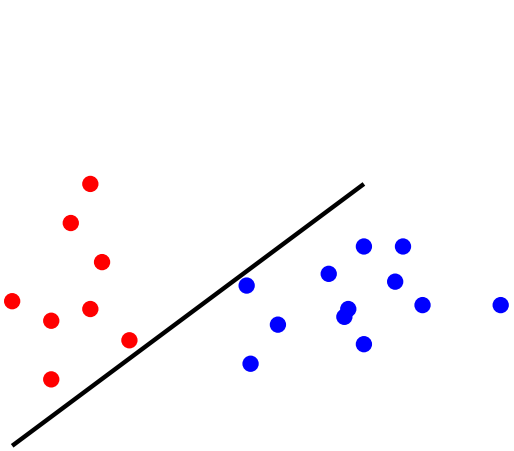
Maximum margin idea

To achieve good generalization (low prediction error), place the hyperplane "in the middle" between the two classes.

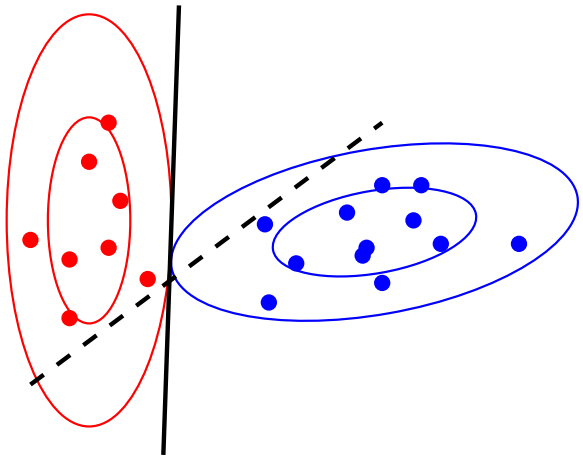
More precisely

Choose plane such that distance to closest point in each class is maximal. This distance is called the *margin*.

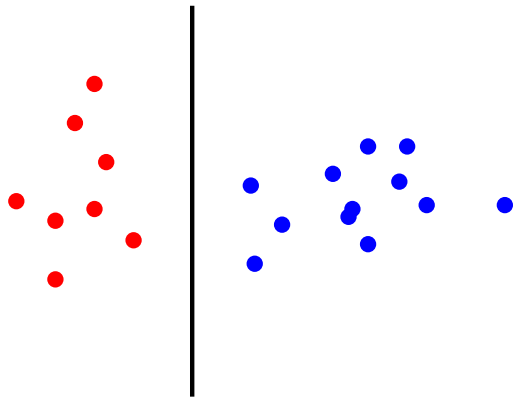
Generalization Error



Possible Perceptron solution



Good generalization under a specific distribution (here: Gaussian)



Maximum margin solution

Example: Gaussian data

- The ellipses represent lines of constant standard deviation (1 and 2 STD respectively).
- The 1 STD ellipse contains $\sim 65\%$ of the probability mass ($\sim 95\%$ for 2 STD; $\sim 99.7\%$ for 3 STD).

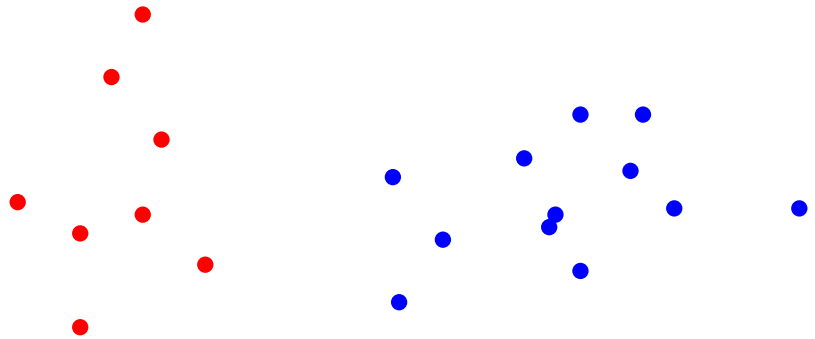
Optimal generalization: Classifier should cut off as little probability mass as possible from either distribution.

Without distributional assumption: Max-margin classifier

- Philosophy: Without distribution assumptions, best guess is symmetric.
- In the Gaussian example, the max-margin solution would *not* be optimal.

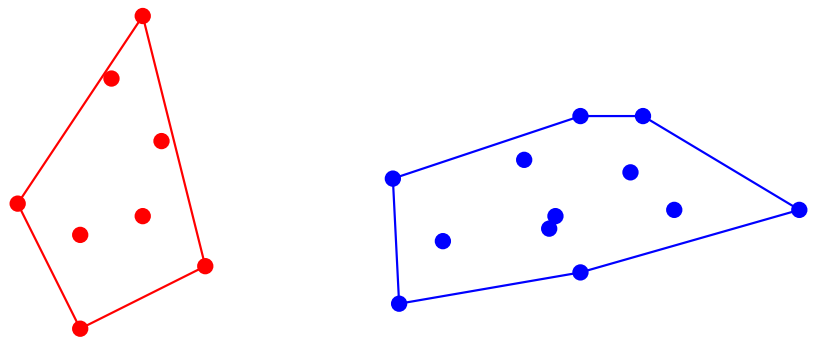
Observation

Where a separating hyperplane may be placed depends on the "outer" points on the sets. Points in the center do not matter.



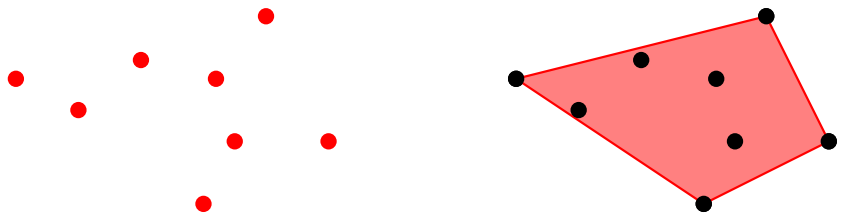
In geometric terms

Substitute each class by the smalles convex set which contains all point in the class:



Definition

If C is a set of points, the smallest convex set containing all points in C is called the **convex hull** of C, denoted $\text{conv}(C)$.

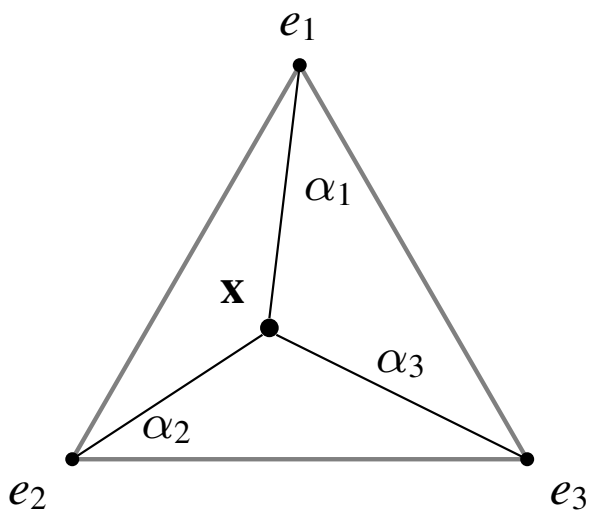


Corner points of the convex set are called **extreme points**.

Barycentric coordinates

Every point x in a convex set can be represented as a convex combination of the extreme points $\{e_1, \ldots, e_m\}$. There are weights $\alpha_1, \ldots, \alpha_m \in \mathbb{R}_+$ such that

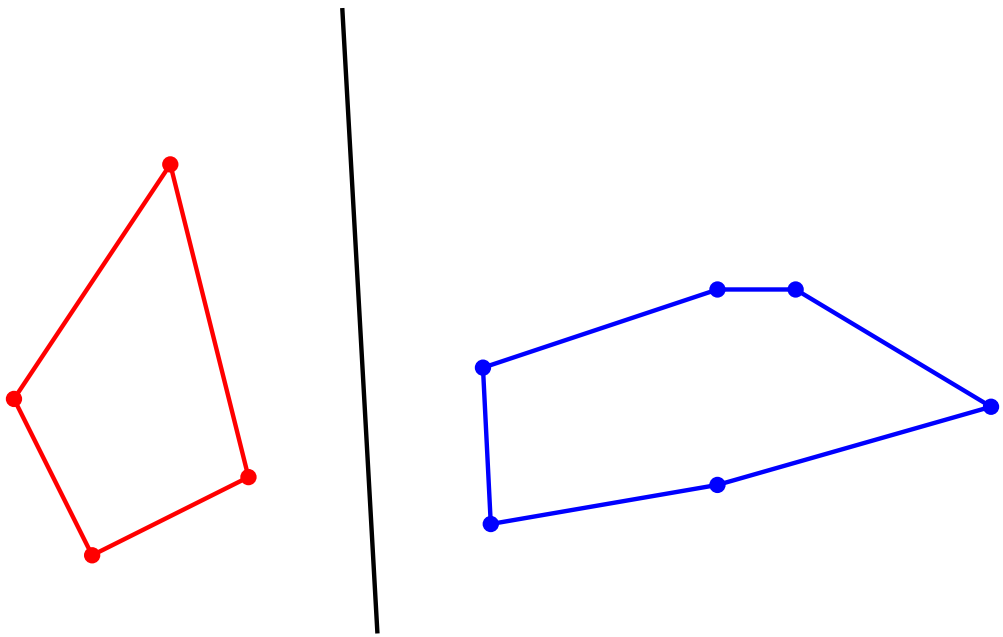
$$\mathbf{x} = \sum_{i=1}^{m} \alpha_i e_i \quad \text{and} \quad \sum_{i=1}^{m} \alpha_i = 1 \ .$$



The coefficients α_i are called **barycentric coordinates** of x.

Key idea

A hyperplane separates two classes if and only if it separates their convex hull.



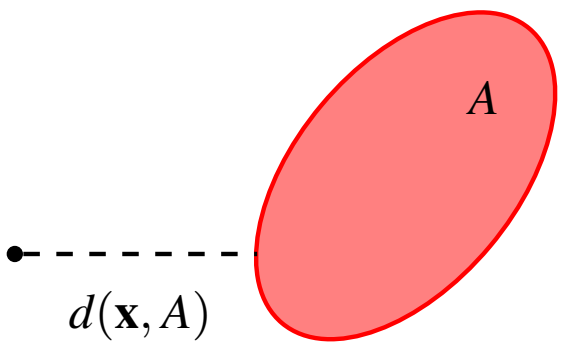
Next: We have to formalize what it means for a hyperplane to be "in the middle" between to classes.

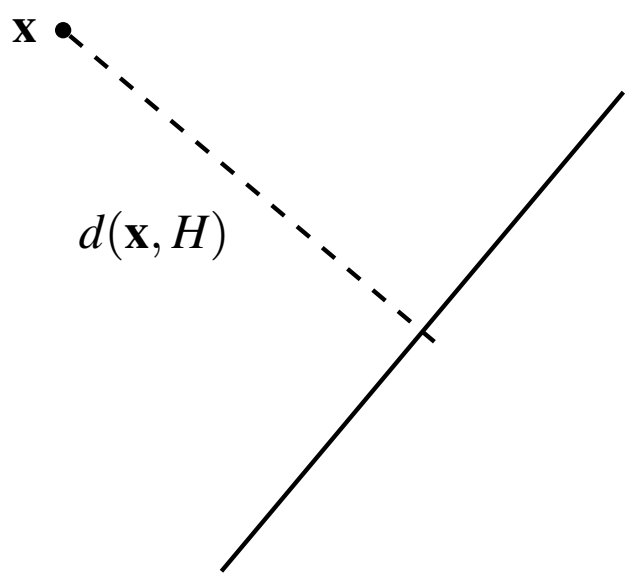
Definition

The **distance** between a point $\mathbf{x}$ and a set A the Euclidean distance between x and the closest point in A:

$$d(\mathbf{x}, A) := \min_{\mathbf{y} \in A} \|\mathbf{x} - \mathbf{y}\|$$

In particular, if $A = H$ is a hyperplane, $d(\mathbf{x}, H) := \min_{\mathbf{y} \in H} \|\mathbf{x} - \mathbf{y}\|$.





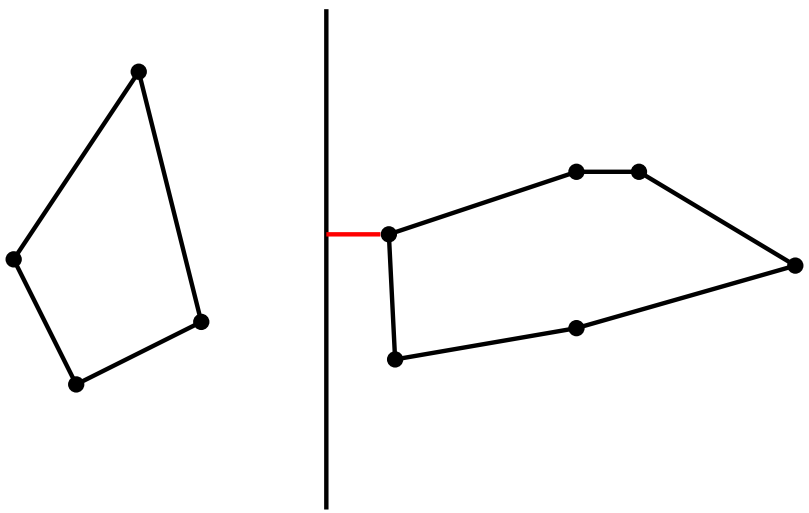
Definition

The **margin** of a classifier hyperplane H given two training classes $\mathcal{X}_1, \mathcal{X}_2$ is the shortest distance between the plane and any point in either set:

$$\text{margin} = \min_{x \in \mathcal{X}_1 \cup \mathcal{X}_2} d(x, H)$$

Equivalently: The shortest distance to either of the convex hulls.

$$\text{margin} = \min\{d(H, \text{conv}(\mathcal{X}_1)), d(H, \text{conv}(\mathcal{X}_2))\}$$



Idea in the following: H is "in the middle" when margin maximal.

Recall: Specifying affine plane

Normal vector $\mathbf{v}_H$.

$$\langle \mathbf{v}_H, \mathbf{x} \rangle - c \begin{cases} > 0 & \mathbf{x} \text{ on positive side} \\ < 0 & \mathbf{x} \text{ on negative side} \end{cases}$$

Scalar $c \in \mathbb{R}$ specifies shift (plane through origin if $c = 0$).

Plane with margin

Demand

$$\langle \mathbf{v}_H, \mathbf{x} \rangle - c > 1 \text{ or } < -1$$

$\{-1, 1\}$ on the right works for any margin: Size of margin determined by $\|\mathbf{v}_H\|$. To increase margin, scale down $\mathbf{v}_H$.

Classification

Concept of margin applies only to training, not to classification. Classification works as for any linear classifier. For a test point $\mathbf{x}$:

$$y = \text{sign}\left(\langle \mathbf{v}_H, \mathbf{x} \rangle - c\right)$$

Finding the hyperplane

For n training points $(\tilde{\mathbf{x}}_i, \tilde{y}_i)$ with labels $\tilde{y}_i \in \{-1, 1\}$, solve optimization problem:

$$\begin{aligned} \min_{\mathbf{v}_\mathrm{H}, c} \quad & \|\mathbf{v}_\mathrm{H}\| \\ \text{s.t.} \quad & \tilde{y}_i(\langle \mathbf{v}_\mathrm{H}, \tilde{\mathbf{x}}_i \rangle - c) \geq 1 \quad \text{for } i = 1, \ldots, n \end{aligned}$$

Definition

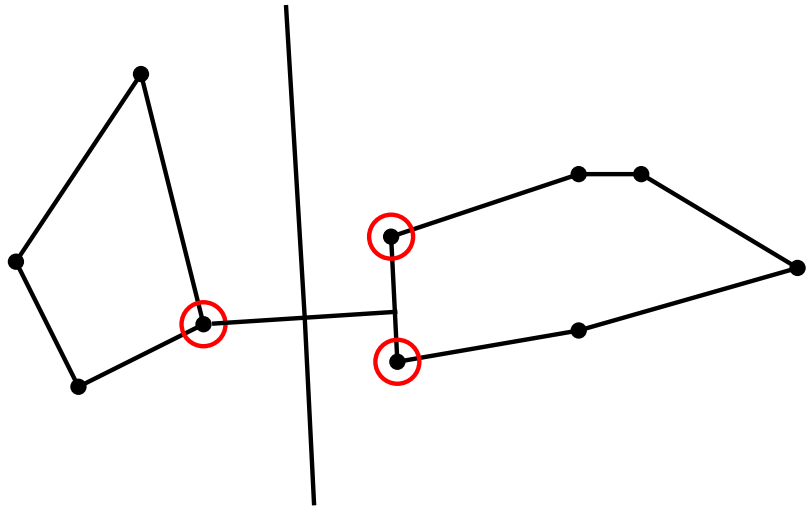
The classifier obtained by solving this optimization problem is called a **support vector machine**.

Support Vectors

Definition

Those extreme points of the convex hulls which are closest to the hyperplane are called the **support vectors**.

There are at least two support vectors, one in each class.



Implications

- The maximum-margin criterion focuses all attention to the area closest to the decision surface.
- Small changes in the support vectors can result in significant changes of the classifier.
- In practice, the approach is combined with "slack variables" to permit overlapping classes. As a side effect, slack variables soften the impact of changes in the support vectors.

Dual Optimization Problem

Solving the SVM opimization problem

$$\begin{aligned} \min_{\mathbf{v}_H, c} \quad & \|\mathbf{v}_H\| \\ \text{s.t.} \quad & \tilde{y}_i(\langle \mathbf{v}_H, \tilde{\mathbf{x}}_i \rangle - c) \geq 1 \quad \text{for } i = 1, \ldots, n \end{aligned}$$

is difficult, because the constraint is a function. It is possible to transform this problem into a problem which seems more complicated, but has simpler constraints:

$$\begin{aligned} \max_{\boldsymbol{\alpha} \in \mathbb{R}^n} \quad & W(\boldsymbol{\alpha}) := \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j \tilde{y}_i \tilde{y}_j \langle \tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_j \rangle \\ \text{s.t.} \quad & \sum_{i=1}^{n} \tilde{y}_i \alpha_i = 0 \\ & \alpha_i \geq 0 \quad \text{for } i = 1, \ldots, n \end{aligned}$$

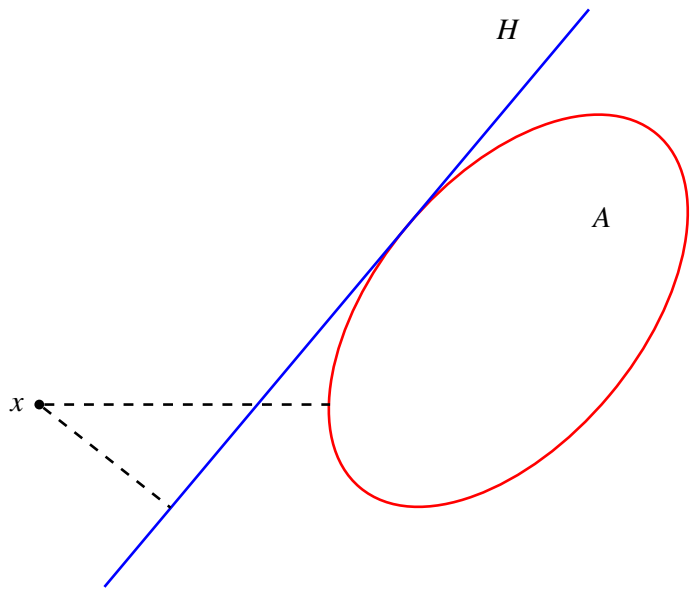
This is called the optimization problem **dual** to the minimization problem above. It is usually derived using Lagrange multipliers. We will use a more geometric argument.

Sets and Planes

Many dual relations in convex optimization can be traced back to the following fact:

The closest distance between a point x and a convex set A is the maximum over the distances between x and all hyperplanes which separate x and A.

$$d(x, A) = \sup_{H \text{ separating}} d(x, H)$$



Idea

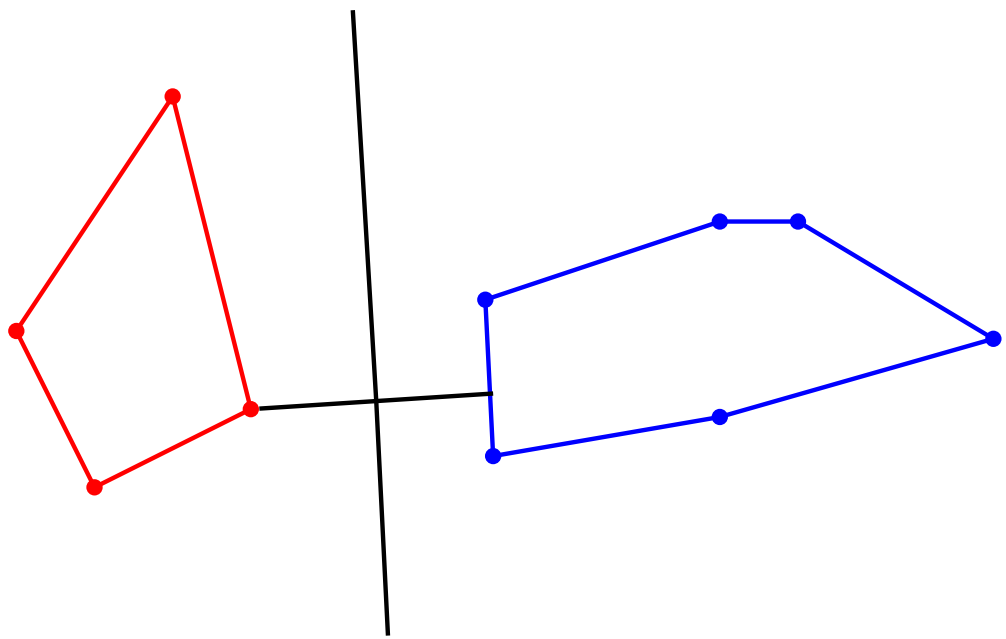
As a consequence of duality on previous slide, we can find the maximum-margin plane as follows:

1. Find shortest line connecting the convex hulls.
2. Place classifier orthogonal to line in the middle.

Convexity of sets ensures that this classifier has correct orientation.

As optimization problem

$$\min_{\substack{\mathbf{u}\in\mathrm{conv}(\mathcal{X}_1)\\ \mathbf{v}\in\mathrm{conv}(\mathcal{X}_2)}} \quad \|\mathbf{u}-\mathbf{v}\|^2$$



Dual optimization problem

$$\min_{\substack{\mathbf{u}\in\mathrm{conv}(\mathcal{X}_1)\\ \mathbf{v}\in\mathrm{conv}(\mathcal{X}_2)}} \quad \|\mathbf{u}-\mathbf{v}\|^2$$

The points $\mathbf{u}$ and $\mathbf{v}$ are in the convex hulls, and can be represented by barycentric coordinates:

$$\mathbf{u}=\sum_{i=1}^{n_1}\alpha_i\tilde{\mathbf{x}}_i \qquad \mathbf{v}=\sum_{i=n_1+1}^{n_1+n_2}\alpha_i\tilde{\mathbf{x}}_i \qquad (\text{where } n_1=|\mathcal{X}_1|, n_2=|\mathcal{X}_2|)$$

Substitute into minimization problem:

$$\begin{aligned}\min_{\alpha_1,\ldots,\alpha_n} \quad & \|\sum_{i\in C_1}\alpha_i\tilde{\mathbf{x}}_i-\sum_{i\in C_2}\alpha_i\tilde{\mathbf{x}}_i\|_2^2\\ \text{s.t.} \quad & \sum_{i\in C_1}\alpha_i=\sum_{i\in C_2}\alpha_i=1\\ & \alpha_i\geq 0\end{aligned}$$

Dual problem

$$\begin{aligned} \| \sum_{i \in C_1} \alpha_i \tilde{\mathbf{x}}_i - \sum_{i \in C_2} \alpha_i \tilde{\mathbf{x}}_i \|_2^2 &= \| \sum_{i \in C_1} \tilde{y}_i \alpha_i \tilde{\mathbf{x}}_i + \sum_{i \in C_2} \tilde{y}_i \alpha_i \tilde{\mathbf{x}}_i \|_2^2 \\ &= \left\langle \sum_{i=1}^{n} \tilde{y}_i \alpha_i \tilde{\mathbf{x}}_i \, , \, \sum_{i=1}^{n} \tilde{y}_i \alpha_i \tilde{\mathbf{x}}_i \right\rangle = \sum_{i,j} \tilde{y}_i \tilde{y}_j \alpha_i \alpha_j \left\langle \tilde{\mathbf{x}}_i | \tilde{\mathbf{x}}_j \right\rangle \end{aligned}$$

Note: Minimizing this term under the constraints is equivalent to *maximizing*

$$\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \tilde{y}_i \tilde{y}_j \alpha_i \alpha_j \left\langle \tilde{\mathbf{x}}_i | \tilde{\mathbf{x}}_j \right\rangle$$

under the same constraints, since $\sum_i \alpha_i = 2$ is constant. That is just the dual problem defined four slides back.

Computing c

Output of dual problem

$\mathbf{v}_{\mathrm{H}}^* = \sum_{i=1}^n \tilde{y}_i \alpha_i^* \tilde{\mathbf{x}}_i$. This vector describes a non-affine hyperplane.

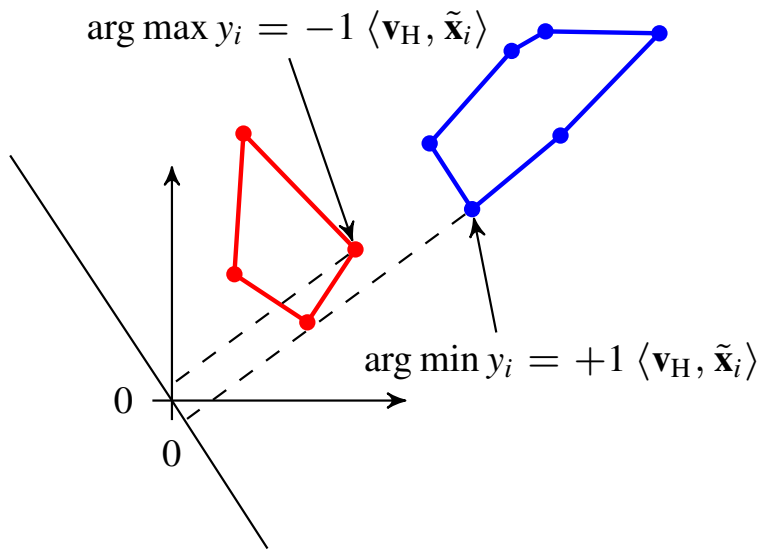
Computing the offset

The offset c is given by

$$c^* := -\frac{\max_{\tilde{y}_i=-1} \langle \mathbf{v}_{\mathrm{H}}^*, \tilde{\mathbf{x}}_i \rangle + \min_{\tilde{y}_i=+1} \langle \mathbf{v}_{\mathrm{H}}^*, \tilde{\mathbf{x}}_i \rangle}{2}$$

Explanation

- The max and min are computed with respect to the $\mathbf{v}_{\mathrm{H}}$ plane *containing the origin*.
- That means the max and min determine a support vector in each class.
- We then compute the shift as the mean of the two distances.



Output of dual optimization

- Optimal values α_i^* for the variables α_i
- If $\tilde{\mathbf{x}}_i$ support vector: $\alpha_i > 0$, if not: $\alpha_i = 0$

SVM Classifier

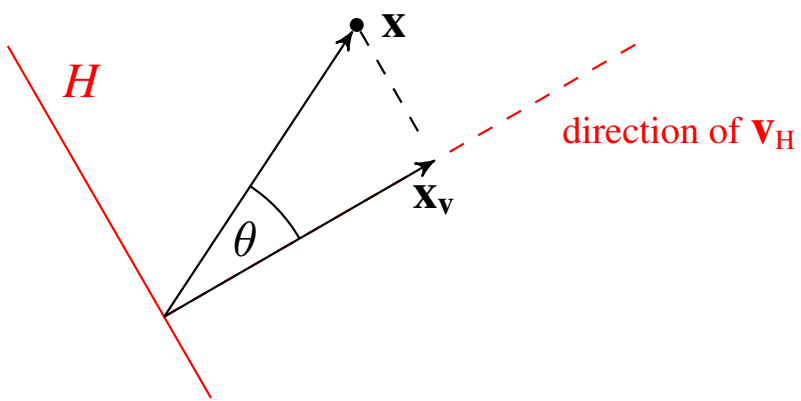
The classification function can be expressed in terms of the variables α_i:

$$f(\mathbf{x}) = \operatorname{sgn}\left(\sum_{i=1}^{n} \tilde{y}_i \alpha_i^* \langle \tilde{\mathbf{x}}_i, \mathbf{x} \rangle - c^*\right)$$

Intuitively: To classify a data point, it is sufficient to know which side of each support vector it is on.

WHY *minimize* $\|\mathbf{v}_H\|$?

We can project a vector $\mathbf{x}$ (think: data point) onto the direction of $\mathbf{v}_H$ and obtain a vector $\mathbf{x}_v$.



- If H has no offset ($c = 0$), the Euclidean distance of $\mathbf{x}$ from H is

$$d(\mathbf{x}, H) = \|\mathbf{x}_v\| = \cos\theta \cdot \|\mathbf{x}\| \ .$$

  It does not depend on the length of $\mathbf{v}_H$.
- The scalar product $\langle \mathbf{x}, \mathbf{v}_H \rangle$ *does* increase if the length of $\mathbf{v}_H$ increases.
- To compute the distance $\|\mathbf{x}_v\|$ from $\langle \mathbf{x}, \mathbf{v}_H \rangle$, we have to scale out $\|\mathbf{v}_H\|$:

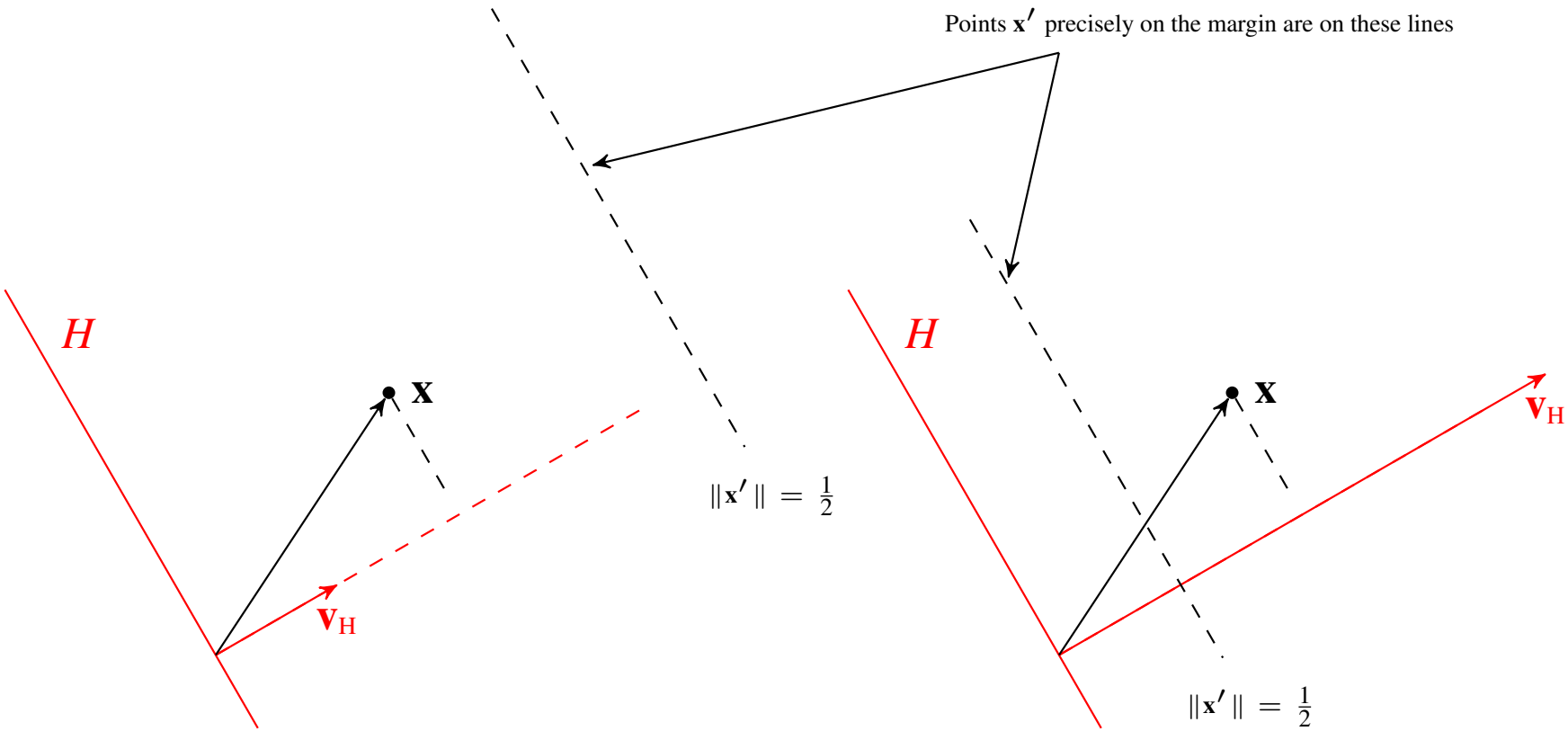
$$\|\mathbf{x}_v\| = \cos\theta \cdot \|\mathbf{x}\| = \frac{\langle \mathbf{x}, \mathbf{v}_H \rangle}{\|\mathbf{v}_H\|}$$

WHY *minimize* $\|\mathbf{v}_H\|$?

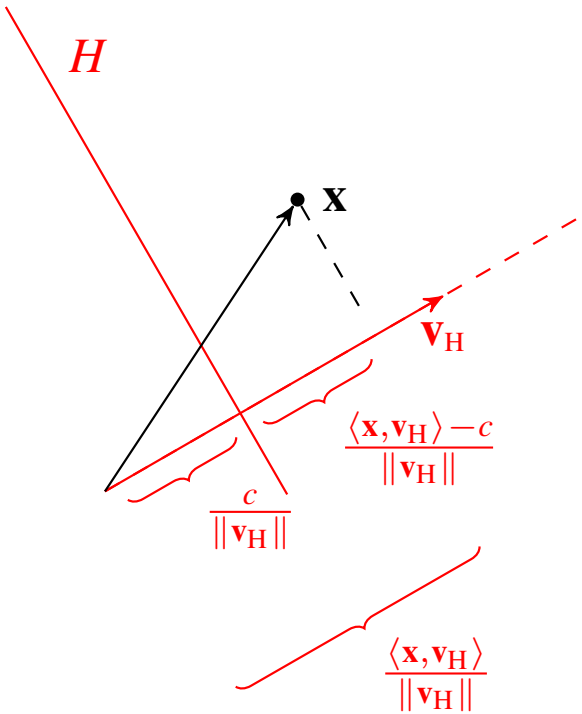
If we scale $\mathbf{v}_H$ by α, we have to scale $\mathbf{x}$ by $1/\alpha$ to keep $\langle \mathbf{v}_H, \mathbf{x} \rangle$ constant, e.g.:

$$1 = \langle \mathbf{v}_H, \mathbf{x} \rangle = \left\langle \alpha \mathbf{v}_H, \frac{1}{\alpha} \mathbf{x} \right\rangle .$$

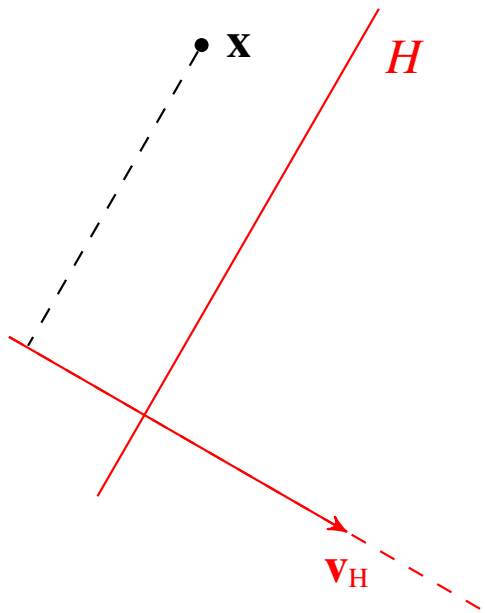
A point $\mathbf{x}'$ is precisely on the margin if $\langle \mathbf{x}', \mathbf{v}_H \rangle = 1$.
Look at what happens if we scale $\mathbf{v}_H$:



Distance With Offset



For an affine plane, we have to substract the offset.



The optimization algorithm can also rotate the vector $\mathbf{v}_H$, which rotates the plane.

Soft-margin classifiers are maximum-margin classifiers which permit some points to lie on the wrong side of the margin, or even of the hyperplane.

Motivation 1: Nonseparable data

SVMs are linear classifiers; without further modifications, they cannot be trained on a non-separable training data set.

Motivation 2: Robustness

- Recall: Location of SVM classifier depends on position of (possibly few) support vectors.
- Suppose we have two training samples (from the same joint distribution on (X, Y)) and train an SVM on each.
- If locations of support vectors vary significantly between samples, SVM estimate of $\mathbf{v}_{\mathrm{H}}$ is "brittle" (depends too much on small variations in training data). $\longrightarrow$ Bad generalization properties.
- Methods which are not susceptible to small variations in the data are often referred to as **robust**.

Slack Variables

Idea

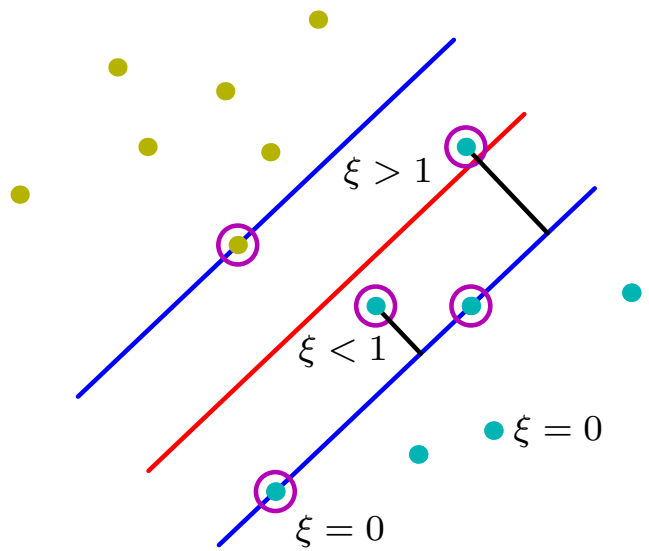
Permit training data to cross the margin, but impose cost which increases the further beyond the margin we are.

Formalization

We replace the training rule $\tilde{y}_i(\langle \mathbf{v}_H, \tilde{\mathbf{x}}_i \rangle - c) \geq 1$ by

$$\tilde{y}_i(\langle \mathbf{v}_H, \tilde{\mathbf{x}}_i \rangle - c) \geq 1 - \xi_i$$

with $\xi_i \geq 0$. The variables ξ_i are called **slack variables**.



Soft-margin optimization problem

$$\begin{aligned} \min_{\mathbf{v}_H, c} \quad & \|\mathbf{v}_H\|^2 + \gamma \sum_{i=1}^{n} \xi^2 \\ \text{s.t.} \quad & \tilde{y}_i(\langle \mathbf{v}_H, \tilde{\mathbf{x}}_i \rangle - c) \geq 1 - \xi_i \quad \text{for } i = 1, \ldots, n \\ & \xi_i \geq 0, \quad \text{for } i = 1, \ldots, n \end{aligned}$$

The training algorithm now has a **parameter** $\gamma > 0$ for which we have to choose a "good" value. γ is usually set by a method called *cross validation* (discussed later). Its value is fixed before we start the optimization.

Role of γ

- Specifies the "cost" of allowing a point on the wrong side.
- If γ is very small, many points may end up beyond the margin boundary.
- For $\gamma \to \infty$, we recover the original SVM.

Soft-margin dual problem

The slack variables vanish in the dual problem.

$$\begin{aligned} \max_{\boldsymbol{\alpha}\in\mathbb{R}^n} \quad & W(\boldsymbol{\alpha}) := \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{n} \alpha_i\alpha_j\tilde{y}_i\tilde{y}_j\left(\langle\tilde{\mathbf{x}}_i,\tilde{\mathbf{x}}_j\rangle + \frac{1}{\gamma}\mathbb{I}\{i=j\}\right) \\ \text{s.t.} \quad & \sum_{i=1}^{n}\tilde{y}_i\alpha_i = 0 \\ & \alpha_i \geq 0 \quad \text{for } i = 1,\ldots,n \end{aligned}$$

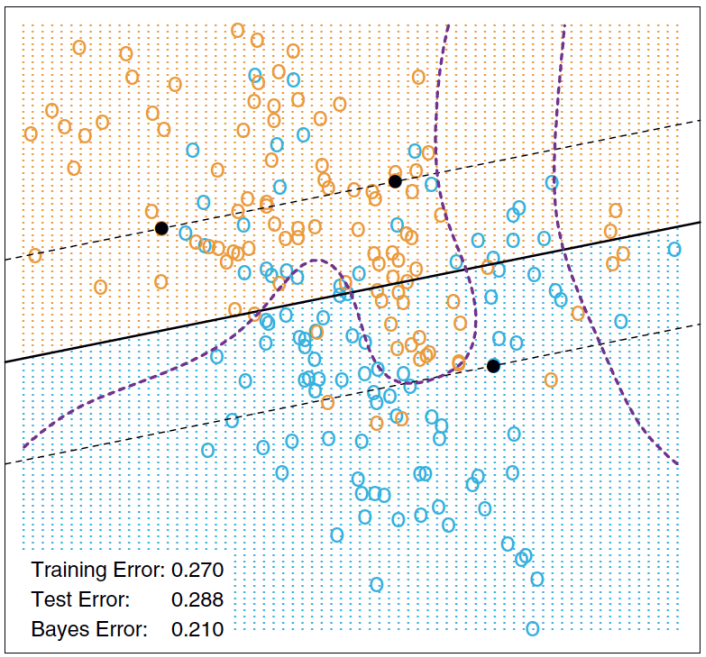
Soft-margin classifier

The classifier looks exactly as for the original SVM:

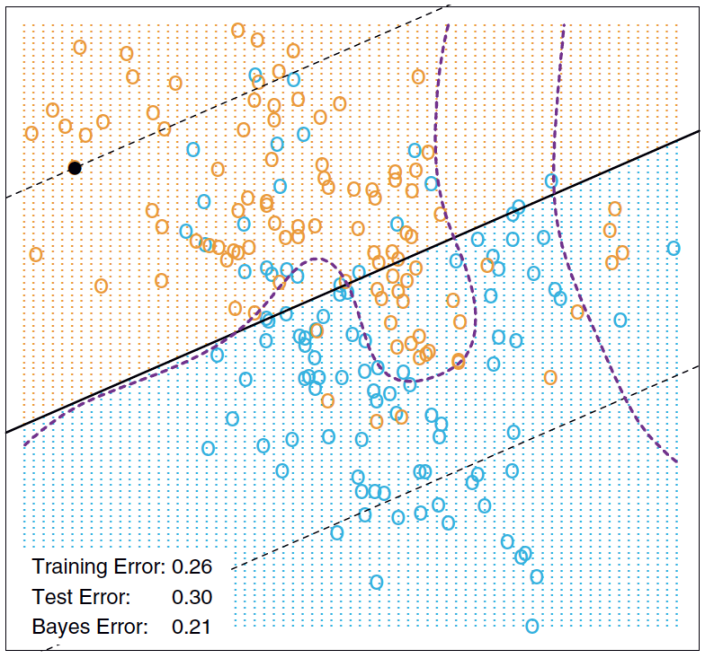
$$f(\mathbf{x}) = \operatorname{sgn}\left(\sum_{i=1}^{n}\tilde{y}_i\alpha_i^*\,\langle\tilde{\mathbf{x}}_i,\mathbf{x}\rangle - c\right)$$

Note: Each point on wrong side of the margin is an additional support vector ($\alpha_i^* \neq 0$), so the ratio of support vectors can be substantial when classes overlap.

Influence of Margin Parameter



$\gamma = 100000$



$\gamma = 0.01$

Changing γ significantly changes the classifier (note how the slope changes in the figures). We need a method to select an appropriate value of γ, in other words: to learn γ from data.

Tools: Optimization Methods

Terminology

An **optimization problem** for a given function $f : \mathbb{R}^d \to \mathbb{R}$ is a problem of the form

$$\min_{\mathbf{x}} f(\mathbf{x})$$

which we read as "find $\mathbf{x}_0 = \arg\min_{\mathbf{x}} f(\mathbf{x})$".
A **constrained optimization problem** adds additional requirements on $\mathbf{x}$,

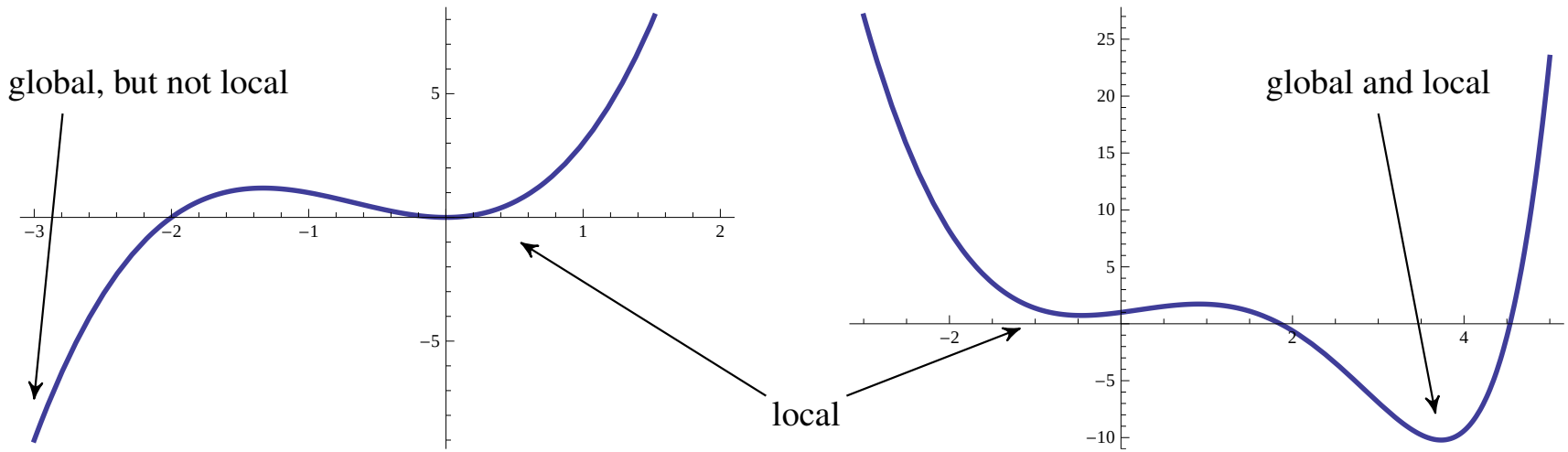
$$\begin{aligned} & \min_{\mathbf{x}} f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{x} \in G \ , \end{aligned}$$

where $G \subset \mathbb{R}^d$ is called the **feasible set**. The set G is often defined by equations, e.g.

$$\begin{aligned} & \min_{\mathbf{x}} f(\mathbf{x}) \\ \text{subject to} \quad & g(\mathbf{x}) \geq 0 \end{aligned}$$

The equation g is called a **constraint**.

Types of Minima



Local and global minima

A minimum of f at x is called:

- **Global** if f assumes no smaller value on its domain.
- **Local** if there is some open neighborhood U of x such that $f(x)$ is a global minimum of f restricted to U.

Analytic criteria for local minima

Recall that $\mathbf{x}$ is a local minimum of f if

$$f'(\mathbf{x}) = 0 \qquad \text{and} \qquad f''(\mathbf{x}) > 0\ .$$

In $\mathbb{R}^d$,

$$\nabla f(\mathbf{x}) = 0 \qquad \text{and} \qquad H_f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_i \partial x_j}(\mathbf{x})\right)_{i,j=1,\dots,n} \text{ positive definite.}$$

The $d \times d$-matrix $H_f(\mathbf{x})$ is called the **Hessian matrix** of f at $\mathbf{x}$.

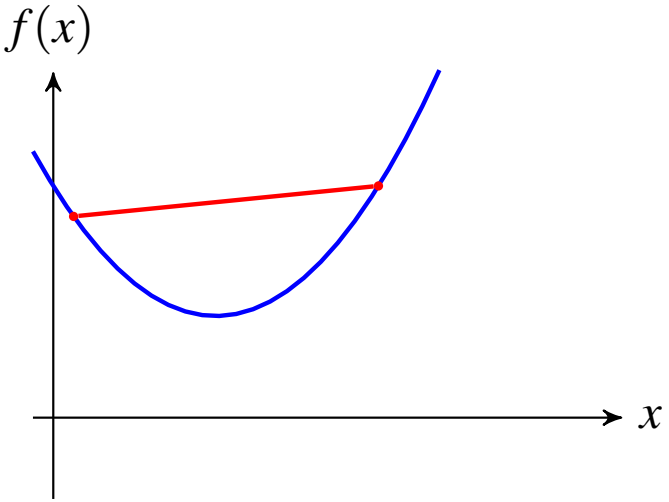
Numerical methods

All numerical minimization methods perform roughly the same steps:

- Start with some point x_0.
- Our goal is to find a sequence $x_0, \dots, x_m$ such that $f(x_m)$ is a minimum.
- At a given point x_n, compute properties of f (such as $f'(x_n)$ and $f''(x_n)$).
- Based on these values, choose the next point x_{n+1}.

The information $f'(x_n), f''(x_n)$ etc is always *local at* x_n, and we can only decide whether a point is a local minimum, not whether it is global.

Convex Functions



Definition

A function f is **convex** if every line segment between function values lies above the graph of f.

Analytic criterion

A twice differentiable function is convex if $f''(x) \geq 0$ (or $H_f(\mathbf{x})$ positive semidefinite) for all $\mathbf{x}$.

Implications for optimization

If f is convex, then:

- $f'(x) = 0$ is a sufficient criterion for a minimum.
- Local minima are global.
- If f is **strictly convex** ($f'' > 0$ or H_f positive definite), there is only one minimum (which is both gobal and local).

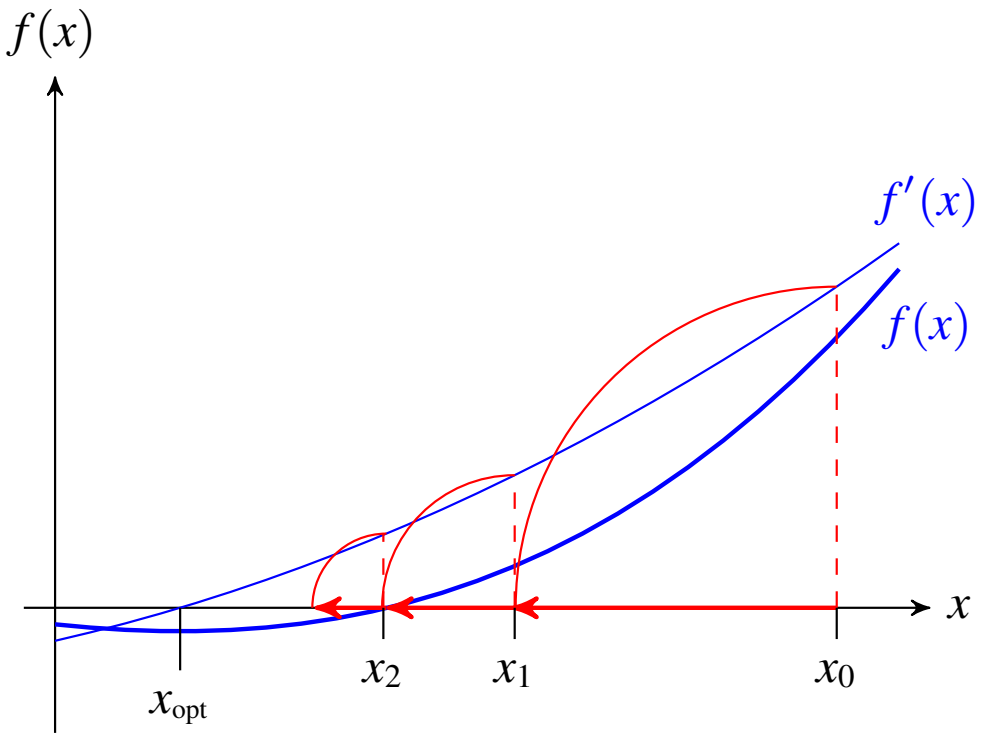
Algorithm

Gradient descent searches for a minimum of f.

1. Start with some point $x \in \mathbb{R}$ and fix a precision $\varepsilon > 0$.
2. Repeat for $n = 1, 2, \ldots$

$$x_{n+1} := x_n - f'(x_n)$$

3. Terminate when $|f'(x_n)| < \varepsilon$.



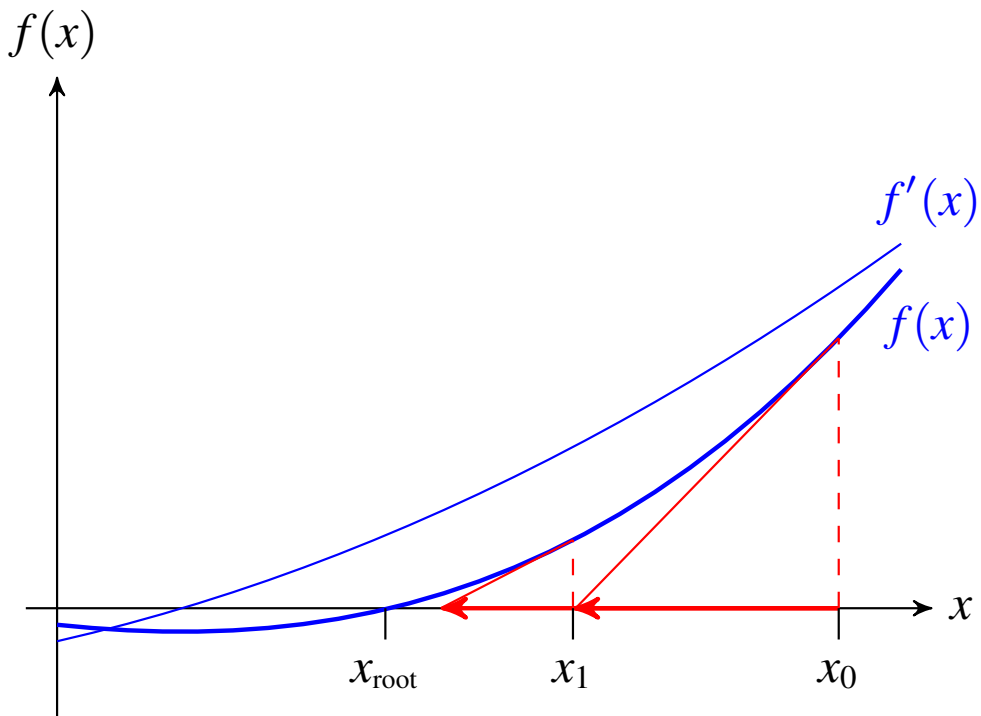
Algorithm

Newton's method searches for a **root** of f, i.e. it solves the equation $f(\mathbf{x}) = 0$.

1. Start with some point $x \in \mathbb{R}$ and fix a precision $\varepsilon > 0$.
2. Repeat for $n = 1, 2, \ldots$

$$x_{n+1} := x_n - f(x_n)/f'(x_n)$$

3. Terminate when $|f(x_n)| < \varepsilon$.



Function evaluation

Most numerical evaluations of functions ($\sqrt{a}$, $\sin(a)$, $\exp(a)$, etc) are implemented using Newton's method. To evaluate g at a, we have to transform $x = g(a)$ into an equivalent equation of the form

$$f(x, a) = 0 \ .$$

We then fix a and solve for x using Newton's method for roots.

Example: Square root

To eveluate $g(a) = \sqrt{a}$, we can solve

$$f(x, a) = x^2 - a = 0 \ .$$

This is essentially how `sqrt()` is implemented in the standard C library.

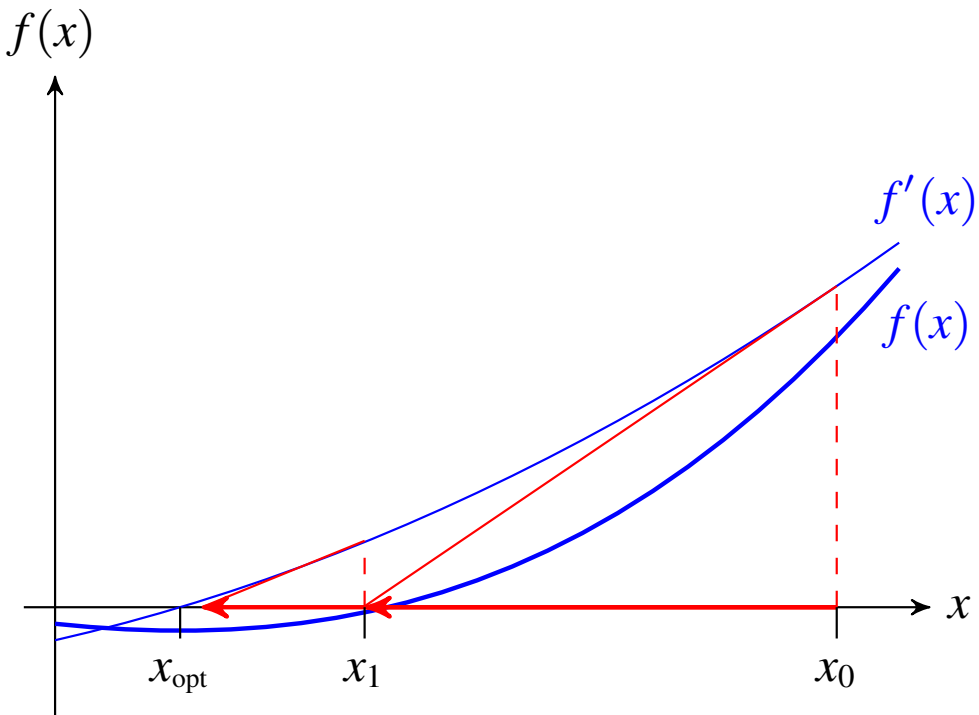
Algorithm

We can use Newton's method for minimization by applying it to solve $f'(\mathbf{x}) = 0$.

1. Start with some point $x \in \mathbb{R}$ and fix a precision $\varepsilon > 0$.
2. Repeat for $n = 1, 2, \ldots$

$$x_{n+1} := x_n - f'(x_n)/f''(x_n)$$

3. Terminate when $|f'(x_n)| < \varepsilon$.



In $\mathbb{R}^d$ we have to replace the derivatives by their vector space analogues.

Gradient descent

$$\mathbf{x}_{n+1} := \mathbf{x}_n - \nabla f(\mathbf{x}_n)$$

Newton's method for minima

$$\mathbf{x}_{n+1} := \mathbf{x}_n - H_f^{-1}(\mathbf{x}_n) \cdot \nabla f(\mathbf{x}_n)$$

The inverse of $H_f(\mathbf{x})$ exists only if the matrix is positive definite (not if it is only semidefinite), i.e. f has to be strictly convex.

The Hessian measures the curvature of f.

Effect of the Hessian

Multiplication by H_f^{-1} in general changes the direction of $\nabla f(\mathbf{x}_n)$. The correction takes into account how $\nabla f(\mathbf{x})$ changes away from $\mathbf{x}_n$, as estimated using the Hessian at $\mathbf{x}_n$.

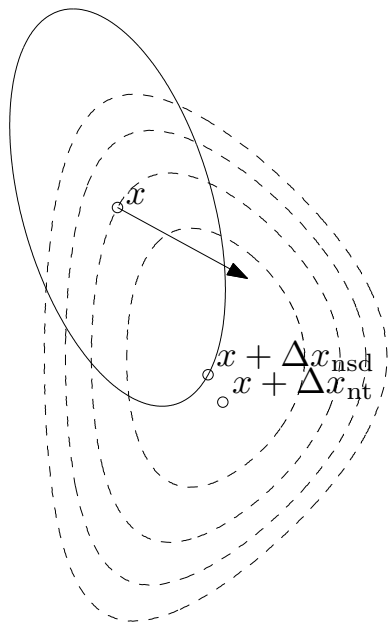


Figure: Arrow is ∇f, $x + \Delta x_{\text{nt}}$ is Newton step.

Newton: Properties

Convergence

- The algorithm always converges if $f'' > 0$ (or H_f positive definite).
- The speed of convergence separates into two phases:
  - In a (possibly small) region around the minimum, f can always be approximated by a quadratic function.
  - Once the algorithm reaches that region, the error decreases at quadratic rate. Roughly speaking, the number of correct digits in the solution doubles in each step.
  - Before it reaches that region, the convergence rate is linear.

High dimensions

- The required number of steps hardly depends on the dimension of $\mathbb{R}^d$. Even in $\mathbb{R}^{10000}$, you can usually expect the algorithm to reach high precision in half a dozen steps.
- Caveat: The individual steps can become very expensive, since we have to invert H_f in each step, which is of size $d \times d$.

Next: Constrained Optimization

So far

- If f is differentiable, we can search for local minima using gradient descent.
- If f is sufficiently nice (convex and twice differentiable), we know how to speed up the search process using Newton's method.

Constrained problems

- The numerical minimizers use the criterion $\nabla f(x) = 0$ for the minimum.
- In a constrained problem, the minimum is *not* identified by this criterion.

Next steps

We will figure out how the constrained minimum can be identified. We have to distinguish two cases:

- Problems involving only equalities as constraints (easy).
- Problems also involving inequalities (a bit more complex).

Objective

$$\min f(\mathbf{x})$$
$$\text{subject to } g(\mathbf{x}) = 0$$

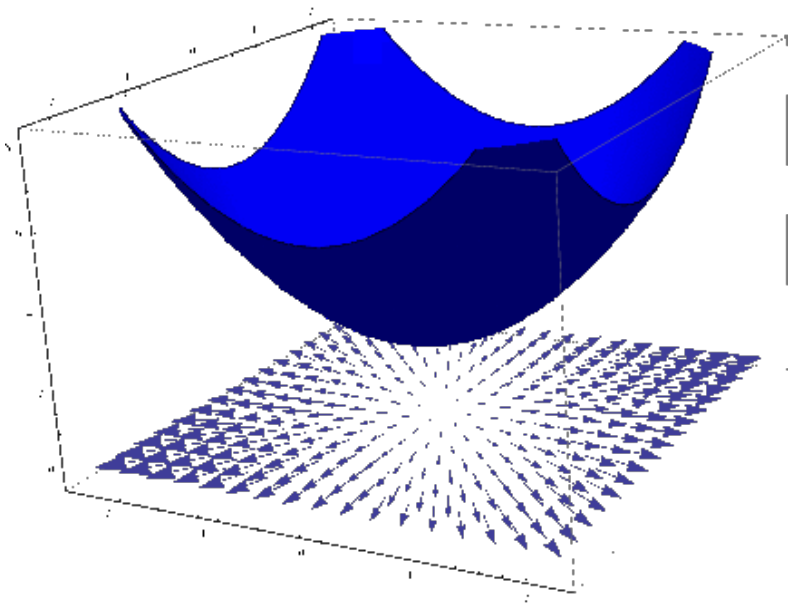
Idea

- The feasible set is the set of points $\mathbf{x}$ which satisfy $g(\mathbf{x}) = 0$,

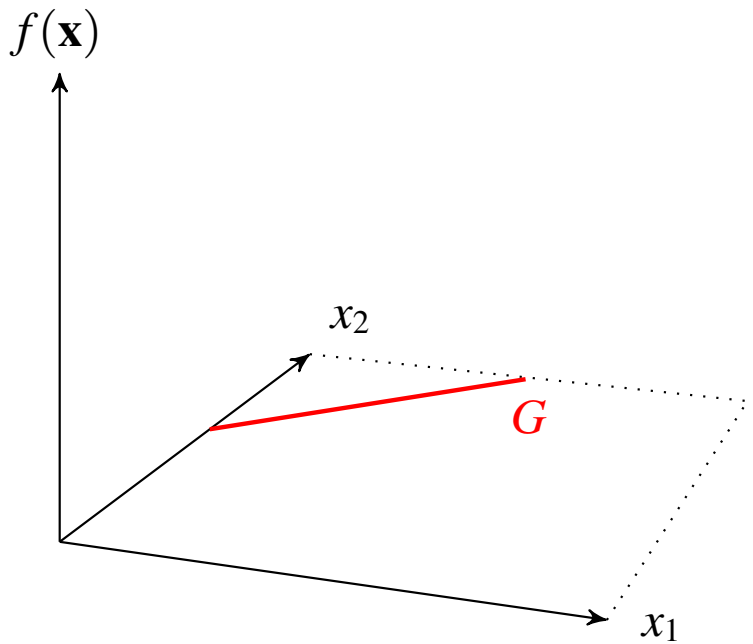
  $$G := \{\mathbf{x} \mid g(\mathbf{x}) = 0\} .$$

  If g is reasonably smooth, G is a smooth surface in $\mathbb{R}^d$.
- We restrict the function f to this surface and call the restricted function f_g.
- The constrained optimization problem says that we are looking for the minimum of f_g.

Lagrange Optimization

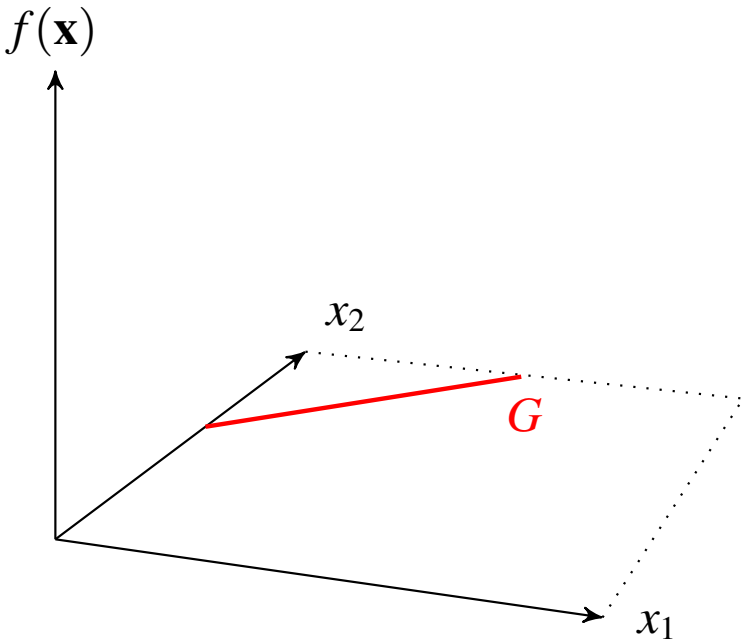


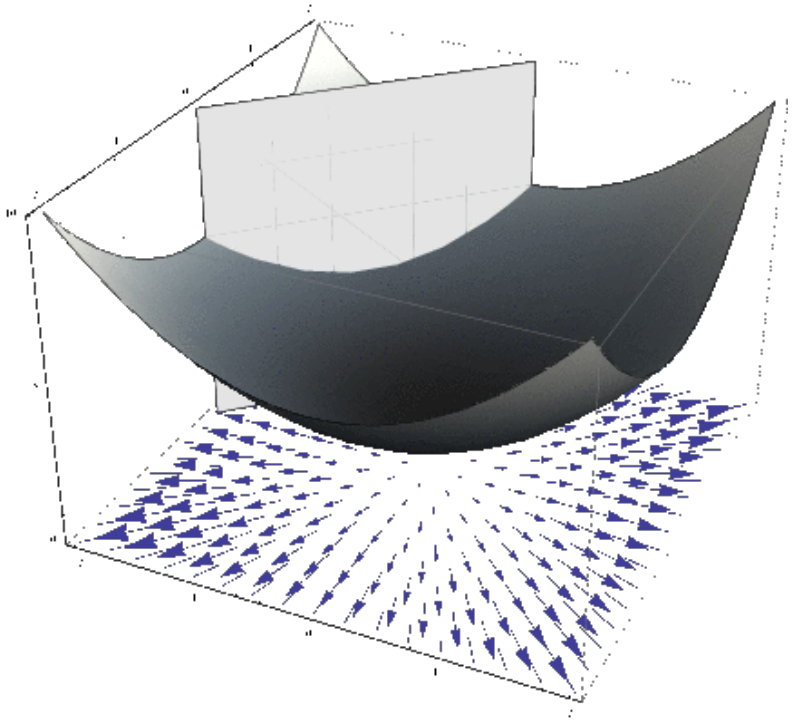
$$f(\mathbf{x}) = x_1^2 + x_2^2$$



Constraint g: The intersection of the plane with the x_1-x_2-plane is the set G of all points with $g(\mathbf{x}) = 0$.

Lagrange Optimization





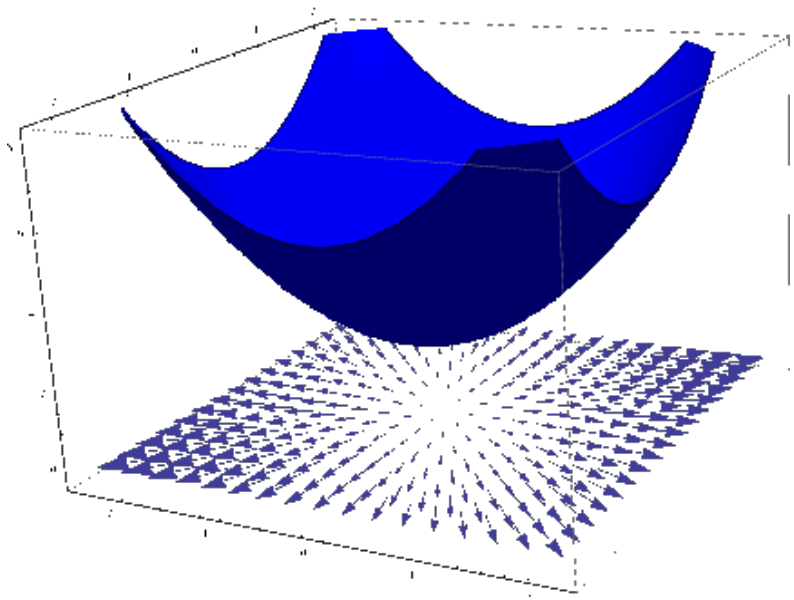
- We can make the function f_g given by the constraint $g(\mathbf{x}) = 0$ visible by placing a plane vertically through G. The graph of f_g is the intersection of the graph of f with the plane.
- Here, f_g has parabolic shape.
- The gradient of f at the miniumum of f_g is *not* 0.

Fact

Gradients are orthogonal to contour lines.

Intuition

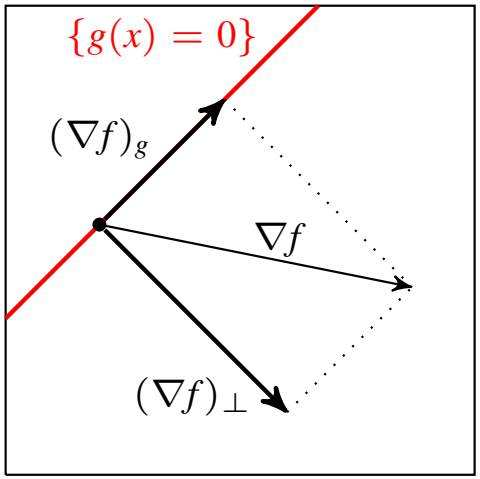
- The gradient points in the direction in which f grows most rapidly.
- Contour lines are sets along which f does not change.



The Crucial Bit

Idea

- Decompose ∇f into a component $(\nabla f)_g$ in the set $\{x \mid g(\mathbf{x}) = 0\}$ and a remainder $(\nabla f)_\perp$.
- The two components are orthogonal.
- If f_g is minimal within $\{x \mid g(\mathbf{x}) = 0\}$, the component within the set vanishes.
- The remainder need not vanish.



Consequence

We need a criterion for $(\nabla f)_g = 0$.

Solution

- If $(\nabla f)_g = 0$, then ∇f is orthogonal to the set $g(\mathbf{x}) = 0$.
- Since gradients are orthogonal to contours, and the set is a contour of g, ∇g is also orthogonal to the set.
- Hence: At a minimum of f_g, the two gradients point in the same direction: $\nabla f + \lambda \nabla g = 0$ for some scalar $\lambda \neq 0$.

Solution

The constrained optimization problem

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & g(\mathbf{x}) = 0 \end{aligned}$$

is solved by solving the equation system

$$\begin{aligned} \nabla f(\mathbf{x}) + \lambda \nabla g(\mathbf{x}) &= 0 \\ g(\mathbf{x}) &= 0 \end{aligned}$$

The vectors ∇f and ∇g are D-dimensional, so the system contains $D + 1$ equations for the $D + 1$ variables $x_1, \ldots, x_D, \lambda$.

Objective

For a function f and a convex function g, solve

$$\begin{aligned} &\min f(\mathbf{x}) \\ &\text{subject to } g(\mathbf{x}) \leq 0 \end{aligned}$$

i.e. we replace $g(\mathbf{x}) = 0$ as previously by $g(\mathbf{x}) \leq 0$. This problem is called an optimization problem with **inequality constraint**.

Feasible set

We again write G for the set of all points which satisfy the constraint,

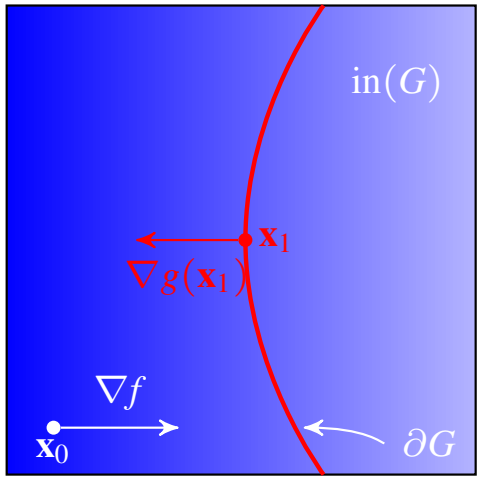
$$G := \{\mathbf{x} \,|\, g(\mathbf{x}) \leq 0\} \ .$$

G is often called the **feasible set** (the same name is used for equality constraints).

Two Cases

Case distinction

1. The location $\mathbf{x}$ of the minimum can be in the *interior* of G
2. $\mathbf{x}$ may be on the *boundary* of G.



lighter shade of blue = larger value of f

Decomposition of G

$$G = \text{in}(G) \cup \partial G = \text{ interior } \cup \text{ boundary}$$

Note: The interior is given by $g(\mathbf{x}) < 0$, the boundary by $g(\mathbf{x}) = 0$.

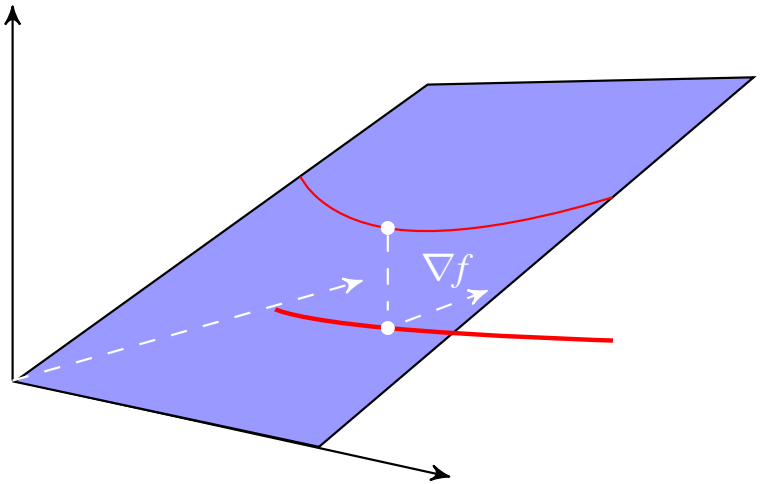
Criteria for minimum

1. **In interior:** $f_g = f$ and hence $\nabla f_g = \nabla f$. We have to solve a standard optimization problem with criterion $\nabla f = 0$.
2. **On boundary:** Here, $\nabla f_g \neq \nabla f$. Since $g(\mathbf{x}) = 0$, the geometry of the problem is the same as we have discussed for equality constraints, with criterion $\nabla f = \lambda \nabla g$.
   However: In this case, the sign of λ matters.

On the Boundary

Observation

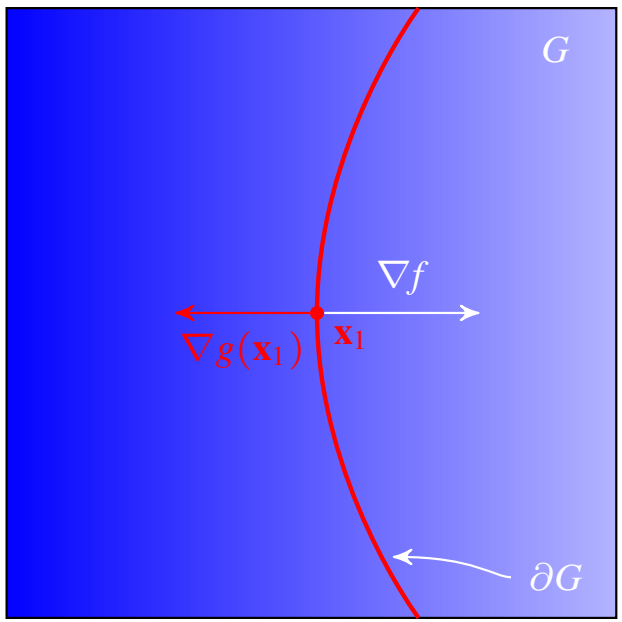
- An extremum on the boundary is a minimum onlyl if ∇f points *into* G.
- Otherwise, it is a maximum instead.



Criterion for minimum on boundary

Since ∇g points *away* from G (since g increases away from G), ∇f and ∇g have to point in opposite directions:

$$\nabla f = \lambda \nabla g \qquad \text{with } \lambda < 0$$



Convention

To make the sign of λ explicit, we constrain λ to positive values and instead write:

$$\nabla f = -\lambda \nabla g$$
$$\text{s.t. } \lambda > 0$$

Combined problem

$$\begin{aligned} & \nabla f = -\lambda \nabla g \\ \text{s.t.} \quad & g(\mathbf{x}) \leq 0 \\ & \lambda = 0 \text{ if } \mathbf{x} \in \text{in}(G) \\ & \lambda > 0 \text{ if } \mathbf{x} \in \partial G \end{aligned}$$

Can we get rid of the "if $\mathbf{x} \in \cdot$" distinction?

Yes: Note that $g(\mathbf{x}) < 0$ if $\mathbf{x}$ in interior and $g(\mathbf{x}) = 0$ on boundary. Hence, we always have either $\lambda = 0$ or $g(\mathbf{x}) = 0$ (and never both).

That means we can substitute

$$\begin{aligned} & \lambda = 0 \text{ if } \mathbf{x} \in \text{in}(G) \\ & \lambda > 0 \text{ if } \mathbf{x} \in \partial G \end{aligned}$$

by

$$\lambda \cdot g(\mathbf{x}) = 0 \qquad \text{and} \qquad \lambda \geq 0 \, .$$

Combined solution

The optimization problem with inequality constraints

$$\begin{aligned} &\min f(\mathbf{x}) \\ &\text{subject to } g(\mathbf{x}) \leq 0 \end{aligned}$$

can be solved by solving

$$\text{s.t.} \quad \left.\begin{aligned} \nabla f(\mathbf{x}) &= -\lambda \nabla g(\mathbf{x}) \\ \lambda g(\mathbf{x}) &= 0 \end{aligned}\right\} \longleftarrow \text{system of } d+1 \text{ equations for } d+1 \text{ variables } x_1, \ldots, x_D, \lambda$$
$$\begin{aligned} g(\mathbf{x}) &\leq 0 \\ \lambda &\geq 0 \end{aligned}$$

These conditions are known as the **Karush-Kuhn-Tucker** (or **KKT**) conditions.

Haven't we made the problem more difficult?

- To simplify the minimization of f for $g(\mathbf{x}) \leq 0$, we have made f more complicated and added a variable and two constraints. Well done.
- However: In the original problem, we *do not know how to minimize* f, since the usual criterion $\nabla f = 0$ does not work.
- By adding λ and additional constraints, we have reduced the problem to solving a system of equations.

Summary: Conditions

Condition	Ensures that...	Purpose
$\nabla f(\mathbf{x}) = -\lambda \nabla g(\mathbf{x})$	If $\lambda = 0$: ∇f is 0	Opt. criterion inside G
	If $\lambda > 0$: ∇f is anti-parallel to ∇g	Opt. criterion on boundary
$\lambda g(\mathbf{x}) = 0$	$\lambda = 0$ in interior of G	Distinguish cases in(G) and ∂G
$\lambda \geq 0$	∇f cannot flip to orientation of ∇g	Optimum on ∂G is minimum

Why Should g be Convex?

More precisely

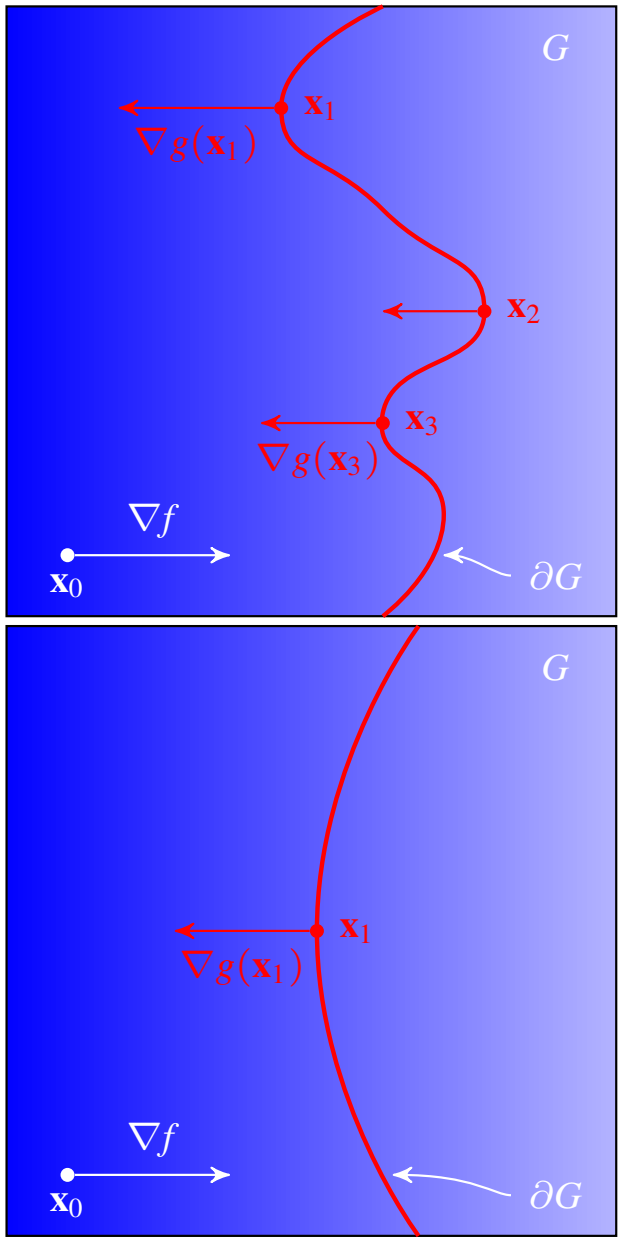
If g is a convex function, then $G = \{\mathbf{x} \mid g(\mathbf{x}) \leq 0\}$ is a convex set. Why do we require convexity of G?

Problem

If G is not convex, the KKT conditions do not guarantee that $\mathbf{x}$ is a minimum. (The conditions still hold, i.e. if G is not convex, they are necessary conditions, but not sufficient.)

Example (Figure)

- f is a linear function (lighter color = larger value)
- ∇f is identical everywhere
- If G is not convex, there can be several points (here: $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$) which satisfy the KKT conditions. Only $\mathbf{x}_1$ minimizes f on G.
- If G is convex, such problems cannot occur.



Numerical methods for constrained problems

Once we have transformed our problem using Lagrange multipliers, we still have to solve a problem of the form

$$\begin{aligned} & \nabla f(\mathbf{x}) = -\lambda \nabla g(\mathbf{x}) \\ \text{s.t.} \quad & \lambda g(\mathbf{x}) = 0 \quad \text{and} \quad g(\mathbf{x}) \leq 0 \quad \text{and} \quad \lambda \geq 0 \end{aligned}$$

numerically.

Idea

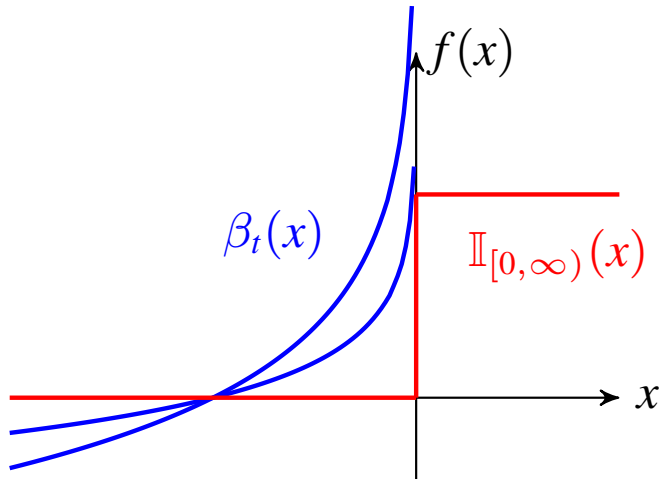
A constraint in the problem

$$\min f(x) \qquad \text{s.t.} \quad g(x) < 0$$

can be expressed as an indicator function:

$$\min f(x) + const. \cdot \mathbb{I}_{[0,\infty)}(g(x))$$

The constant must be chosen large enough to enforce the constraint.



Problem: The indicator function is piece-wise constant and not differentiable at 0. Newton or gradient descent are not applicable.

Barrier function

A **barrier function** approximates $\mathbb{I}_{[0,\infty)}$ by a smooth function, e.g.

$$\beta_t(x) := -\frac{1}{t}\log(-x) \ .$$

Interior point methods

We can (approximately) solve

$$\min f(x) \text{ s.t.} \quad g_i(x) < 0 \quad \text{for } i = 1, \ldots, m$$

by solving

$$\min f(x) + \sum_{i=1}^{m} \beta_t(x) \ .$$

We do not have to adjust a multiplicative constant since $\beta_t(x) \to \infty$ as $x \nearrow 0$.

Constrained problems: General solution strategy

1. Convert constraints into solvable problem using Lagrange multipliers.
2. Convert constraints of transformed problem into barrier functions.
3. Apply numerical optimization (usually Newton's method).

Original optimization problem

$$\min_{\mathbf{v}_H, c} \|\mathbf{v}_H\|_2 \qquad \text{s.t.} \quad y_i(\langle \mathbf{v}_H, \tilde{\mathbf{x}}_i \rangle - c) \geq 1 \quad \text{for } i = 1, \ldots, n$$

Problem with inequality constraints $g(\mathbf{v}_H) < 0$ for $g(\mathbf{v}_H) := 1 - y_i(\langle \mathbf{v}_H, \tilde{\mathbf{x}}_i \rangle - c)$.

Transformed problem

If we transform the problem using Lagrange multipliers $\alpha_1, \ldots, \alpha_n$, we obtain:

$$\max_{\boldsymbol{\alpha} \in \mathbb{R}^n} \qquad W(\boldsymbol{\alpha}) := \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j \tilde{y}_i \tilde{y}_j \langle \tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_j \rangle$$

$$\text{s.t.} \qquad \sum_{i=1}^{n} y_i \alpha_i = 0$$

$$\alpha_i \geq 0 \quad \text{for } i = 1, \ldots, n$$

This is precisely the "dual problem" we obtained before using geometric arguments.
We can find the max-margin hyperplane using an interior point method.

Minimization problems

Most methods that we encounter in this class can be phrased as minimization problem. For example:

Problem	Objective function
ML estimation	negative log-likelihood
Classification	empirical risk
Regression	fitting or prediction error
Unsupervised learning	suitable cost function (later)

More generally

The lion's share of algorithms in statistics or machine learning fall into either of two classes:

1. Optimization methods.
2. Simulation methods (e.g. Markov chain Monte Carlo algorithms).

Multiple Classes

More than two classes

For some classifiers, multiple classes are natural. We have already seen one:

- Simple classifier fitting one Gaussian per class.

We will discuss more examples soon:

- Trees.
- Ensembles: Number of classes is determined by weak learners.

Exception: All classifiers based on hyperplanes.

Linear Classifiers

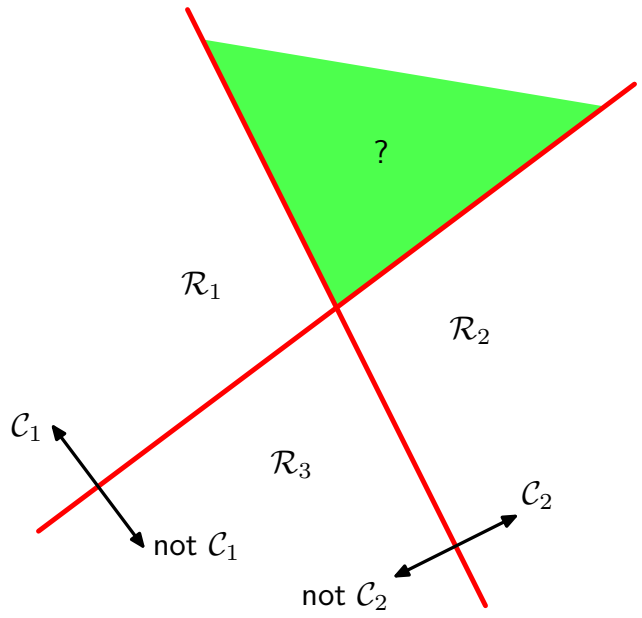
Approaches:

- One-versus-one classification.
- One-versus-all (more precisely: one-versus-the-rest) classification.
- Multiclass discriminants.

The SVM is particularly problematic.

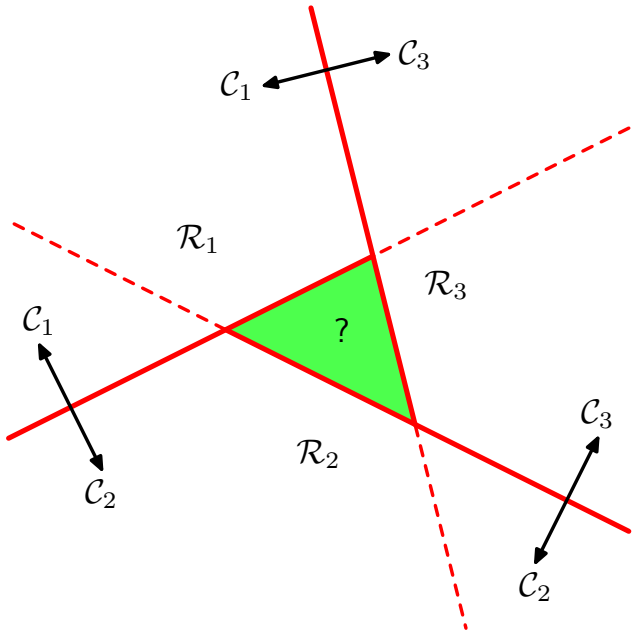
One-Versus-X Classification

One-versus-all



- One linear classifier per class.
- Classifies "in class k" versus "not in class k".
- Positive class = $\mathcal{C}_k$.
  Negative class = $\cup_{j \neq k} \mathcal{C}_j$.
- Problem: Ambiguous regions (green in figure).

One-versus-one



- One linear classifier for each pair of classes (i.e. $\frac{K(K-1)}{2}$ in total).
- Classify by majority vote.
- Problem again: Ambiguous regions.

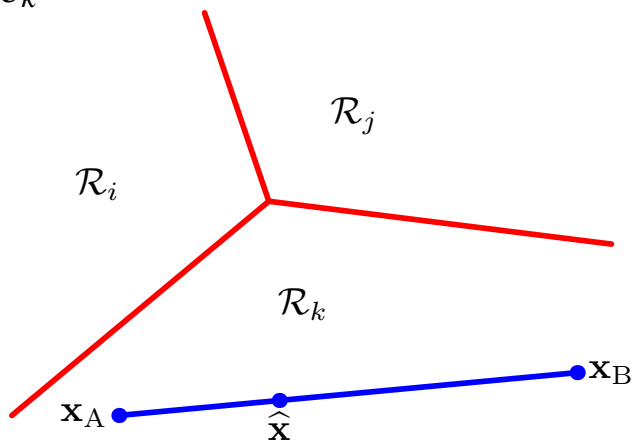
Linear classifier

- Recall: Decision rule is $f(\mathbf{x}) = \operatorname{sgn}(\langle \mathbf{x}, \mathbf{v}_{\mathrm{H}} \rangle - c)$
- Idea: Combine classifiers *before* computing sign. Define

$$g_k(\mathbf{x}) := \langle \mathbf{x}, \mathbf{v}_k \rangle - c_k$$

Multiclass linear discriminant

- Use one classifier g_k (as above) for each class k.
- Trained e.g. as one-against-rest.



- Classify according to

$$f(\mathbf{x}) := \arg\max_k \{g_k(\mathbf{x})\}$$

- If $g_k(\mathbf{x})$ is positive for several classes, a larger value of g_k means that $\mathbf{x}$ lies "further" into class k than into any other class j.
- If $g_k(\mathbf{x})$ is negative for all k, the maximum means we classify $\mathbf{x}$ according to the class represented by the closest hyperplane.
- Regions are convex.

Problem

- Multiclass discriminant idea: Compare distances to hyperplanes.
- Works if the orthogonal vectors $\mathbf{v}_H$ determining the hyperplanes are normalized.
- SVM: The K classifiers in multiple discriminant approach are trained on separate problems, so the individual lengths of $\mathbf{v}_H$ computed by max-margin algorithm are not comparable.

Workarounds

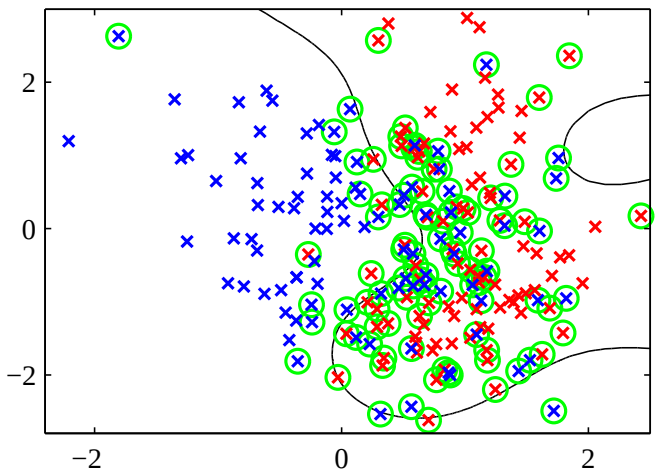
- Often: One-against-all approaches.
- It is possible to define a single optimization problem for all classes, but training time scales quadratically in number of classes.

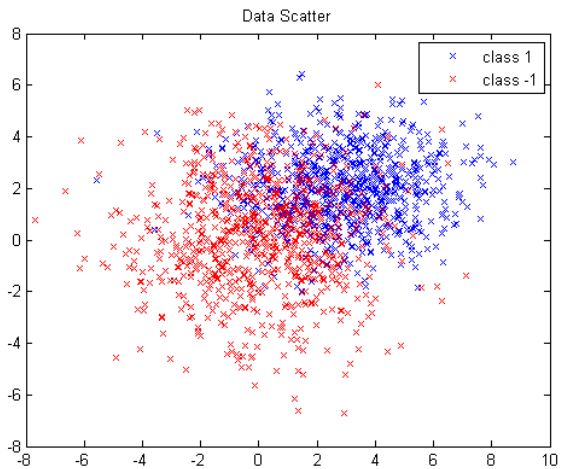
KERNELS

Classifiers discussed so far

- Both assume linear decision boundary
- Perceptron: Linear separability; placement of boundary rather arbitrary

More realistic data





Motivation: Kernels

Idea

- The SVM uses the scalar product $\langle \mathbf{x}, \tilde{\mathbf{x}}_i \rangle$ as a measure of similarity between $\mathbf{x}$ and $\tilde{\mathbf{x}}_i$, and of distance to the hyperplane.
- Since the scalar product is linear, the SVM is a linear method.
- By using a *nonlinear* function instead, we can make the classifier nonlinear.

More precisely

- Scalar product can be regarded as a two-argument function

$$\langle \,.\,,\,. \,\rangle : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$$

- We will replace this function with a function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ and substitute

$$k(\mathbf{x}, \mathbf{x}') \qquad \text{for every occurrence of} \qquad \langle \mathbf{x}, \mathbf{x}' \rangle$$

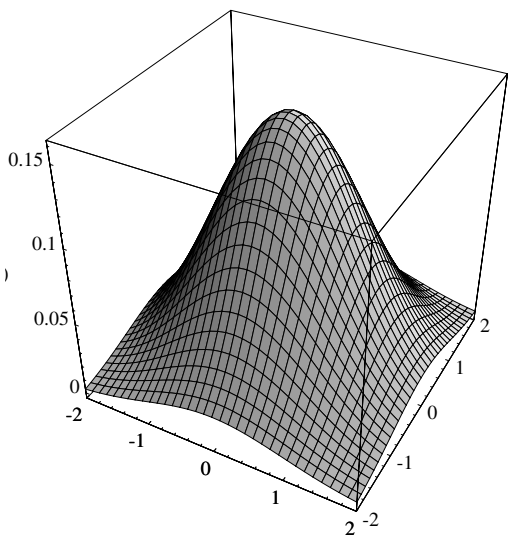
  in the SVM formulae.
- Under certain conditions on k, all optimization/classification results for the SVM still hold. Functions that satisfy these conditions are called **kernel functions**.

RBF Kernel

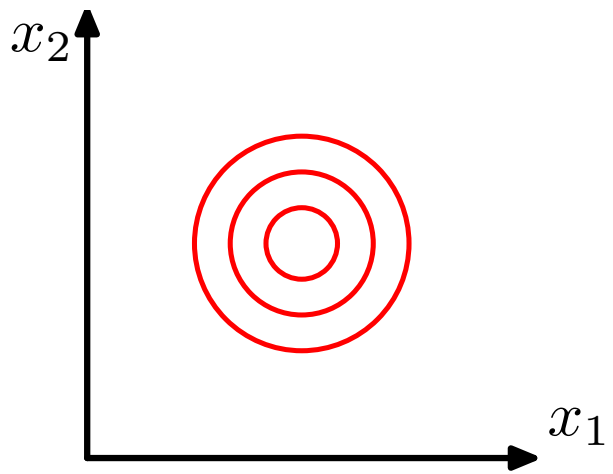
$$k_{\text{RBF}}(\mathbf{x}, \mathbf{x}') := \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2\sigma^2}\right) \qquad \text{for some } \sigma \in \mathbb{R}_+$$

is called an **RBF kernel** (RBF = radial basis function). The parameter σ is called **bandwidth**.
Other names for k_{RBF}: Gaussian kernel, squared-exponential kernel.

If we fix $\mathbf{x}'$, the function $k_{\text{RBF}}(\,.\,, \mathbf{x}')$ is (up to scaling) a spherical Gaussian density on $\mathbb{R}^d$, with mean $\mathbf{x}'$ and standard deviation σ.



function surface (d=2)



contours (d=2)

Choosing a kernel

Theory

To define a kernel:

- We have to define a function of two arguments and prove that it is a kernel.
- This is done by checking a set of necessary and sufficient conditions known as "Mercer's theorem".

Practice

The data analyst does not define a kernel, but tries some well-known standard kernels until one seems to work. Most common choices:

- The RBF kernel.
- The "linear kernel" $k_{\text{SP}}(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}' \rangle$, i.e. the standard, linear SVM.

Once kernel is chosen

- Classifier can be trained by solving the optimization problem using standard software.
- SVM software packages include implementations of most common kernels.

Which Functions work as Kernels?

Formal definition

A function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is called a **kernel** on $\mathbb{R}^d$ if there is *some* function $\phi : \mathbb{R}^d \to \mathcal{F}$ into *some* space $\mathcal{F}$ with scalar product $\langle \,.\,,\,.\,\rangle_{\mathcal{F}}$ such that

$$k(\mathbf{x}, \mathbf{x}') = \big\langle \phi(\mathbf{x}), \phi(\mathbf{x}') \big\rangle_{\mathcal{F}} \qquad \text{for all } \mathbf{x}, \mathbf{x}' \in \mathbb{R}^d \ .$$

In other words

- k is a kernel if it can be interpreted as a scalar product on some other space.
- If we substitute $k(\mathbf{x}, \mathbf{x}')$ for $\langle \mathbf{x}, \mathbf{x}' \rangle$ in all SVM equations, we implicitly train a *linear* SVM on the space $\mathcal{F}$.
- The SVM still works: It still uses scalar products, just on another space.

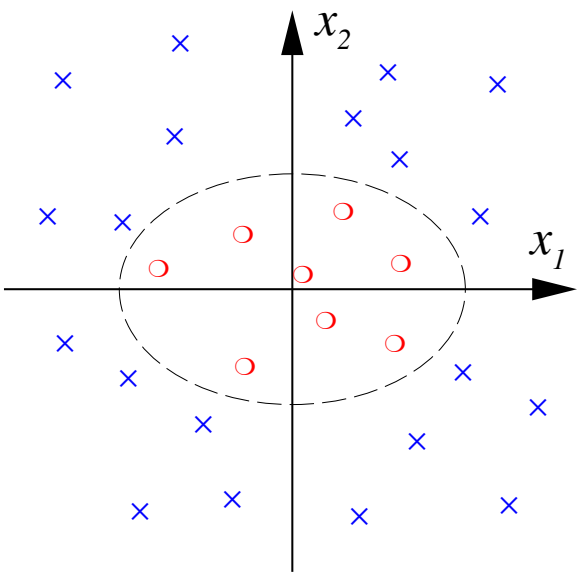
The mapping ϕ

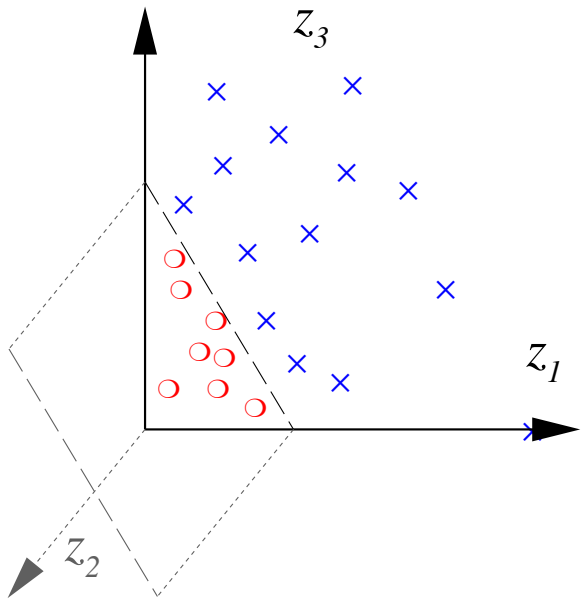
- ϕ has to transform the data into data on which a linear SVM works well.
- This is usually achieved by choosing $\mathcal{F}$ as a higher-dimensional space than $\mathbb{R}^d$.

Example

How can a map into higher dimensions make class boundary (more) linear?
Consider

$$\phi : \mathbb{R}^2 \to \mathbb{R}^3 \qquad \text{where} \qquad \phi \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} := \begin{pmatrix} x_1^2 \\ 2x_1x_2 \\ x_2^2 \end{pmatrix}$$





Problem

In previous example: We have to know what the data looks like to choose ϕ!

Solution

- Choose high dimension h for $\mathcal{F}$.
- Choose components ϕ_i of $\phi(\mathbf{x}) = (\phi_1(\mathbf{x}), \ldots, \phi_h(\mathbf{x}))$ as different nonlinear mappings.
- If two points differ in $\mathbb{R}^d$, some of the nonlinear mappings will amplify differences.

The RBF kernel is an extreme case

- The function k_{RBF} can be shown to be a kernel, however:
- $\mathcal{F}$ is infinite-dimensional for this kernel.

Mercer's theorem

A mathematical result called *Mercer's theorem* states that, if the function k is positive, i.e.

$$\int_{\mathbb{R}^d \times \mathbb{R}^d} k(\mathbf{x}, \mathbf{x}') f(\mathbf{x}) f(\mathbf{x}') d\mathbf{x} d\mathbf{x}' \geq 0$$

for all functions f, then it can be written as

$$k(\mathbf{x}, \mathbf{x}') = \sum_{j=1}^{\infty} \lambda_j \phi_j(\mathbf{x}) \phi_j(\mathbf{x}') \ .$$

The ϕ_j are functions $\mathbb{R}^d \to \mathbb{R}$ and $\lambda_i \geq 0$. This means the (possibly infinite) vector $\phi(\mathbf{x}) = (\sqrt{\lambda_1}\phi_1(\mathbf{x}), \sqrt{\lambda_2}\phi_2(\mathbf{x}), \ldots)$ is a feature map.

Kernel arithmetic

Various functions of kernels are again kernels: If k_1 and k_2 are kernels, then e.g.

$$k_1 + k_2 \qquad\qquad k_1 \cdot k_2 \qquad\qquad \text{const.} \cdot k_1$$

are again kernels.

Kernels in general

- Many linear machine learning and statistics algorithms can be "kernelized".
- The only conditions are:
  1. The algorithm uses a scalar product.
  2. In all relevant equations, the data (and all other elements of $\mathbb{R}^d$) appear *only inside a scalar product*.
- This approach to making algorithms non-linear is known as the "kernel trick".

Optimization problem

$$\begin{aligned} \min_{\mathbf{v}_H, c} \quad & \|\mathbf{v}_H\|_{\mathcal{F}}^2 + \gamma \sum_{i=1}^{n} \xi^2 \\ \text{s.t.} \quad & y_i(\langle \mathbf{v}_H, \phi(\tilde{\mathbf{x}}_i) \rangle_{\mathcal{F}} - c) \geq 1 - \xi_i \quad \text{and } \xi_i \geq 0 \end{aligned}$$

Note: $\mathbf{v}_H$ now lives in $\mathcal{F}$, and $\|\,.\,\|_{\mathcal{F}}$ and $\langle\,.\,,\,.\,\rangle_{\mathcal{F}}$ are norm and scalar product on $\mathcal{F}$.

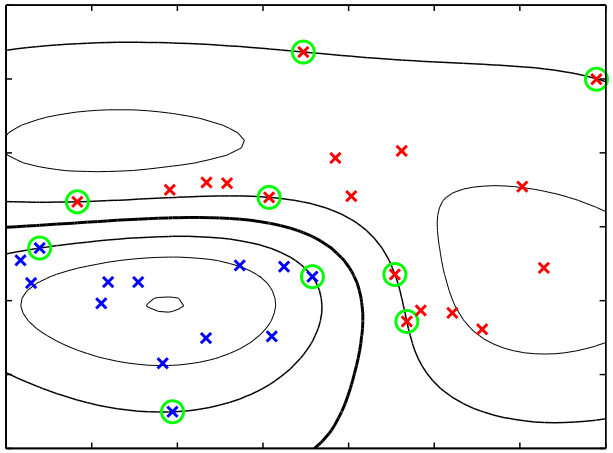
Dual optimization problem

$$\begin{aligned} \max_{\boldsymbol{\alpha} \in \mathbb{R}^n} \quad & W(\boldsymbol{\alpha}) := \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j \tilde{y}_i \tilde{y}_j (k(\tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_j) + \frac{1}{\gamma} \mathbb{I}\{i = j\}) \\ \text{s.t.} \quad & \sum_{i=1}^{n} y_i \alpha_i = 0 \qquad \text{and} \qquad \alpha_i \geq 0 \end{aligned}$$

Classifier

$$f(\mathbf{x}) = \operatorname{sgn}\left(\sum_{i=1}^{n} \tilde{y}_i \alpha_i^* k(\tilde{\mathbf{x}}_i, \mathbf{x}) - c \right)$$

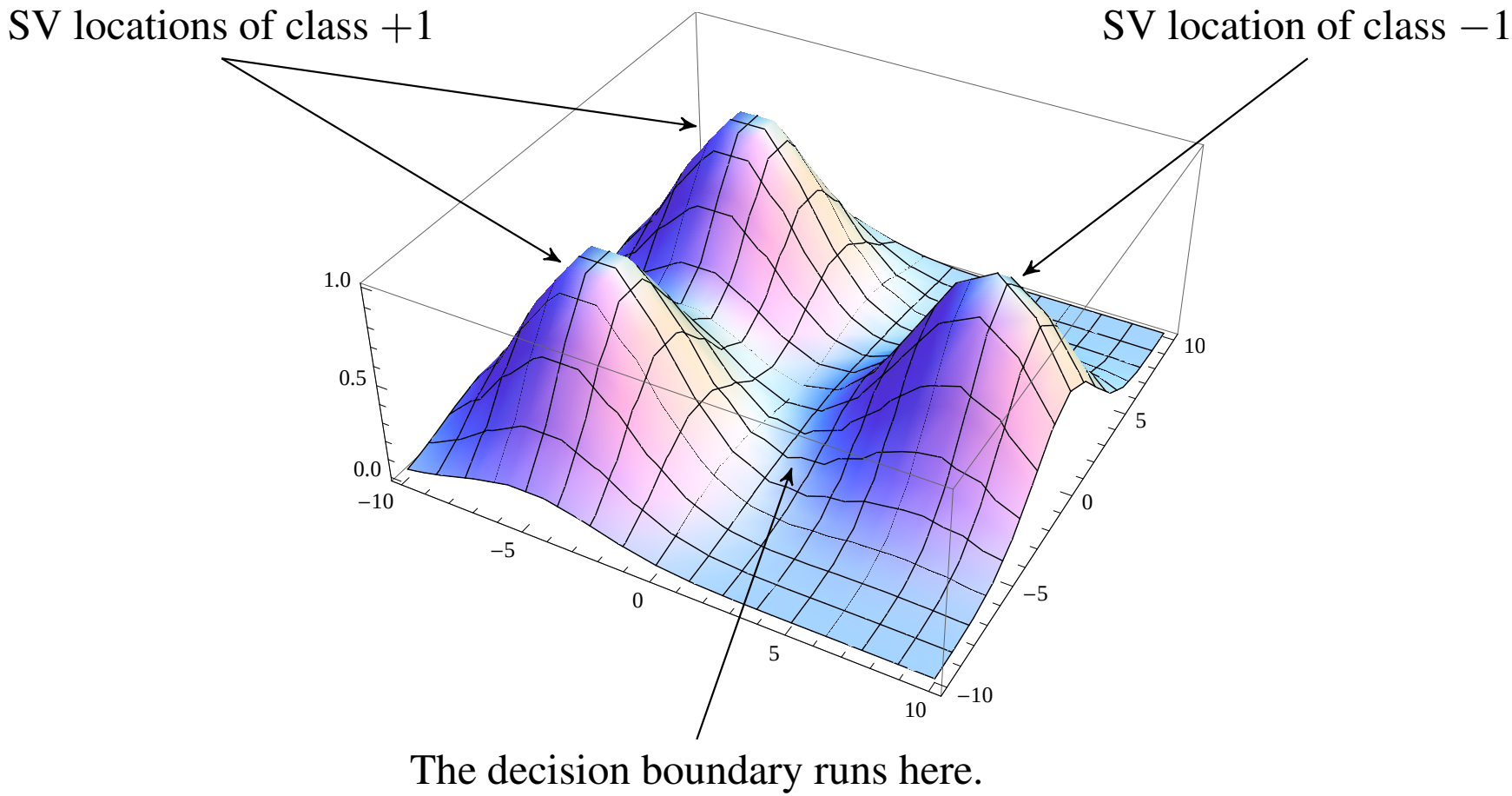
SVM WITH RBF KERNEL



$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{n} y_i \alpha_i^* k_{\text{RBF}}(\mathbf{x}_i, \mathbf{x})\right)$$

- Circled points are support vectors. The the two contour lines running through support vectors are the nonlinear counterparts of the convex hulls.
- The thick black line is the classifier.
- Think of a Gaussian-shaped function $k_{\text{RBF}}(\,.\,, \mathbf{x}')$ centered at each support vector $\mathbf{x}'$. These functions add up to a function surface over $\mathbb{R}^2$.
- The lines in the image are contour lines of this surface. The classifier runs along the bottom of the "valley" between the two classes.
- Smoothness of the contours is controlled by σ

Decision Boundary with RBF Kernel



The decision boundary of the classifier coincides with the set of points where the surfaces for class $+1$ and class -1 have equal value.

SUMMARY: SVMS

Basic SVM

- Linear classifier for linearly separable data.
- Positions of affine hyperplane is determined by maximizing margin.
- Maximizing the margin is a convex optimization problem.

Full-fledged SVM

Ingredient	Purpose
Maximum margin	Good generalization properties
Slack variables	Overlapping classes
	Robustness against outliers
Kernel	Nonlinear decision boundary

Use in practice

- Software packages (e.g. libsvm, SVMLite)
- Choose a kernel function (e.g. RBF)
- Cross-validate margin parameter γ and kernel parameters (e.g. bandwidth)

History

- Ca. 1957: Perceptron (Rosenblatt)

- 1970s: Vapnik and Chervonenkis develop learning theory
- 1986: Neural network renaissance (backpropagation algorithm by Rumelhart, Hinton, Williams)
- 1993: SVM (Boser, Guyon, Vapnik)
- 1997: Boosting (Freund and Schapire)

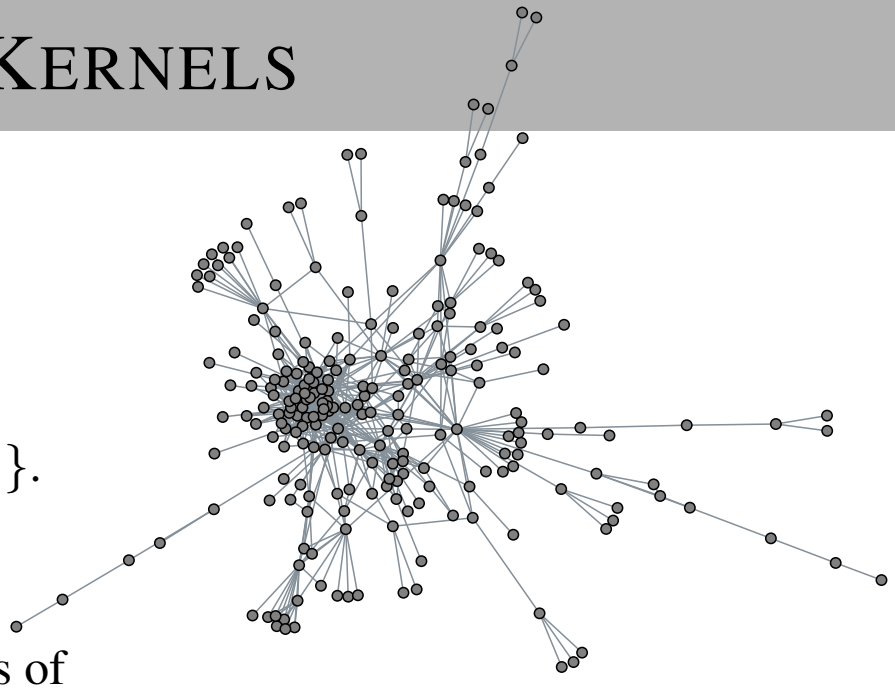
Unusual Example: Graph Kernels

Terminology

A **graph** $G = (V, E)$ is defined by two sets:

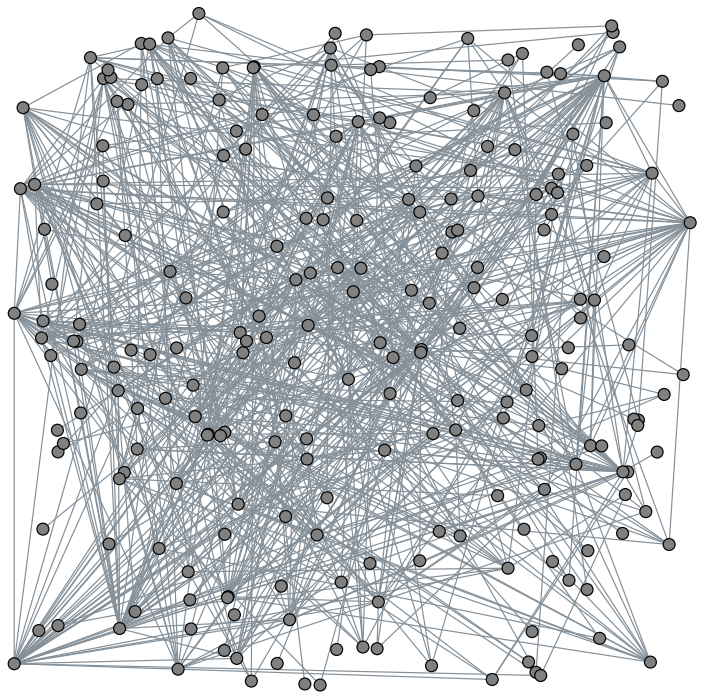
1. A set of V **vertices** $v_1, \ldots, v_m$.
2. A set E of **edges**, i.e. variables $e_{ij} \in \{0, 1\}$.

$e_{ij} = 1$ means that v_i and v_j are connected.



The graph is **undirected** if $e_{ij} = e_{ji}$ for all pairs of vertices. (The graphs in the figure are undirected.)

We write $\mathcal{G}$ for the set of undirected graphs of finite size.



Problem setting

- Training data $(\tilde{G}_i, \tilde{y}_i)_{i \in [n]}$, where each $\tilde{G}_i$ is a graph in $\mathcal{G}$.
- Can we learn a classifier f that classifies an unlabeled graph G?

Example 1: Social Networks

- Each vertex v_j is a user.
- $e_{ij} = 1$ indicates that users i and j are "friends".

This data is graph-valued, but the data set typically consists of a single, very large graph.

Example 2: Biology

There are dozens of types of graph-valued data in biology. One example is protein-protein interaction data:

- Each vertex v_j is a protein.
- $e_{ij} = 1$ indicates that proteins i and j interact in the given system.

(The graph on the previous slide shows such a data set.)

Graph kernels are designed for problems where we observe a set of graphs.

Modeling assumption

Graph G is characterized by how often certain patterns (= subgraphs) occur in G.

Feature map

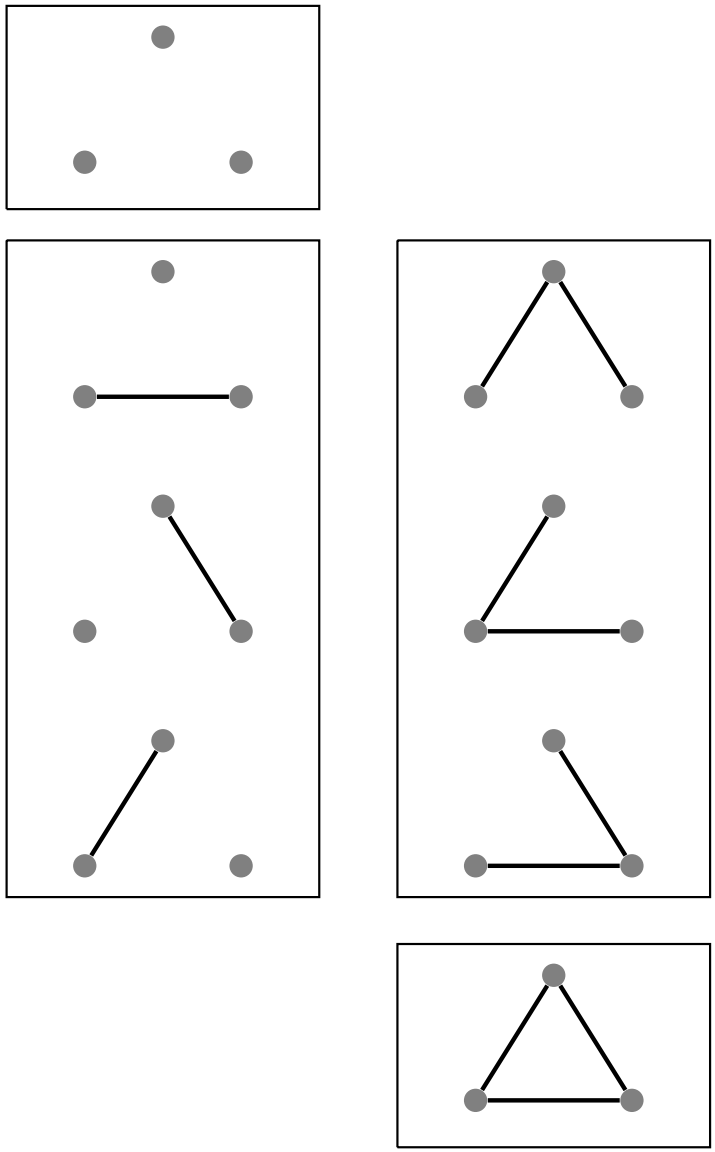
- Fix a set $\mathcal{K}$ of patterns.
  Example: All subgraphs of size 3.
- For graphs $G \in \mathcal{G}$, define

$$\phi_F(G) := \frac{\#\text{ occurrences of } F \text{ in } G}{\#\text{subgraphs of size } |F| \text{ in } G}$$

- Define the feature map ϕ as the vector

$$\phi(G) = (\phi_F(G))_{F \in \mathcal{K}}$$

This is a mapping $\phi : \mathcal{G} \to \mathbb{R}_+^d$.
The dimension is $d = |\mathcal{K}|$.



All subgraphs of size 3

Kernel

The kernel defined by ϕ is

$$k(G, G') := \langle \phi(G), \phi(G') \rangle = \sum_{F \in \mathcal{K}} \phi_F(G) \cdot \phi_F(G')$$

A large value of k indicates there is a subgraph in $\mathcal{K}$ that occurs often in *both* graphs.

Classification

We can now train an SVM as usual. For training data $(\tilde{G}_1, \tilde{y}_1), \ldots, (\tilde{G}_n, \tilde{y}_n)$, the resulting classifier is

$$f(G) = \operatorname{sgn}\left(\sum_{i=1}^{n} \tilde{y}_i \alpha_i^* k(\tilde{G}_i, G) - c \right)$$

Other graph kernels

- There are various other ways to define graph kernels. For example, k could compare G and G' in terms of the probability that a random walk on G and a random walk on G' take the same path. (Such kernels are called *random walk kernels*.)
- Each choice of kernel emphasizes a different property in terms of which the graphs are compared.

More generally: Kernels for non-Euclidean data

- We have used the kernel to transform non-Euclidean data (graphs) so that it fits into our classification framework.
- There are other, similar methods, e.g. string kernels.
- Note that we have not used the kernel for implicit representation, but rather compute ϕ explicitly.

Model Selection and Cross Validation

Objective

- Cross validation is a method which tries to select the best model from a given set of models.
- Assumption: Quality measure is predictive performance.
- "Set of models" can simply mean "set of different parameter values".

Terminology

The problem of choosing a good model is called **model selection**.

Model selection problem for SVM

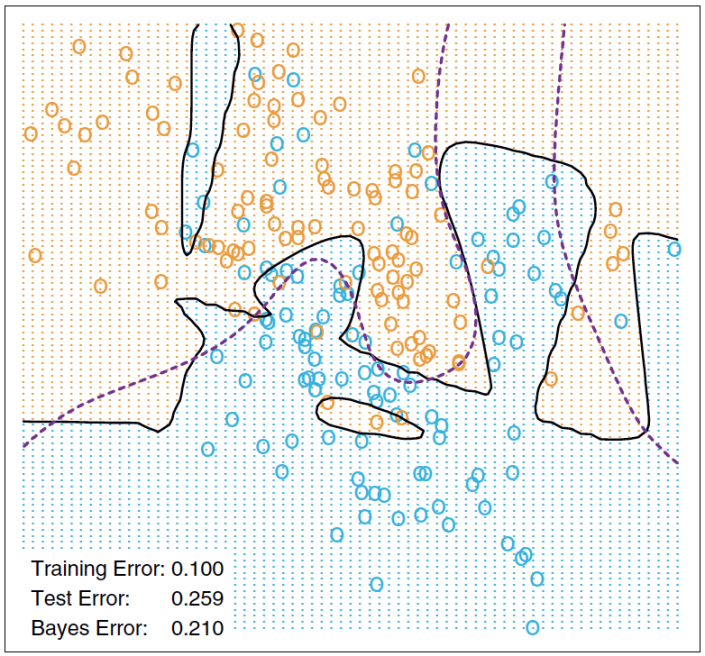
- The SVM is a *family* of models indexed by the margin parameter γ and the kernel parameter(s) σ.
- Our goal is to find a value of (γ, σ) for which we can expect small generalization error.

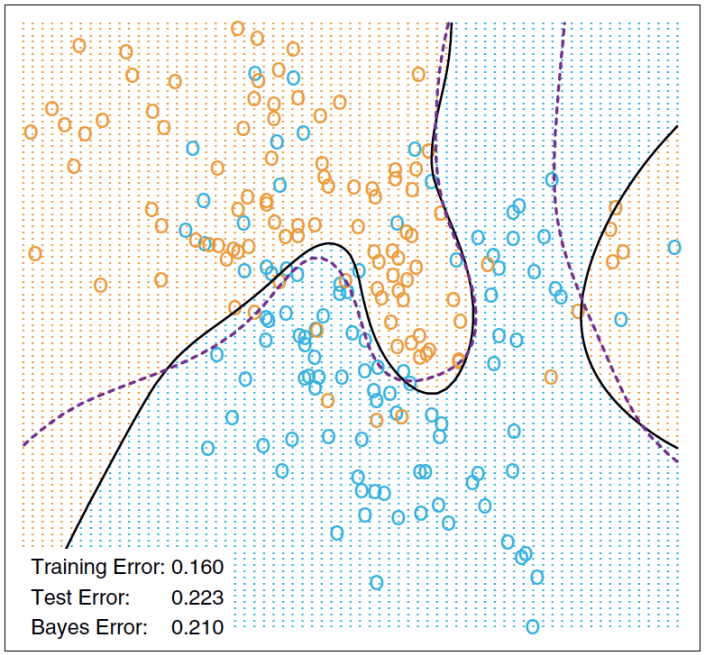
Naive approach

- We could include (γ, σ) into the optimization problem, i.e. train by minimizing over α *and* (γ, σ).
- This leads to a phenomenon called **overfitting**: The classifier adapts too closely to specific properties of the training data, rather than the underlying distribution.

Overfitting: Illustration

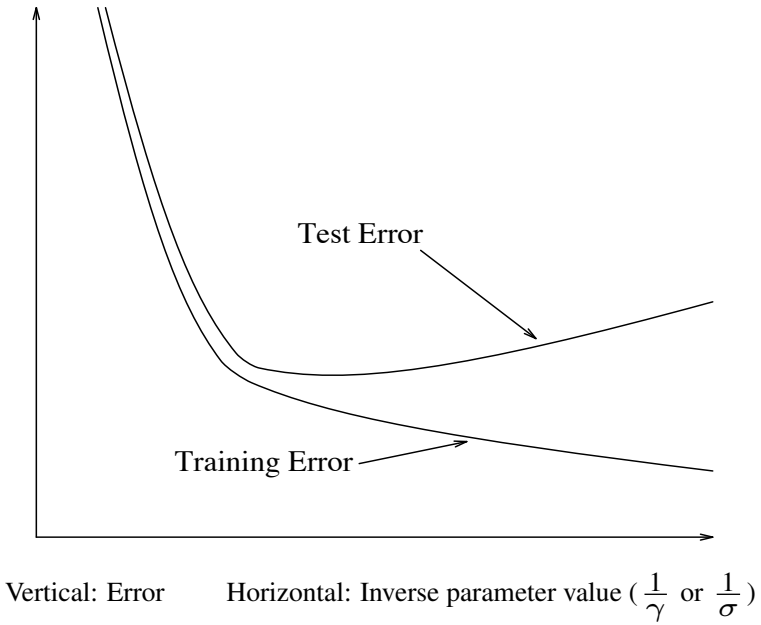
Overfitting is best illustrated with a *nonlinear* classifier.





- The classifier in this example only has a "bandwidth" parameter σ, similar to the parameter σ of the RBF kernel.
- Small σ permits curve with sharp bends; large σ: Smooth curve.

Conceptual illustration



- If classifier can adapt (too) well to data: Small training error, but possibly large test error.
- If classifier can hardly adapt at all: Large training and test error.
- Somewhere in between, there is a sweet spot.
- Trade-off is controlled by the parameter.

(From now on, we just write γ to denote the entire set of model parameters.)

Cross Validation: Procedure

Model selection:

1. Randomly split data into three sets: training, validation and test data.

2. Train classifier on training data for different values of γ.
3. Evaluate each trained classifier on validation data (ie compute error rate).
4. Select the value of γ with lowest error rate.

Model assessment:

5. Finally: Estimate the error rate of the selected classifier on test data.

Meaning

- The quality measure by which we are comparing different classifiers $f(\,.\,;\gamma)$ (for different parameter values γ) is the risk

$$R(f(\,.\,;\gamma)) = \mathbb{E}[L(y, f(x;\gamma))]\,.$$

- Since we do not know the true risk, we estimate it from data as $\hat{R}(f(\,.\,;\gamma))$.

Importance of model assessment step

- We always have to assume: Classifier is better adapted to *any* data used to select it than to actual data distribution.
- Model selection: Adapts classifier to *both* training and validation data.
- If we estimate error rate on this data, we will in general underestimate it.

Procedure in detail

We consider possible parameter values $\gamma_1, \ldots, \gamma_m$.

1. For each value γ_j, train a classifier $f(\,.\,;\gamma_j)$ on the training set.
2. Use the validation set to estimate $R(f(\,.\,;\gamma_j))$ as the empirical risk

$$\hat{R}(f(x;\gamma_j)) = \frac{1}{n_v}\sum_{i=1}^{n_v} L(\tilde{y}_i, f(\tilde{\mathbf{x}}_i, \gamma_j)) \ .$$

   n_v is the size of the validation set.
3. Select the value γ^* which achieves the smallest estimated error.
4. Re-train the classifier with parameter γ^* on all data except the test set (i.e. on training + validation data).
5. Report error estimate $\hat{R}(f(\,.\,;\gamma^*))$ computed on the *test* set.

K-FOLD CROSS VALIDATION

Idea

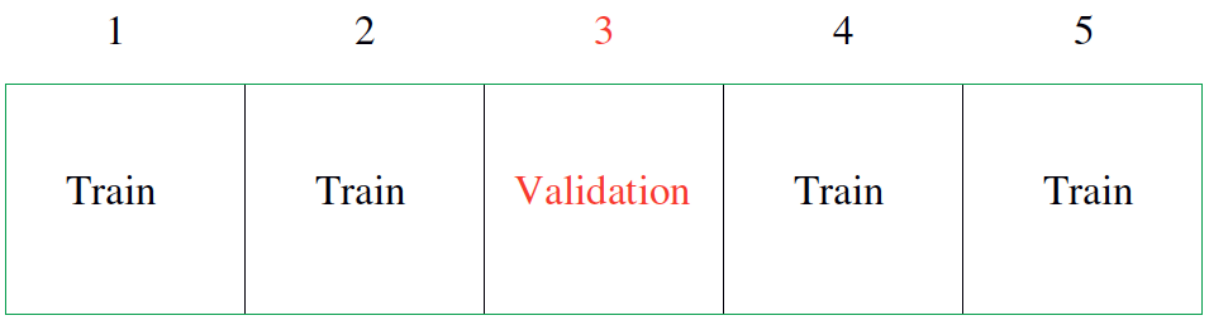
Each of the error estimates computed on validation set is computed from a single example of a trained classifier. Can we improve the estimate?

Strategy

- Set aside the test set.
- Split the remaining data into K blocks (called "folds").
- Use each fold in turn as validation set. Perform cross validation and average the results over all K combinations.

This method is called **K-fold cross validation**.

Example: K=5, step k=3



K-FOLD CROSS VALIDATION: PROCEDURE

Risk estimation

To estimate the risk of a classifier $f(\,.\,,\gamma_j)$:

1. Split data into K equally sized parts.
2. Train an instance $f_k(\,.\,,\gamma_j)$ of the classifier, using all folds except fold k as training data.
3. Compute the cross validation estimate

$$\hat{R}_{\mathrm{CV}}(f(\,.\,,\gamma_j)) := \frac{1}{K}\sum_{k=1}^{K}\frac{1}{|\text{fold } k|}\sum_{(\tilde{\mathbf{x}},\tilde{y})\in \text{ fold } k} L(\tilde{y},f_k(\tilde{\mathbf{x}},\gamma_j))$$

Repeat this for all parameter values $\gamma_1,\ldots,\gamma_m$.

Selecting a model

Choose the parameter value γ^* for which estimated risk is minimal.

Model assessment

Report risk estimate for $f(\,.\,,\gamma^*)$ computed on *test* data.

Extremal cases

- $K = n$, called **leave one out cross validation** (loocv)
- $K = 2$

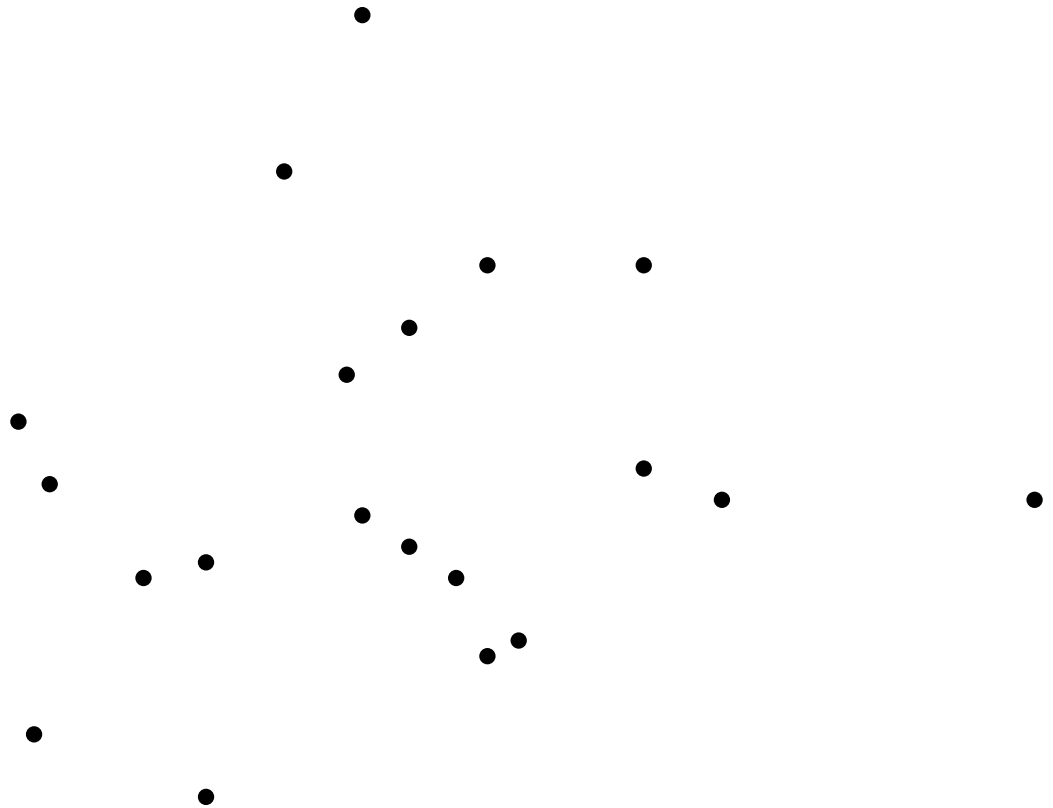
An often-cited problem with loocv is that we have to train many (= n) classifiers, but there is also a deeper problem.

Argument 1: K should be small, e.g. $K = 2$

- Unless we have a lot of data, variance between two distinct training sets may be considerable.
- **Important concept:** By removing substantial parts of the sample in turn and at random, we can simulate this variance.
- By removing a single point (loocv), we cannot make this variance visible.

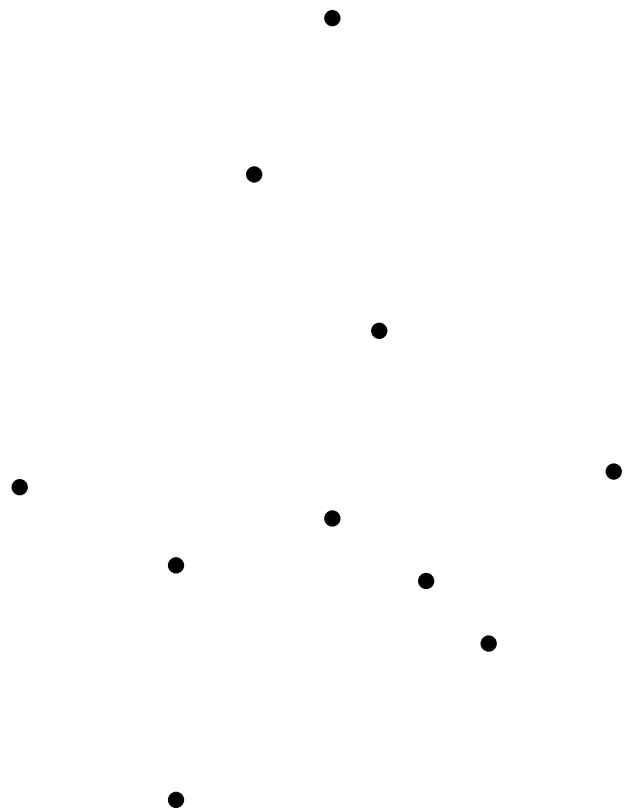
Illustration

$K = 2, n = 20$



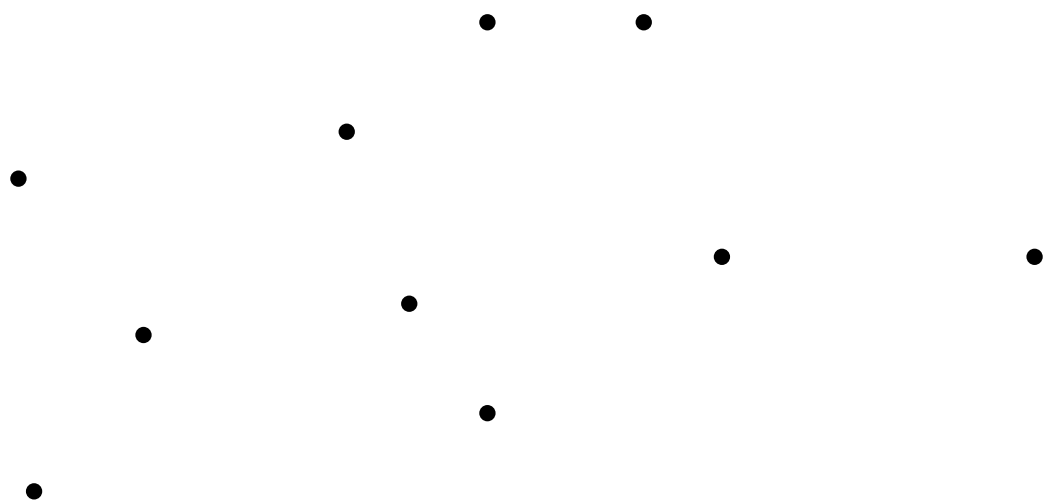
Illustration

$K = 2, n = 20$



Illustration

$K = 2, n = 20$



How to Choose K?

Argument 2: K should be large, e.g. $K = n$

- Classifiers generally perform better when trained on larger data sets.
- A small K means we substantially reduce the amount of training data used to train each f_k, so we may end up with weaker classifiers.
- This way, we will systematically overestimate the risk.

Common recommendation: $K = 5$ to $K = 10$

Intuition:

- $K = 10$ means number of samples removed from training is one order of magnitude below training sample size.
- This should not weaken the classifier considerably, but should be large enough to make measure variance effects.

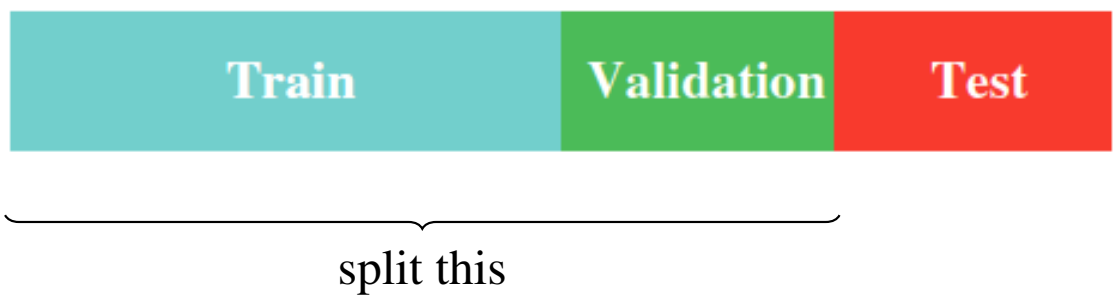
SUMMARY: CROSS VALIDATION

Purpose

Estimates the risk $R(f) = \mathbb{E}[L(y, f(x))]$ of a classifier (or regression function) from data.

Application to parameter tuning

- Compute one cross validation estimate of $R(f)$ for each parameter value.
- Example above is margin parameter γ, but can be used for any parameter of a supervised learning algorithm.
- Note: Cross validation procedure does not involve the test data.



TREE CLASSIFIERS

Idea

- Recall: Classifiers classify according to location in $\mathbb{R}^d$
- Linear classifiers: Divide space into two halfspaces
- What if we are less sophisticated and divide space only along axes? We could classify e.g. according to

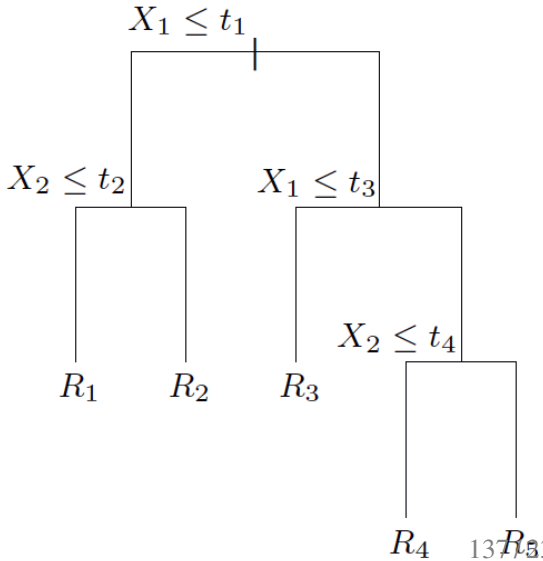
$$\mathbf{x} \in \begin{cases} \text{Class +} & \text{if } x_3 > 0.5 \\ \text{Class -} & \text{if } x_3 \leq 0.5 \end{cases}$$

- This decision would correspond to an affine hyperplane perpendicular to the x_3-axis, with offset 0.5.

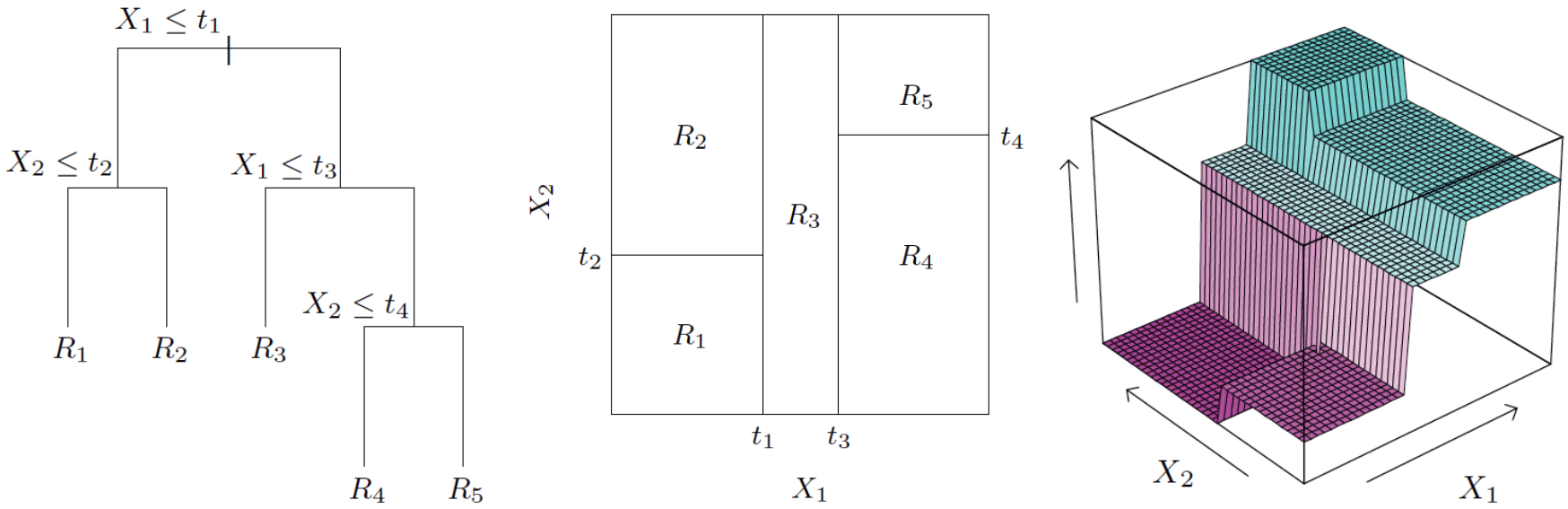
Tree classifier

A **tree classifier** is a binary tree in which

- Each inner node is a rule of the form $x_i > t_i$.
- The threshold values t_i are the parameters which specify the tree.
- Each leaf is a class label.



TREES



- Each leaf of the tree corresponds to a region R_m of $\mathbb{R}^d$.
- Classes $k \in \{1, \ldots, K\}$ (not restricted to two classes).
- Training: Each node is assigned class to which most points in R_m belong,

$$k(m) := \arg\max_k \#\{x_i \in R_m \text{ with } y_i = k\}$$

- In training algorithm, we have to fix a region R_m and split it along an axis j at a point t_j.
- The split results in two new regions R_m^1 and R_m^2.
- On each region, we obtain a new class assignment $k^1(m)$ and $k^2(m)$.
- Strategy is again: Define cost of split at t_j and minimize it to find t_j.

Cost of a split

$$Q(R_m, t_j) := \frac{\sum_{\tilde{\mathbf{x}}_i \in R_m^1} \mathbb{I}\{\tilde{y}_i \neq k^1(m)\} + \sum_{\tilde{\mathbf{x}}_i \in R_m^2} \mathbb{I}\{\tilde{y}_i \neq k^2(m)\}}{\#\{x_i \in R_m\}}$$

In words:

$$Q = \text{proportion of training points in } R_m \text{ that get misclassified if we choose to split at } t_j$$

Overall algorithm

- At each step: Current tree leaves define regions $R_1, \ldots, R_M$.
- For each R_m, find the best split.
- Continue splitting regions until tree has depth D (input parameter).

Step of training algorithm

At each step: Current tree leaves define regions $R_1, \ldots, R_M$.

For each region R_m:

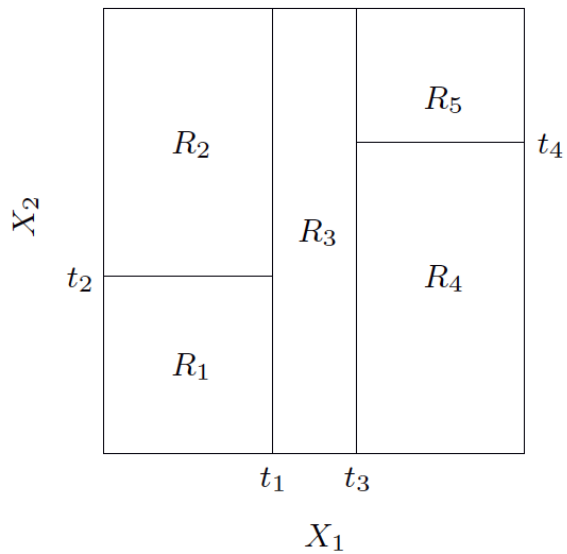
1. For each axis j: Compute best splitting point t_j as

$$t_j := \arg\min Q(R_m, t_j)$$

2. Select best splitting axis:

$$j := \arg\min_j Q(R_m, t_j)$$

3. Split R_m along axis j at t_j



EXAMPLE: SPAM FILTERING

Data

- 4601 email messages
- Classes: email, spam

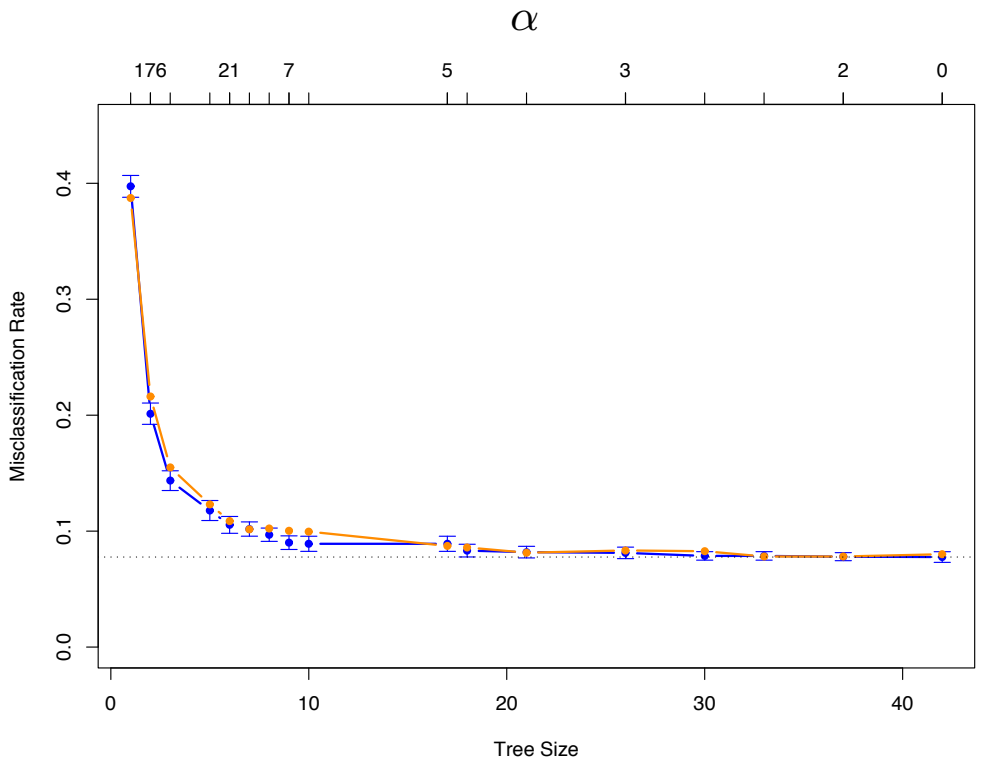
	george	you	your	hp	free	hpl	!	our	re	edu	remove
spam	0.00	2.26	1.38	0.02	0.52	0.01	0.51	0.51	0.13	0.01	0.28
email	1.27	1.27	0.44	0.90	0.07	0.43	0.11	0.18	0.42	0.29	0.01

Tree classifier

- 17 nodes
- Performance:

	Predicted	
True	Email	Spam
Email	57.3%	4.0%
Spam	5.3%	33.4%

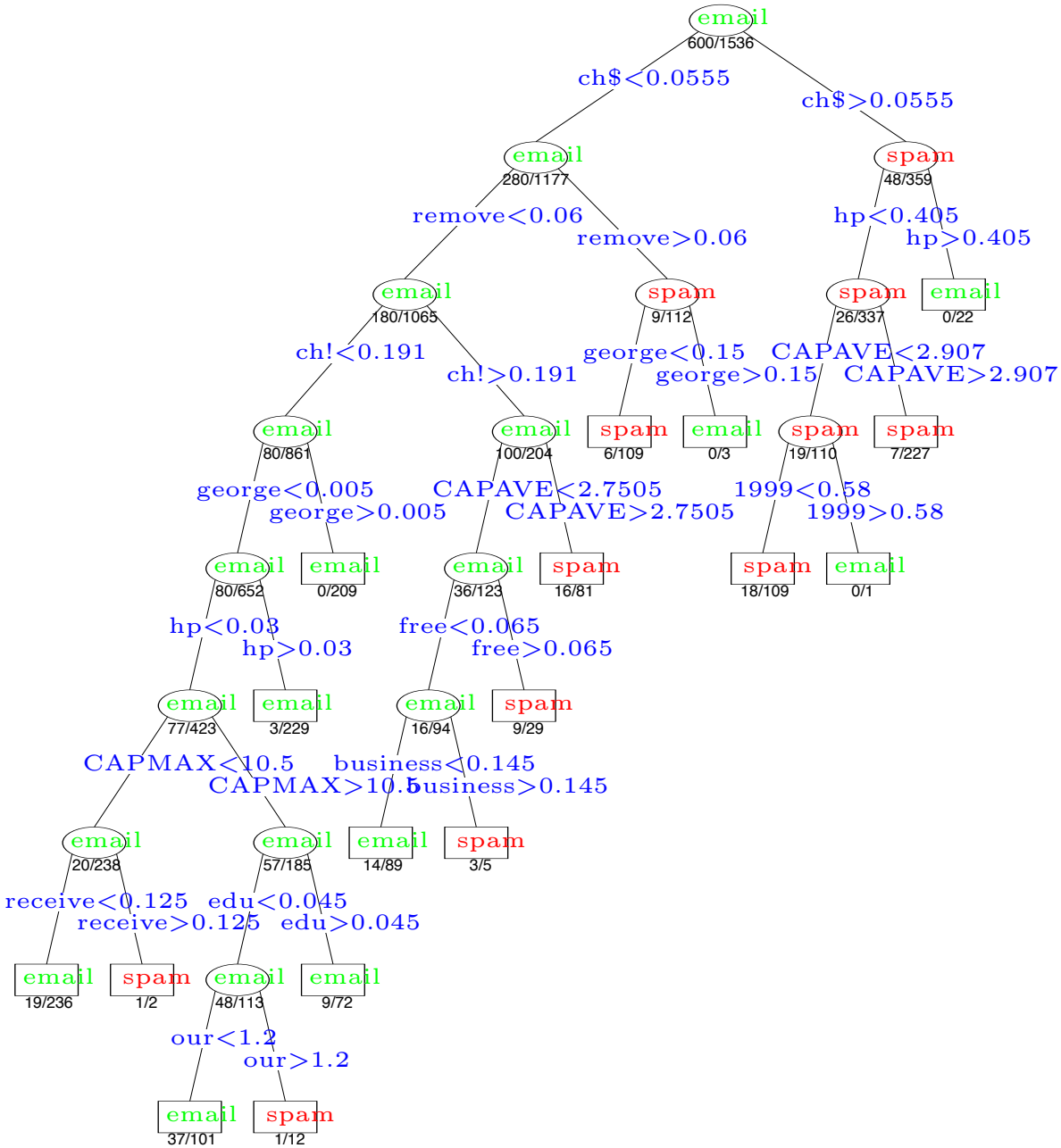
Influence of Tree Size



Tree Size

- Tree of height D defines 2^D regions.
- D too small: Insufficient accuracy. D too large: Overfitting.
- D can be determined by cross validation or more sophisticated methods ("complexity pruning" etc), which we will not discuss here.

SPAM FILTERING: TREE

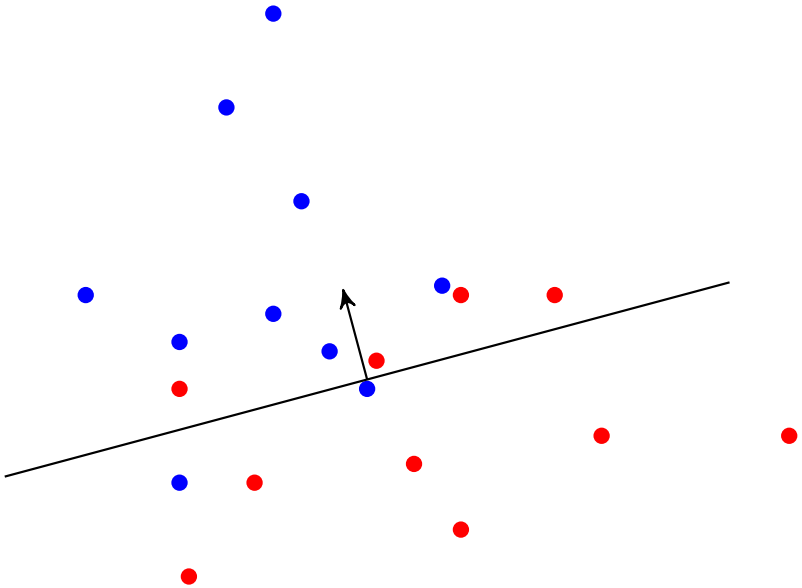


Decision Stumps

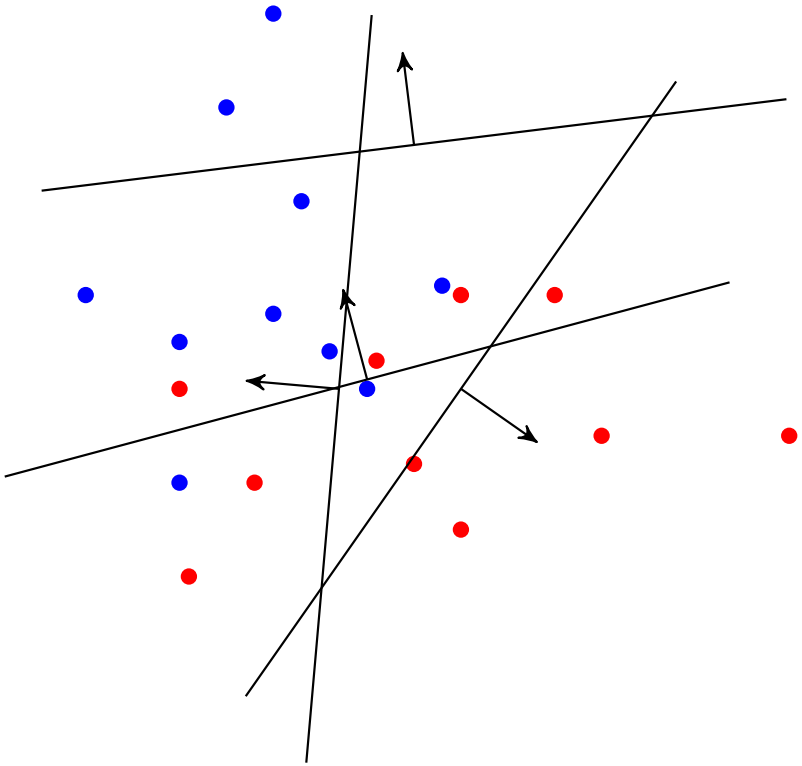
- The simplest possible tree classifier is a tree of depth 1. Such a classifier is called a **decision stump**.
- A decision stump is parameterized by a pair (j, t_j) of an axis j and a splitting point t_j.
- Splits $\mathbb{R}^d$ into two regions.
- Decision boundary is an affine hyperplane which is perpendicular to axis j and intersects the axis at t_j.
- Often used in Boosting algorithms and other ensemble methods.

BOOSTING

A *randomly* chosen hyperplane classifier has an *expected* error of 0.5 (i.e. 50%).

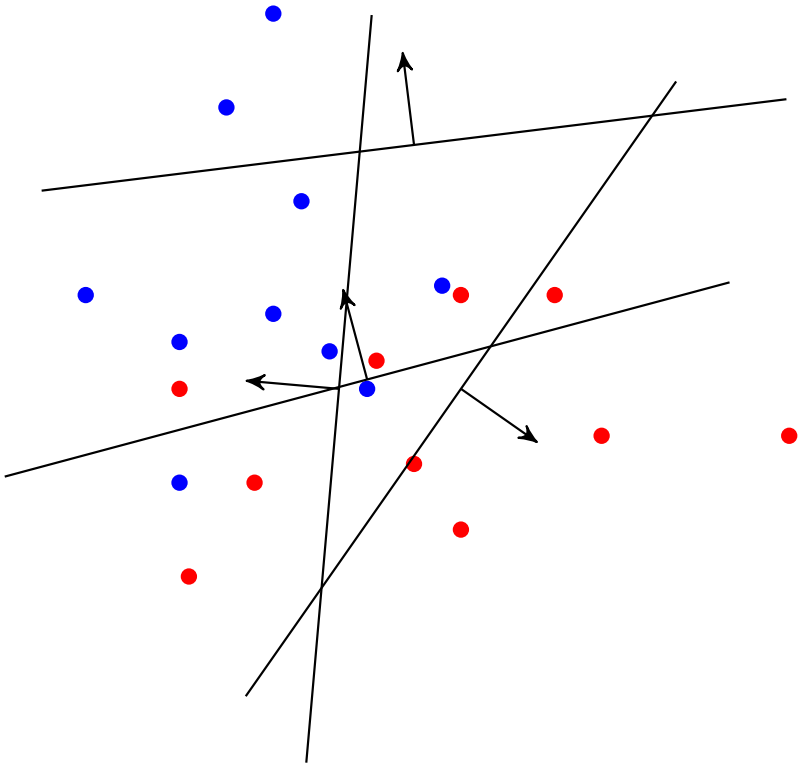


A *randomly* chosen hyperplane classifier has an *expected* error of 0.5 (i.e. 50%).



ENSEMBLES

A *randomly* chosen hyperplane classifier has an *expected* error of 0.5 (i.e. 50%).



- Many random hyperplanes combined by majority vote: Still 0.5.
- A single classifier slightly better than random: $0.5 + \varepsilon$.
- What if we use m such classifiers and take a majority vote?

Decision by majority vote

- m individuals (or classifiers) take a vote. m is an odd number.
- They decide between two choices; one is correct, one is wrong.
- After everyone has voted, a decision is made by simple majority.

Note: For two-class classifiers $f_1, \ldots, f_m$ (with output ± 1):

$$\text{majority vote} = \operatorname{sgn}\left(\sum_{j=1}^{m} f_j\right)$$

Assumptions

Before we discuss ensembles, we try to convince ourselves that voting can be beneficial. We make some simplifying assumptions:

- Each individual makes the right choice with probability $p \in [0, 1]$.
- The votes are *independent*, i.e. stochastically independent when regarded as random outcomes.

Does the Majority Make the Right Choice?

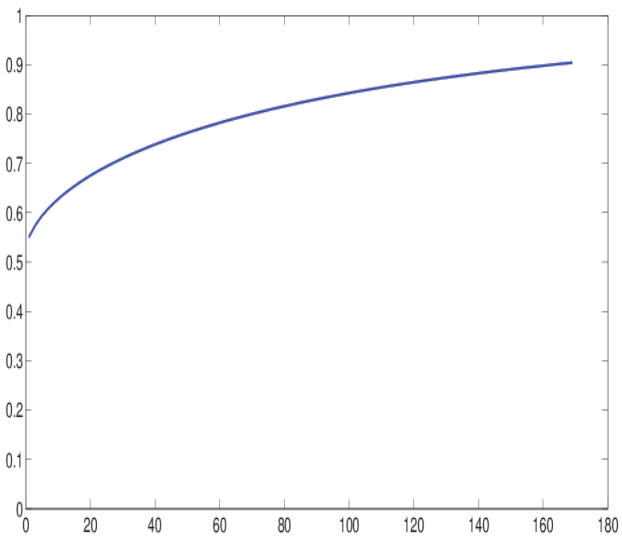
Condorcet's rule

If the individual votes are independent, the answer is

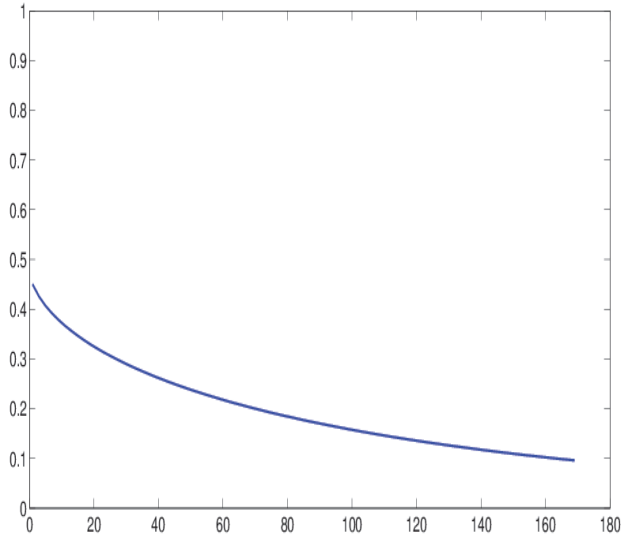
$$\Pr\{\text{ majority makes correct decision }\} = \sum_{j=\frac{m+1}{2}}^{m} \frac{m!}{j!(m-j)!} p^j (1-p)^{m-j}$$

This formula is known as **Condorcet's jury theorem**.

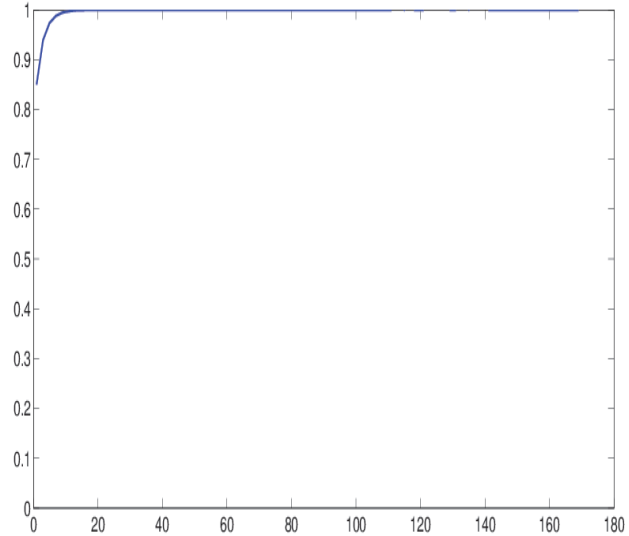
Probability as function of the number of votes



$p = 0.55$



$p = 0.45$



$p = 0.85$

Terminology

- An **ensemble method** makes a prediction by combining the predictions of many classifiers into a single vote.
- The individual classifiers are usually required to perform only slightly better than random. For two classes, this means slightly more than 50% of the data are classified correctly. Such a classifier is called a **weak learner**.

Strategy

- We have seen above that if the weak learners are random and independent, the prediction accuracy of the majority vote will increase with the number of weak learners.
- Since the weak learners all have to be trained on the training data, producing random, independent weak learners is difficult.
- Different ensemble methods (e.g. Boosting, Bagging, etc) use different strategies to train and combine weak learners that behave relatively independently.

Methods We Will Discuss

Boosting

- After training each weak learner, data is modified using weights.
- Deterministic algorithm.

Bagging

Each weak learner is trained on a random subset of the data.

Random forests

- Bagging with tree classifiers as weak learners.
- Uses an additional step to remove dimensions in $\mathbb{R}^d$ that carry little information.

Boosting

- Arguably the most popular (and historically the first) ensemble method.
- Weak learners can be trees (decision stumps are popular), Perceptrons, etc.
- Requirement: It must be possible to train the weak learner on a *weighted* training set.

Overview

- Boosting adds weak learners one at a time.
- A weight value is assigned to each training point.
- At each step, data points which are currently classified correctly are weighted down (i.e. the weight is smaller the more of the weak learners already trained classify the point correctly).
- The next weak learner is trained on the *weighted* data set: In the training step, the error contributions of misclassified points are multiplied by the weights of the points.
- Roughly speaking, each weak learner tries to get those points right which are currently not classified correctly.

Example: Decision stump

A decision stump classifier for two classes is defined by

$$f(\mathbf{x} \,|\, j, t) := \begin{cases} +1 & x^{(j)} > t \\ -1 & \text{otherwise} \end{cases}$$

where $j \in \{1, \ldots, d\}$ indexes an axis in $\mathbb{R}^d$.

Weighted data

- Training data $(\tilde{\mathbf{x}}_1, \tilde{y}_1), \ldots, (\tilde{\mathbf{x}}_n, \tilde{y}_n)$.
- With each data point $\tilde{\mathbf{x}}_i$ we associate a weight $w_i \geq 0$.

Training on weighted data

Minimize the *weighted* misclassifcation error:

$$(j^*, t^*) := \arg\min_{j,t} \frac{\sum_{i=1}^n w_i \mathbb{I}\{\tilde{y}_i \neq f(\tilde{\mathbf{x}}_i | j, t)\}}{\sum_{i=1}^n w_i}$$

Input

- Training data $(\tilde{\mathbf{x}}_1, \tilde{y}_1), \ldots, (\tilde{\mathbf{x}}_n, \tilde{y}_n)$
- Algorithm parameter: Number M of weak learners

Training algorithm

1. Initialize the observation weights $w_i = \frac{1}{n}$ for $i = 1, 2, ..., n$.
2. For $m = 1$ to M:
   - 2.1 Fit a classifier $g_m(x)$ to the training data using weights w_i.
   - 2.2 Compute
   $$\text{err}_m := \frac{\sum_{i=1}^{n} w_i \mathbb{I}\{y_i \neq g_m(x_i)\}}{\sum_i w_i}$$
   - 2.3 Compute $\alpha_m = \log(\frac{1-\text{err}_m}{\text{err}_m})$
   - 2.4 Set $w_i \leftarrow w_i \cdot \exp(\alpha_m \cdot \mathbb{I}(y_i \neq g_m(x_i)))$ for $i = 1, 2, ..., n$.
3. Output
$$f(x) := \text{sign}\left(\sum_{m=1}^{M} \alpha_m g_m(x)\right)$$

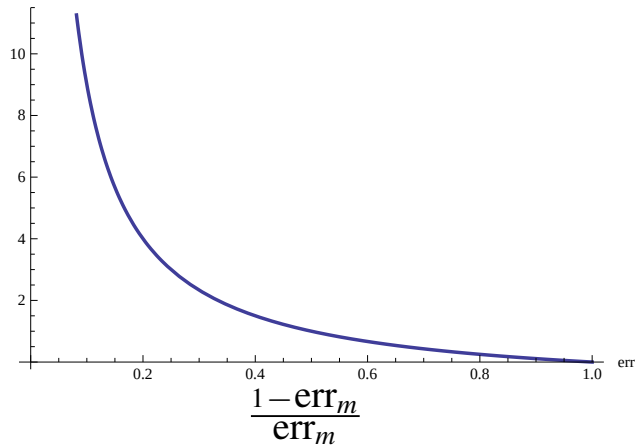
Weight updates

$$\alpha_m = \log\left(\frac{1-\text{err}_m}{\text{err}_m}\right)$$

$$w_i^{(\text{m})} = w_i^{(\text{m-1})} \cdot \exp(\alpha_m \cdot \mathbb{I}(y_i \neq g_m(x_i)))$$

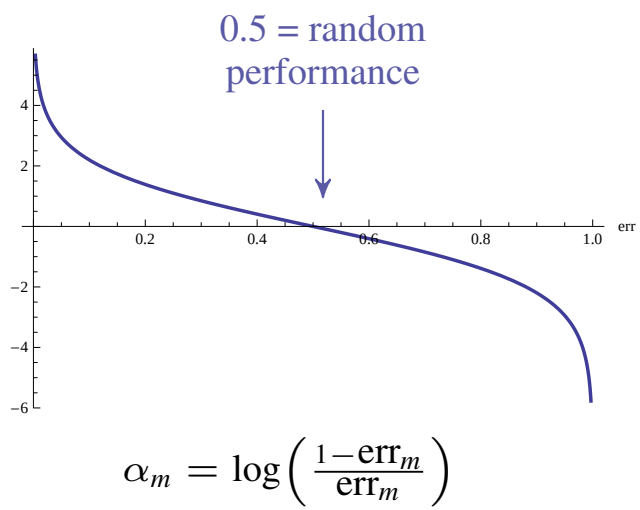
Hence:

$$w_i^{(\text{m})} = \begin{cases} w_i^{(\text{m-1})} & \text{if } g_m \text{ classifies } x_i \text{ correctly} \\ w_i^{(\text{m-1})} \cdot \frac{1-\text{err}_m}{\text{err}_m} & \text{if } g_m \text{ misclassifies } x_i \end{cases}$$

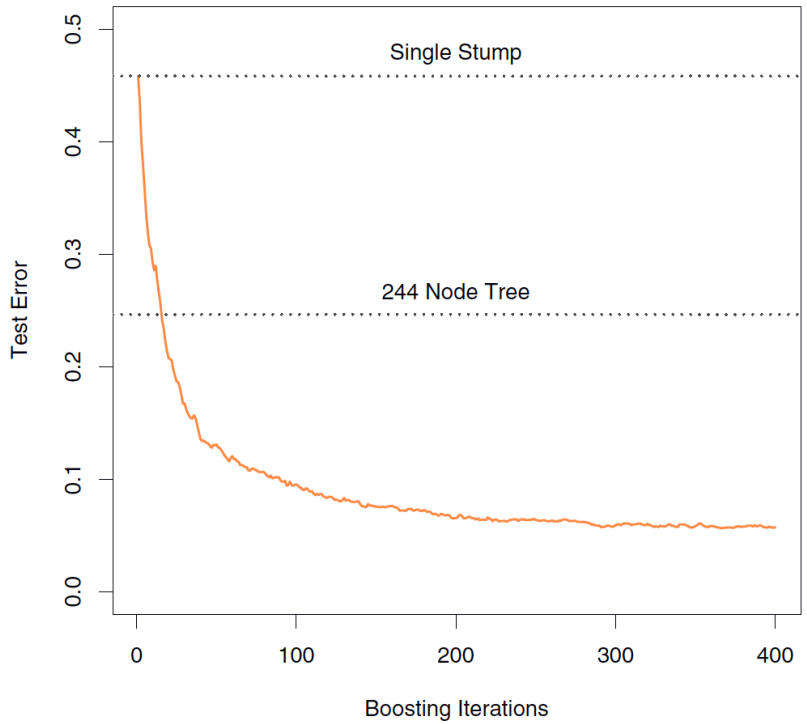


Weighted classifier

$$f(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m g_m(x)\right)$$



AdaBoost test error (simulated data)



- Weak learners used are decision stumps.
- Combining many trees of depth 1 yields much better results than a single large tree.

Properties

- AdaBoost is one of most widely used classifiers in applications.
- Decision boundary is non-linear.
- Can handle multiple classes if weak learner can do so.

Test vs training error

- Most training algorithms (e.g. Perceptron) terminate when training error reaches minimum.
- AdaBoost weights keep changing even if training error is minimal.
- Interestingly, the *test error* typically keeps decreasing even *after* training error has stabilized at minimal value.
- It can be shown that this behavior can be interpreted in terms of a margin:
  - Adding additional classifiers slowly pushes overall f towards a maximum-margin solution.
  - May not improve training error, but improves generalization properties.
- This does *not* imply that boosting magically outperforms SVMs, only that minimal test error does not imply an optimal solution.

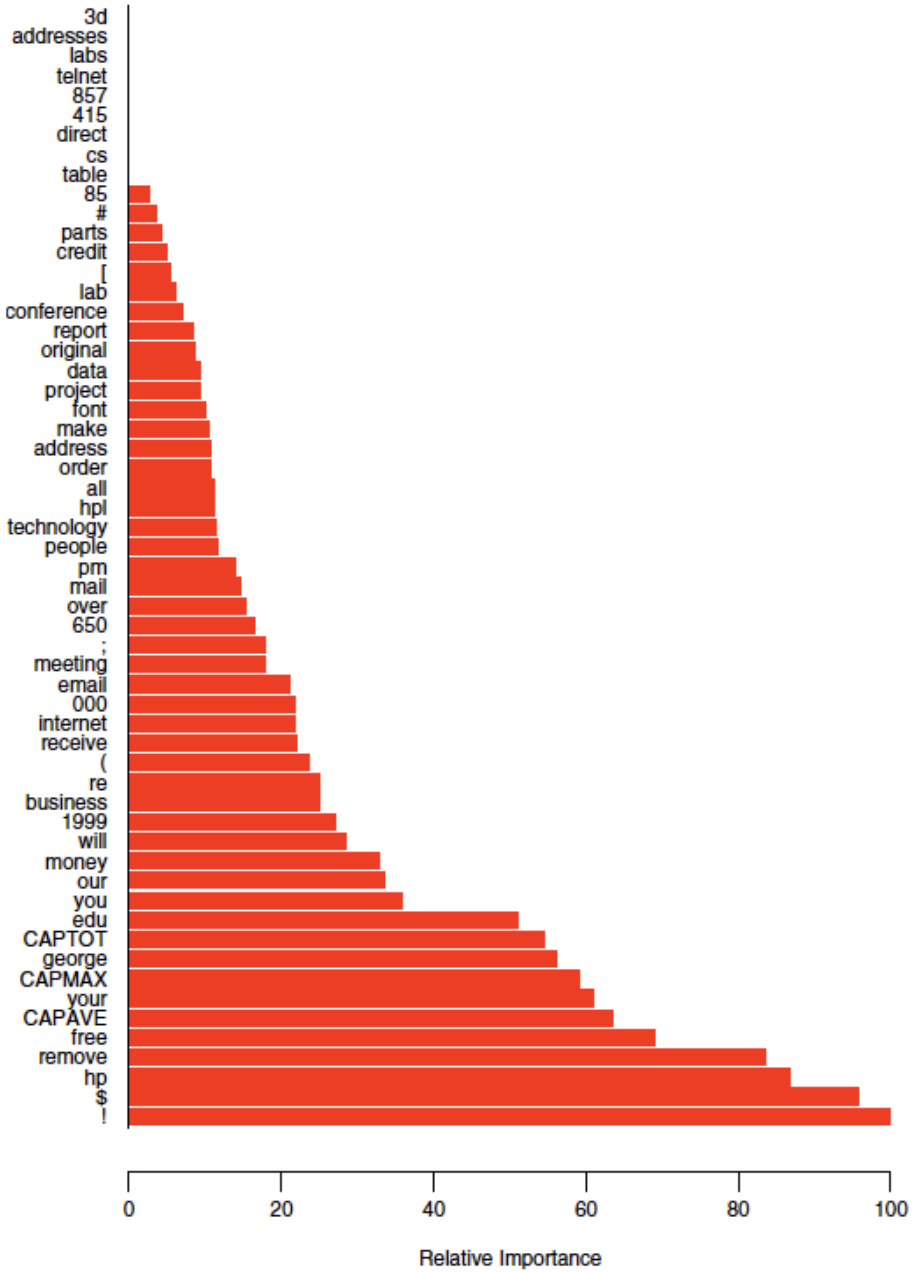
AdaBoost with Decision Stumps

- Once AdaBoost has trained a classifier, the weights α_m tell us which of the weak learners are important (i.e. classify large subsets of the data well).
- If we use Decision Stumps as weak learners, each f_m corresponds to one axis.
- From the weights α, we can read off which axis are important to separate the classes.

Terminology

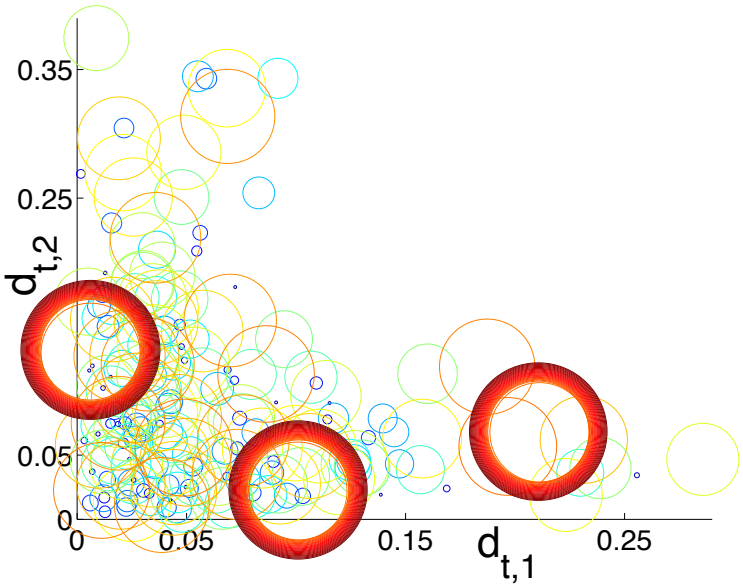
The dimensions of $\mathbb{R}^d$ (= the measurements) are often called the **features** of the data. The process of selecting features which contain important information for the problem is called **feature selection**. Thus, AdaBoost with Decision Stumps can be used to perform feature selection.

SPAM DATA



- Tree classifier: 9.3% overall error rate
- Boosting with decision stumps: 4.5%
- Figure shows feature selection results of Boosting.

Cycles



- An odd property of AdaBoost is that it can go into a cycle, i.e. the same sequence of weight configurations occurs over and over.
- The figure shows weights (called d_t by the authors of the paper, with t=iteration number) for two weak learners.
- Circle size indicates iteration number, i.e. larger circel indicates larger t.

Application: Face Detection

Searching for faces in images

Two problems:

- **Face detection** Find locations of all faces in image. Two classes.
- **Face recognition** Identify a person depicted in an image by recognizing the face. One class per person to be identified + background class (all other people).

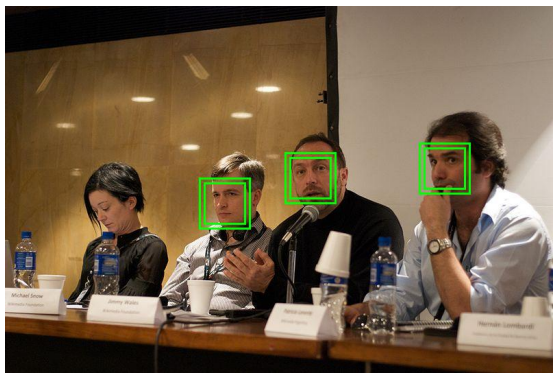
Face detection can be regarded as a solved problem. Face recognition is not solved.

Face detection as a classification problem

- Divide image into patches.
- Classify each patch as "face" or "not face"

Unbalanced Classes

- Our assumption so far was that both classes are roughly of the same size.
- Some problems: One class is much larger.
- Example: Face detection.
  - Image subdivided into small quadratic patches.
  - Even in pictures with several people, only small fraction of patches usually represent faces.



Standard classifier training

Suppose positive class is very small.

- Training algorithm can achieve good error rate by classifiying *all* data as negative.
- The error rate will be precisely the proportion of points in positive class.

Addressing class imbalance

- We have to change cost function: False negatives (= classify face as background) expensive.
- Consequence: Training algorithm will focus on keeping proportion of false negatives small.
- Problem: Will result in many false positives (= background classified as face).

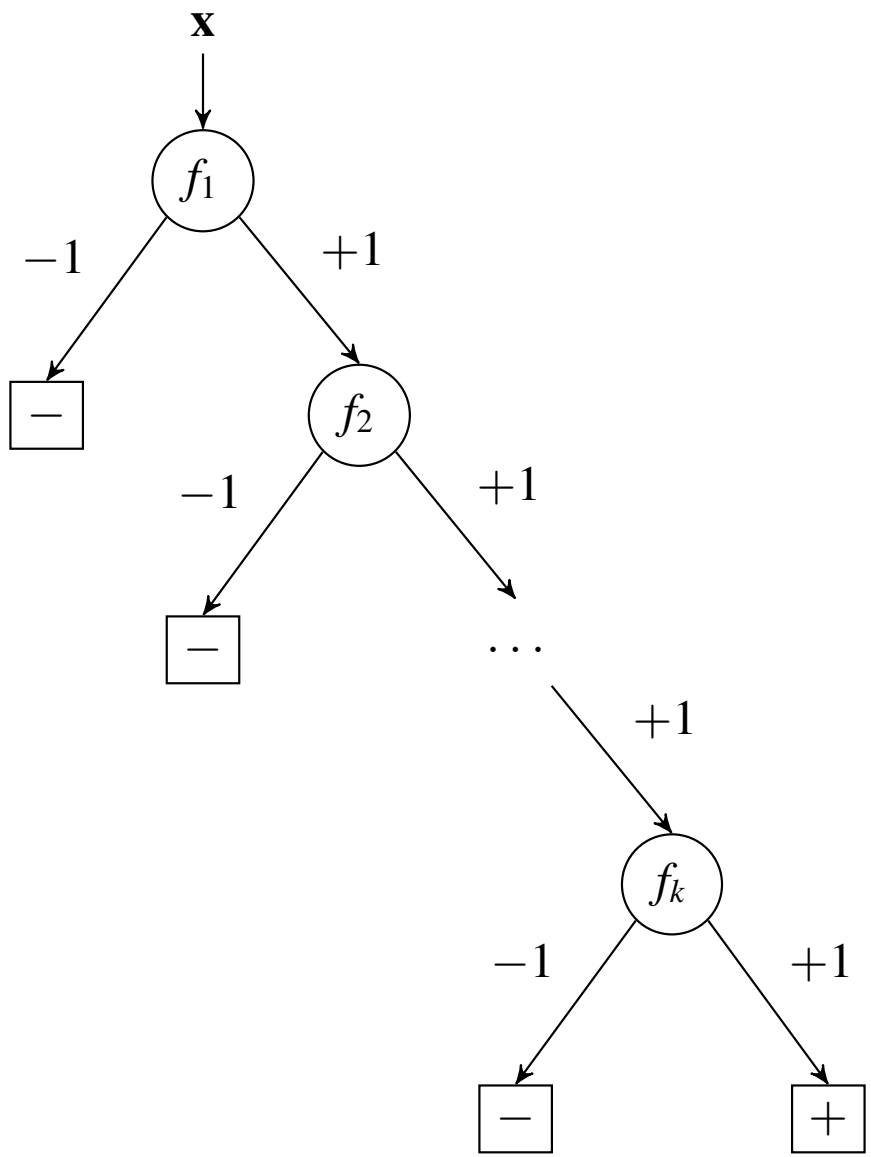
Cascade approach

- Use many classifiers linked in a chain structure ("cascade").
- Each classifier eliminates part of the negative class.
- With each step down the cascade, class sizes become more even.

Training a cascade

Use imbalanced loss (very low false negative rate for each f_j).

1. Train classifier f_1 on entire training data set.
2. Remove all $\tilde{\mathbf{x}}_i$ in negative class which f_1 classifies correctly from training set.
3. On smaller training set, train f_2.
4. ...
5. On remaining data at final stage, train f_k.



Classifying with a cascade

- If any f_j classifies $\mathbf{x}$ as negative, $f(\mathbf{x}) = -1$.
- Only if all f_j classify $\mathbf{x}$ as positive, $f(\mathbf{x}) = +1$.

We have to consider two rates

$$\text{false positive rate} \qquad \mathrm{FPR}(f_j) = \frac{\#\text{negative points classified as "+1"}}{\#\text{negative training points at stage } j}$$

$$\text{detection rate} \qquad \mathrm{DR}(f_j) = \frac{\#\text{correctly classified positive points}}{\#\text{positive training points at stage } j}$$

We want to achieve a low value of $\mathrm{FPR}(f)$ and a high value of $\mathrm{DR}(f)$.

Class imbalance

In face detection example:

- Number of faces classified as background is (size of face class) $\times\ (1 - \mathrm{DR}(f))$
- We would like to see a decently high detection rate, say 90%
- Number of background patches classified as faces is
  (size of background class) $\times\ (\mathrm{FPR}(f))$
- Since background class is huge, $\mathrm{FPR}(f)$ has to be *very* small to yield roughly the same amount of errors in both classes.

Cascade detection rate

The rates of the overall cascade classifier f are

$$\mathrm{FPR}(f) = \prod_{j=1}^{k} \mathrm{FPR}(f_j) \qquad \mathrm{DR}(f) = \prod_{j=1}^{k} \mathrm{DR}(f_j)$$

- Suppose we use a 10-stage cascade ($k = 10$)
- Each $\mathrm{DR}(f_j)$ is 99% and we permit $\mathrm{FPR}(f_j)$ of 30%.
- We obtain $\mathrm{DR}(f) = 0.99^{10} \approx 0.90$ and $\mathrm{FPR}(f) = 0.3^{10} \approx 6 \times 10^{-6}$

Objectives

- Classification step should be computationally efficient.
- Expensive training affordable.

Strategy

- Extract very large set of measurements (features), i.e. d in $\mathbb{R}^d$ large.
- Use Boosting with decision stumps.
- From Boosting weights, select small number of important features.
- Class imbalance: Use Cascade.

Classification step

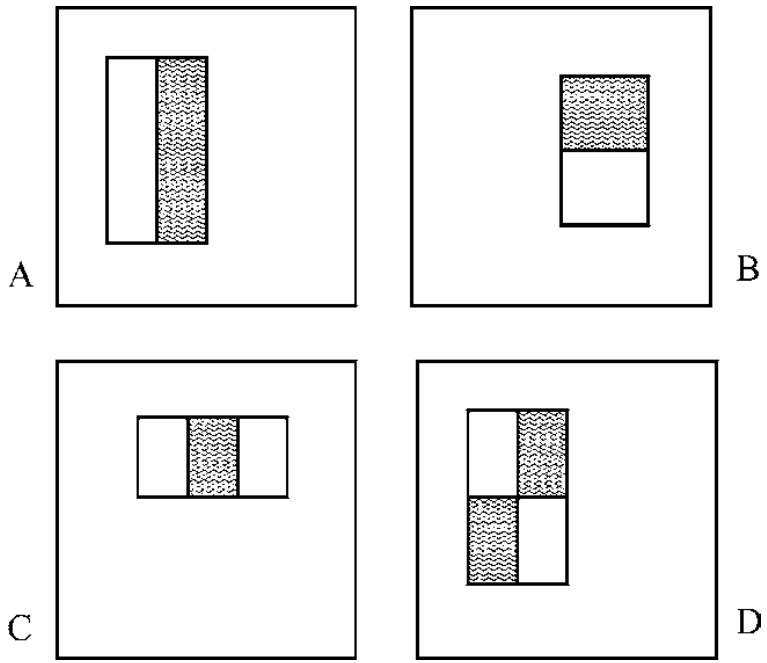
Compute only the selected features from input image.

Extraction method

1. Enumerate possible windows (different shapes and locations) by $j = 1, \ldots, d$.
2. For training image i and each window j, compute

   x_{ij} := average of pixel values in gray block(s)
   − average of pixel values in white block(s)

3. Collect values for all j in a vector $\mathbf{x}_i := (x_{i1}, \ldots, x_{id}) \in \mathbb{R}^d$.

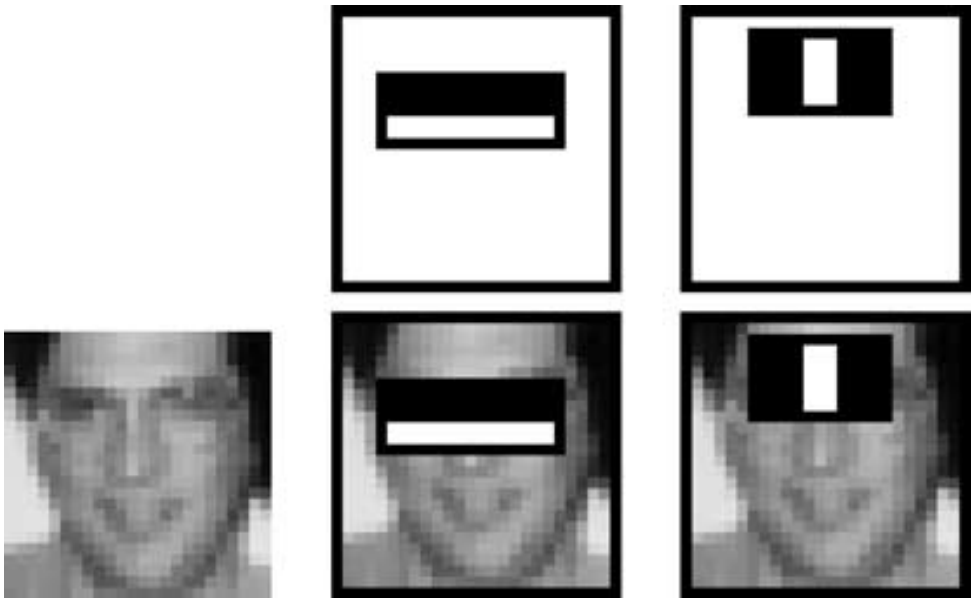


The dimension is huge

- One entry for (almost) every possible location of a rectangle in image.
- Start with small rectangles and increase edge length repeatedly by 1.5.
- In Viola-Jones paper: Images are 384×288 pixels, $d \approx 160000$.

SELECTED FEATURES

First two selected features



200 features are selected in total.

Training procedure

1. User selects acceptable rates (FPR and DR) for each level of cascade.
2. At each level of cascade:
   - Train boosting classifier.
   - Gradually increase number of selected features until rates achieved.

Use of training data

Each training step uses:

- All positive examples (= faces).
- Negative examples (= non-faces) misclassified at previous cascade layer.

Example Results

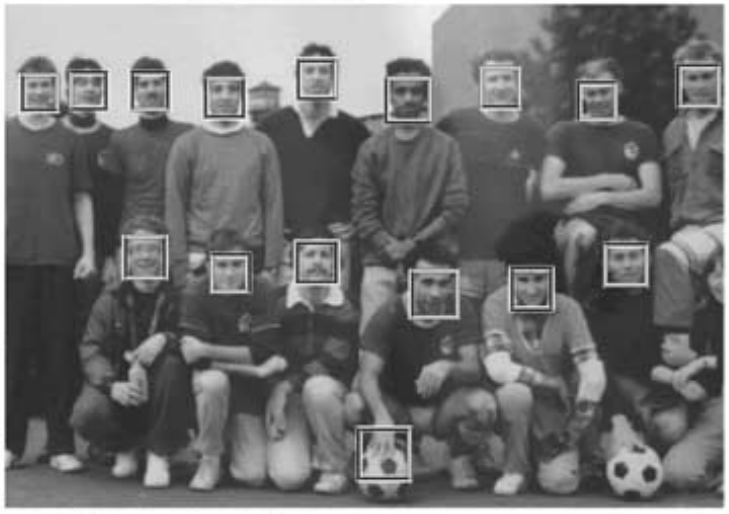





Table 3. Detection rates for various numbers of false positives on the MIT + CMU test set containing 130 images and 507 faces.

	False detections							
Detector	10	31	50	65	78	95	167	422
Viola-Jones	76.1%	88.4%	91.4%	92.0%	92.1%	92.9%	93.9%	94.1%
Viola-Jones (voting)	81.1%	89.7%	92.1%	93.1%	93.1%	93.2%	93.7%	–
Rowley-Baluja-Kanade	83.2%	86.0%	–	–	–	89.2%	90.1%	89.9%
Schneiderman-Kanade	–	–	–	94.4%	–	–	–	–
Roth-Yang-Ahuja	–	–	–	–	(94.8%)	–	–	–

Basis function interpretation

The boosting classifier is of the form

$$f(\mathbf{x}) = \operatorname{sgn}(F(\mathbf{x})) \qquad \text{where} \qquad F(\mathbf{x}) := \sum_{m=1}^{M} \alpha_m g_m(\mathbf{x}) \ .$$

- A linear combination of functions $g_1, \ldots, g_m$ can be interpreted as a representation of F using the **basis functions** $g_1, \ldots, g_m$.
- We can interpret the linear combination $F(\mathbf{x})$ as an approximation of the decision boundary using a basis of weak classifiers.
- To understand the approximation, we have to understand the coefficients α_m.

Boosting as a stage-wise minimization procedure

It can be shown that α_m is obtained by minimizing a risk,

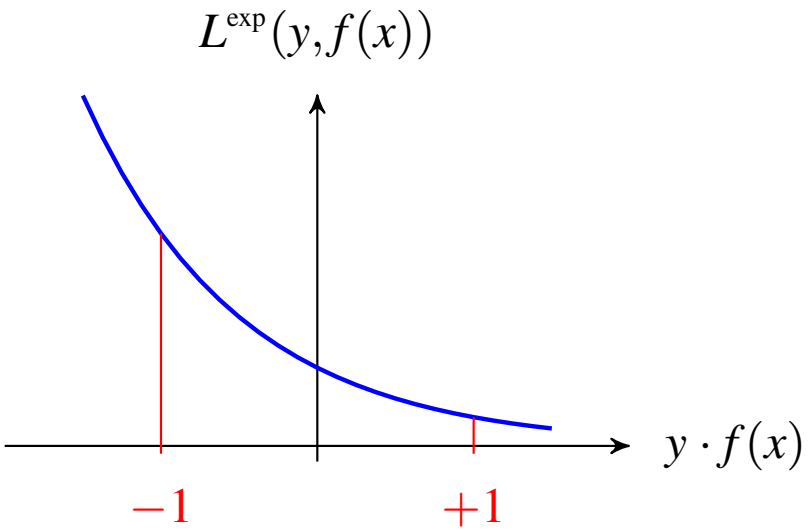
$$(\alpha_m, g_m) := \arg \min_{\alpha'_m, g'_m} \hat{R}_n(F^{(\text{m-1})} + \alpha'_m g'_m)$$

under a specific loss function, the **exponential loss**. Notation: $F^{(\text{m})} := \sum_{j \leq m} \alpha_m g_m$.

Exponential Loss

Definition

$$L^{\exp}(y, f(x)) := \exp(-y \cdot f(x))$$



Relation to indicator function

$$y \cdot f(x) = \begin{cases} +1 & x \text{ correctly classified} \\ -1 & x \text{ misclassified} \end{cases}$$

This is related to the indicator function we have used so far by

$$-y \cdot f(x) = 2 \cdot \mathbb{I}\{f(x) \neq y\} - 1$$

Exponential loss risk of additive classifier

Our claim is that AdaBoost minimizes the empirical risk under $L^{\exp}$,

$$\hat{R}_n(F^{(\text{m-1})} + \beta_m g_m) = \frac{1}{n} \sum_{i=1}^{n} \exp(\overbrace{-y_i F^{(\text{m-1})}}^{\text{fixed in } m\text{th step}} \underbrace{- y_i \beta_m g_m(\mathbf{x}_i)}_{\text{we only have to minimize here}})$$

Relation to AdaBoost

It can be shown that the classifier obtained by solving

$$\arg \min_{\beta_m, g_m} \hat{R}_n(F^{(\text{m-1})} + \beta_m g_m)$$

at each step m yields the AdaBoost classifier.

More precisely, it can be shown:

If we build a classifier $F(\mathbf{x}) := \sum_{m=1}^{M} \beta_m g_m(\mathbf{x})$ which minimizes

$$\hat{R}_n(F^{(\text{m-1})}(\mathbf{x}) + \beta_m g_m(\mathbf{x}))$$

at each step m, we have to choose:

- g_m as the classifier which minimizes the weighted misclassifiation rate.
- $\beta_m = \frac{1}{2} \log \frac{1-\text{err}_m}{\text{err}_m} = \frac{1}{2}\alpha_m$
- $w_i^{(\text{m+1})} := w_i^{(\text{m})} \exp(-y_i \beta_m g_m(\mathbf{x}_i))$

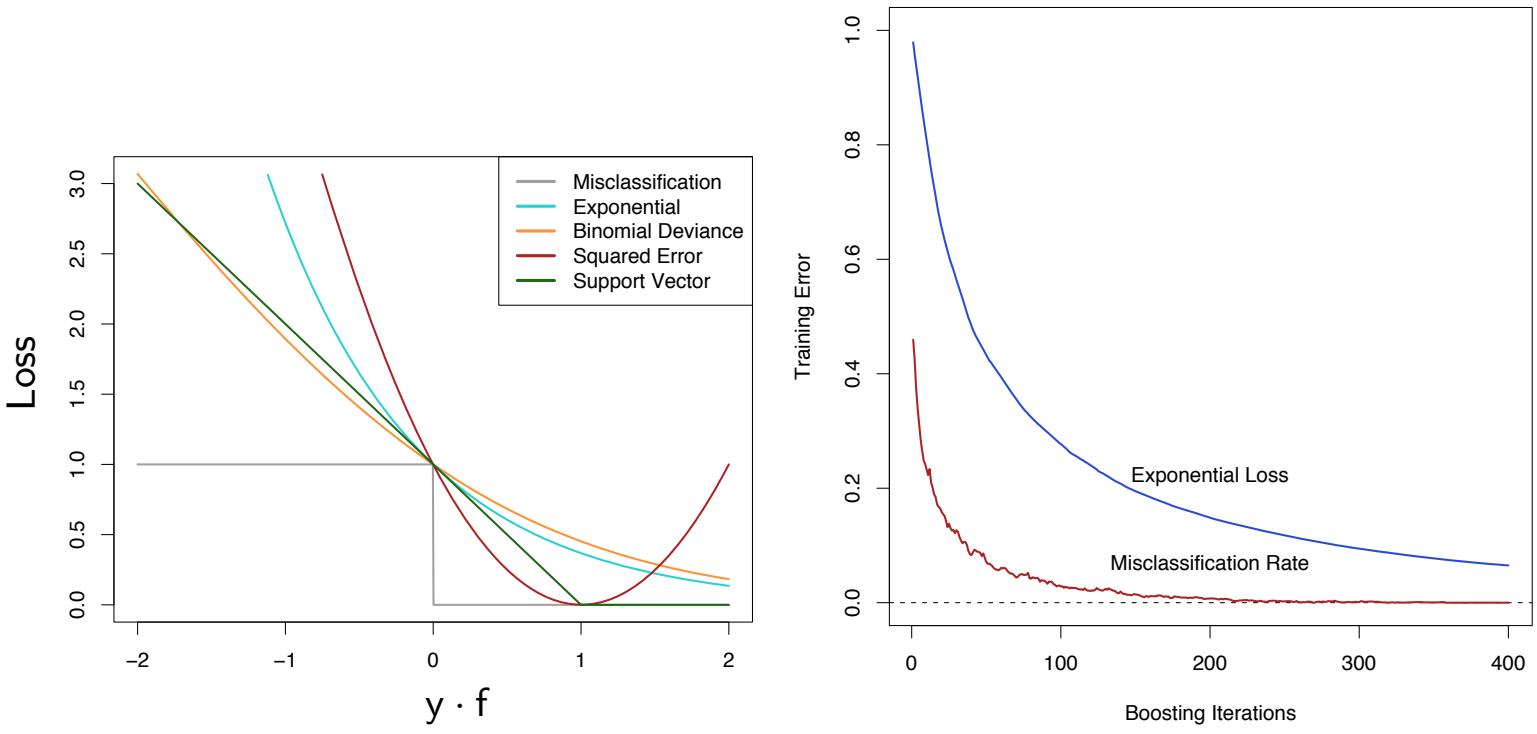
This is precisely equivalent to what AdaBoost does.

In other words

AdaBoost approximates the Bayes-optimal classifier (under exponential loss) using a basis of weak classifiers.

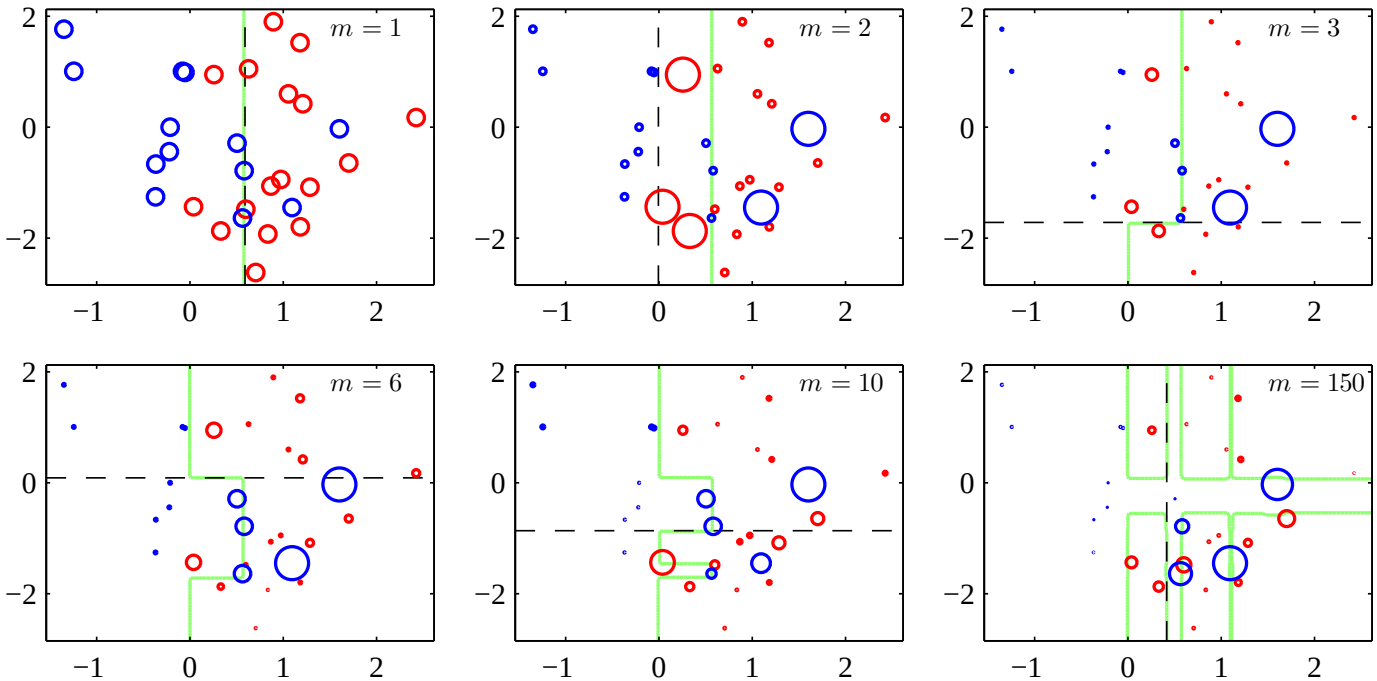
- Since we do not know the true risk, we approximate by the empirical risk.
- Each weak learner optimizes 0-1 loss on *weighted* data.
- Weights are chosen so that procedure overall optimizes *exponential* loss risk.

Loss functions



- The right figure shows the misclassification rate and the average exponential loss on the same data as number of weak learners increases.
- From the additive model perspective, the exponential loss helps explain why prediction error continues to improve when training error is already optimal.

ILLUSTRATION



Circle = data points, circle size = weight.

Dashed line: Current weak learner. Green line: Aggregate decision boundary.

Bagging and Random Forests

We briefly review a technique called bootstrap on which Bagging and random forests are based.

Bootstrap

Bootstrap (or **resampling**) is a technique for improving the quality of estimators.

Resampling = sampling from the empirical distribution

Application to ensemble methods

- We will use resampling to generate weak learners for classification.
- We discuss two classifiers which use resampling: Bagging and random forests.
- Before we do so, we consider the traditional application of Bootstrap, namely improving estimators.

Bootstrap: Basic algorithm

Given

- A sample $\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_n$.
- An estimator $\hat{S}$ for a statistic S.

Bootstrap algorithm

1. Generate B **bootstrap samples** $\mathcal{B}_1, \ldots, \mathcal{B}_B$. Each bootstrap sample is obtained by sampling n times with replacement from the sample data. (Note: Data points can appear multiple times in any $\mathcal{B}_b$.)
2. Evaluate the estimator on each bootstrap sample:

$$\hat{S}_b := \hat{S}(\mathcal{B}_b)$$

   (That is: We estimate S pretending that $\mathcal{B}_b$ is the data.)
3. Compute the **bootstrap estimate** of S by averaging over all bootstrap samples:

$$\hat{S}_{\text{BS}} := \frac{1}{B} \sum_{b=1}^{B} \hat{S}_b$$

Example: Variance Estimation

Mean and Variance

$$\mu := \int_{\mathbb{R}^d} \mathbf{x}\, p(\mathbf{x}) d\mathbf{x} \qquad \sigma^2 := \int_{\mathbb{R}^d} (\mathbf{x} - \mu)^2 p(\mathbf{x}) d\mathbf{x}$$

Plug-in estimators for mean and variance

$$\hat{\mu} := \frac{1}{n} \sum_{i=1}^{n} \tilde{\mathbf{x}}_i \qquad \hat{\sigma}^2 := \frac{1}{n} \sum_{i=1}^{n} (\tilde{\mathbf{x}}_i - \hat{\mu})^2$$

Bootstrap algorithm

1. For $b = 1, \ldots, B$, generate a boostrap sample $\mathcal{B}_b$. In detail:
   For $i = 1, \ldots, n$:
   - Sample an index $j \in \{1, \ldots, n\}$.
   - Set $\tilde{\mathbf{x}}_i^{(b)} := \tilde{\mathbf{x}}_j$ and add it to $\mathcal{B}_b$.
2. For each b, compute mean and variance estimates:

$$\hat{\mu}_b := \frac{1}{n} \sum_{i=1}^{n} \tilde{\mathbf{x}}_i^{(b)} \qquad \hat{\sigma}_b^2 := \frac{1}{n} \sum_{i=1}^{n} (\tilde{\mathbf{x}}_i^{(b)} - \hat{\mu}_b)^2$$

3. Compute the bootstrap estimate:

$$\hat{\sigma}_{\mathrm{BS}}^2 := \frac{1}{B} \sum_{b=1}^{B} \hat{\sigma}_b^2$$

How Often Do We See Each Sample?

Sample $\{\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_n\}$, bootstrap resamples $\mathcal{B}_1, \ldots, \mathcal{B}_B$.

In how many sets does a given $\mathbf{x}_i$ occur?

Probability for $\mathbf{x}_i$ *not* to occur in n draws:

$$\Pr\{\tilde{\mathbf{x}}_i \notin \mathcal{B}_b\} = (1 - \frac{1}{n})^n$$

For large n:

$$\lim_{n\to\infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e} \approx 0.3679$$

- Asymptotically, any $\tilde{\mathbf{x}}_i$ will appear in $\sim 63\%$ of the bootstrap resamples.
- Multiple occurrences possible.

How often is $\tilde{\mathbf{x}}_i$ expected to occur?

The *expected* number of occurences of each $\tilde{\mathbf{x}}_i$ is B.

Bootstrap estimate averages over reshuffled samples.

Estimate variance of estimators

- Since estimator $\hat{S}$ depends on (random) data, it is a random variable.
- The more this variable scatters, the less we can trust our estimate.
- If scatter is high, we can expect the values $\hat{S}_b$ to scatter as well.
- In previous example, this means: Estimating the variance of the variance estimator.

Variance reduction

- Averaging over the individual bootstrap samples can reduce the variance in $\hat{S}$.
- In other words: $\hat{S}_{\text{BS}}$ typically has lower variance than $\hat{S}$.
- This is the property we will use for classicifation in the following.

As alternative to cross validation

To estimate prediction error of classifier:

- For each b, train on $\mathcal{B}_b$, estimate risk on points not in $\mathcal{B}_b$.
- Average risk estimates over bootstrap samples.

Idea

- Recall Boosting: Weak learners are deterministic, but selected to exhibit high variance.
- Strategy now: Randomly distort data set by resampling.
- Train weak learners on resampled training sets.
- Resulting algorithm: **Bagging** (= **B**ootstrap **agg**regation)

REPRESENTATION OF CLASS LABELS

For Bagging with K classes, we represent class labels as vectors:

$$\mathbf{x}_i \text{ in class } k \qquad \text{as} \qquad y_i = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \longleftarrow k\text{th entry}$$

This way, we can average together multiple class labels:

$$\frac{1}{n}(y_1 + \ldots + y_n) = \begin{pmatrix} p_1 \\ \vdots \\ p_k \\ \vdots \\ p_K \end{pmatrix}$$

We can interpret p_k as the probability that one of the n points is in class k.

Bagging: Algorithm

Training

For $b = 1, \ldots, B$:

1. Draw a bootstrap sample $\mathcal{B}_b$ of size n from training data.
2. Train a classifier f_b on $\mathcal{B}_b$.

Classification

- Compute

$$f_{\text{avg}}(\mathbf{x}) := \frac{1}{B} \sum_{b=1}^{B} f_b(\mathbf{x})$$

  This is a vector of the form $f_{\text{avg}}(\mathbf{x}) = (p_1(\mathbf{x}), \ldots, p_k(\mathbf{x}))$.

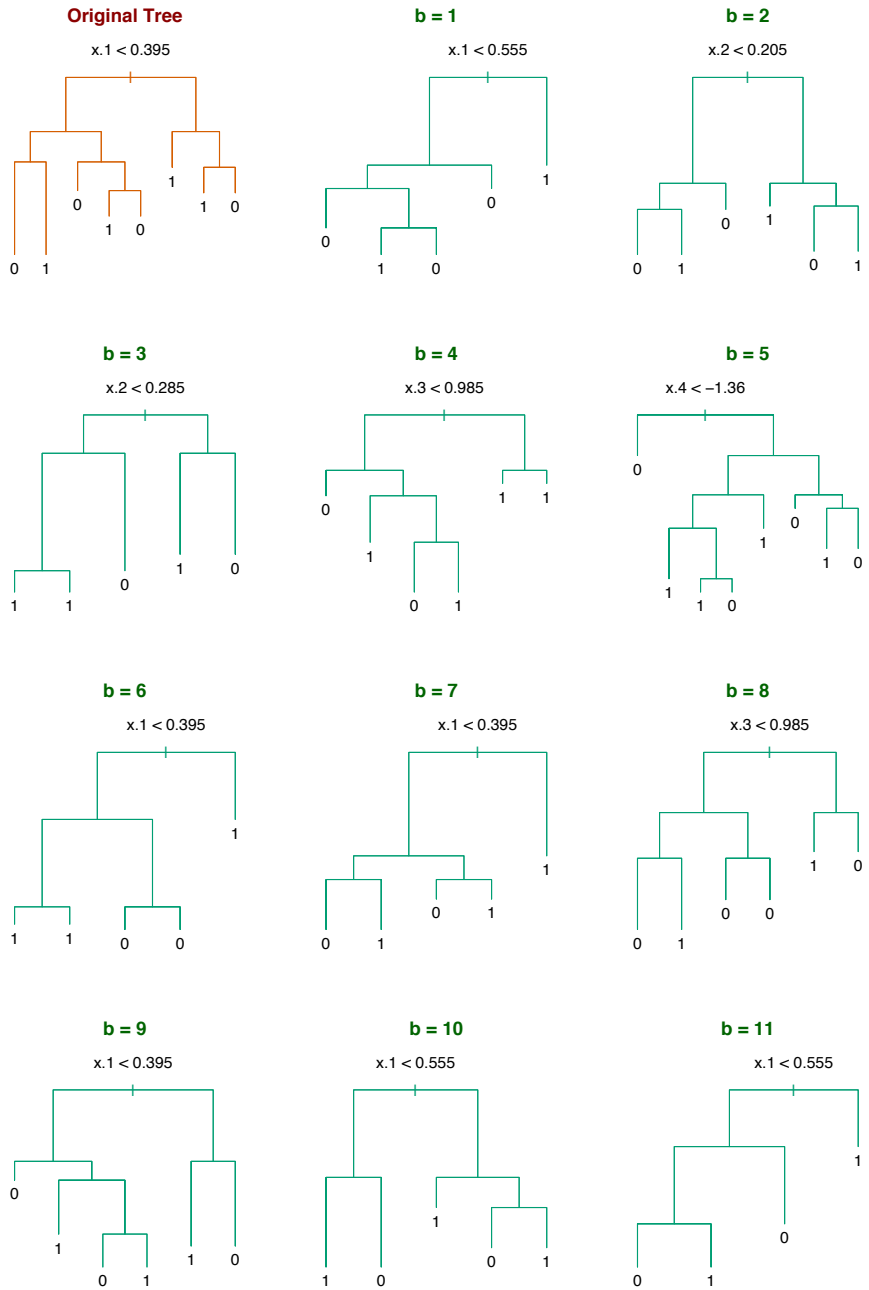
- The Bagging classifier is given by

$$f_{\text{Bagging}}(\mathbf{x}) := \arg\max_k \{p_1(\mathbf{x}), \ldots, p_k(\mathbf{x})\} \ ,$$

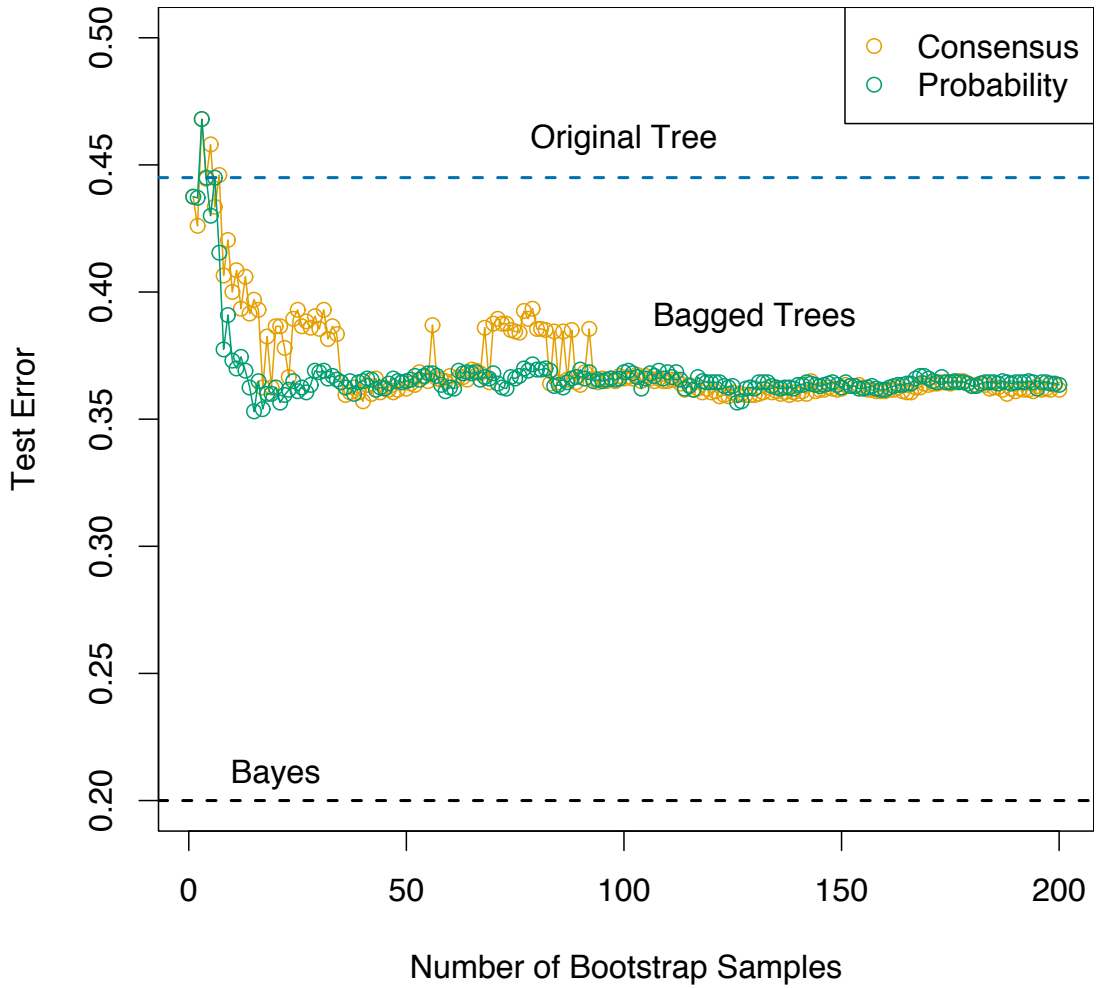
  i.e. we predict the class label which most weak learners have voted for.

Example: Bagging Trees

- Two classes, each with Gaussian distribution in $\mathbb{R}^5$.
- Note the variance between bootstrapped trees.



EXAMPLE: BAGGING TREES



- "Original tree" = single tree trained on original data.
- The orange dots correspond to the bagging classifier.

Bagging vs. Boosting

- Bagging works particularly well for trees, since trees have high variance.
- Boosting typically outperforms bagging with trees.
- The main culprit is usually dependence: Boosting is better at reducing correlation between the trees than bagging is.

Random Forests

Modification of bagging with trees designed to further reduce correlation.

- Tree training optimizes each split over all dimensions.
- Random forests choose a different subset of dimensions *at each split*.
- Optimal split is chosen within the subset.
- The subset is chosen at random out of all dimensions $\{1, \ldots, d\}$.

Training

Input parameter: m (positive integer with $m < d$)

For $b = 1, \ldots, B$:

1. Draw a bootstrap sample $\mathcal{B}_b$ of size n from training data.
2. Train a tree classifier f_b on $\mathcal{B}_b$, where each split is computed as follows:
   - Select m axes in $\mathbb{R}_d$ at random.
   - Find the best split (j^*, t^*) on this subset of dimensions.
   - Split current node along axis j^* at t^*.

Classification

Exactly as for bagging: Classify by majority vote among the B trees. More precisely:

- Compute $f_{\text{avg}}(\mathbf{x}) := (p_1(\mathbf{x}), \ldots, p_k(\mathbf{x})) := \frac{1}{B} \sum_{b=1}^{B} f_b(\mathbf{x})$
- The Random Forest classification rule is

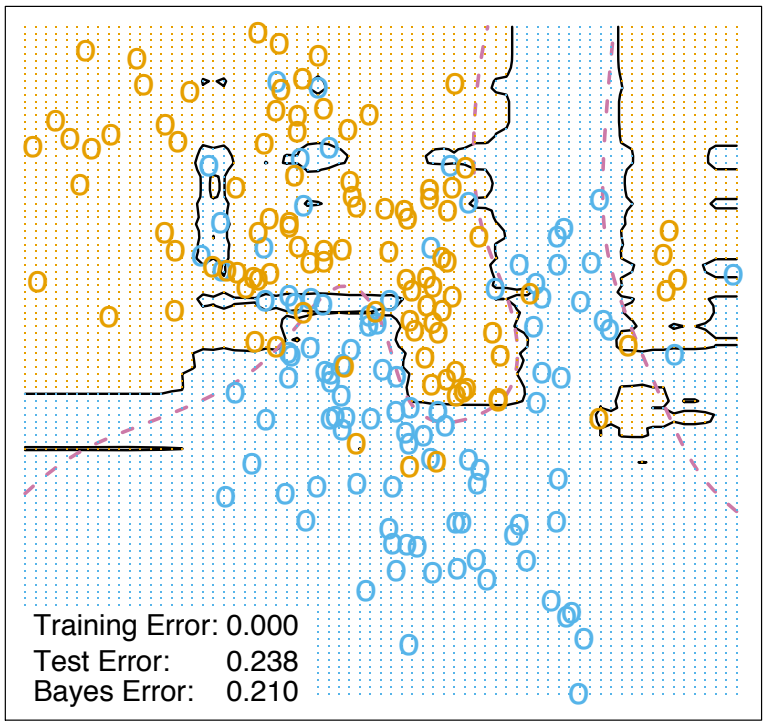
$$f_{\text{Bagging}}(\mathbf{x}) := \arg\max_k \{p_1(\mathbf{x}), \ldots, p_k(\mathbf{x})\}$$

Remarks

- Recommended value for m is $m = \lfloor \sqrt{d} \rfloor$ or smaller.
- RF typically achieve similar results as boosting. Implemented in most packages, often as standard classifier.

Example: Synthetic Data

- This is the RF classification boundary on the synthetic data we have already seen a few times.
- Note the bias towards axis-parallel alignment.



SUMMARY: CLASSIFICATION

SUMMARY

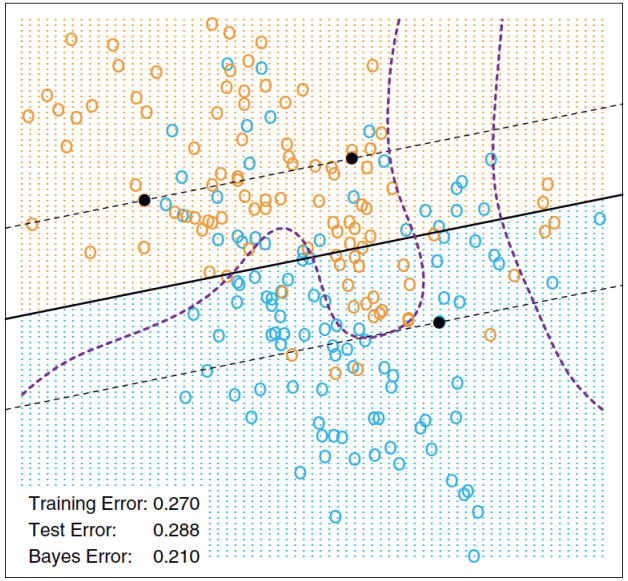
Approaches we have discussed

- Linear classifiers
  - Perceptron, SVM
  - Nonlinear versions using kernels
- Trees (depth 1: linear and axis-parallel, depth ≥ 2: non-linear)
- Ensemble methods

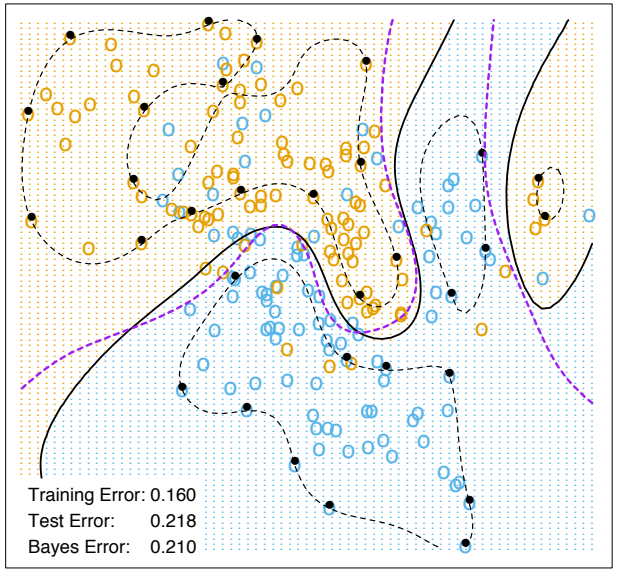
What should we use?

- RBF SVMs, AdaBoost and Random Forests perform well on many problems.
- All have strengths and weaknesses. E.g.:
  - High dimension, limited data: SVM may have the edge.
  - Many dimensions, but we believe only a few are important: AdaBoost with stumps.
- In general: Feature extraction (what do we measure?) is crucial.
- Consider combination of different methods by voting.

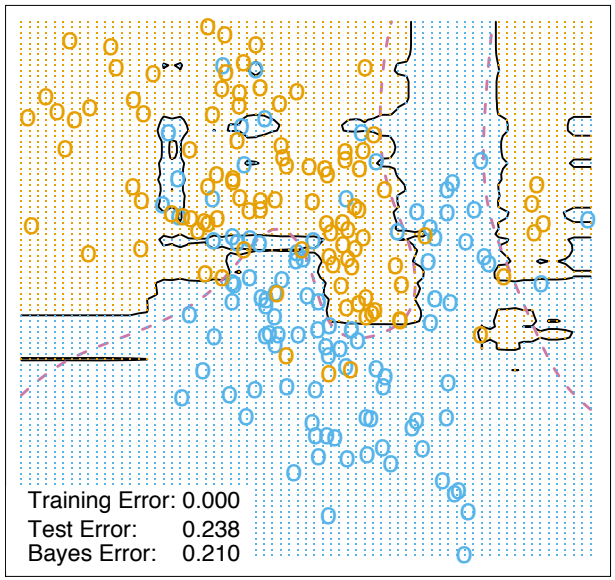
Every Method has its Idiosyncrasies



Linear SVM



RBF SVM



Random forest

Regression

Regression: Problem Definition

Data

- Measurements: $\mathbf{x} \in \mathbb{R}^d$ (also: independent variable, covariate)
- Labels: $y \in \mathbb{R}$ (also: dependent variable, response)

Task

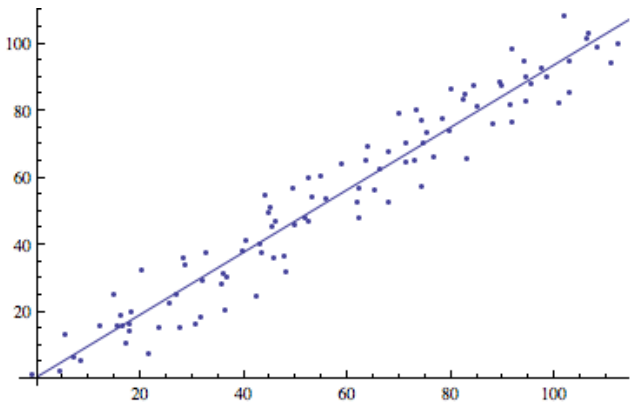
Find a predictor $f : \mathbb{R}^d \to \mathbb{R}$ such that (approximately) $f(x) = y$ for data (x, y). The predictor is called a **regression function**.

Definition: Linear regression

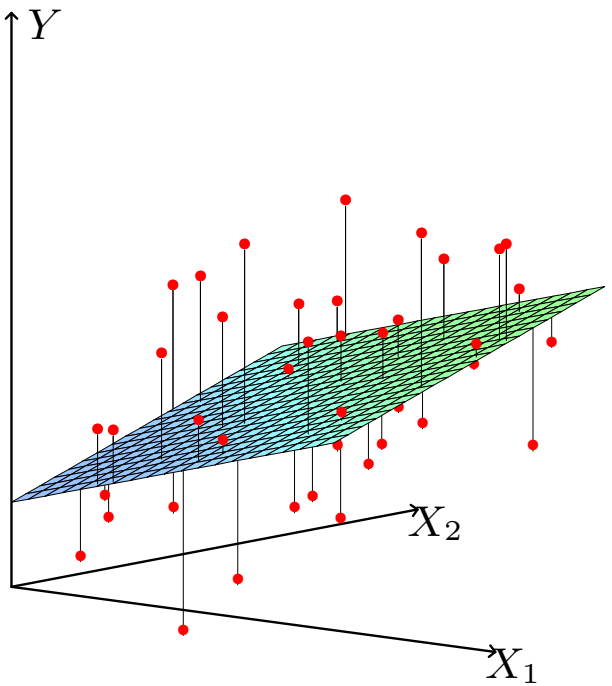
A regression method is called **linear** if the predictor f is a linear function, i.e. a line if $d = 1$ (more generally, an affine hyperplane).

Linear Regression

$$\mathbf{x} \in \mathbb{R}^d \quad \text{and} \quad y \in \mathbb{R}$$



$d = 1$



$d = 2$

Implications of linearity

A linear function $f : \mathbb{R}^d \to \mathbb{R}$ is always of the form

$$f(\mathbf{x}) = \beta_0 + \sum_{j=1}^{d} \beta_j x_j \qquad \text{for } \beta_0, \beta_1, \ldots, \beta_d \in \mathbb{R} \,,$$

where x_j is the jth entry of $\mathbf{x}$. Recall representation of hyperplanes in classification!

Consequence

Finding f boils down to finding $\boldsymbol{\beta} \in \mathbb{R}^{d+1}$.

Relation to classification

- Classification is a regression problem with $\{1, \ldots, K\}$ substituted for $\mathbb{R}$.
- Don't get confused—the role of the hyperplane (for, say, $d = 2$) is different:
  - Regression: Graph of regression function is hyperplane in $\mathbb{R}^{d+1}$.
  - Classification: Regression function is piece-wise constant. The classifier hyperplane lives in $\mathbb{R}^d$ and marks where the regression function jumps.

Squared-error loss

We use the **squared-error loss function**

$$L^{\text{se}}(y, f(x)) := \|y - f(x)\|_2^2 \ .$$

Regression methods that determine f by minimizing L^{se} are called **least-squares regression** methods.

Least-squares linear regression

For training data $(\tilde{\mathbf{x}}_1, \tilde{y}_1), \ldots, (\tilde{\mathbf{x}}_n, \tilde{y}_n)$, we have to find the parameter vector $\boldsymbol{\beta} \in \mathbb{R}^{d+1}$ which solves

$$\hat{\boldsymbol{\beta}} := \arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} L^{\text{se}}(\tilde{y}_i, f(\tilde{\mathbf{x}}_i\,; \boldsymbol{\beta}))$$

where

$$f(\mathbf{x}\,; \boldsymbol{\beta}) = \beta_0 + \sum_{j=1}^{d} \beta_j x_j = \langle \boldsymbol{\beta}, (1, \mathbf{x}) \rangle \ .$$

Data matrix

Since $f(\mathbf{x}\,;\beta) = \langle \beta, (1, \mathbf{x}) \rangle$, we write the data as a matrix:

$$\tilde{\mathbf{X}} := \begin{pmatrix} 1 & (\tilde{\mathbf{x}}_1)_1 & \dots & (\tilde{\mathbf{x}}_1)_j & \dots & (\tilde{\mathbf{x}}_1)_d \\ \vdots & & & \vdots & & \vdots \\ 1 & (\tilde{\mathbf{x}}_i)_1 & \dots & (\tilde{\mathbf{x}}_i)_j & \dots & (\tilde{\mathbf{x}}_i)_d \\ \vdots & & & \vdots & & \vdots \\ 1 & (\tilde{\mathbf{x}}_n)_1 & \dots & (\tilde{\mathbf{x}}_n)_j & \dots & (\tilde{\mathbf{x}}_n)_d \end{pmatrix} \leftarrow \tilde{\mathbf{x}}_i$$

We write $\tilde{\mathbf{X}}_j^{\text{col}}$ for the column vectors with $\tilde{\mathbf{X}}_0^{\text{col}} = (1, \dots, 1)$ and $j = 1, \dots, d$.

$$\tilde{\mathbf{X}}\beta = \begin{pmatrix} f(\tilde{\mathbf{x}}_1\,;\beta) \\ \vdots \\ f(\tilde{\mathbf{x}}_n\,;\beta) \end{pmatrix}$$

Least-squares linear regression: Matrix form

We have to minimize

$$\sum_{i=1}^{n} L^{\text{se}}(\tilde{y}_i, f(\tilde{\mathbf{x}}_i\,;\boldsymbol{\beta})) = \sum_{i=1}^{n} (\tilde{y}_i - f(\tilde{\mathbf{x}}_i\,;\boldsymbol{\beta}))^2 = \|\tilde{\mathbf{y}} - \tilde{\mathbf{X}}\boldsymbol{\beta}\|_2^2$$

The solution to the linear regression problem is now $\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \|\tilde{\mathbf{y}} - \tilde{\mathbf{X}}\boldsymbol{\beta}\|^2$.

Solving the minimization problem

Recall:

- We have to solve for a zero derivative, $\frac{\partial L^{\text{se}}}{\partial \boldsymbol{\beta}}(\hat{\boldsymbol{\beta}}) = 0$.
- That means that $\hat{\boldsymbol{\beta}}$ is an extremum.
- To ensure that the extremum is a minimum, we have to ensure the second derivative $\frac{\partial^2 L^{\text{se}}}{\partial \boldsymbol{\beta}^2}(\hat{\boldsymbol{\beta}})$ is positive. For matrices: Positive definite.

Solution

$$\frac{\partial L^{\text{se}}}{\partial \boldsymbol{\beta}}(\hat{\boldsymbol{\beta}}) = -2\tilde{\mathbf{X}}^t(\tilde{\mathbf{y}} - \tilde{\mathbf{X}}\boldsymbol{\beta})$$

Equating to zero gives the **least-squares solution**:

$$\hat{\boldsymbol{\beta}} = (\tilde{\mathbf{X}}^t\tilde{\mathbf{X}})^{-1}\tilde{\mathbf{X}}^t y$$

(Recall: The transpose $\mathbf{X}^t$ is the matrix with $(\mathbf{X}^t)_{ij} := \mathbf{X}_{ji}$.)

Second derivative

$$\frac{\partial^2 L^{\text{se}}}{\partial \boldsymbol{\beta}^2}(\hat{\boldsymbol{\beta}}) = 2\tilde{\mathbf{X}}^t\tilde{\mathbf{X}}$$

- $\tilde{\mathbf{X}}^t\tilde{\mathbf{X}}$ is always positive semi-definite. If it is also invertible, it is positive definite.
- In other words: If $\tilde{\mathbf{X}}^t\tilde{\mathbf{X}}$ is invertible (which we also need to compute $\hat{\boldsymbol{\beta}}$), then $\hat{\boldsymbol{\beta}}$ is the unique global minimum of the squared-error loss.

Tools: Linear Algebra Basics

Images of Linear Mappings (1)

Linear mapping

A matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$ defines a linear mapping $f_{\mathbf{X}} : \mathbb{R}^m \to \mathbb{R}^n$.

Image

Recall: The **image** of a mapping f is the set of all possible function values, here

$$\text{image}(f_{\mathbf{X}}) := \{\mathbf{y} \in \mathbb{R}^n \mid \mathbf{Xz} = \mathbf{y} \text{ for some } \mathbf{z} \in \mathbb{R}^m\}$$

Image of a linear mapping

- The image of a linear mapping $\mathbb{R}^m \to \mathbb{R}^n$ is a linear subspace of $\mathbb{R}^n$.
- The columns of $\mathbf{X}$ form a basis of the image space:

$$\text{image}(\tilde{\mathbf{X}}) = \text{span}\{\mathbf{X}_1^{\text{col}}, \ldots, \mathbf{X}_m^{\text{col}}\}$$

- This is one of most useful things to remember about matrices, so, again:

The columns span the image.

Dimension of the image space

Clearly: The number of linearly independent column vectors. This number is called the **column rank** of $\mathbf{X}$.

Invertible mappings

Recall: A mapping f is invertible if it is one-to-one, i.e. for each function value $\tilde{\mathbf{y}}$ there is exactly one input value with $f(\mathbf{z}) = \tilde{\mathbf{y}}$.

Invertible matrices

The matrix $\tilde{\mathbf{X}}$ is called invertible if $f_{\tilde{\mathbf{X}}}$ is invertible.

- Only square matrices can be invertible.
- For a *linear* mapping: If $\tilde{\mathbf{X}}$ is a square matrix $f_{\tilde{\mathbf{X}}}$ is invertible if the image has the same dimension as the input space.
- Even if $\tilde{\mathbf{X}} \in \mathbb{R}^{n \times m}$, the matrix $\tilde{\mathbf{X}}^t \tilde{\mathbf{X}}$ is in $\mathbb{R}^{m \times m}$ (a square matrix).
- So: $\tilde{\mathbf{X}}^t \tilde{\mathbf{X}}$ is invertible if $\tilde{\mathbf{X}}$ has full column rank.

Recall: Transpose

The transpose A^{T} of a matrix $A \in \mathbb{R}^m$ is the matrix with entries

$$(A^{\mathrm{T}})_{ij} := A_{ji}$$

Orthogonal matrices

A matrix $O \in \mathbb{R}^{m \times m}$ is called **orthogonal**

$$O^{-1} = O^T$$

Orthogonal matrices describe two types of operations:

1. Rotations of the coordinate system.
2. Permutations of the coordinate axes.

Symmetric matrices

A matrix $A \in \mathbb{R}^{m \times m}$ is called **symmetric**

$$A = A^{\mathrm{T}}$$

Note: Symmetric and orthogonal matrices are very different objects. Only the identity is both.

Recall: ONB

A basis $\{v_1, \ldots, v_m\}$ of $\mathbb{R}^m$ is called an **orthonormal basis** if

$$\langle v_i, v_j \rangle = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

In other words, the v_i are pairwise orthogonal and each of length 1.

Orthogonal matrices

A matrix is orthogonal precisely if its rows form an ONB. Any two ONBs can be transformed into each other by an orthogonal matrix.

Representation of a vector

Suppose $\mathcal{E} = \{e_1, \ldots, e_d\}$ is a basis of a vector space. Then a vector x is represented as

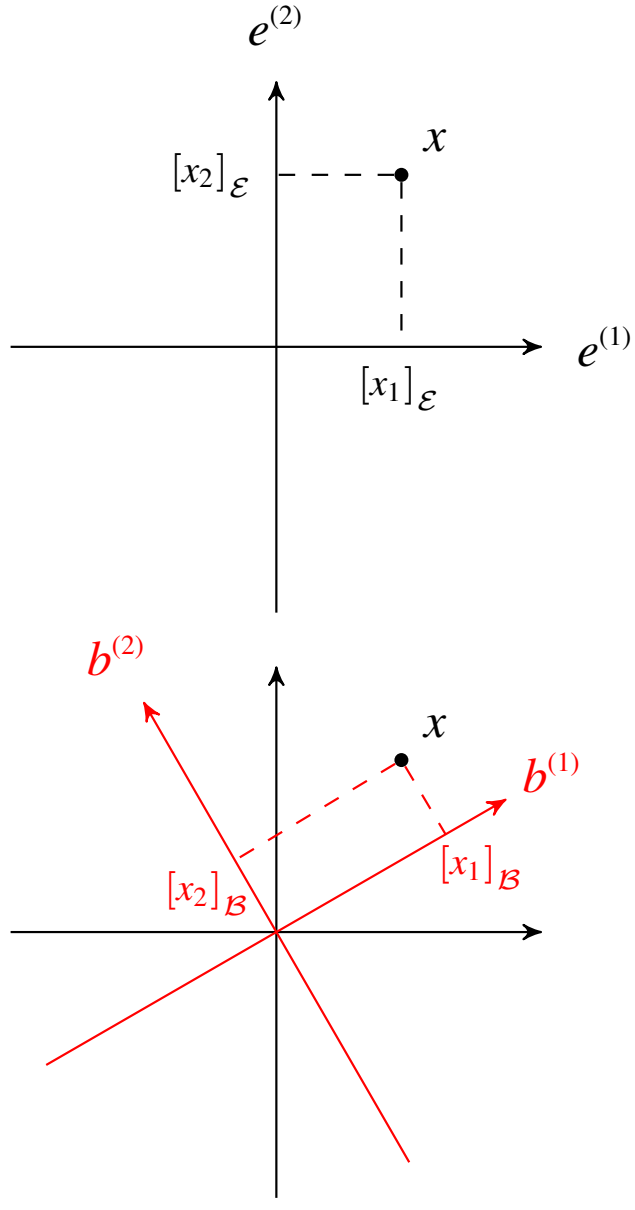
$$x = \sum_{j=1}^{d} [x_j]_{\mathcal{E}} \, e^{(j)}$$

$[x_j]_{\mathcal{E}} \in \mathbb{R}$ are the coordinates of x w.r.t. $\mathcal{E}$.

Other bases

If $\mathcal{B} = \{b_1, \ldots, b_d\}$ is another basis, x can be represented alternatively as

$$x = \sum_{j=1}^{d} [x_j]_{\mathcal{B}} \, b^{(j)}$$



Change-of-basis matrix

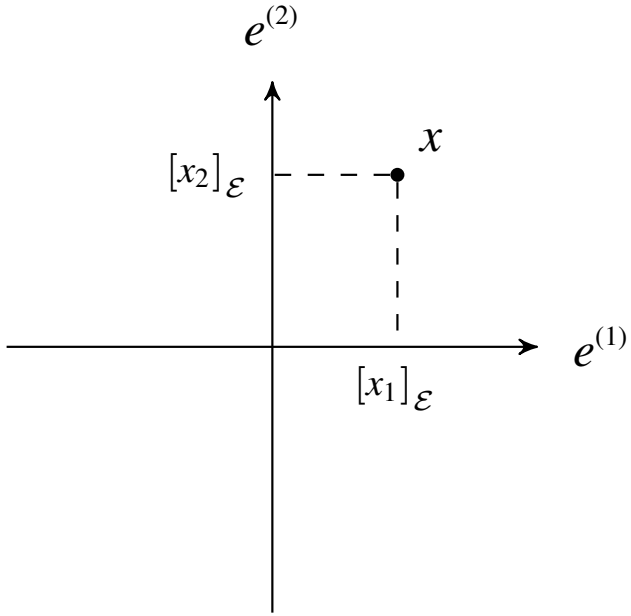
The matrix

$$M := \left([e^{(1)}]_{\mathcal{B}}, \ldots, [e^{(\mathrm{d})}]_{\mathcal{B}} \right)$$

transforms between the bases, i.e.

$$M[x]_{\mathcal{E}} = [x]_{\mathcal{B}} \ .$$

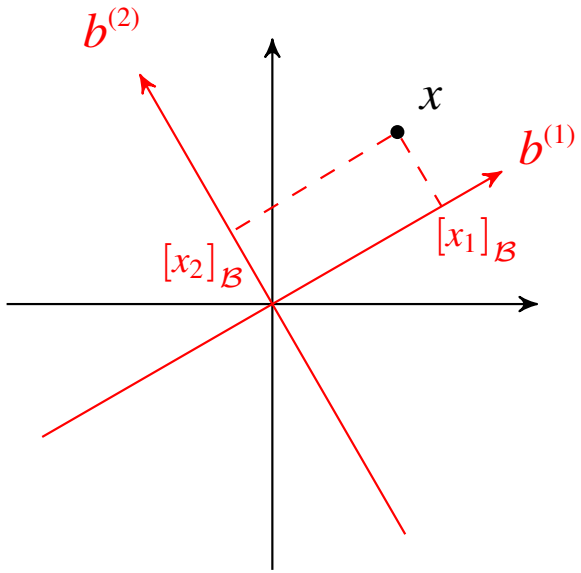
If both $\mathcal{E}$ and $\mathcal{B}$ are ONBs, M is orthogonal.



Representation of matrices

The matrix representing a linear mapping $A : \mathbb{R}^d \to \mathbb{R}^d$ in the basis $\mathcal{E}$ is computed as

$$[A]_{\mathcal{E}} := \left([A(e^{(1)})]_{\mathcal{E}}, \ldots, [A(e^{(\mathrm{d})})]_{\mathcal{E}} \right)$$

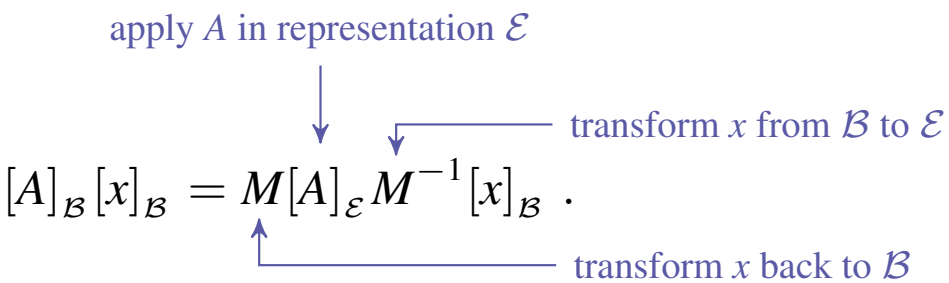


Transforming matrices

The matrix representing a linear mapping also changes when we change bases:

$$[A]_{\mathcal{B}} = M[A]_{\mathcal{E}} M^{-1} .$$

Applied to a vector x, this means:



Transforming between ONBs

If $\mathcal{V} = \{v_1, \ldots, v_m\}$ and $\mathcal{W} = \{w_1, \ldots, w_m\}$ are any two ONBs, there is an orthogonal matrix O such that

$$[A]_{\mathcal{V}} = O[A]_{\mathcal{W}} O^{-1}$$

for any linear mapping A.

Tools: Eigenvalues and Gaussian Distributions

We consider a square matrix $A \in \mathbb{R}^{m \times m}$.

Definition

A vector $\xi \in \mathbb{R}^m$ is called an **eigenvector** of A if the direction of ξ does not change under application of A. In other words, if there is a scalar λ such that

$$A\xi = \lambda\xi \ .$$

λ is called an **eigenvalue** of A for the eigenvector ξ.

Properties in general

- In general, eigenvalues are complex numbers $\lambda \in \mathbb{C}$.
- The class of matrices with the nicest eigen-structure are symmetric matrices, for which all eigenvalues are real numbers.

If a matrix is symmetric:

- All eigenvalues and eigenvectors are real, i.e. $\lambda \in \mathbb{R}$ and $\xi \in \mathbb{R}^m$.
- There are $\operatorname{rank}(A)$ distinct eigenvectors.
- The eigenvectors are pair-wise orthogonal.
- If $\operatorname{rank}(A) = m$, there is an ONB of $\mathbb{R}^m$ consisting of eigenvectors of A.

Definiteness

type	if ...
positive definite	all eigenvalues > 0
positive semi-definite	all eigenvalues ≥ 0
negative semi-definite	all eigenvalues ≤ 0
negative definite	all eigenvalues < 0
indefinite	none of the above

EIGENVECTOR ONB

Setting

- Suppose A symmetric, $\xi_1, \ldots, \xi_m$ are eigenvectors and form an ONB.
- $\lambda_1, \ldots, \lambda_m$ are the corresponding eigenvalues.

How does A act on a vector $v \in \mathbb{R}^m$?

1. Represent v in basis $\xi_1, \ldots, \xi_m$:

$$v = \sum_{j=1}^{m} v_j^{\mathrm{A}} \xi_j \qquad \text{where } v_j^{\mathrm{A}} \in \mathbb{R}$$

2. Multiply by A: Eigenvector definition (recall: $A\xi_j = \lambda\xi_j$) yields

$$Av \quad = \quad A\Big(\sum_{j=1}^{m} v_j^{\mathrm{A}} \xi_j\Big) \quad = \quad \sum_{j=1}^{m} v_j^{\mathrm{A}} A\xi_j \quad = \quad \sum_{j=1}^{m} v_j^{\mathrm{A}} \lambda_j \xi_j$$

Conclusion

A symmetric matrix acts by scaling the directions ξ_j.

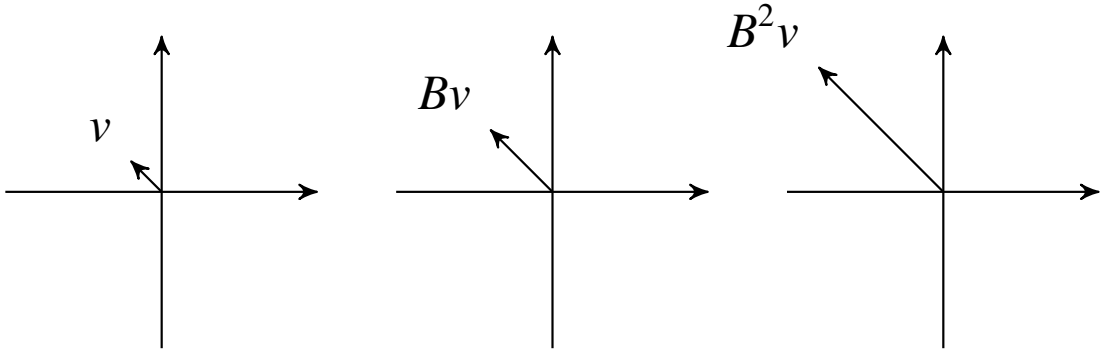
Setting

We *repeatedly* apply a symmetric matrix B to some vector $v \in \mathbb{R}^m$, i.e. we compute

$$Bv, \quad B(Bv) = B^2v, \quad B(B(Bv))) = B^3v, \quad \dots$$

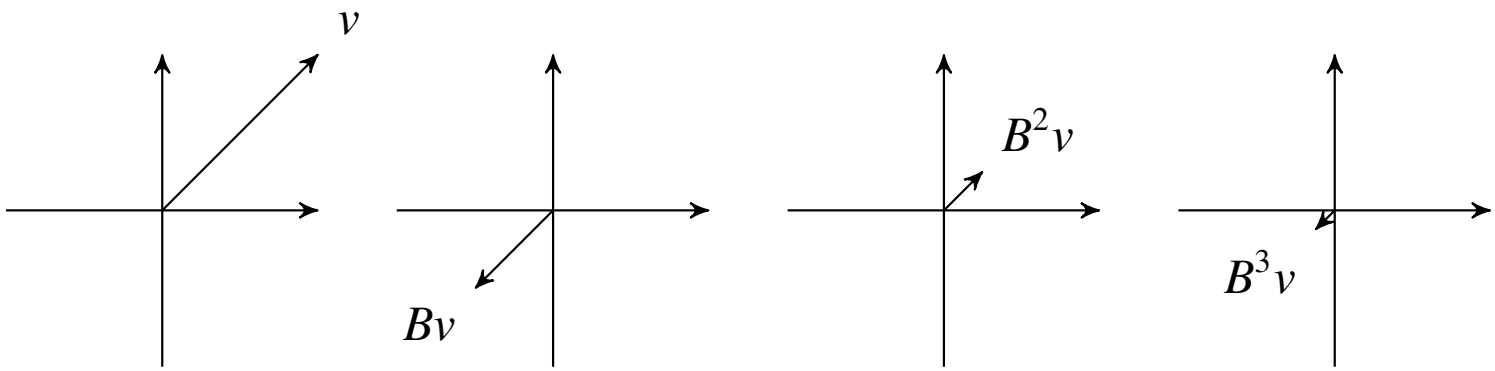
How does v change?

Example 1: v is an eigenvector with eigenvalue 2



The direction of v does not change, but its length doubles with each application of B.

Example 2: v is an eigenvector with eigenvalue $-\frac{1}{2}$



For an arbitrary vector v

$$B^n v = \sum_{j=1}^{m} v_j^{\mathrm{B}} \lambda_j^n \xi_j$$

- The weight λ_j^n grows most rapidly for eigenvalue with largest absolute value.
- Consequence:

The direction of $B^n v$ converges to the direction of the eigenvector with largest eigenvalue as n grows large.

Quadratic Forms

In applications, symmetric matrices often occur in quadratic forms.

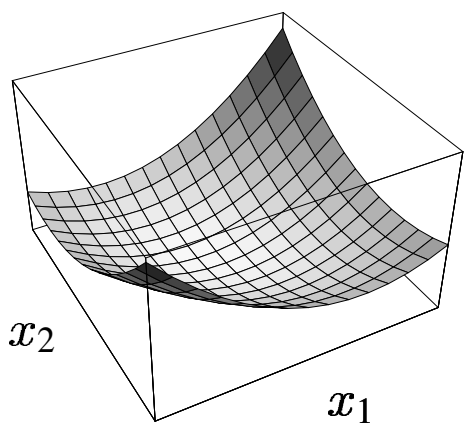
Definition

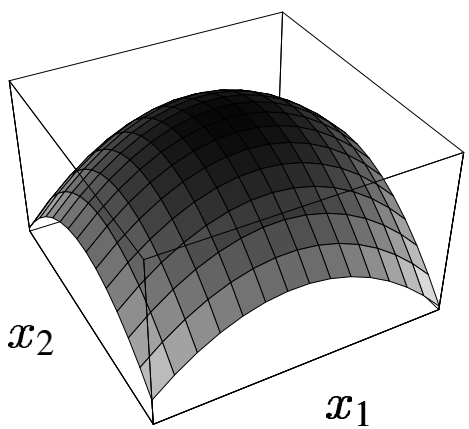
The **quadratic form** defined by a matrix A is the function

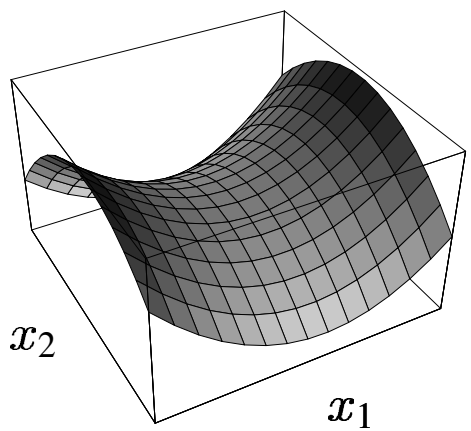
$$\begin{aligned} q_A : \mathbb{R}^m &\to \mathbb{R} \\ x &\mapsto \langle x, Ax \rangle \end{aligned}$$

Intuition

A quadratic form is the m-dimensional analogue of a quadratic function ax^2, with a vector substituted for the scalar x and the matrix A substituted for the scalar $a \in \mathbb{R}$.

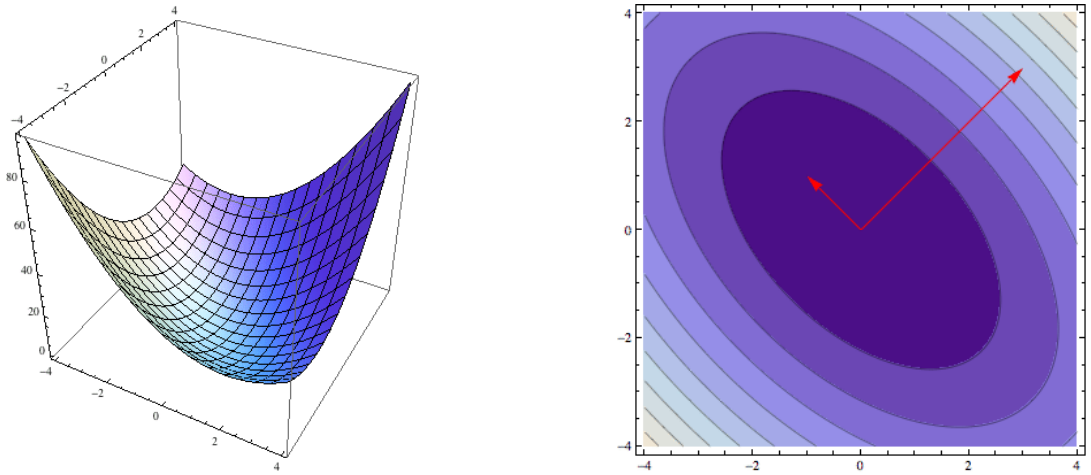






QUADRATIC FORMS

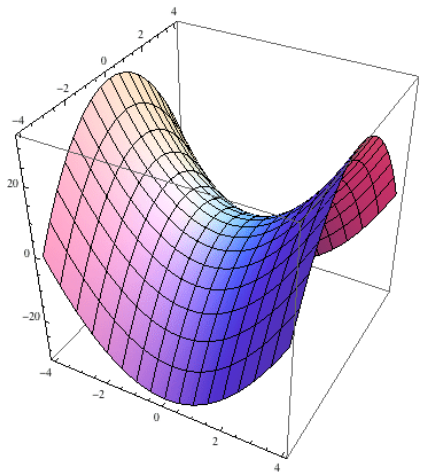
Here is the quadratic form for the matrix $A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$:

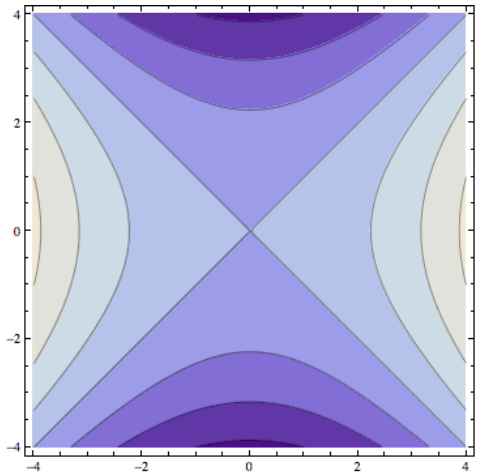


- Left: The function value q_A is graphed on the vertical axis.
- Right: Each line in $\mathbb{R}^2$ corresponds to a constant function value of q_A.
  Dark color = small values.
- The red lines are eigenvector directions of A. Their lengths represent the (absolute) values of the eigenvalues.
- In this case, both eigenvalues are positive. If all eigenvalues are positive, the contours are ellipses. So:

$$\text{positive definite matrices} \quad \leftrightarrow \quad \text{elliptic quadratic forms}$$

In this plot, the eigenvectors are axis-parallel, and one eigenvalue is negative:





The matrix here is $A = \begin{pmatrix} 2 & 0 \\ 0 & -2 \end{pmatrix}$.

Intuition

- If we change the sign of one of the eigenvalue, the quadratic function along the corresponding eigen-axis flips.
- There is a point which is a minimum of the function along one axis direction, and a maximum along the other. Such a point is called a *saddle point*.

Recall: Covariance

The covariance of two random variables X_1, X_2 is

$$\mathrm{Cov}[X_1, X_2] = \mathbb{E}[(X_1 - \mathbb{E}[X_1])(X_2 - \mathbb{E}[X_2])] \ .$$

If $X_1 = X_2$, the covariance is the variance: $\mathrm{Cov}[X, X] = \mathrm{Var}[X]$.

Covariance matrix

If $X = (X_1, \ldots, X_m)$ is a random vector with values in $\mathbb{R}^m$, the matrix of all covariances

$$\mathrm{Cov}[X] := (\mathrm{Cov}[X_i, X_j])_{i,j} = \begin{pmatrix} \mathrm{Cov}[X_1, X_1] & \cdots & \mathrm{Cov}[X_1, X_m] \\ \vdots & & \vdots \\ \mathrm{Cov}[X_m, X_1] & \cdots & \mathrm{Cov}[X_m, X_m] \end{pmatrix}$$

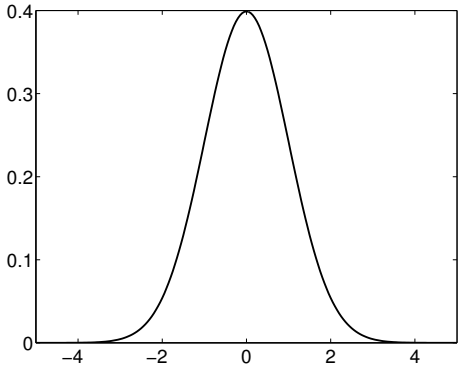
is called the **covariance matrix** of X.

Notation

It is customary to denote the covariance matrix $\mathrm{Cov}[X]$ by Σ.

Gaussian density in one dimension

$$p(x;\mu,\sigma) := \frac{1}{\sqrt{2\pi}\sigma}\exp\Big(-\frac{(x-\mu)^2}{2\sigma^2}\Big)$$

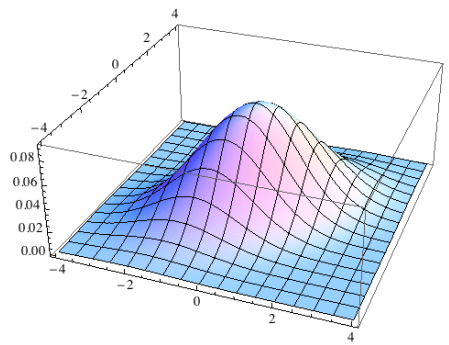


- μ = expected value of x, σ^2 = variance, σ = standard deviation
- The quotient $\frac{x-\mu}{\sigma}$ measures deviation of x from its expected value in units of σ (i.e. σ defines the length scale)

Gaussian density in m dimensions

The quadratric function

$$-\frac{(x-\mu)^2}{2\sigma^2} \quad = \quad -\frac{1}{2}(x-\mu)(\sigma^2)^{-1}(x-\mu)$$

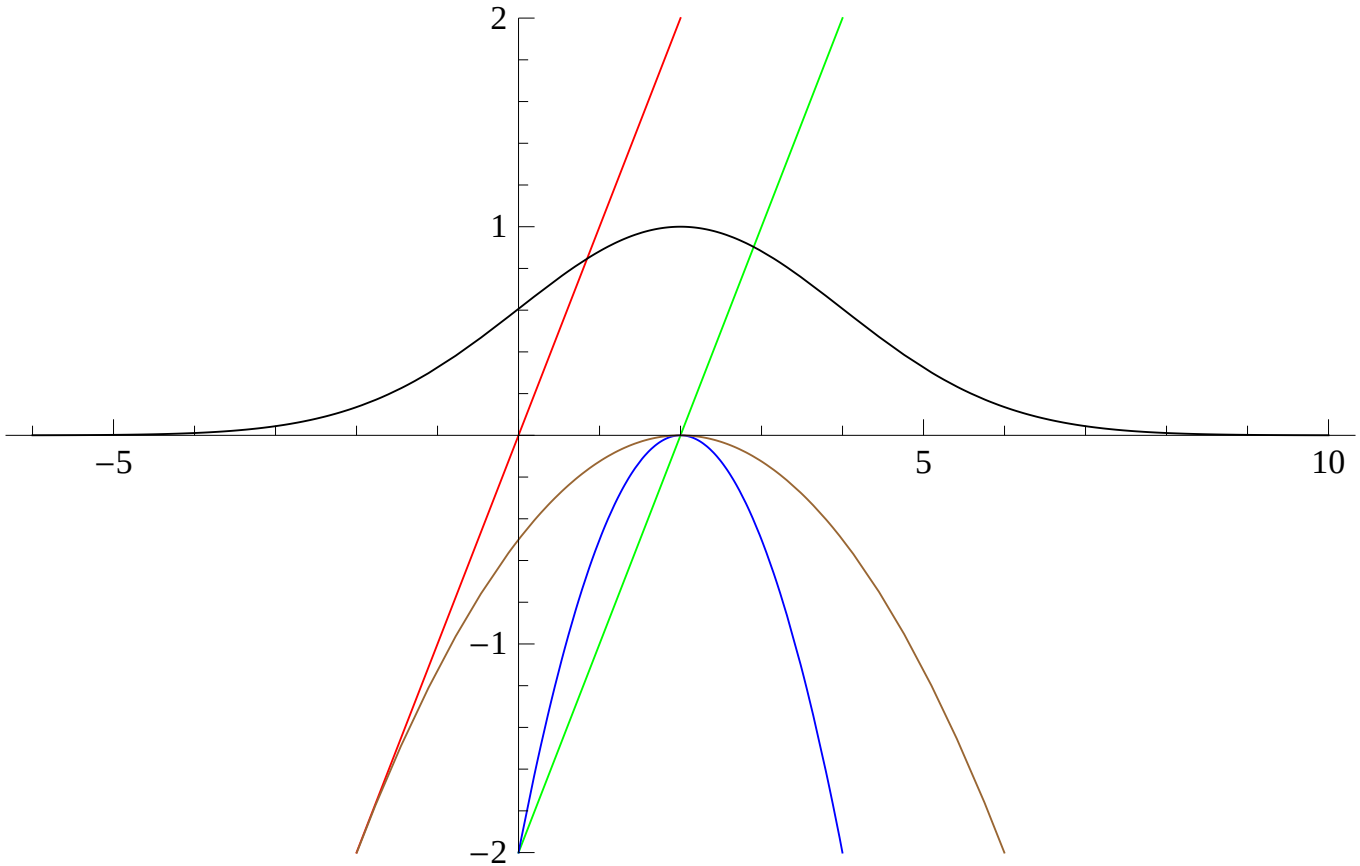


is replaced by a quadratic form:

$$p(\mathbf{x};\boldsymbol{\mu},\Sigma) := \frac{1}{\sqrt{2\pi\,\det(\Sigma)}}\exp\Big(-\frac{1}{2}\Big\langle(\mathbf{x}-\boldsymbol{\mu}),\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\Big\rangle\Big)$$

Components of a 1D Gaussian

$\mu = 2, \sigma = 2$



- Red: $x \mapsto x$
- Green: $x \mapsto x - \mu$
- Blue: $x \mapsto -\frac{1}{2}(x-\mu)^2$
- Brown: $x \mapsto -\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2$
- Black: $x \mapsto \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$

Covariance matrix of a Gaussian

If a random vector $X \in \mathbb{R}^m$ has Gaussian distribution with density $p(\mathbf{x}; \mu, \Sigma)$, its covariance matrix is $\mathrm{Cov}[X] = \Sigma$. In other words, a Gaussian is parameterized by its covariance.

Observation

Since $\mathrm{Cov}[X_i, X_j] = \mathrm{Cov}[X_j, X_i]$, the covariance matrix is symmetric.

What is the eigenstructure of Σ?

- We know: Σ symmetric $\Rightarrow$ there is an eigenvector ONB
- Call the eigenvectors in this ONB $\xi_1, \ldots, \xi_m$ and their eigenvalues $\lambda_1, \ldots, \lambda_m$
- We can rotate the coordinate system to $\xi_1, \ldots, \xi_m$. In the new coordinate system, Σ has the form

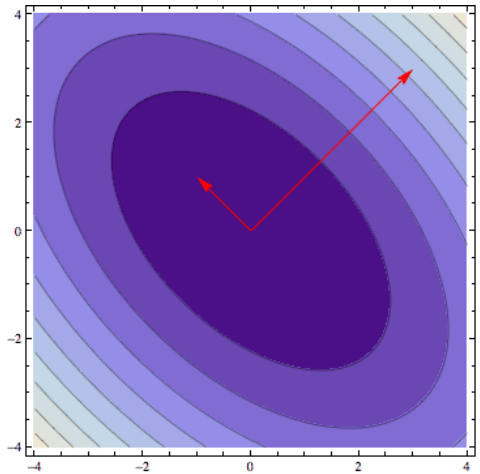
$$\Sigma_{[\xi_1,\ldots,\xi_n]} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_m \end{pmatrix} = \mathrm{diag}(\lambda_1, \ldots, \lambda_m)$$

Example

Quadratic form

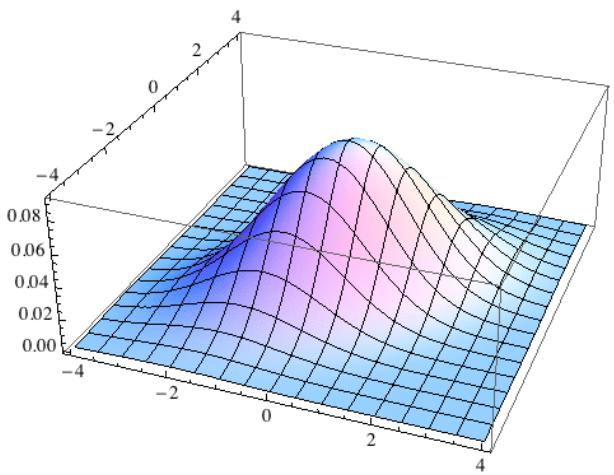
$$\langle \mathbf{x}, \Sigma \mathbf{x} \rangle \quad \text{with} \quad \Sigma = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

The eigenvectors are $(1,1)$ and $(-1,1)$ with eigenvalues 3 and 1.

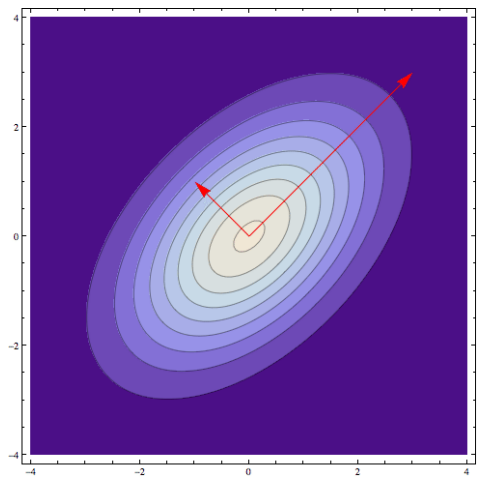


Gaussian density

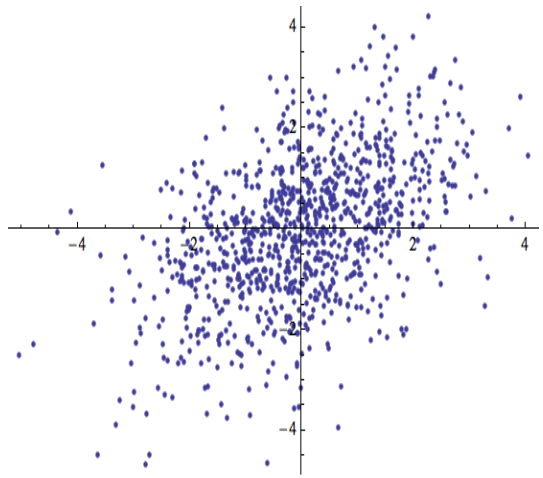
$p(\mathbf{x}; \boldsymbol{\mu}, \Sigma)$ with $\boldsymbol{\mu} = (0,0)$.



Density graph



Density contour



1000 sample points

The ξ_i as random variables

Write $e_1, \ldots, e_m$ for the ONB of axis vectors. We can represent each ξ_i as

$$\xi_i = \sum_{j=1}^{m} \alpha_{ij} e_j$$

Then $O = (\alpha_{ij})$ is the orthogonal transformation matrix between the two bases. We can represent random vector $X \in \mathbb{R}^m$ sampled from the Gaussian in the eigen-ONB as

$$X_{[\xi_1,\ldots,\xi_m]} = (X'_1, \ldots, X'_m) \qquad \text{with} \qquad X'_i = \sum_{j=1}^{m} \alpha_{ij} X_j$$

Since the X_j are random variables (and the α_{ij} are fixed), each X'_i is a scalar random variable.

Meaning of the random variables ξ_i

For any Gaussian $p(\mathbf{x}; \boldsymbol{\mu}, \Sigma)$, we can

1. shift the origin of the coordinate system into $\boldsymbol{\mu}$
2. rotate the coordinate system to the eigen-ONB of Σ.

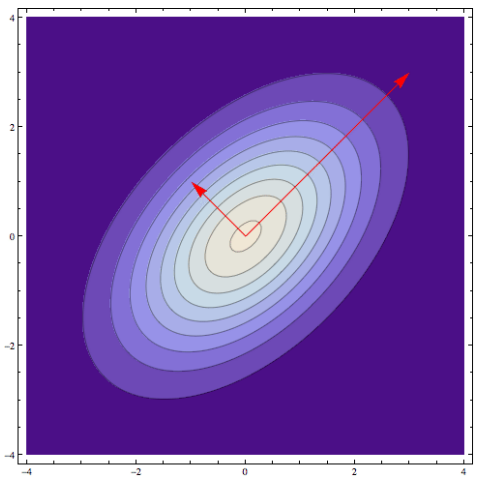
In this new coordinate system, the Gaussian has covariance matrix

$$\Sigma_{[\xi_1,\ldots,\xi_m]} = \operatorname{diag}(\lambda_1, \ldots, \lambda_m)$$

where λ_i are the eigenvalues of Σ.

Gaussian in the new coordinates

A Gaussian vector $X_{[\xi_1,\ldots,\xi_m]}$ represented in the new coordinates consists of m *independent* 1D Gaussian variables X_i'. Each X_i' has mean 0 and variance λ_i.



SHRINKAGE

Issues with Least Squares

Robustness

- Least squares works only if $\tilde{\mathbf{X}}$ has full column rank, i.e. if $\tilde{\mathbf{X}}^t\tilde{\mathbf{X}}$ is invertible.
- If $\tilde{\mathbf{X}}^t\tilde{\mathbf{X}}$ *almost* not invertible, least squares is numerically unstable.
  Statistical consequence: High variance of predictions.

Not suited for high-dimensional data

- Modern problems: Many dimensions/features/predictors (possibly thousands)
- Only a few of these may be important
  $\rightarrow$ need some form of feature selection
- Least squares:
  - Treats all dimensions equally
  - Relevant dimensions are averaged with irrelevant ones
  - Consequence: Signal loss

Regularity

A matrix which is not invertible is also called a **singular** matrix. A matrix which is invertible (not singular) is called **regular**.

In computations

Numerically, matrices can be "almost singular". Intuition:

- A singular matrix maps an entire linear subspace into a single point.
- If a matrix maps points far away from each other to points very close to each other, it almost behaves like a singular matrix.

Regularity of Symmetric Matrices

Recall: A positive semi-definite matrix A is singluar $\Leftrightarrow$ smallest EValue is 0

Illustration

If smallest EValue $\lambda_{\min} > 0$ but very small (say $\lambda_{\min} \approx 10^{-10}$):

- Suppose x_1, x_2 are two points in subspace spanned by $\xi_{\min}$ with $\|x_1 - x_2\| \approx 1000$.
- Image under A: $\|Ax_1 - Ax_2\| \approx 10^{-7}$

In this case

- A has an inverse, but A behaves almost like a singular matrix
- The inverse A^{-1} can map almost identical points to points with large distance, i.e.

  small change in input $\quad\rightarrow\quad$ large change in output

  $\rightarrow$ unstable behavior

Consequence for Statistics

If a statistical prediction involves the inverse of an almost-singular matrix, the predictions become unreliable (high variance).

Recall: Prediction in linear regression

For a point $\mathbf{x}_{\text{new}} \in \mathbb{R}^d$, we predict the corresponding function value as

$$\hat{y}_{\text{new}} = \left\langle \hat{\beta}, (1, \mathbf{x}) \right\rangle = (\tilde{\mathbf{X}}^t \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^t \mathbf{y}$$

Effect of unstable inversion

- Suppose we choose an arbitrary training point $\tilde{\mathbf{x}}_i$ and make a small change to its response value $\tilde{y}_i$.
- Intuitively, that should not have a big impact on $\hat{\beta}$ or on prediction.
- If $\tilde{\mathbf{X}}^t \tilde{\mathbf{X}}$ is almost singular, a small change to $\tilde{y}_i$ can prompt a huge change in $\hat{\beta}$, and hence in the predicted value $\hat{y}_{\text{new}}$.

Symmetric matrices

Denote by $\lambda_{\max}$ and $\lambda_{\min}$ the eigenvalues of A with largest/smallest *absolute* value. If A is symmetric, then

$$A \text{ regular} \quad \Leftrightarrow \quad |\lambda_{\min}| > 0 \, .$$

Idea

- We can use $|\lambda_{\min}|$ as a measure of regularity:

$$\text{larger value of } \lambda_{\min} \quad \leftrightarrow \quad \text{"more regular" matrix } A$$

- We need a notion of scale to determine whether $|\lambda_{\min}|$ is large.
- The relevant scale is how A scales a vector. Maximal scaling coefficient: $\lambda_{\max}$.

Regularity measure

$$c(A) := \frac{|\lambda_{\min}|}{|\lambda_{\max}|}$$

The function $c(\,.\,)$ is called the **spectral condition** ("spectral" since the set of eigenvalues is also called the "spectrum").

Objective

Ridge regression is a modification of least squares. We try to make least squares more robust if $\tilde{\mathbf{X}}^t\tilde{\mathbf{X}}$ is almost singular.

Ridge regression solution

The ridge regression solution to a linear regression problem is defined as

$$\hat{\boldsymbol{\beta}}^{\text{ridge}} := (\tilde{\mathbf{X}}^t\tilde{\mathbf{X}} + \lambda\mathbb{I})^{-1}\tilde{\mathbf{X}}^t\mathbf{y}$$

λ is a tuning parameter.

Recall

$\tilde{\mathbf{X}}^t\tilde{\mathbf{X}} \in \mathbb{R}^{(d+1)\times(d+1)}$ is positive definite.

Spectral shift

Suppose $\xi_1, \ldots, \xi_{d+1}$ are EVectors of $\tilde{\mathbf{X}}^t\tilde{\mathbf{X}}$ with EValues $\lambda_1, \ldots, \lambda_{d+1}$.
Then:

$$(\tilde{\mathbf{X}}^t\tilde{\mathbf{X}} + \lambda\mathbb{I})\xi_i = (\tilde{\mathbf{X}}^t\tilde{\mathbf{X}})\xi_i + \lambda\mathbb{I}\xi_i = (\lambda_i + \lambda)\xi_i$$

Hence: $(\tilde{\mathbf{X}}^t\tilde{\mathbf{X}} + \lambda\mathbb{I})$ is positive definite with EValues $\lambda_1 + \lambda, \ldots, \lambda_{d+1} + \lambda$.

Conclusion

$\tilde{\mathbf{X}}^t\tilde{\mathbf{X}} + \lambda\mathbb{I}$ is a *regularized* version of $\tilde{\mathbf{X}}^t\tilde{\mathbf{X}}$.

Effect of regularization

- We deliberately distort prediction:
  - If least squares ($\lambda = 0$) predicts perfectly, the ridge regression prediction has an error that increases with λ.
  - Hence: Biased estimator, bias increases with λ.
- Spectral shift regularizes matrix $\rightarrow$ decreases variance of predictions.

Bias-variance trade-off

- We decrease the variance (improve robustness) at the price of incurring a bias.
- λ controls the trade-off between bias and variance.