Machine Learning

– winter term 2016/17 –

Chapter 10: Instance-based Learning

Prof. Adrian Ulges
Masters “Computer Science”
DCSM Department
University of Applied Sciences RheinMain

ML Strategies so Far

Our ML Models so far...

- Learning based on recursive splits *(decision trees)*
- Learning based on hyperplanes *(logistic regression)*
- Learning based on stacked hyperplanes *(neural networks)*
- Learning based on projection to subdimensions *(PCA)*
- Learning based on finding clusters of close-by points *(K-Means/EM)*

In this Chapter

- Learning based on **comparing instances** (=samples)
- Required: **similarity/distance measure** (Euclidean?)

1. k-Nearest Neighbor Classification
2. fast nearest neighbor search
3. Support Vector Machines

Outline

k-Nearest Neighbor

Y'old Classification Setting

- Training samples $\mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^d$ with labels $y_1, ..., y_n \in \{1, ..., C\}$
- **Goal**: classify a sample $\mathbf{x}$

Approach

- Compute each training sample $\mathbf{x}_i$'s (Euclidean) distance to $\mathbf{x}$, $d(\mathbf{x}_i, \mathbf{x})$
- Sort the training samples by (increasing) distance to $\mathbf{x}$

$$\mathbf{x}_{\pi(1)}, \mathbf{x}_{\pi(2)}, ..., \mathbf{x}_{\pi(k)}, \mathbf{x}_{\pi(k+1)}, ..., \mathbf{x}_{\pi(n)}$$

with

$$\begin{aligned}
\text{(closest training sample)} \quad & \pi(1) = \arg\min_i \, d(\mathbf{x}_i, \mathbf{x}) \\
\text{(2nd closest training sample)} \quad & \pi(2) = \arg\min_{i \neq \pi(1)} \, d(\mathbf{x}_i, \mathbf{x}) \\
\text{(3rd closest training sample)} \quad & \pi(3) = \arg\min_{i \neq \pi(1), i \neq \pi(2)} \, d(\mathbf{x}_i, \mathbf{x}) \\
& ...
\end{aligned}$$

k-Nearest Neighbor

Approach (cont'd)

- We call the k closest training samples the **nearest neighbors** to $\mathbf{x}$

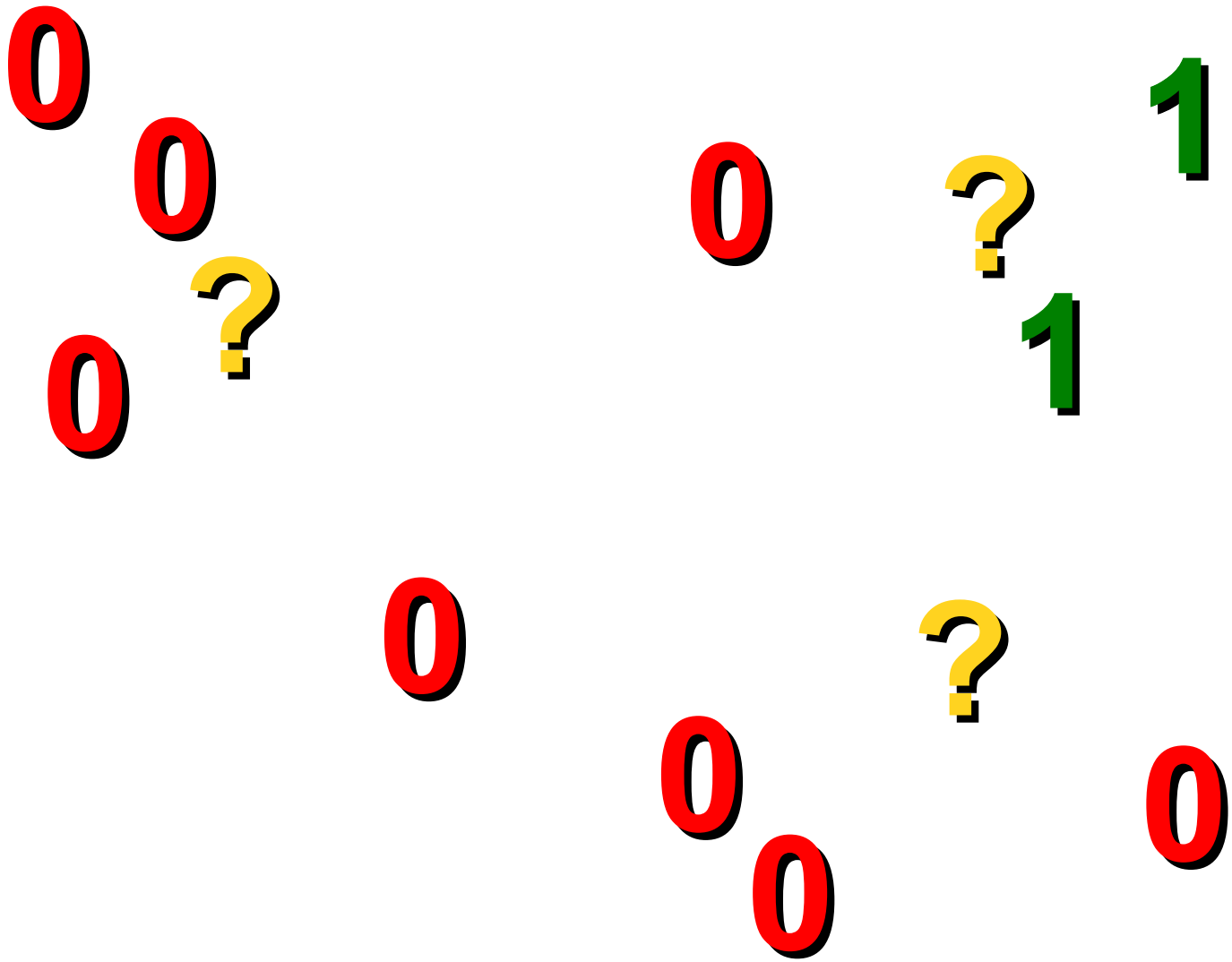
$$\mathbf{x}_{\pi(1)}, \mathbf{x}_{\pi(2)}, \cdots, \mathbf{x}_{\pi(k)}, \mathbf{x}_{\pi(k+1)}, \cdots, \mathbf{x}_{\pi(n)}$$

- We estimate the class score (or *posterior*) by a simple voting over the nearest neighbors

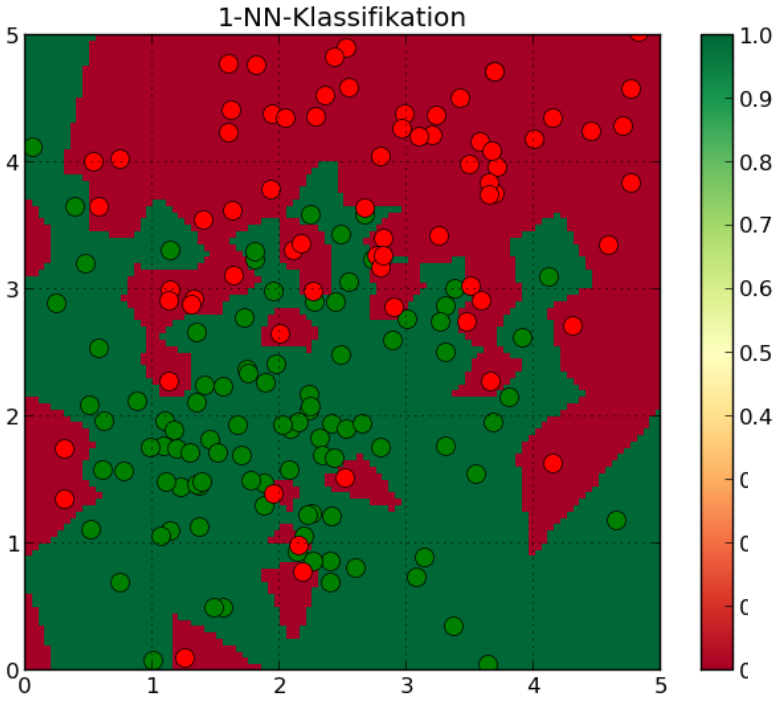
$$P(c|\mathbf{x}) = \frac{\sum_{j=1}^{k} \mathbf{1}_{c=y_{\pi(j)}}}{k}$$

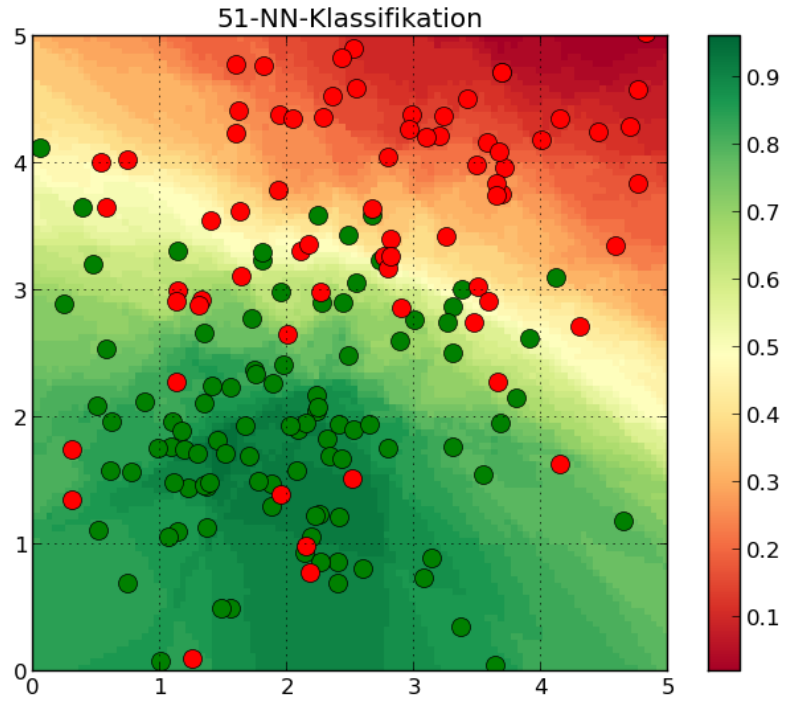
$$\left(= \frac{\#\text{ neighbors with class c}}{\#\text{ neighbors total}} \right)$$

k-Nearest Neighbor: Do-it-Yourself



k-Nearest Neighbor: Examples





k-Nearest Neighbor: Discussion

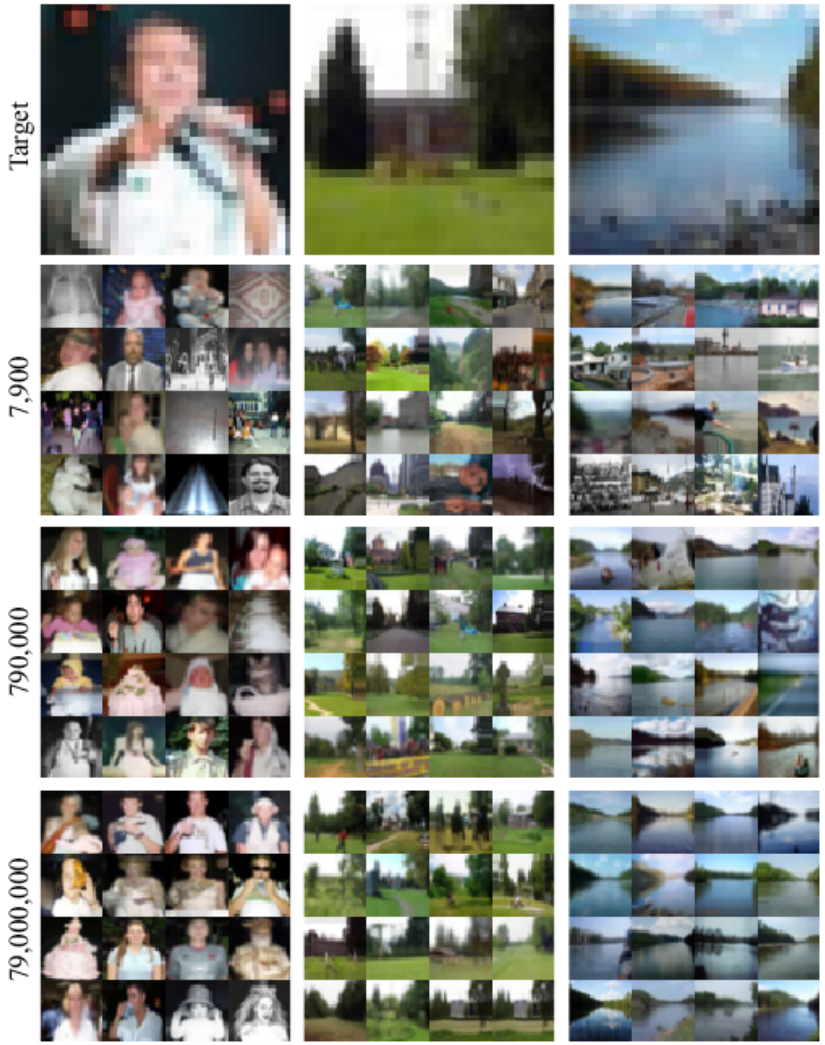
k-NN Example: Image Annotation

- **Given**: a training set of annotated images and a test image **x** (to be annotated)
- **Approach**: Find the k training images most similar to **x** and transfer their labels

k-NN Example: Image Annotation

A sample Approach (Torralba et al.)[1]

- **Scale** (color) images to 32×32 pixels
- Store **pixel values** in a $32 \times 32 \times 3$ feature vector
- Calculate Euclidean distance between vectors *(improvements by invariance to flipping and small shifts)*
- **Observation**: The bigger the training set, the 'better' neighbors+classification!

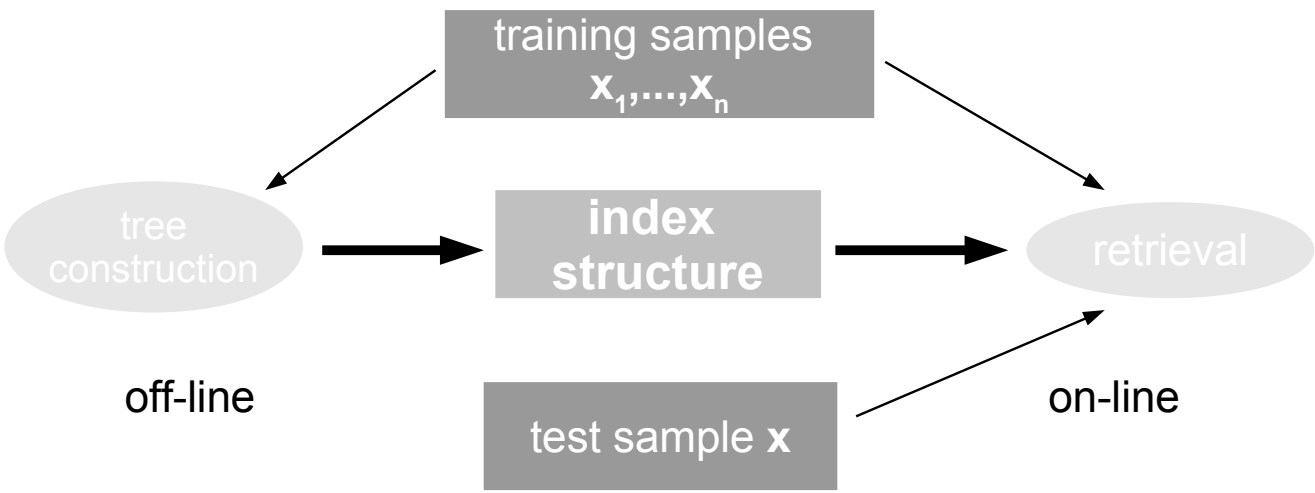


[1] Torralba et al.: „80 Mio. Tiny Images – A large-scale Dataset for Non-parametric Object and Scene Recognition“, CVPR 2008.

Outline

KD-Trees: Approach

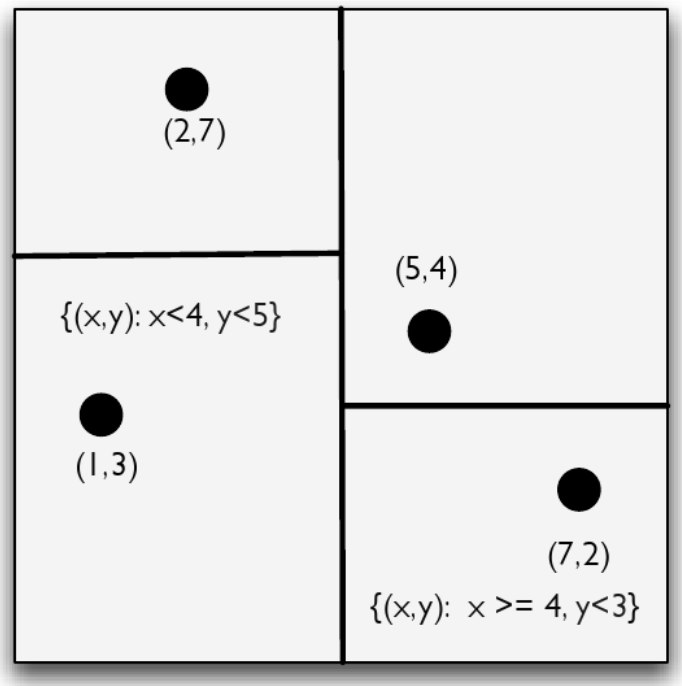
- **Tree-based indexing** is a standard approach towards scalable NN search, with applications in computer graphics, geo-search, machine learning, ...
- **Approach (space partitioning)**: Recursively subdivide feature space (similar to *binary search*)
- KD-trees are **index-based**: The KD-tree is constructed off-line, and used for fast search on-line

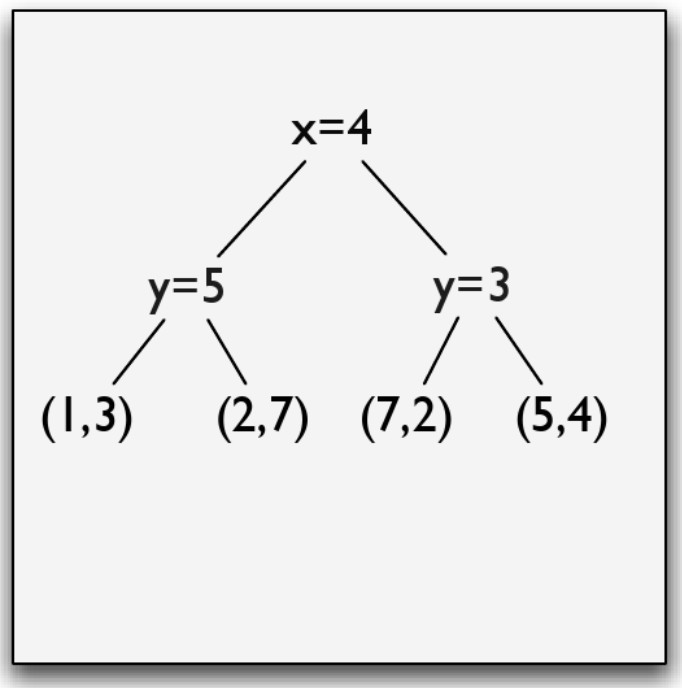


KD-Trees: Basics

For now, we assume ...

- ... feature vectors to be **real-valued**
- ... the target distance to be the **Euclidean** distance
- ... $\mathbf{k = 1}$ (only one nearest neighbor)





KD-Trees: Construction

```
1  function construct_kdtree(samples):
2      if #samples==1:          // reached a leaf
3          return KDTree(samples)
4      (d*, t) := choose_split(samples)
5      samples_0 := {x ∈ samples | x_d* < t}
6      samples_1 := {x ∈ samples | x_d* ≥ t}
7      tree_0 := construct_kdtree(samples_0)
8      tree_1 := construct_kdtree(samples_1)
9      return KDTree(d*, t, samples, tree_0, tree_1)
10
```

- Every node in the tree represents a **bounding box**

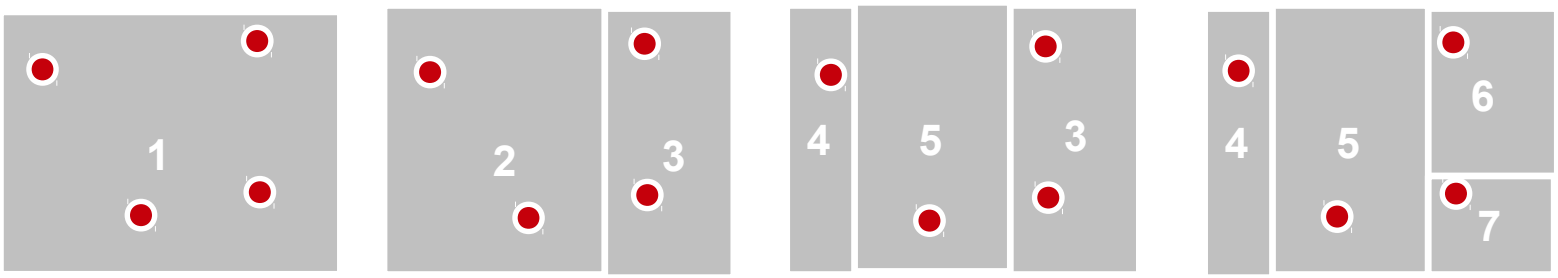
$$[min_1, max_1] \times ... \times [min_d, max_d]$$

- The root bounding box covers *all* training samples
- We recursively...
  - ... pick a dimension $d^* \in \{1, ..., d\}$ and a threshold $t \in \mathbb{R}$
  - ... and **split** the bounding box into two parts

$$[min_1, max_1] \times ...[min_d, \mathbf{t}[\times ... \times [min_d, max_d]$$

$$[min_1, max_1] \times ...[\mathbf{t}, max_d] \times ... \times [min_d, max_d]$$

KD-Trees: Do-it-Yourself



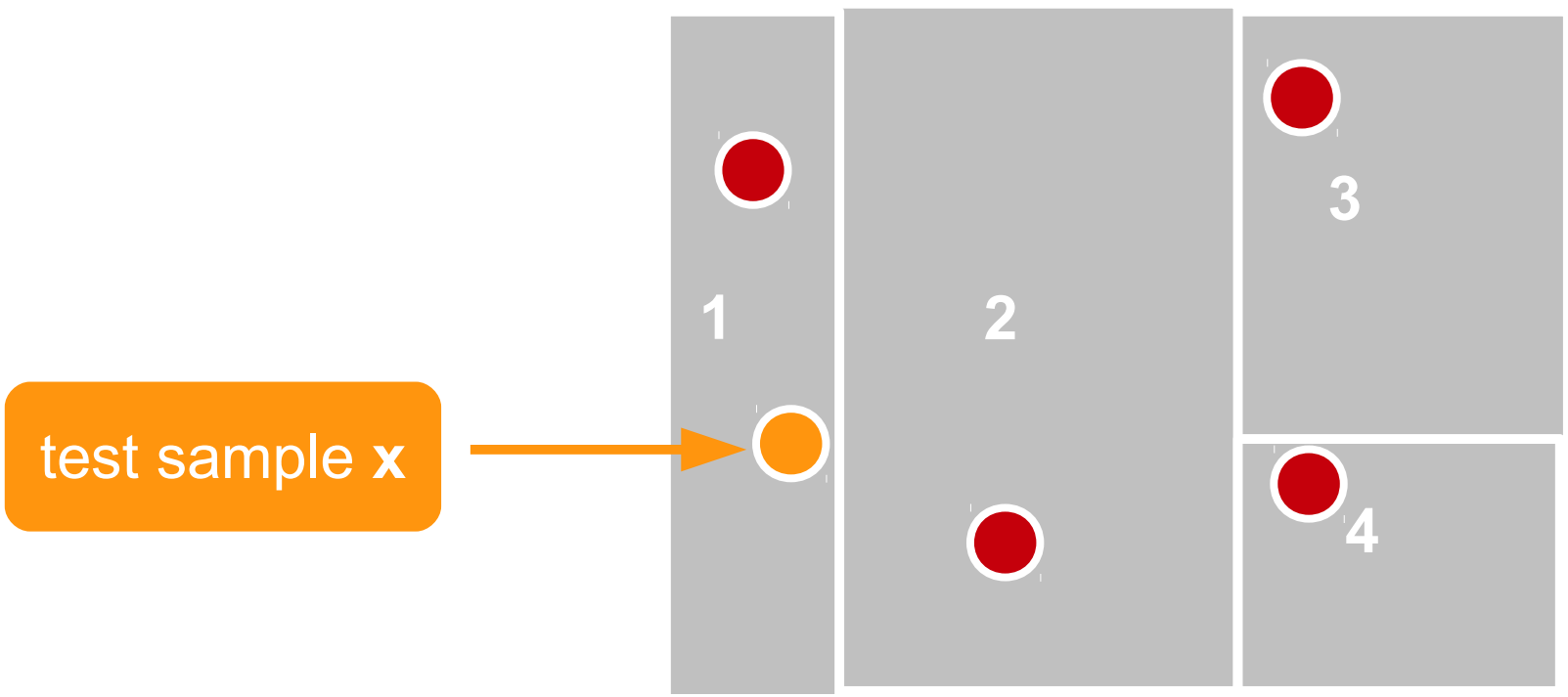
- What are good strategies for choosing d^* and t?

KD-Trees: Search

- Search works by recursing until we reach a **leaf node**
- We return the corresponding sample as the nearest neighbor
- Effort: $O(log(n))$ (if splitting by the median)

Challenge

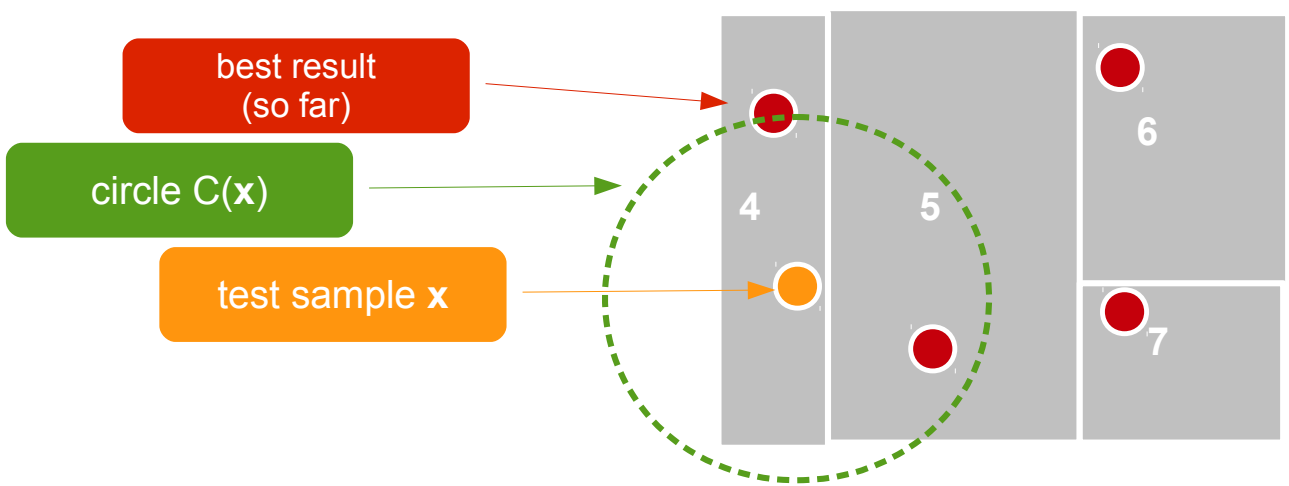
- The found neighbor may not be the best one



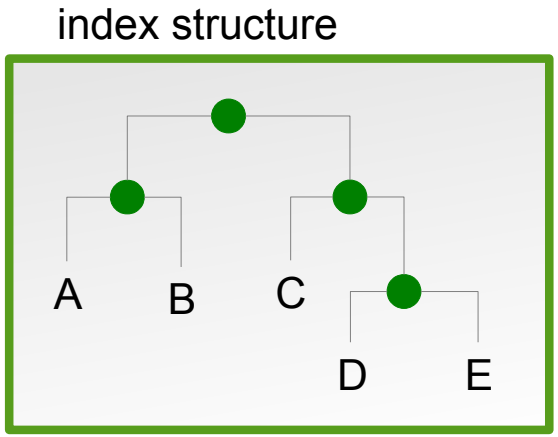
KD-Trees: Search (Backtracking)

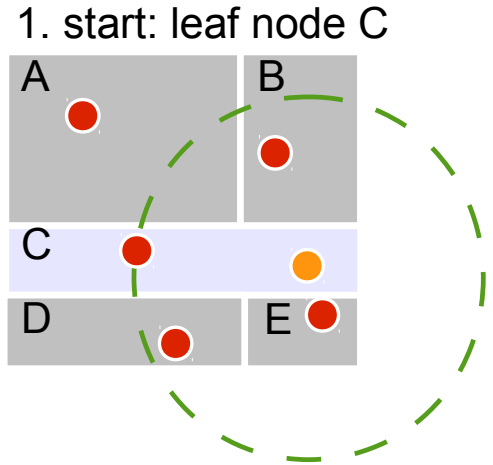
Extension: Backtracking

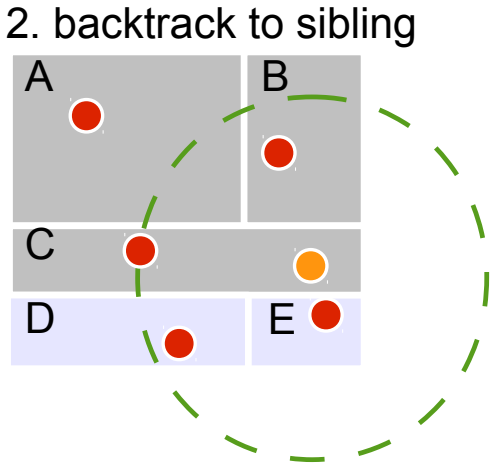
- **Observation**: Any potentially better neighbor than the one found would have to lie in a **circle C(x)**
- **Backtracking**: Recurse up the tree, and check each node whose bounding box intersects with $C(\mathbf{x})$
- Whenever we find a better neighbor, remember it and shrink $C(\mathbf{x})$

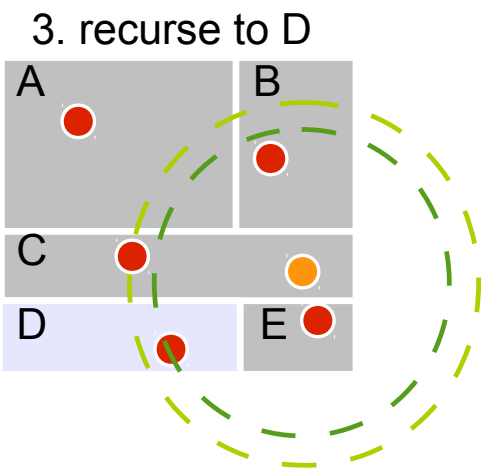


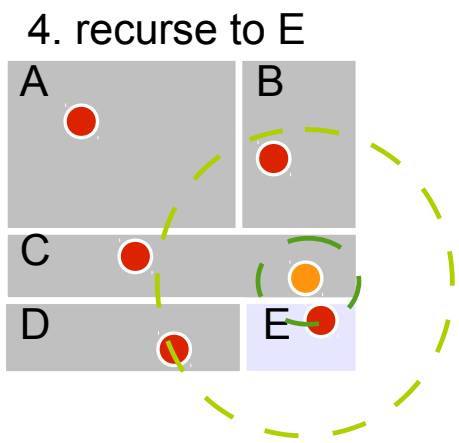
KD-Trees: Search (Backtracking Example)

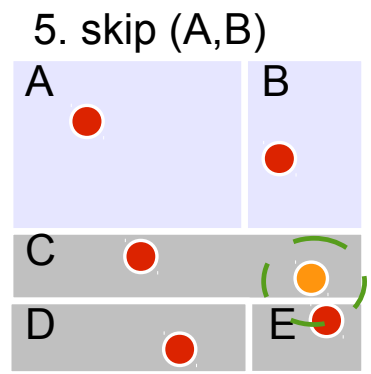










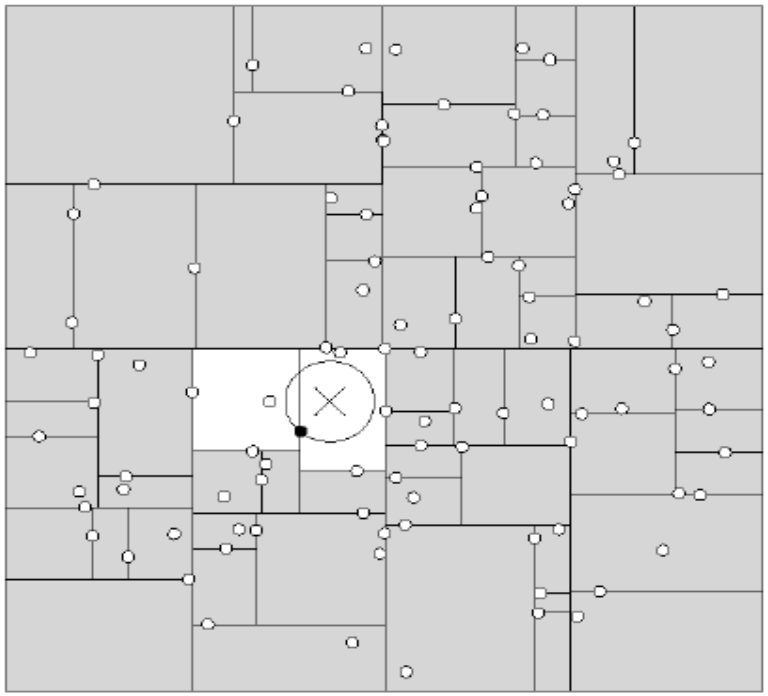


→ return E

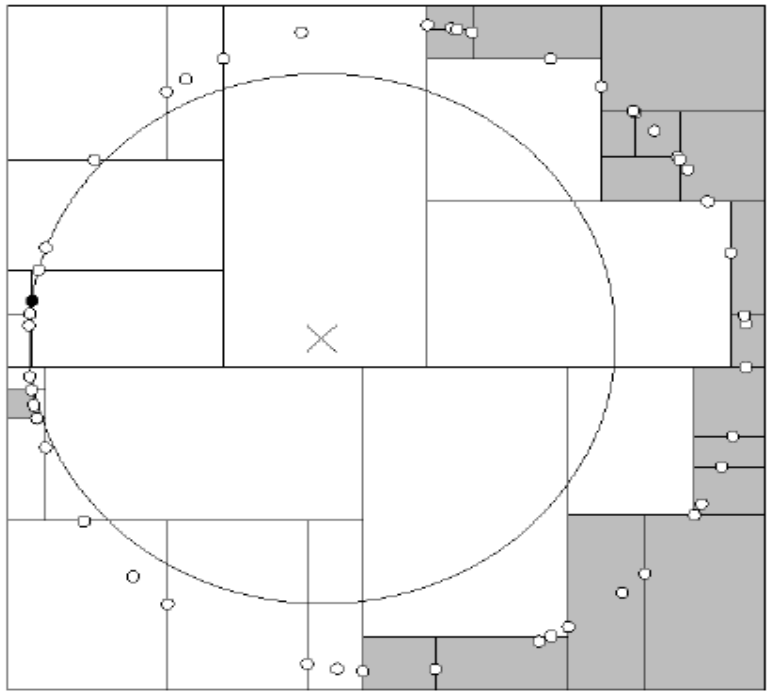
KD-Trees Search: Do-it-Yourself

- Do we always find the **best neighbor** by backtracking?
- What is the **O-class** when searching with backtracking?

KD-Trees: Search (Backtracking Example)



good case



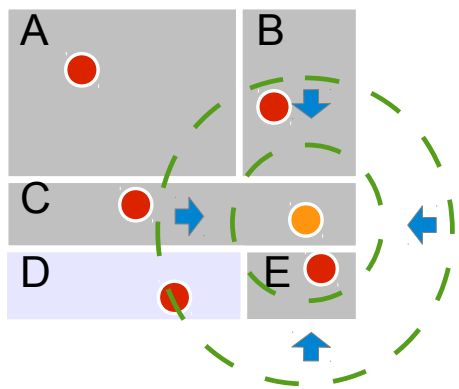
bad case

KD-Trees: Approximate Search

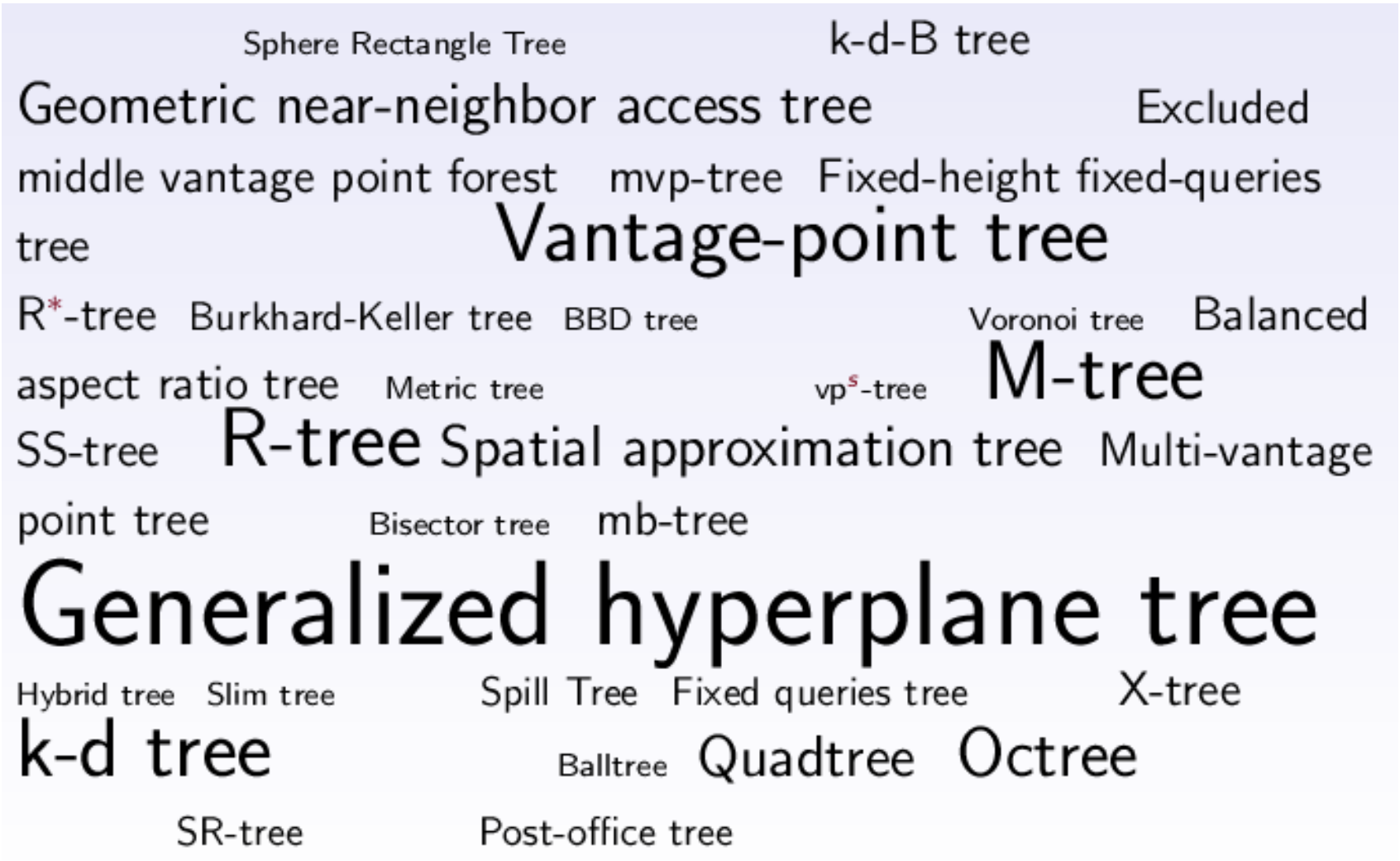
Approximate NN Search

- **Same approach as before**: We backtrack the tree and search regions intersecting with the circle $C(x)$
- Idea: **reduce the circle** by a factor ϵ (for example, $\epsilon = \frac{1}{3}$)
- This leads to a faster search *(more nodes are pruned)*
- **Quality garantee** (kd-tree result $\mathbf{x}'$ vs. best neighbor $\mathbf{x}^*$):

$$||\mathbf{x} - \mathbf{x}'|| \leq \frac{1}{\epsilon} \cdot ||\mathbf{x} - \mathbf{x}^*||$$



Tree Structures for fast NN Search



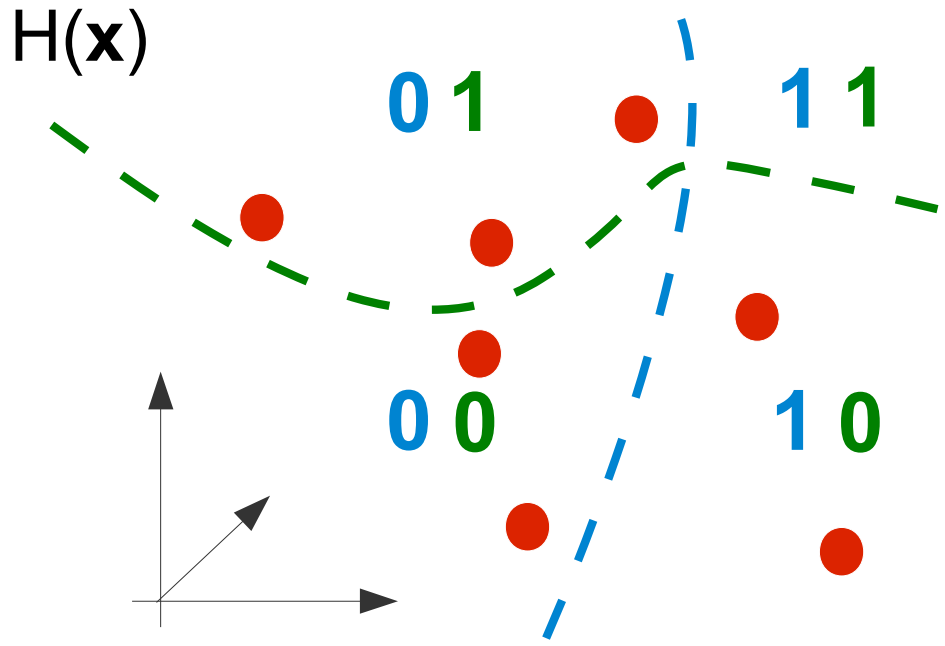
Outline

Locallity-sensitive Hashing (LSH)

Locality-sensitive hashing (LSH) is a **space partitioninig** approach, similar to KD-trees

Differences to KD-trees

- Partitioning is (usually) **sequential**, not recursive
- No backtracking (LSH search is **approximate**)
- Subdivisions are **randomized**



LSH: Formalization

- **Given**: training samples $\mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^d$
- **Given**: a set (or *family*) of hash functions, each of the form

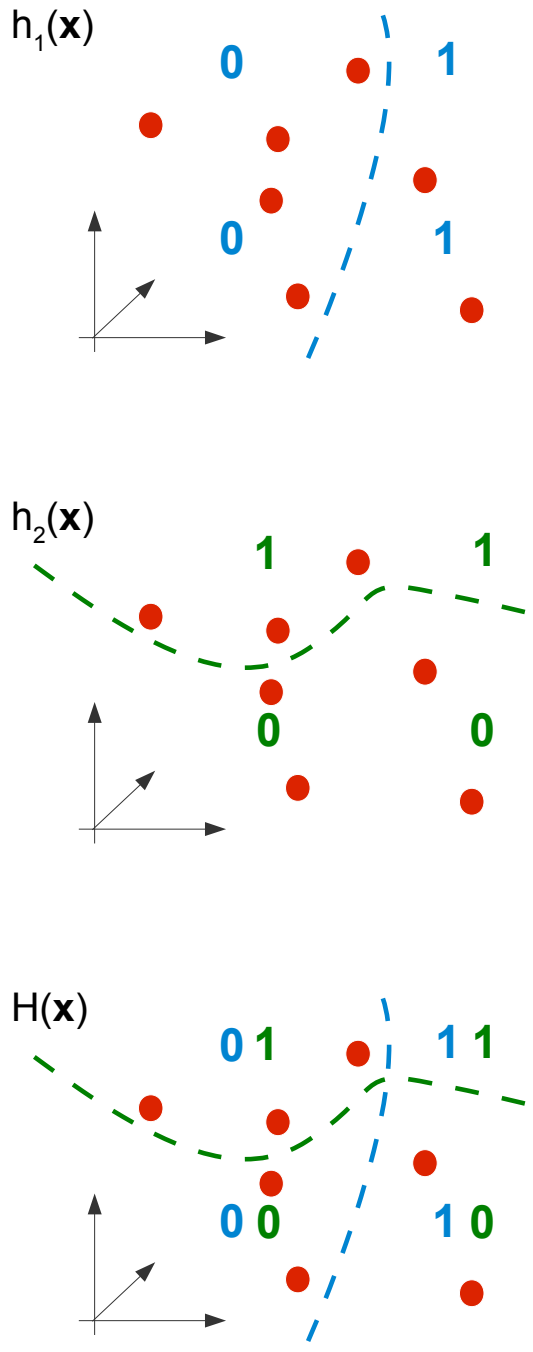
$$h : \mathbb{R}^d \to \{0, ..., N\}$$

- We usually choose $N = 1$ (i.e., hash functions = "bits")

$$h : \mathbb{R}^d \to \{0, 1\}$$

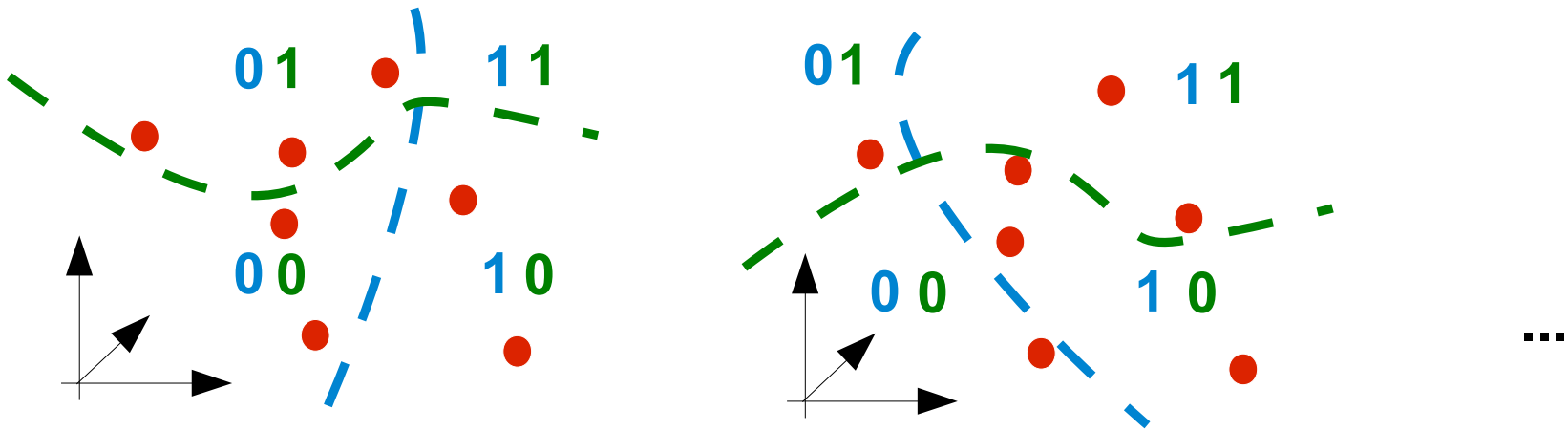
- We randomly choose k hash functions $h_1, ..., h_k$, and map each sample to a **hash code**

$$\boxed{H(\mathbf{x}) := \Big(h_1(\mathbf{x}), ..., h_k(\mathbf{x})\Big)}$$



LSH: Indexing

- Training samples $\mathbf{x_1}, ..., \mathbf{x_n}$ are stored in a **hash table**, with their hash codes $H(\mathbf{x_1}), ..., H(\mathbf{x_n})$ as keys
- We repeat this process t times, obtaining t hash codes $H_1, ..., H_t$ leading to **t (randomized) tables**



table$_1$

00	● ●
01	● ● ●
10	● ●
11	

table$_2$

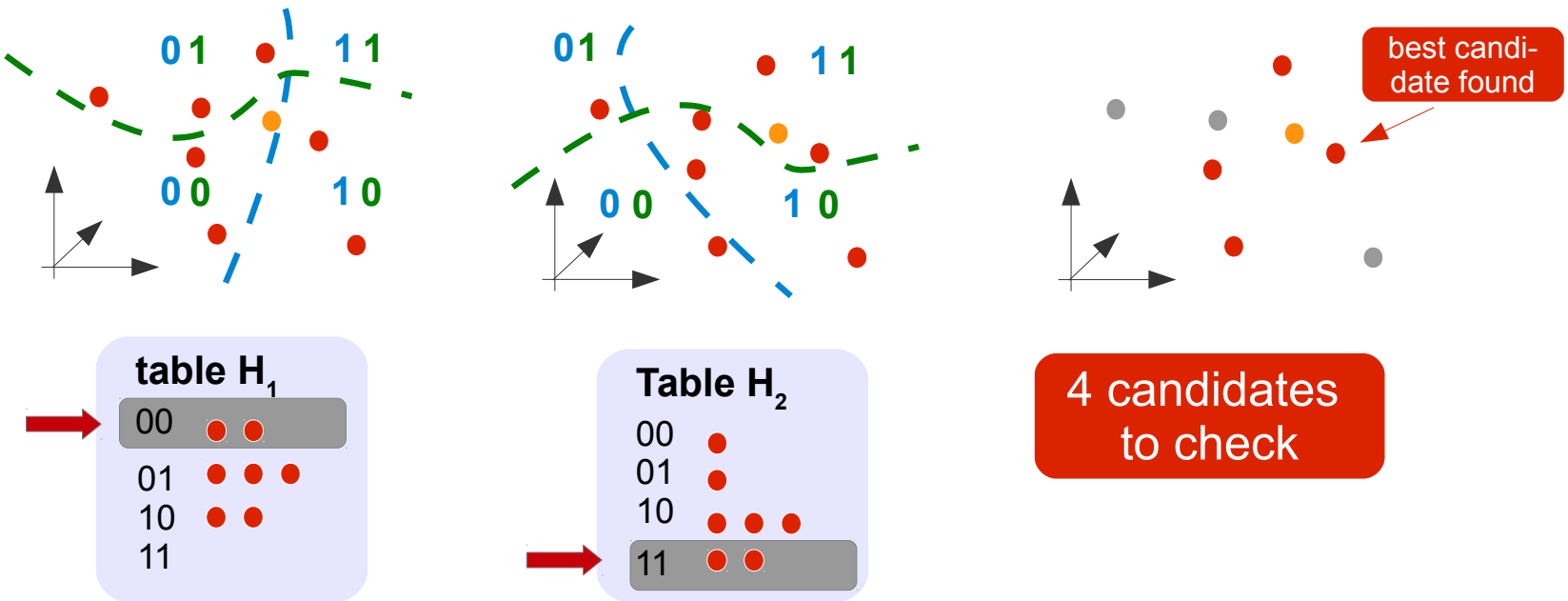
00	●
01	●
10	● ● ●
11	● ●

LSH: Search

Given a test sample **x**, we ...

- ... compute all hash codes $H_1(\mathbf{x}), ..., H_t(\mathbf{x})$
- ... lookup **candidates** in all t tables
- ... do a **linear scan** over all candidates from all tables
  (and return the best candidate found)

Example



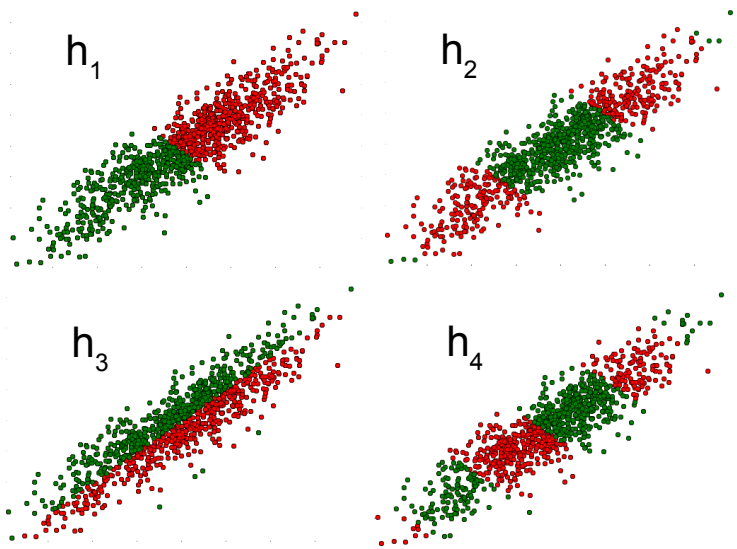
LSH: Discussion

Do-it-yourself

- What happens when increasing the number of bits k?
- What happens when increasing the number of tables t?

Outlook: Spectral Hashing [4]

- Hash functions derived from PCA
- better "goodness-of-fit" of hash functions

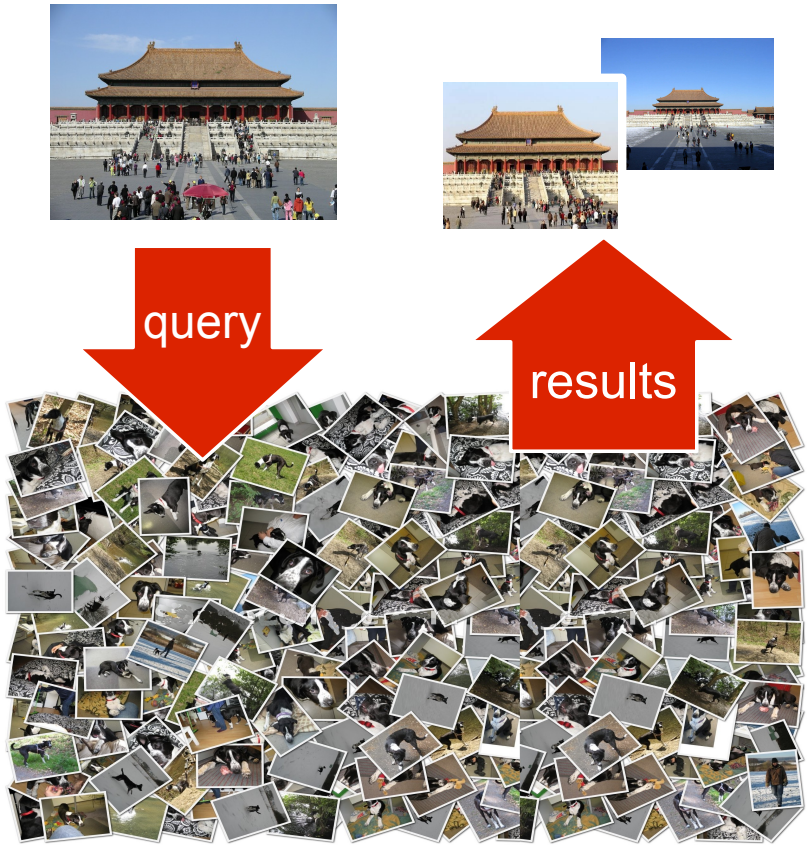


LSH: A Sample Experiment

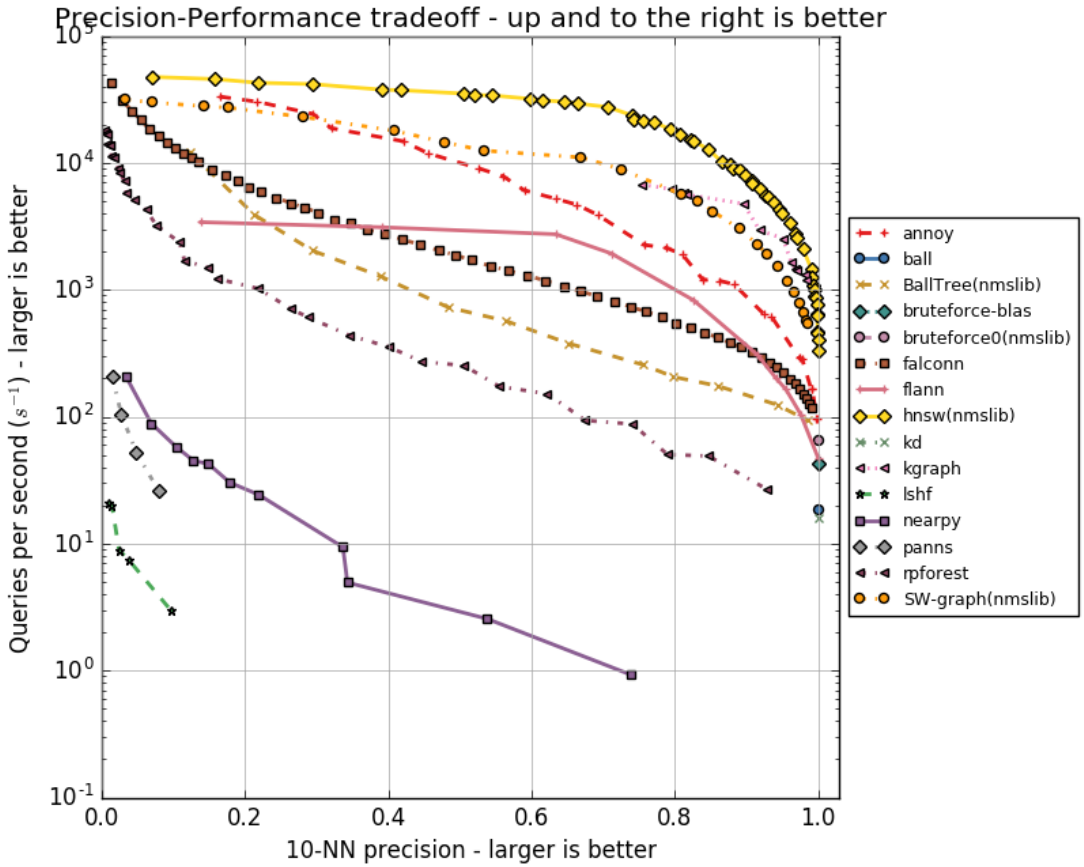
Application: Image Search

- 200,000 training images, 2,000 test images (each with 9 *targets* in the training images)
- 600-dimensional color-based **features** (color histograms, color correlograms)
- Use **LSH** to reduce the number of distance calculation (e.g., from 200,000 to 1,000)

LSH ?	-	10 bits	16 bits
time (s)	3.30	0.54	0.06
PREC@10 (%)	46.6	45.1	34.1



Approximate NN Search in Practice image from [1]



Some Nearest Neighbor Libraries

- sklearn *(not found to be very fast)*
- FLANN *(OpenCV, with Python links, but buggy)*
- annoy *(good solution, randomized trees, fast disk I/O)*

Outline

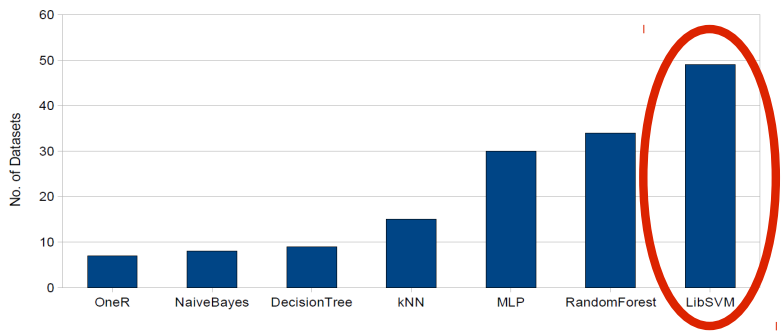
Support Vector Machines (SVMs) image from [2]

Support Vector Machines...

- ... are (still) **very popular classifiers** in machine learning
- ... have been introduced by *Vladimir Vapnik (top right)* in 1992
- ... often provide significantly better **generalization** than other classifiers
- ... follow an **instance-based** approach, similar to nearest neighbors

A Classifier Benchmark (2010) [2]

- 103 datasets from the UCI machine learning repository
- **7 classifiers** *(parameters optimized using cross-validated grid search)*
- For each classifier, count the datasets on which it is **the best**



[2] provided by Matthias Reif

Support Vector Machines (SVMs)[3]

SVMs are based on two **fundamental concepts**

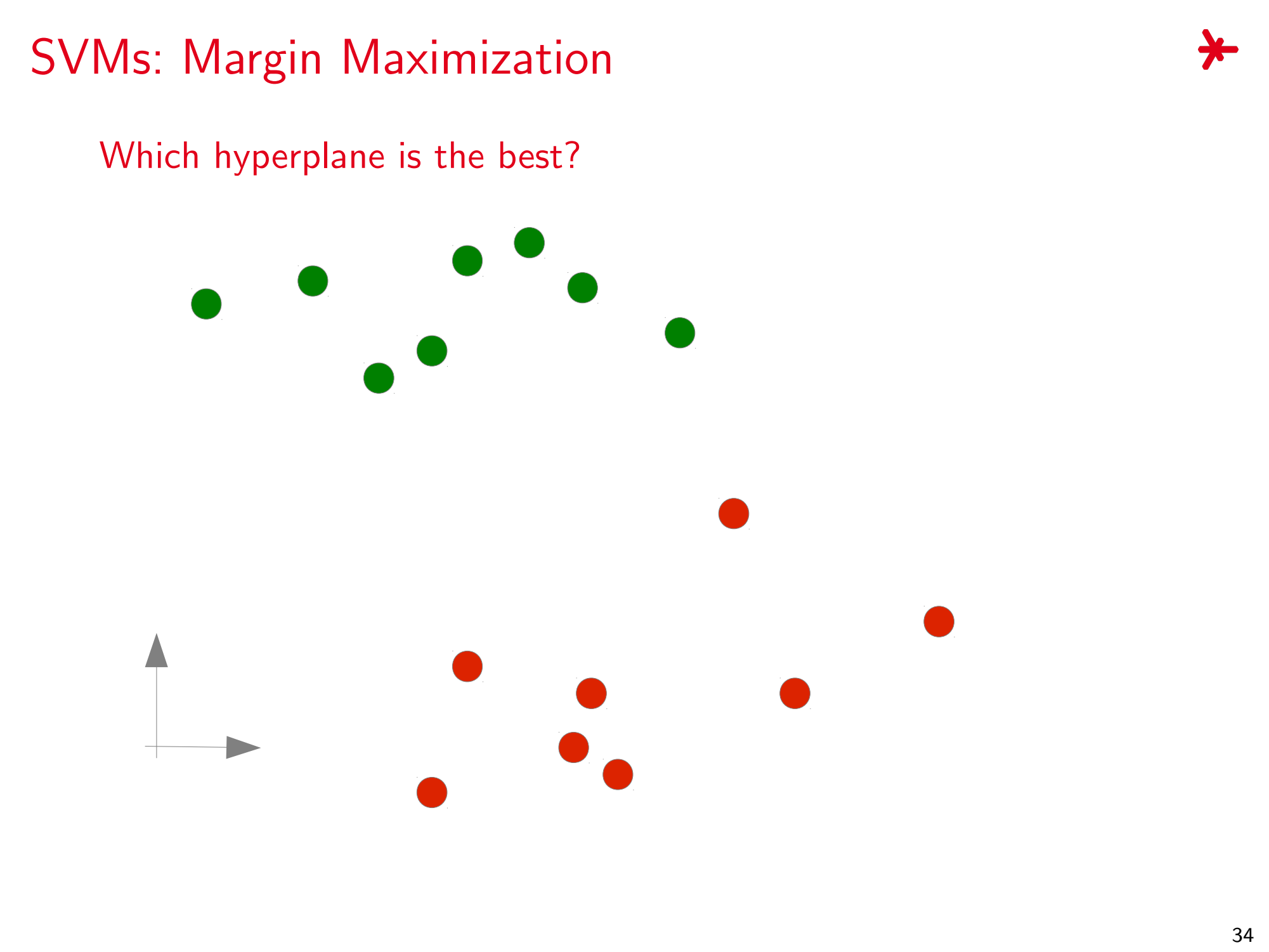
- **margin maximization**
- **kernel functions**

Formalization

- Training samples $\mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^d$
- Training labels $y_1, ..., y_n \in \{-1, 1\}$
  (multi-class problems → one-vs-rest, one-vs-one)
- **Geometric approach**: Find a separating **hyperplane**

[3] based on Christoph Lampert's excellent tutorial on Kernel methods [3]

SVMs: Margin Maximization



Which hyperplane is the best?

SVMs: Margin Maximization

To find the hyperplane $(\mathbf{w}, b)$ that maximizes the margin, we formulate a **constrained optimization problem**

- We require all samples to be on the correct side of the plane, plus a bit of *margin*
- We obtain the following **constraints**

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x}_i + b &\geq 1 \qquad \text{if } y_i = \quad 1 \\ \mathbf{w} \cdot \mathbf{x}_i + b &\leq 1 \qquad \text{if } y_i = -1 \end{aligned}$$

- Or (clever):

$$y_i \cdot \Big(\mathbf{w} \cdot \mathbf{x}_i + b \Big) \geq 1 \qquad \text{for } \underline{\text{all}}\ i = 1, .., n$$

SVMs: Margin Maximization

Formular for the Margin

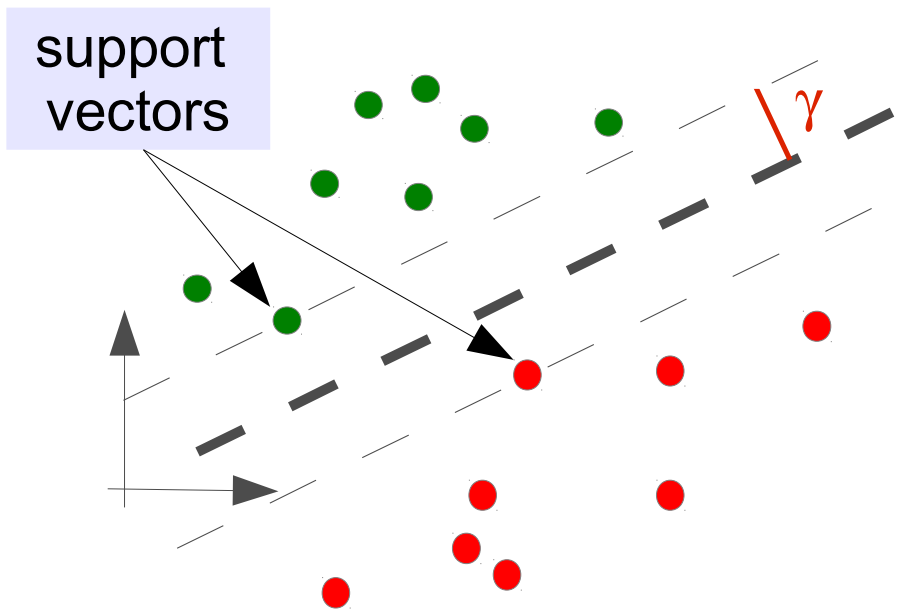
- We choose the two samples $\mathbf{x}^+$ (with label 1) and $\mathbf{x}^-$ (with label -1) "closest" to the separating hyperplane.
- We compute the "distance" of these samples orthogonal to the hyperplane:

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x}^+ + b &= 1 \\ \mathbf{w} \cdot \mathbf{x}^- + b &= -1 \\ \mathbf{w} \cdot (\mathbf{x}^+ - \mathbf{x}^-) &= 2 \\ \frac{\mathbf{w}}{||\mathbf{w}||} \cdot (\mathbf{x}^+ - \mathbf{x}^-) &= \frac{2}{||\mathbf{w}||} \end{aligned}$$

- $\frac{2}{||\mathbf{w}||}$ denotes the full "distance" from $\mathbf{x}^+$ to $\mathbf{x}^-$.
- Ergo: the margin is $\frac{1}{||\mathbf{w}||}$.

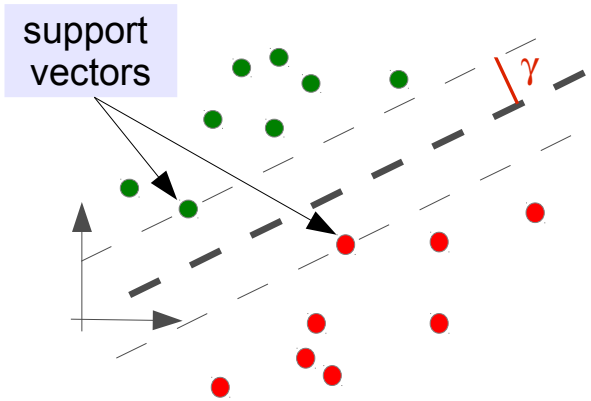
SVMs: Support Vectors

- There are **two kinds** of training samples
  1. "*safe*" samples (which are *far away* from the decision boundary, i.e. $|\mathbf{w} \cdot \mathbf{x}_i + b| > |y_i|$)
  2. **support vectors** (samples that lie on the margin, i.e. $\mathbf{w} \cdot \mathbf{x}_i + b = y_i$)
- The **decision boundary** is determined only by the support vectors (hence, **support vector** machine)



SVMs: The Margin

- Note: Geometrically, the size of the margin is: $\gamma = \frac{1}{||\mathbf{w}||}$!
- This means: Maximizing the margin is equivalent to minimizing $||\mathbf{w}||$



SVMs: Maximum-margin Problem Formulation

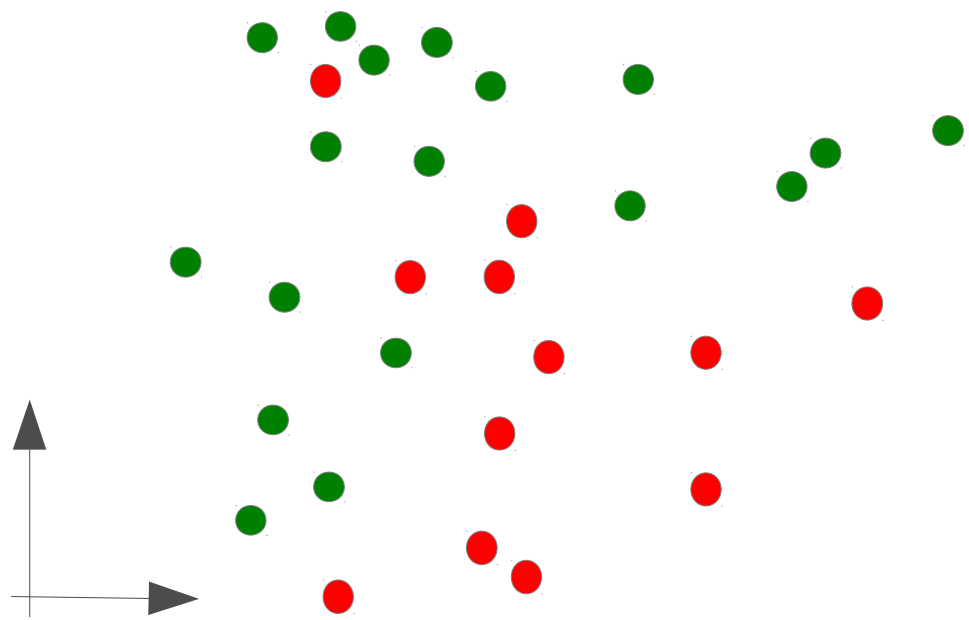
The Maximum-margin Optimization Problem

Remarks

- This is a **quadratic optimization problem** with $d + 1$ variables. The objective function is differentiable and convex.
- We can find a **global optimum**!

How to achieve Non-Linearity?

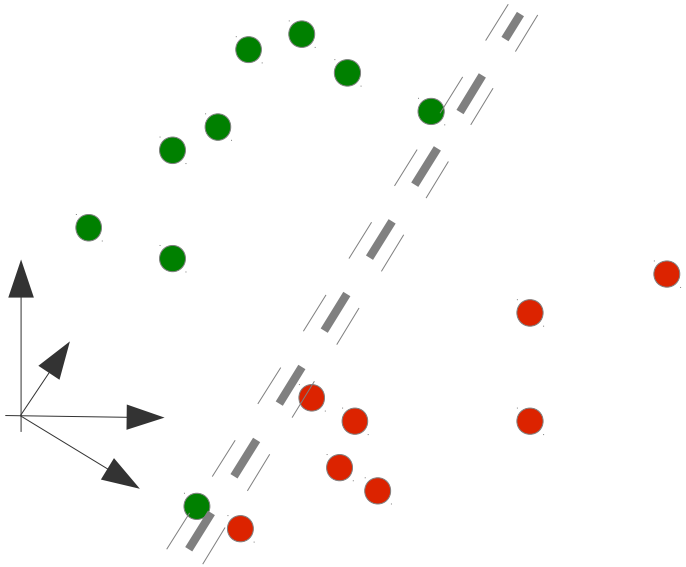
- **Problem**: Usually, datasets are not **linearly separable**
- Some **strategies** to achieve non-linearity
  1. stacking multiple classifiers *(neural networks)*
  2. slack variables *(here)*
  3. data transformation *(here)*

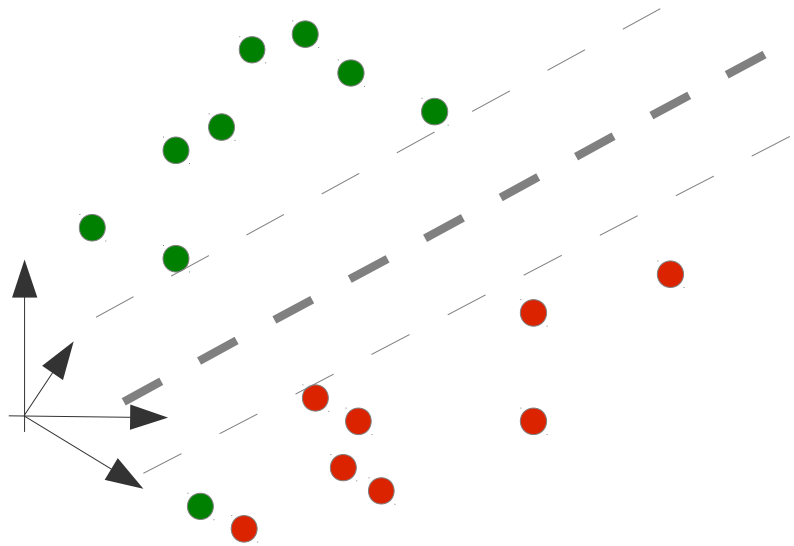


Non-Linearity 1: Slack Variables

Motivation

Which of the two decision boundaries is better?





Slack Variables: Formulation

- Idea: Allow *some* misclassifications
- Introduce so-called **slack variables** $\xi_1, ..., \xi_n \geq 0$
  (one slack variable per training sample)

Maximum-margin Formulation with **slack variables**

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b, \xi_1, \xi_2, ..., \xi_n}{\operatorname{argmin}} ||\mathbf{w}||^2 + \mathbf{C} \cdot \sum_{\mathbf{i}} \xi_{\mathbf{i}}$$

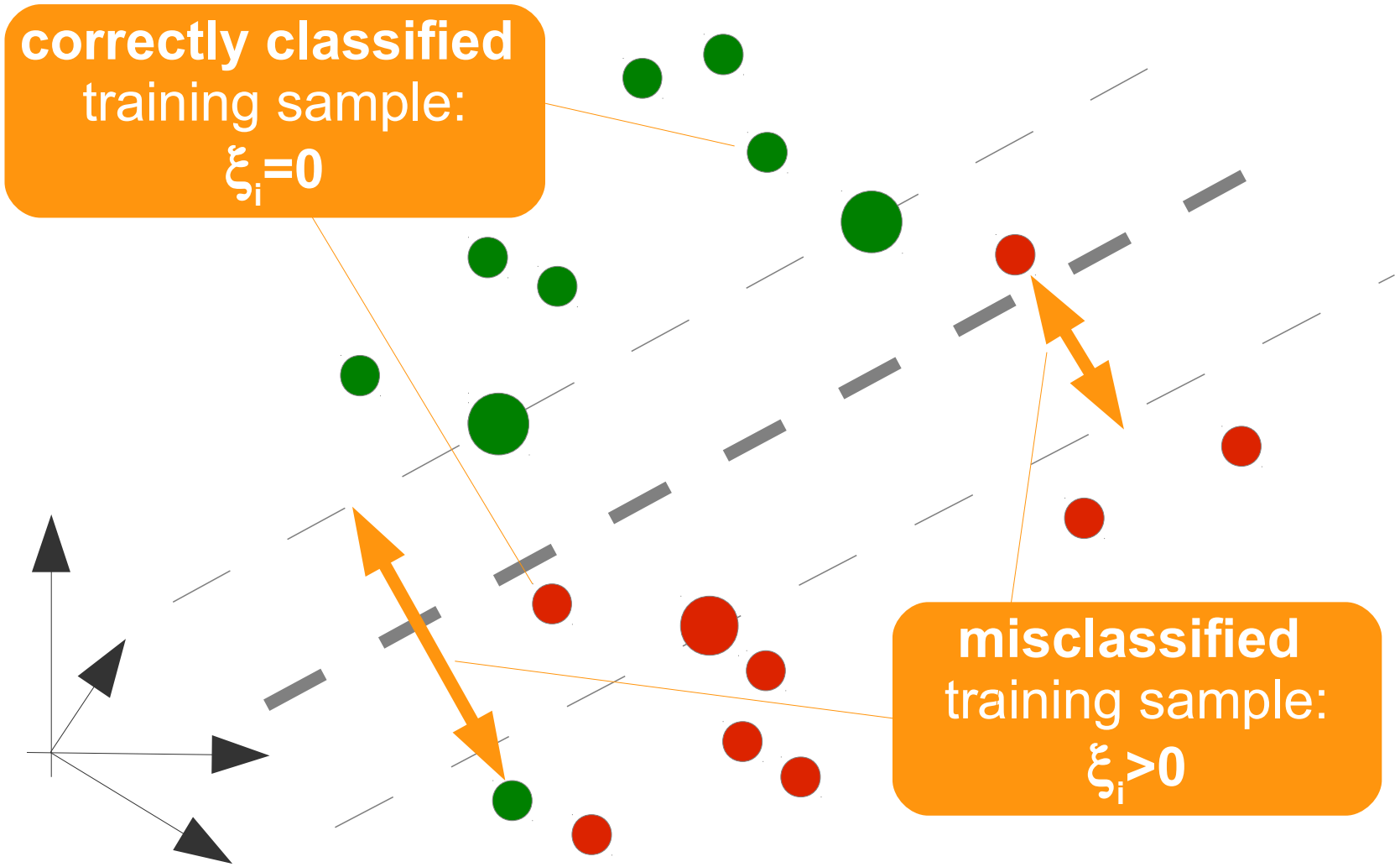
subject to:

$$y_i \cdot \Big(\mathbf{w} \cdot \mathbf{x}_i + b \Big) \geq 1 - \xi_{\mathbf{i}} \qquad \text{for } \underline{\text{all}}\ i = 1, .., n$$

Remarks

- Each slack variable ξ_i allows a training sample $\mathbf{x}_i$ to be misclassified – *at some cost*.
- The free parameter C balances the cost of misclassifications vs. margin size *(later)*.

Slack Variables: Illustration



Slack Variables

The cost factor C realizes a **trade-off** between training error and generalization

When choosing a high C ($C \to \infty$)...

- $\xi_1, ..., \xi_n \to 0$
- *hard* margin
- no training errors

When choosing a low C ($C \to 0$)...

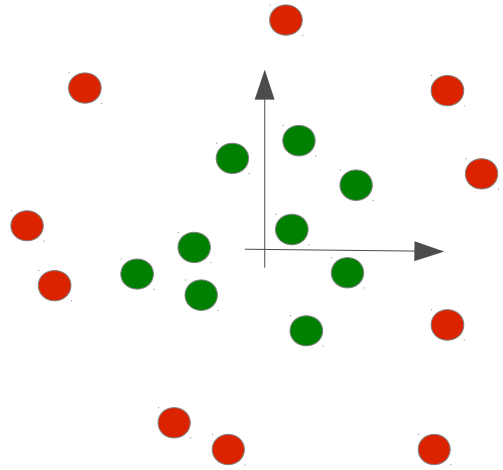
- larger, *soft* margin
- more incorrectly classified training samples

How to find a 'good' C?

- C is usually optimized using **cross-validation**
- Optimization is still 'simple', as the target function is still convex *(but there are $n + d + 1$ dimensions instead of $d + 1$: the slack variables need to be optimized too)*

Non-Linearity 2: Data Transformation

How can we transform this training set so it becomes linearly separable?



Data Transformation: Formalization

- We define a data transformation $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^m$
- We train on $\phi(\mathbf{x}_1), ..., \phi(\mathbf{x}_n)$ *(rather than* $\mathbf{x}_1, ..., \mathbf{x}_n$*)*
- We apply classification on $\phi(\mathbf{x})$ *(rather than* $\mathbf{x}$*)*

Maximum-margin Problem with Slack Variables and Data Transformation

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}\in\mathbb{R}^{\mathbf{m}}, b, \xi_1, \xi_2, ..., \xi_n}{\operatorname{argmin}} ||\mathbf{w}||^2 + C \cdot \sum_i \xi_i$$

subject to:

$$y_i \cdot \Big(\underbrace{\mathbf{w} \cdot \phi(\mathbf{x}_i)}_{=:\mathbf{k}(\mathbf{w},\mathbf{x_i})} + b \Big) \geq 1 - \xi_i \qquad \text{for all } i = 1, .., n$$

Data Transformations and the *Kernel Trick*

- In practice, finding 'good' data transformations can be **tricky**
- Often, it is easier to compute a **similarity** between samples
- We omit ϕ and use **similarity functions** $k(\mathbf{x}, \mathbf{y})$ to compare samples $\mathbf{x}$ and $\mathbf{y}$
- This approach is called the **kernel trick**. We call $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^0_+$ a **kernel function**.

"Kernelizing" our Learning Problem

The Representer Theorem

This theorem tells us that our maximum-margin solution $\mathbf{w}$ lies in the subspace spanned by the training samples, and we can rewrite it as:

$$\mathbf{w} = \sum_i \alpha_i \cdot \phi(\mathbf{x}_i) \quad \text{with } \alpha_1, ..., \alpha_n \in \mathbb{R}$$

'The SVM Problem' (=Maximum-margin Problem with Slack Variables and Kernel Functions)

SVMs: Algorithm

SVM Training

Given: training set $\mathbf{x}_1, ..., \mathbf{x}_n$ with labels $y_1, ..., y_n \in \{-1, 1\}$

1. Choose a kernel function k
2. Estimate $\alpha_1, ..., \alpha_n$ by optimizating the above SVM problem *($\alpha_i \neq 0 \Leftrightarrow \mathbf{x}_i$ is a support vector)*

SVM Classification

Given: a test sample $\mathbf{x}$

- compute $k(\mathbf{x}, \mathbf{x}_i)$ for all support vectors $\mathbf{x}_i$
- compute the classification score

$$f(\mathbf{x}) := \left(\sum_i \alpha_i \cdot k(\mathbf{x}, \mathbf{x}_i) \right) + b$$

- Classify: $\varphi(\mathbf{x}) := \begin{cases} 1 & \text{if } f(\mathbf{x}) \geq 0 \\ -1 & \text{else} \end{cases}$

Outline

Kernel Practice

Key Question: How do we choose **kernel functions** in practice?

- Some popular kernel functions

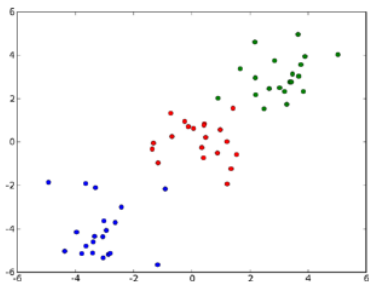
linear	$k(\mathbf{x},\mathbf{y}) := \mathbf{x}\cdot\mathbf{y} = \sum_{i=1}^{d} x_i y_i$
polynomial	$k(\mathbf{x},\mathbf{y}) := (\mathbf{x}\cdot\mathbf{y})^p = \left(\sum_{i=1}^{d} x_i y_i\right)^p$
radial basis function (RBF)	$k(\mathbf{x},\mathbf{y}) := exp\left(-\frac{\|\|\mathbf{x}-\mathbf{y}\|\|^2}{\beta}\right)$
histogram intersection	$k(\mathbf{x},\mathbf{y}) := \sum_{i=1}^{d} min(x_i, y_i)$
χ^2 kernel	$k(\mathbf{x},\mathbf{y}) := exp\left(-\frac{1}{\beta}\sum_{i=1}^{d}\frac{(x_i-y_i)^2}{(x_i+y_i)^2}\right)$ (with $\frac{0}{0} := 0$)

- You can also define **application-specific kernels** for your own type of data (e.g., strings)
- We can construct kernels from **distance functions**: if $d(.,.)$ is a distance function, then $e^{-d(.,.)}$ can be used as a kernel function

Kernel Practice image from [3]

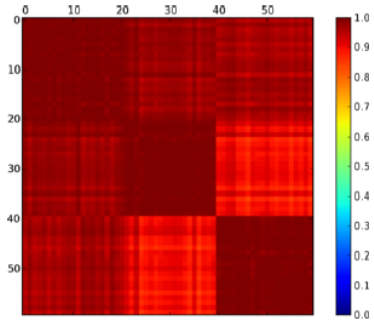
$$k(x,y) = exp\left(-\frac{||x-y||^2}{\beta}\right)$$

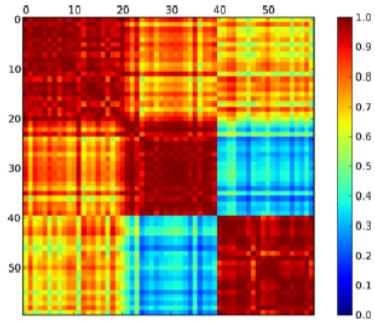
- Some kernels have *parameters* (example: β in the RBF kernel)
- In general, we want kernels to **separate classes well**
- Often a good choice *(bottom right)*: $\beta := \frac{1}{n^2}\sum_{i,j=1}^{n}||x_i - x_j||^2$

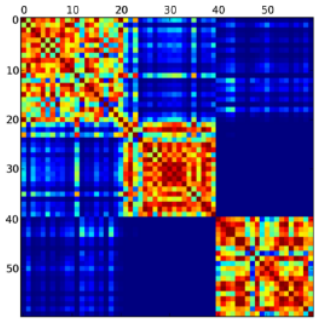


input data

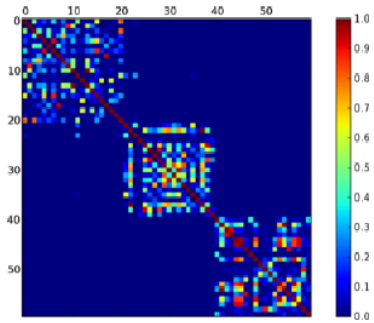
$x_1, \ldots, x_n$

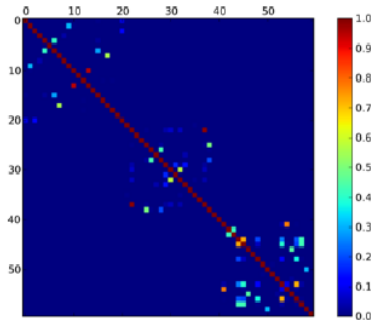




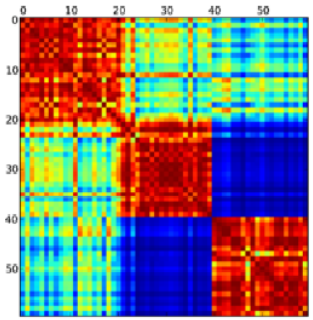


β large...

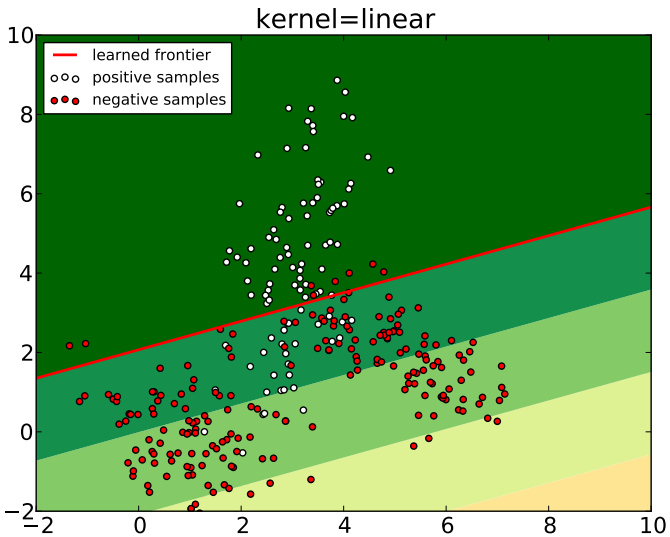


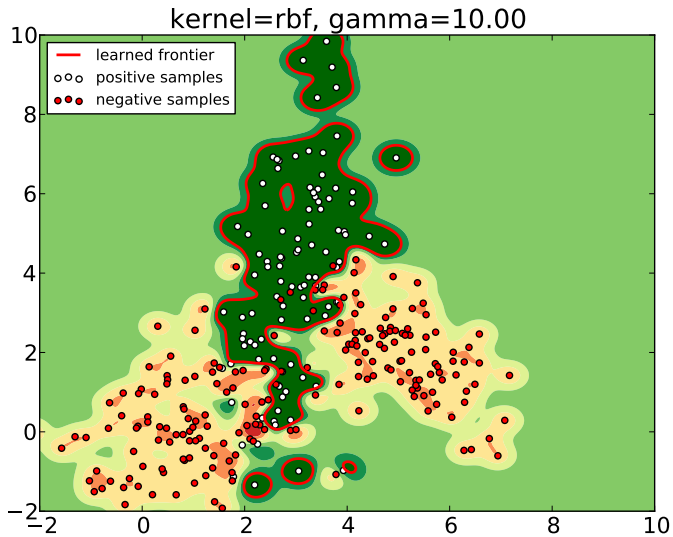


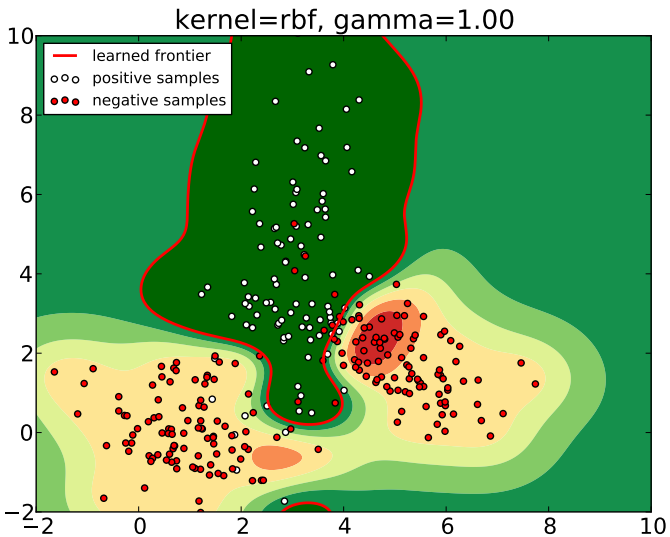
...β small

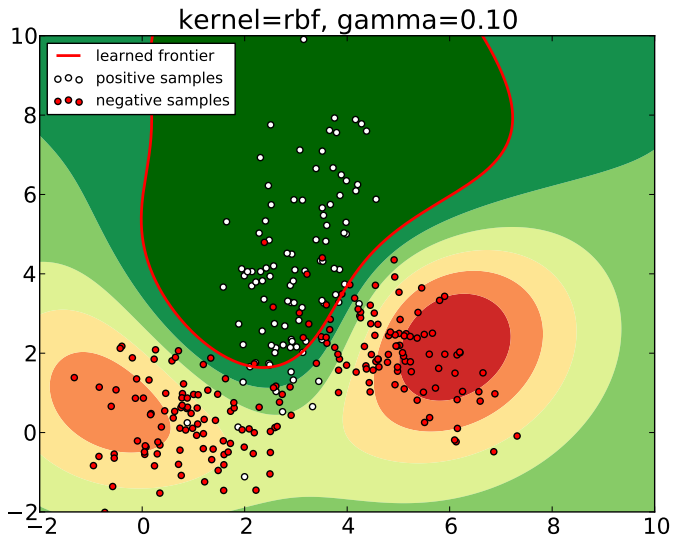


SVM Example (sklearn)







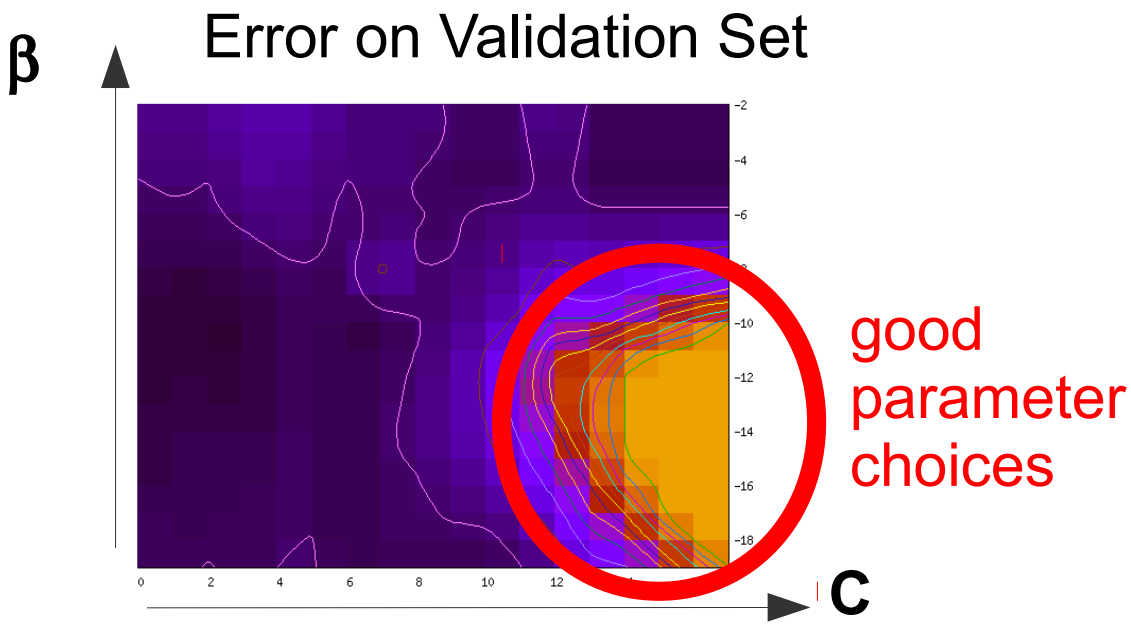


SVMs: Parameter optimization

SVMs usually contain **free parameters**, like C *(weight of slack variables)* and β *(kernel parameter)*

Standard Approach: Grid Search

- test different choices for C and β on regular steps *(a grid)*
- for each (C, β): measure classification accuracy on a held-out validation set, or using cross-validation



SVMs: Unbalanced Training Data

- Sometimes, training sets are highly **imbalanced**
  (e.g., $n_1 = 10$ *positive samples,* $n_{-1} = 10000$ *negative ones)*
- When training an SVM on such data, we may obtain **degenerate solutions**

Strategy 1: Subsampling

- **Subsample** training samples **class-wise** such that they become balanced

Strategy 2: Class-specific Cost

- Replace C with **class-specific cost** C_1, C_{-1}, such that $n_1 \cdot C_1 = n_{-1} \cdot C_{-1}$
- Formally:

$$\alpha_1^*, ..., \alpha_n^*, b = \arg\min_{...} \; ... + C_1 \cdot \sum_{i:y_i=1} \xi_i \; + C_{-1} \cdot \sum_{i:y_i=-1} \xi_i$$

SVM Software

- We have not tackled how to **solve the optimization problems** we formulated. SVM software will do it for you.
- Core software packages exist in C *(libsvm, svmlight)*
- Bindings to python, R, matlab, etc. exist
  (check out scikit-learn)
- Those packages include common **kernel functions**, but also allow you to define your own kernels!

References

[1] E. Bernhardsson.
Benchmark of Approximate Nearest Neighbor libraries.
https://erikbern.com/2015/07/04/benchmark-of-approximate-nearest-neighbor-libraries/ (retrieved: Nov 2016).

[2] Columbia Engineering, The Fu Foundation.
Vladimir Vapnik – Unlocking a Complex World Mathematically.
http://engineering.columbia.edu/files/engineering/Excellentia.pdf (retrieved: Nov 2016).

[3] C. Lampert and M. Blaschko.
Kernel Methods in Computer Vision.
http://www.robots.ox.ac.uk/~blaschko/CVPRTutorial2009/kernel_tutorial-Part1.pdf (retrieved: Nov 2016).

[4] Y. Weiss, A. Torralba, and R. Fergus.
Spectral Hashing.
In Ann. Conf. on Neural Information Processing Systems (NIPS), pages 1753–1760, 2008.