Human-Oriented Robotics

Supervised Learning

Part 3/3

Kai Arras

Social Robotics Lab, University of Freiburg

Winter term 2014/2015

Supervised Learning

Contents

- Introduction and basics
- Bayes Classifier
- Logistic Regression
- Support Vector Machines
- AdaBoost
- k-Nearest Neighbor
- Cross-validation
- Performance measures

Supervised Learning

Ensemble Learning

- So far, we have looked at learning methods in which a **single hypothesis** h for $y = f(\mathbf{x})$ is used to make predictions
- The underlying idea of **ensemble learning** is to select a collection, or **ensemble**, of hypotheses and combine their predictions
- Consider, for instance, an ensemble of $K = 5$ hypotheses and suppose that we combine their predictions using simple majority voting. For the ensemble to misclassify a new sample, at least 3 of 5 hypotheses have to be wrong. This is **much less likely** than a mistake by a single hypothesis
- **Boosting** is the most widely used ensemble learning method. In boosting, simple "rules" or base classifiers are trained in sequence in a way that the performance of the ensemble members is improved, i.e. "boosted"
- Other ensemble methods include bagging, mixture of experts, voting

AdaBoost

Ensemble Learning

- **AdaBoost** is the most popular boosting algorithm
- It learns an **accurate strong classifier** by combining an ensemble of **inaccurate** "rules of thumb"
  - **Inaccurate rule** $h(\mathbf{x})$: **weak classifier** (a.k.a. weak learner, base classifier, feature)
  - **Accurate rule** $H(\mathbf{x})$: **strong classifier**
- Given an ensemble of weak classifiers $h_{1...K}(\mathbf{x})$ the combined strong classifier $H(\mathbf{x})$ is obtained by a **weighted majority voting scheme**

$$f(\mathbf{x}_i) = \sum_{k=1}^{K} \alpha_k h_k(\mathbf{x}_i)$$

Confidence

$$H(\mathbf{x}_i) = sign(f(\mathbf{x}_i))$$

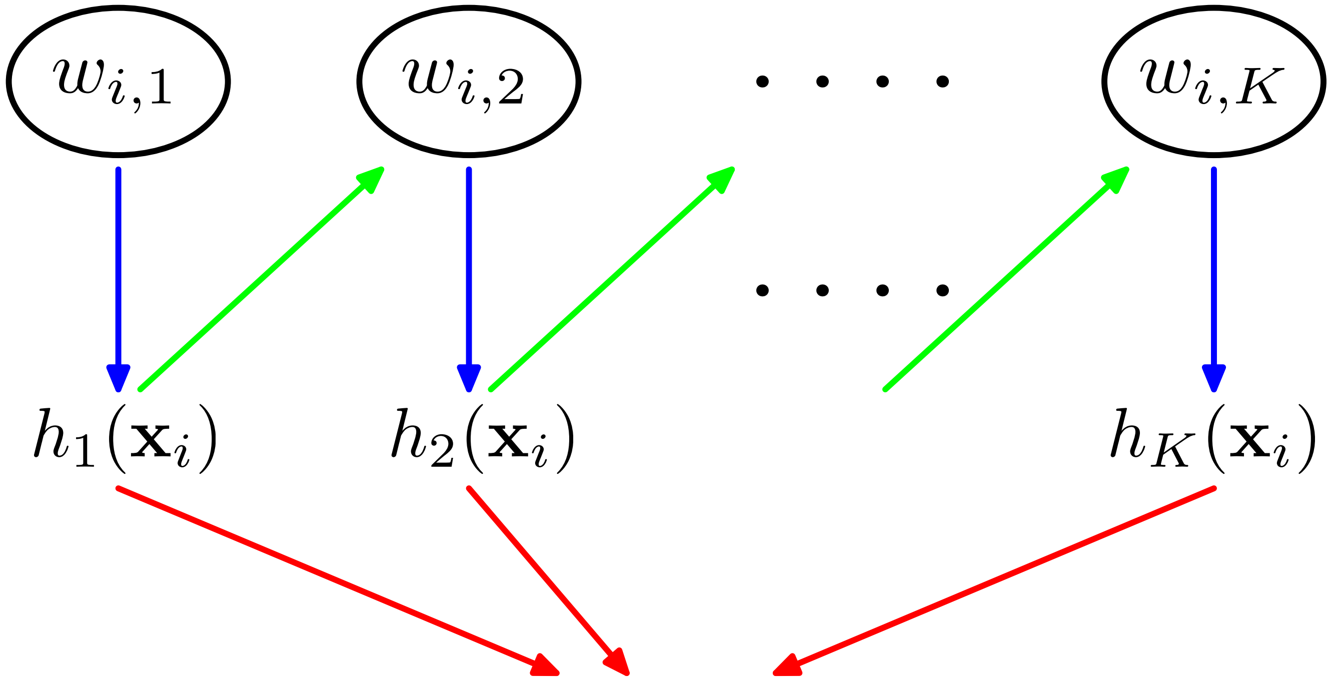
Strong classifier

AdaBoost

Boosting

- Boosting methods define a **weight distribution** over the training samples

- Each weak classifier is trained on **weighted training data** (blue arrows) in which the weights **depend** on the **performance of the previous** weak classifier (green)
- Once all classifiers have been learned, they are **combined** to give a **strong classifier** (red)



$$y' = H(\mathbf{x}') = sign\left(\sum_{k=1}^{K} \alpha_k h_k(\mathbf{x}')\right)$$

Source [4]

AdaBoost

Boosting

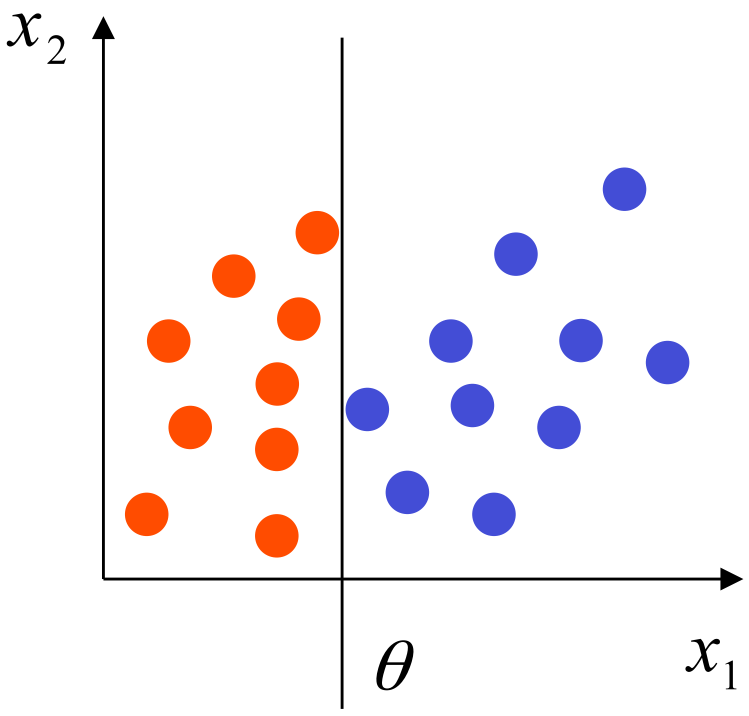
- Weak classifier examples
  - **Decision stump:** single axis-parallel partition of space
  - **Decision tree:** hierarchical partition of space
  - **Multi-layer perceptron:** general non-linear function approximators
  - **Support vector machines:** maximum-margin classifier
- There is a trade-off between **diversity** among weak learners versus their **accuracy**
  - Diversity (“negative correlation”) among weak classifiers is a necessary condition for better ensemble performance
  - Highly accurate weak learners will make very similar predictions
- **Decision stumps** are a popular choice

AdaBoost

Decision Stump

- Simple-most type of **decision tree**
- Linear classifier defined by an **axis-parallel hyperplane** with parameters θ and d
- Hyperplane is orthogonal to axis/dimension d with which it intersects orthogonally at threshold value θ
- Rarely useful on its own due to its simplicity
- Formally,

$$h(\mathbf{x}; d, \theta) = \begin{cases} +1 & x_d > \theta \\ -1 & \text{else} \end{cases}$$



where $\mathbf{x}$ is an m-dimensional training sample, d is the dimension

AdaBoost

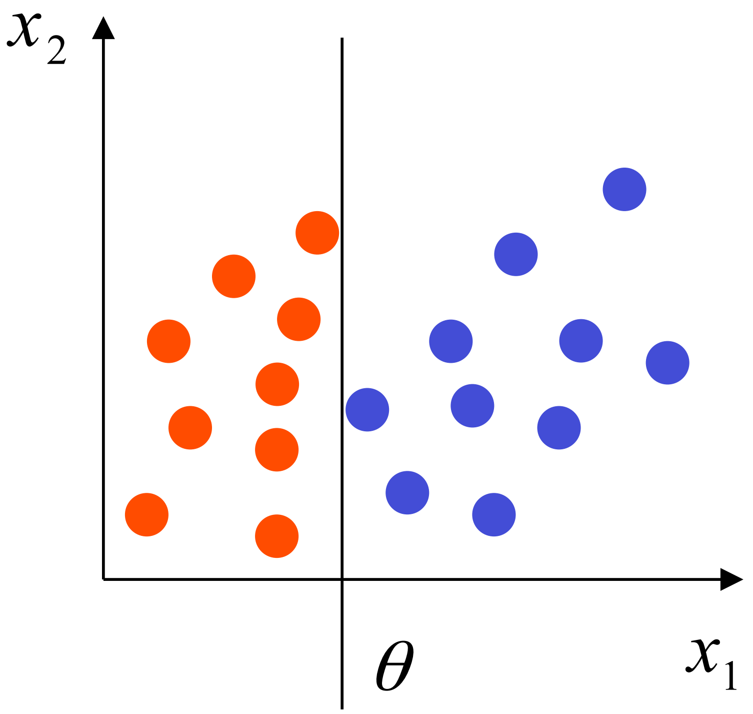
Decision Stump

- **Learning objective** of decision stumps on **weighted data**

$$\theta^*, d^* = \arg\min_{\theta,d} \sum_{i=1}^{N} w_i \, \mathrm{I}(y_i \neq h(\mathbf{x}_i))$$

where I(.) is the indicator function

$$\mathrm{I}(x) = \begin{cases} 1 & x \text{ is true} \\ 0 & x \text{ is false} \end{cases}$$



- The goal is to find parameters θ*, d* that **minimize the weighted error**

AdaBoost

Decision Stump

Learning algorithm for decision stumps on weighted data

- For $d = 1 \dots m$
  1. **Sort** samples $\mathbf{x}_i$ in ascending order along dimension d
  2. For $i = 1 \dots N$

     Compute N **cumulative sums** $w_i^{cum}(d) = \sum_{j=1}^{i} w_j \, y_j$
  3. Threshold θ_d is at extremum of $w^{cum}(d)$
  4. **Sign** of extremum gives direction p_d of inequality

- Global extremum in all m cumulative sums gives **optimal threshold** θ^* and **dimension** d^*

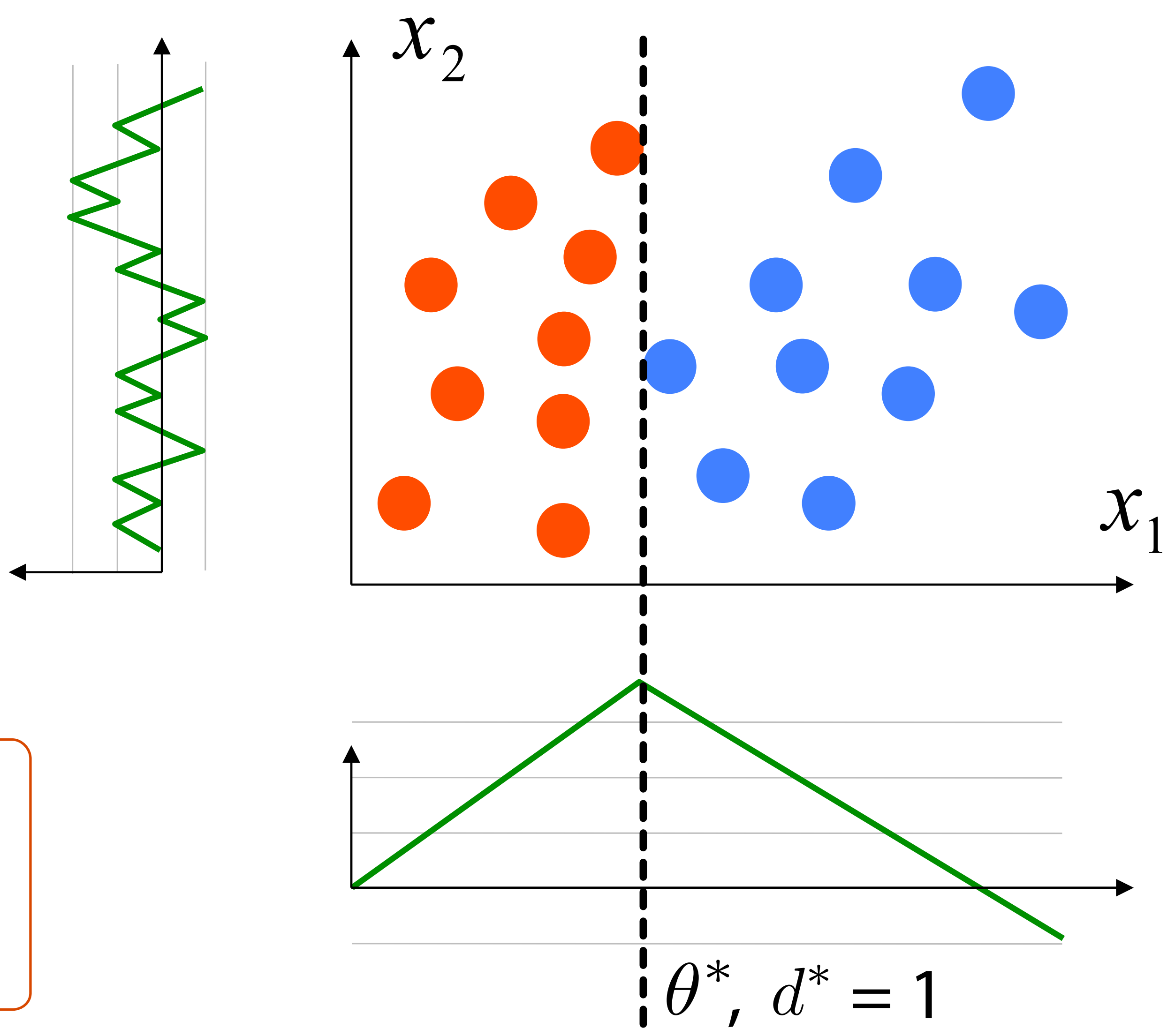
AdaBoost

Decision Stump

Learning algorithm for decision stumps on weighted data

- **Label** y :
  red: $+1$
  blue: -1

- Assume all weights = 1



$$\boxed{w_i^{cum}(d) = \sum_{j=1}^{i} w_j\, y_j}$$

AdaBoost

Learning

Given training set $(\mathbf{x}_1, y_1),\ (\mathbf{x}_2, y_2),\ \cdots\ (\mathbf{x}_N, y_N)$, **learn a strong classifier**

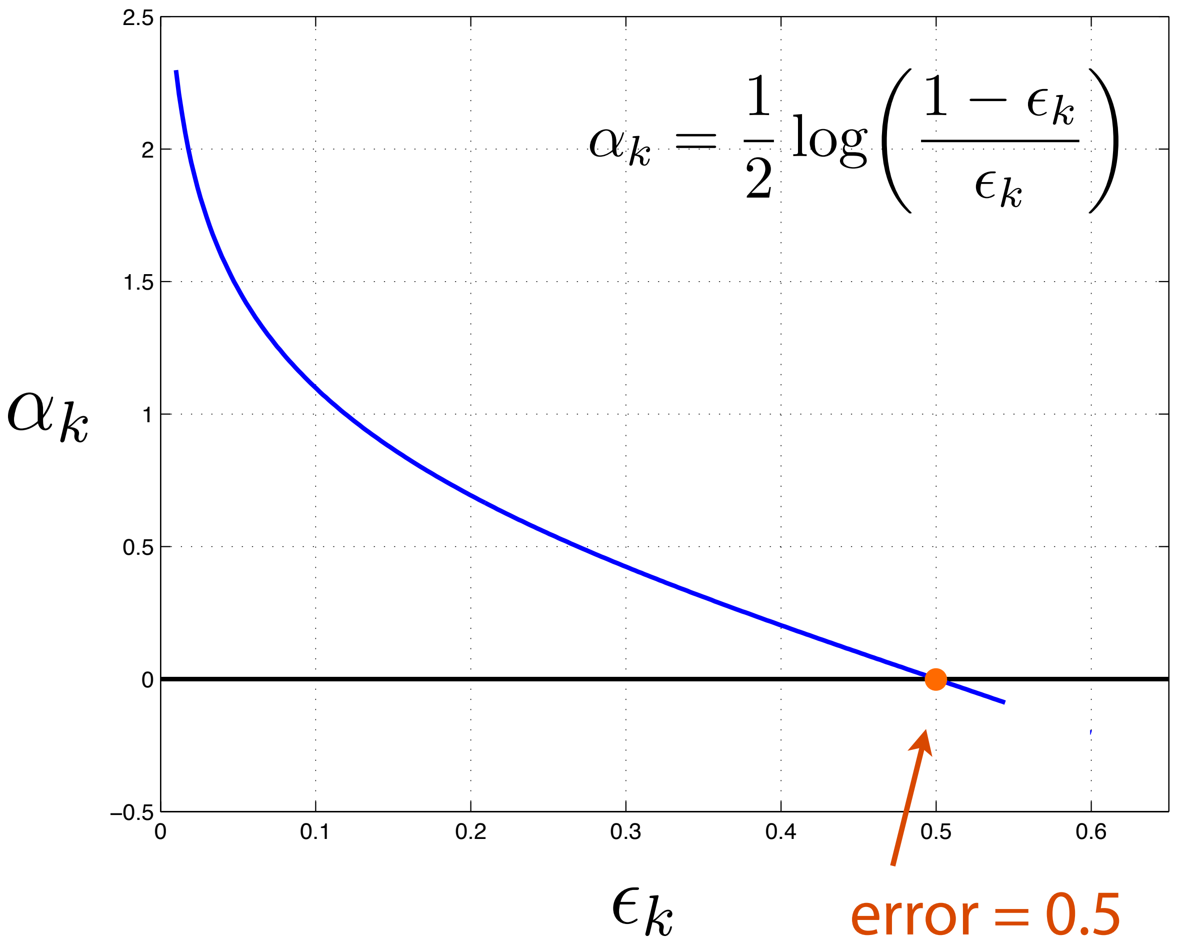
- **Initialize** weights $w_{i,0} = \frac{1}{N} \quad \forall i$
- For $k = 1 \ldots K$

  1. **Learn a weak classifier** $h_k(\mathbf{x})$ on weighted training data minimizing the error $\epsilon_k = \sum_i w_{i,k}\, \mathrm{I}(y_i \neq h_k(\mathbf{x}_i))$
  2. Compute **voting weight of** $h_k(\mathbf{x})$ as $\alpha_k = \frac{1}{2} \log \left(\frac{1-\epsilon_k}{\epsilon_k} \right)$
  3. **Recompute weights** $w_{i,k+1} = w_{i,k} \dfrac{\exp(-\alpha_k y_i h_k(\mathbf{x}_i))}{Z_k}$

AdaBoost

Learning

- Voting weight α_k of a weak classifier as a **function of the error** ϵ_k
- α_k measures the **importance** of classifier $h_k(\mathbf{x})$ and corresponds to the **strength** of its **vote** in the strong classifier
- The expression yields the **optimal voting weight.** Proven later.



- Notice, **training samples** are weighted by weight $w_{i,k}$, **weak classifiers** are weighted by voting weight α_k

AdaBoost

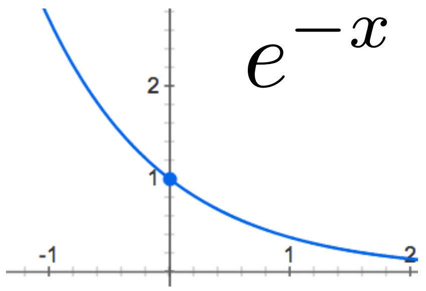
Learning

- Let us take a closer look at the **weight update step**

$$w_{i,k+1} = w_{i,k} \frac{\exp(-\alpha_k y_i h_k(\mathbf{x}_i))}{Z_k}$$

- From

$$\exp(-\alpha_k y_i h_k(\mathbf{x}_i)) = \begin{cases} < 1 & \text{if } \ y_i = h_k(\mathbf{x}_i) \\ > 1 & \text{if } \ y_i \neq h_k(\mathbf{x}_i) \end{cases}$$



  we see that weights of **misclassified** training samples are **increased** and weights of **correctly** classified samples are **decreased**

- Normalizer Z_k makes the weight distribution a probability distribution
- Thus, the learning algorithm generates weak classifier by training the next classifier on the **mistakes of the previous one**
- Hence the name: AdaBoost is derived from **ada**ptive **Boost**ing

AdaBoost

Inference and Decision

- After the learning phase, predictions of new data $\mathbf{x}'$ are made by the **weighted majority voting scheme** of the strong classifier

$$y' = H(\mathbf{x}') = sign\left(\sum_{k=1}^{K} \alpha_k h_k(\mathbf{x}')\right) \qquad y' \in \{-1, +1\}$$

- The **learned model** consists in the K weak learner $h_{1\ldots K}(\mathbf{x})$ with associated voting weights $\alpha_{1\ldots K}$

Learning: why does it work?

- The goal for the strong classifier is to **minimize the training error** defined as the number of misclassified training pairs

$$\mathcal{E} = \frac{1}{N} \sum_{i=1}^{N} \begin{cases} 1 & \text{if } H(\mathbf{x}_i) \neq y_i \\ 0 & \text{if } H(\mathbf{x}_i) = y_i \end{cases}$$

- Using the indicator function

$$\mathrm{I}(x) = \begin{cases} 1 & x \text{ is true} \\ 0 & x \text{ is false} \end{cases}$$

  we can rewrite the error as

$$\mathcal{E} = \frac{1}{N} \sum_{i=1}^{N} \mathrm{I}(H(\mathbf{x}_i) \neq y_i)$$

- Remember our definitions of the confidence $f(\mathbf{x}_i) = \sum_{k=1}^{K} \alpha_k h_k(\mathbf{x}_i)$, the strong classifier $H(\mathbf{x}_i) = sign(f(\mathbf{x}_i))$ and labels $y \in \{-1, +1\}$

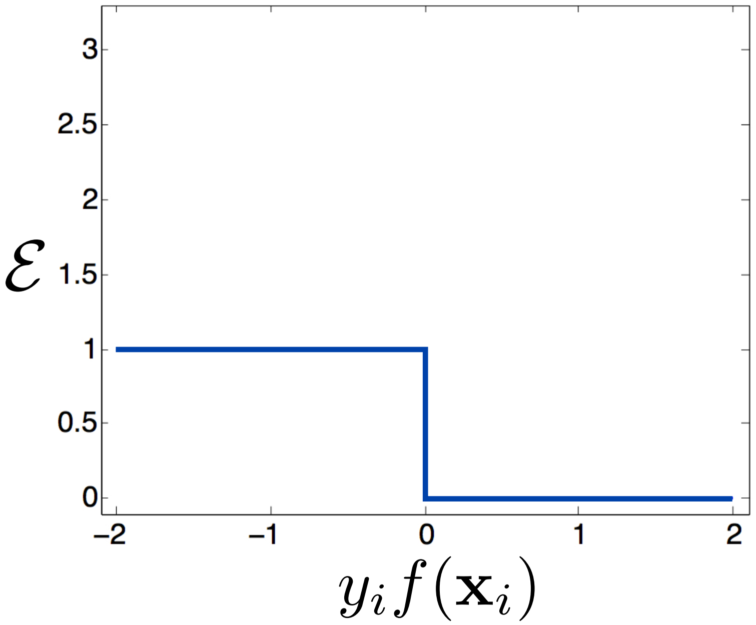
Learning: why does it work?

- Then, we see that $H(\mathbf{x}_i) \neq y_i$ implies $y_i\, f(\mathbf{x}_i) \leq 0$ and the error becomes

$$\mathcal{E} = \frac{1}{N} \sum_{i=1}^{N} \mathrm{I}(y_i f(\mathbf{x}_i) \leq 0)$$

Often called 0/1-loss function

- Plotting the error for the case of a single sample shows that the function is non-differentiable and **difficult to handle** mathematically
- **Idea:** because minimizing the training error directly is difficult, we define an **upper bound** and **minimize this bound** instead



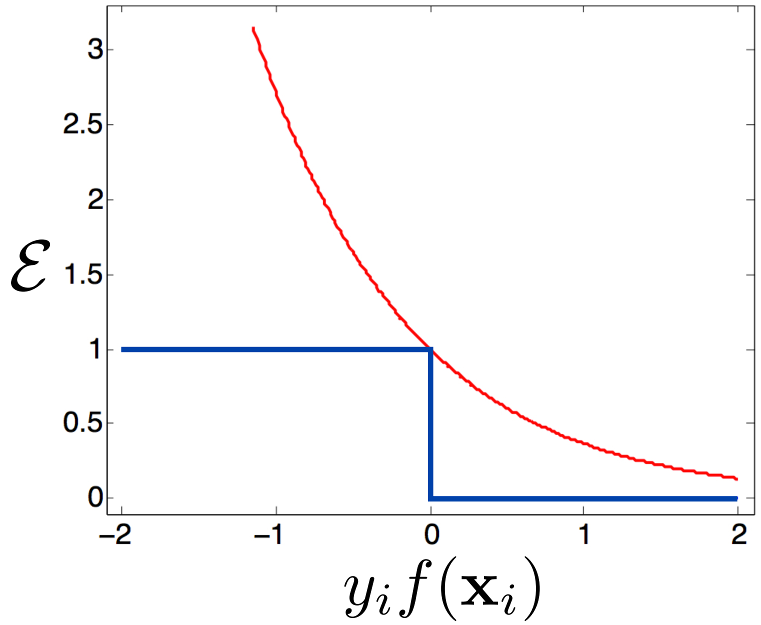
Learning: why does it work?

- Then, we see that $H(\mathbf{x}_i) \neq y_i$ implies $y_i\, f(\mathbf{x}_i) \leq 0$ and the error becomes

$$\mathcal{E} = \frac{1}{N} \sum_{i=1}^{N} \mathrm{I}(y_i f(\mathbf{x}_i) \leq 0)$$

Often called 0/1-loss function

- Plotting the error for the case of a single sample shows that the function is non-differentiable and **difficult to handle** mathematically
- **Idea:** because minimizing the training error directly is difficult, we define an **upper bound** and **minimize this bound** instead
- Using the **exponential loss function** we have for a single sample



$$\mathrm{I}(y_i f(\mathbf{x}_i) \leq 0) \;\leq\; \exp(-y_i f(\mathbf{x}_i))$$

Learning: why does it work?

- The upper bound holds for **all** training samples

$$\mathcal{E} = \frac{1}{N}\sum_{i=1}^{N} \mathrm{I}(H(\mathbf{x}_i) \neq y_i) \;\leq\; \frac{1}{N}\sum_{i=1}^{N} \exp(-y_i f(\mathbf{x}_i))$$

- To proceed from here, we consider the weight update equation and **unravel it recursively** from the back for $k = K$

$$\begin{aligned} w_{i,K+1} &= w_{i,K}\frac{\exp(-\alpha_K y_i h_K(\mathbf{x}_i))}{Z_K} \\ &= w_{i,K-1}\frac{\exp(-\alpha_{K-1} y_i h_{K-1}(\mathbf{x}_i))\exp(-\alpha_K y_i h_K(\mathbf{x}_i))}{Z_{K-1}Z_K} \\ &\;\vdots \\ &= \frac{\exp(-\sum_k \alpha_k y_i h_k(\mathbf{x}_i))}{N\prod_k Z_k} = \frac{\exp(-y_i f(\mathbf{x}_i))}{N\prod_k Z_k} \end{aligned}$$

From $k = 0$

AdaBoost

Learning: why does it work?

- Substitution into the error bound yields

$$\begin{aligned}\mathcal{E} = \frac{1}{N}\sum_{i=1}^{N} \mathrm{I}(H(\mathbf{x}_i) \neq y_i) \;\; &\leq \;\; \frac{1}{N}\sum_{i=1}^{N} N(\textstyle\prod_k Z_k)\, w_{i,K+1} \\ &= \;\; (\textstyle\prod_k Z_k) \displaystyle\sum_{i=1}^{N} w_{i,K+1} \\ &= \;\; \textstyle\prod_k Z_k\end{aligned}$$

- Minimizing the upper bounds is equivalent to **minimizing the product of the K normalizers** or the Z_k in each training round, respectively

$$Z_k = \sum_{i=1}^{N} w_{i,k} \exp(-\alpha_k y_i h_k(\mathbf{x}_i))$$

- This in turn is achieved by choosing the **optimal weak classifier** $h(\mathbf{x})$ and finding the **optimal voting weight** α

AdaBoost

Learning: why does it work?

- **First**, let us go for the **optimal voting weight** α
- To **minimize** $Z_k = \sum_{i=1}^{N} w_{i,k} \exp(-\alpha_k y_i h_k(\mathbf{x}_i))$ we partially differentiate it w.r.t. α and set the derivative to zero (skipping round index k)

$$\frac{\partial Z}{\partial \alpha} = -\sum_i w_i \, y_i h(\mathbf{x}_i) e^{-\alpha \, y_i h(\mathbf{x}_i)} \; = \; 0$$

- Next, we subdivide the sum into a sum over the **correctly predicted samples** (for which $y_i h(\mathbf{x}_i) = 1$) and a sum over the **misclassified samples** (for which $y_i h(\mathbf{x}_i) = -1$)

$$- \sum_{i:y_i = h(\mathbf{x}_i)} w_i \, e^{-\alpha} + \sum_{i:y_i \neq h(\mathbf{x}_i)} w_i \, e^{\alpha} \; = \; 0$$

$$-e^{-\alpha}(1-\epsilon) + e^{\alpha}\epsilon \; = \; 0$$

Learning: why does it work?

- The last step uses the definition of the **error** ϵ **for weak learners** to be the weighted sum over all misclassified training samples. We finally find

$$\begin{aligned} -e^{-\alpha}(1-\epsilon) + e^{\alpha}\epsilon &= 0 \\ e^{2\alpha}\epsilon &= 1-\epsilon \end{aligned}$$

$$\boxed{\alpha = \frac{1}{2}\log\left(\frac{1-\epsilon}{\epsilon}\right)}$$

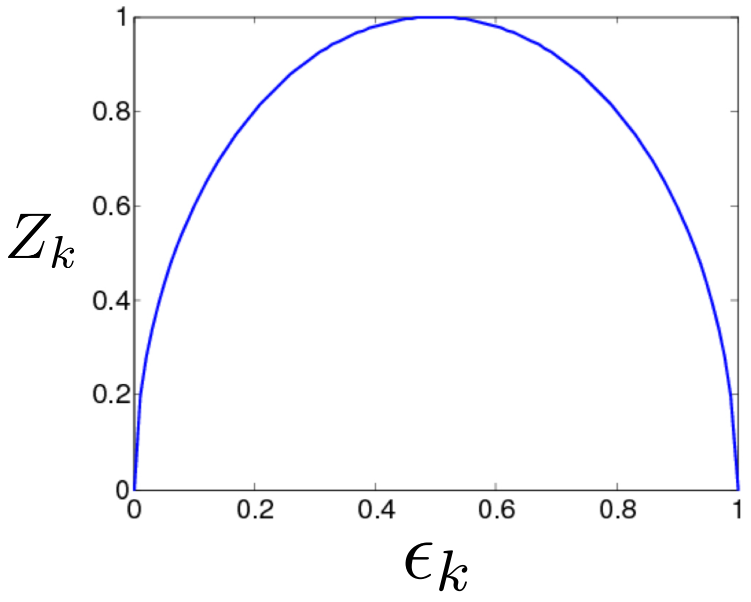
- **Second**, we want to find the **optimal weak classifier** $h(\mathbf{x})$ that minimizes Z_k using this result
- We subdivide Z_k into the same two sums as before, use the definition of the error for weak learners and substitute the optimal voting weight

Learning: why does it work?

- Doing so leads to an expression for Z_k as a function of ϵ_k

$$
\begin{aligned}
Z_k &= \sum_{i=1}^{N} w_{i,k} \exp(-\alpha_k y_i h_k(\mathbf{x}_i)) \\
&= \sum_{i:y_i = h_k(\mathbf{x}_i)} w_{i,k}\, e^{-\alpha_k} + \sum_{i:y_i \neq h_k(\mathbf{x}_i)} w_{i,k}\, e^{\alpha_k} \\
&= e^{-\alpha_k}(1-\epsilon_k) + e^{\alpha_k}\epsilon_k \\
&= 2\sqrt{\epsilon_k(1-\epsilon_k)}
\end{aligned}
$$

having $\alpha_k = \frac{1}{2}\log\left(\frac{1-\epsilon_k}{\epsilon_k}\right)$



- Thus, Z_k is **minimized** by selecting $h(\mathbf{x})$ with **minimal weighted error** ϵ_k

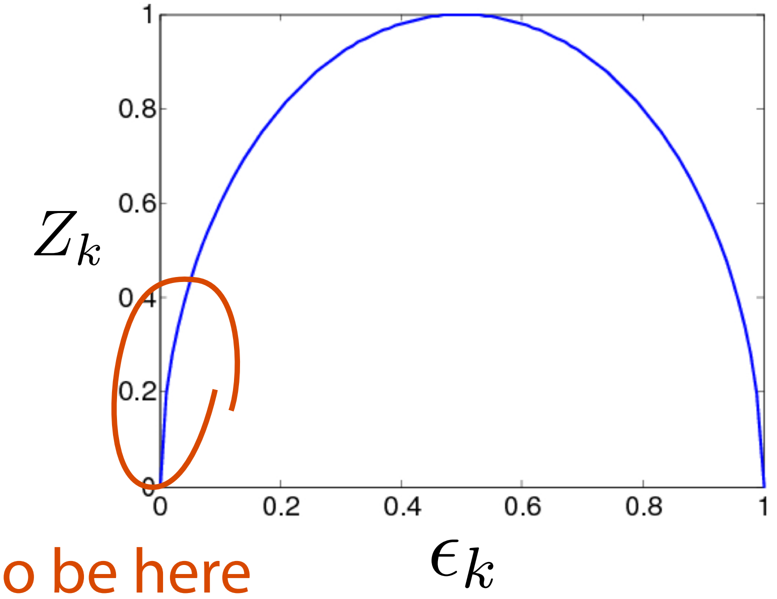
AdaBoost

Learning: why does it work?

- Doing so leads to an expression for Z_k as a function of ϵ_k

$$\begin{aligned} Z_k &= \sum_{i=1}^{N} w_{i,k} \exp(-\alpha_k y_i h_k(\mathbf{x}_i)) \\ &= \sum_{i:y_i = h_k(\mathbf{x}_i)} w_{i,k}\, e^{-\alpha_k} + \sum_{i:y_i \neq h_k(\mathbf{x}_i)} w_{i,k}\, e^{\alpha_k} \\ &= e^{-\alpha_k}(1-\epsilon_k) + e^{\alpha_k}\epsilon_k \\ &= 2\sqrt{\epsilon_k(1-\epsilon_k)} \end{aligned}$$

having $\alpha_k = \frac{1}{2} \log\left(\frac{1-\epsilon_k}{\epsilon_k}\right)$



We want to be here

- Thus, Z_k is **minimized** by selecting $h(\mathbf{x})$ with **minimal weighted error** ϵ_k

AdaBoost

Learning: why does it work?

- The process of selecting α_k and $h_k(\mathbf{x})$ can be interpreted as a **single optimization step** minimizing the upper bound on the error
- The improvement of the bound is **guaranteed** every time the error $\epsilon_k < 0.5$ (in a binary classification problem). This means that weak learners only have to be slightly **better than random guessing!**
- This is an amazingly **light assumption** for AdaBoost to work
- Hence the name "weak" classifier

AdaBoost

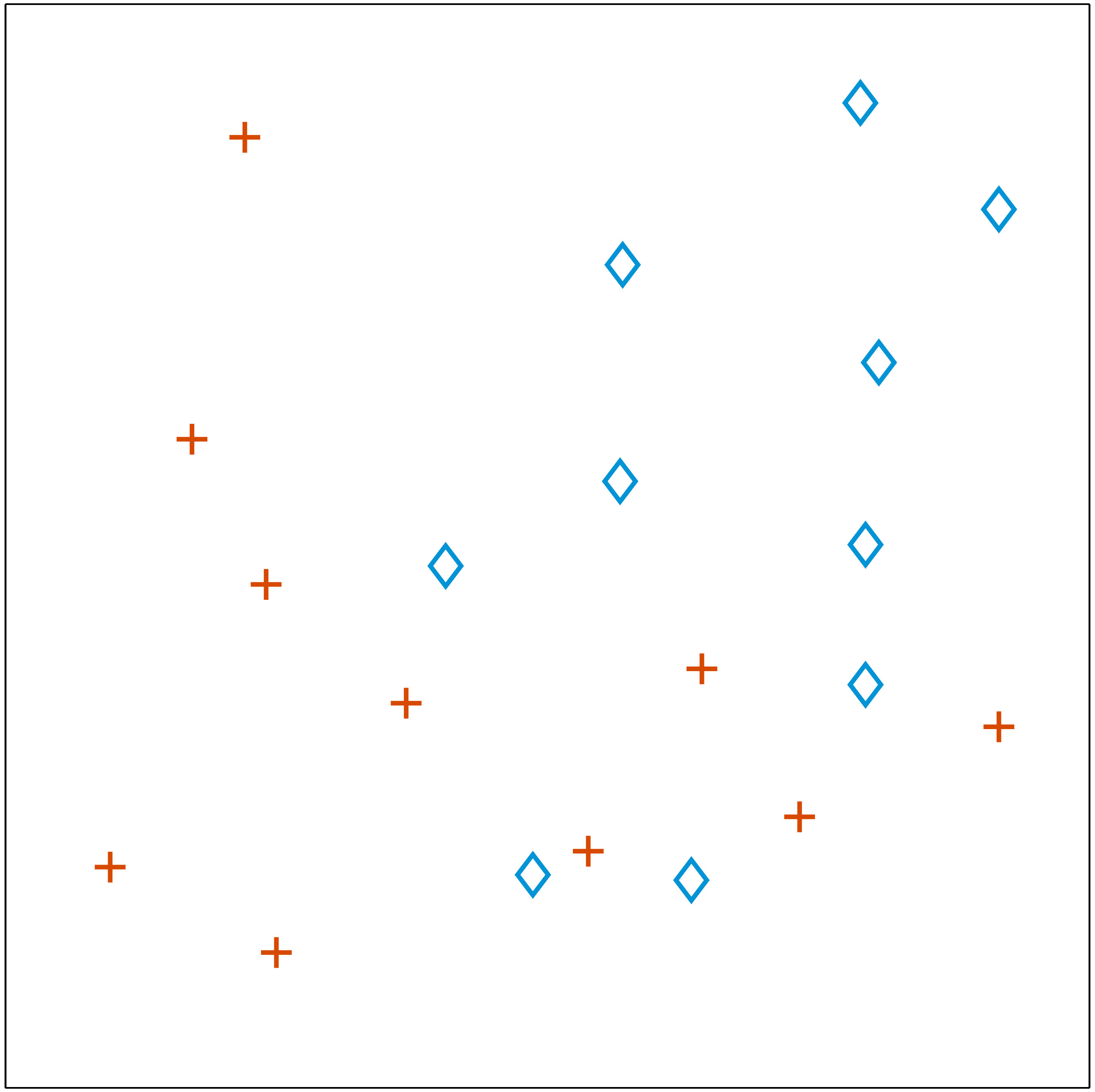
Learning

Given training set $(\mathbf{x}_1, y_1),\ (\mathbf{x}_2, y_2),\ \cdots\ (\mathbf{x}_N, y_N)$, **learn a strong classifier**

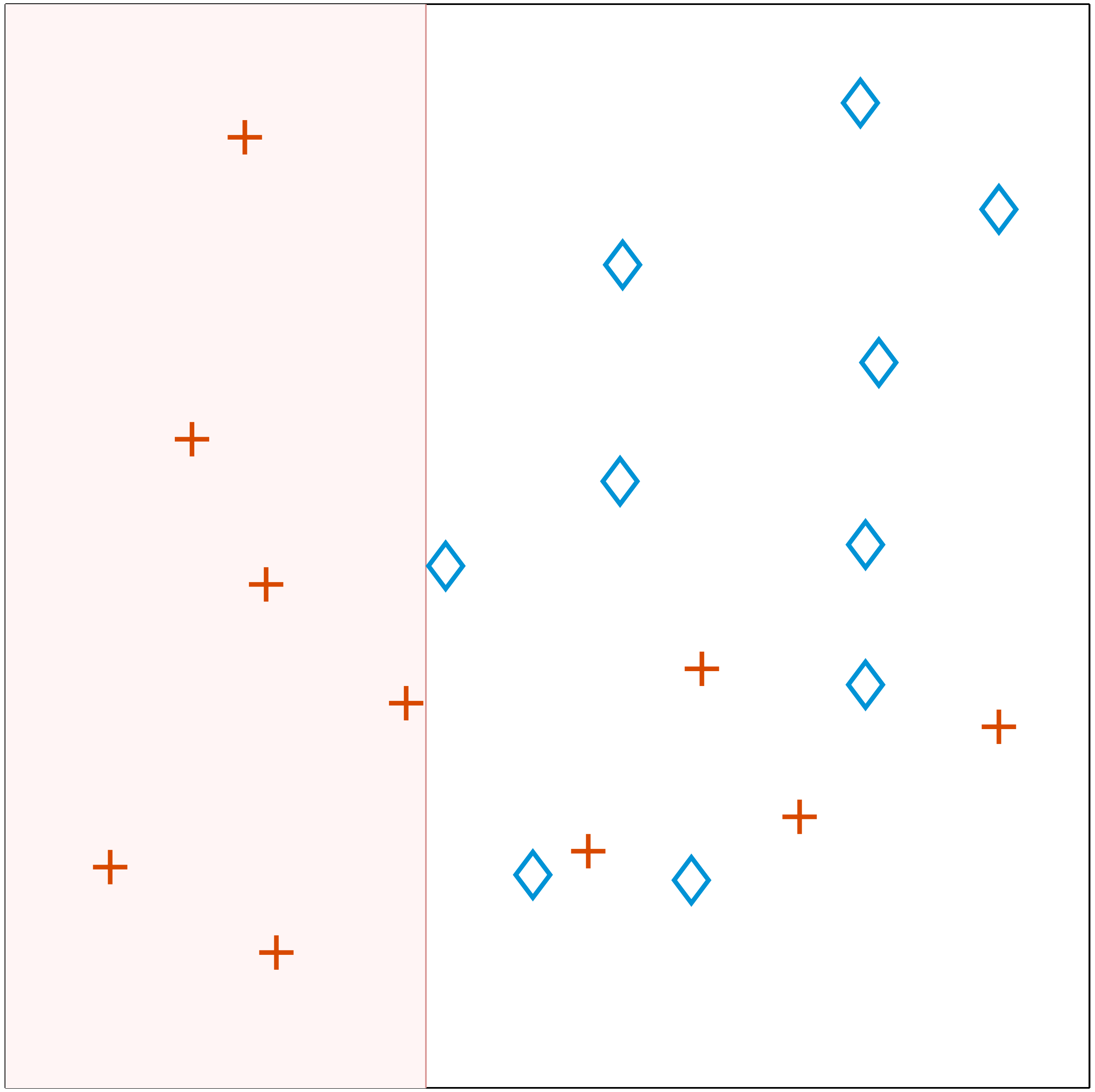
- **Initialize** weights $w_{i,0} = \frac{1}{N} \quad \forall i$
- For $k = 1 \ldots K$

  1. **Learn a weak classifier** $h_k(\mathbf{x})$ on weighted training data minimizing the error $\epsilon_k = \sum_i w_{i,k} \, \mathrm{I}(y_i \neq h_k(\mathbf{x}_i))$
  2. Compute **voting weight of** $h_k(\mathbf{x})$ as $\alpha_k = \frac{1}{2} \log \left(\frac{1-\epsilon_k}{\epsilon_k} \right)$
  3. **Recompute weights** $w_{i,k+1} = w_{i,k} \dfrac{\exp(-\alpha_k y_i h_k(\mathbf{x}_i))}{Z_k}$

Training Data



Iteration 1: train weak classifier 1



Threshold
$\theta^* = 0.37$

Dimension
$j^* = 1$

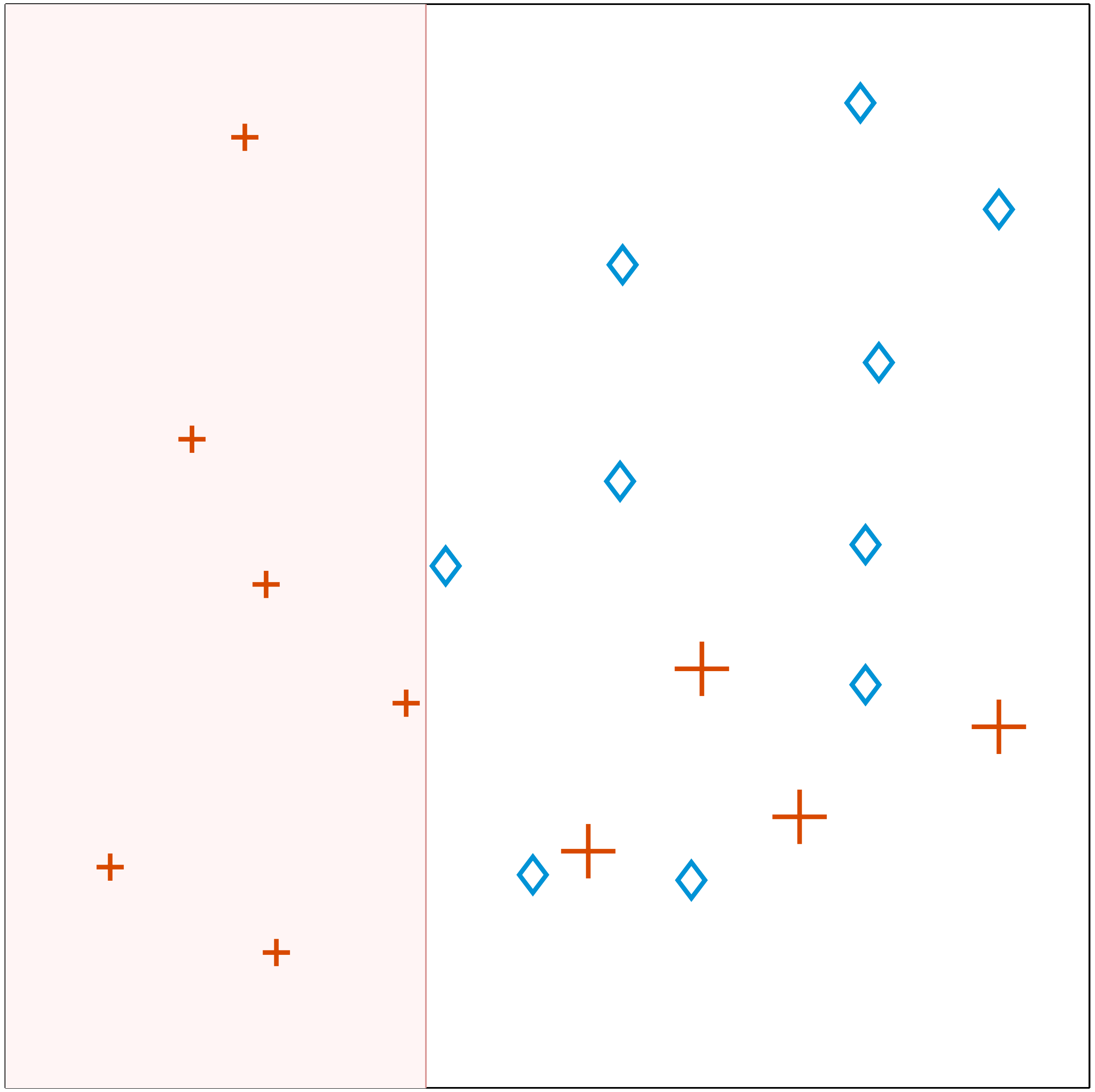
Weighted error
$\varepsilon_k = 0.2$

Voting weight
$\alpha_k = 1.39$

Error = 4

AdaBoost

Iteration 1: recompute weights



Threshold
$\theta^* = 0.37$

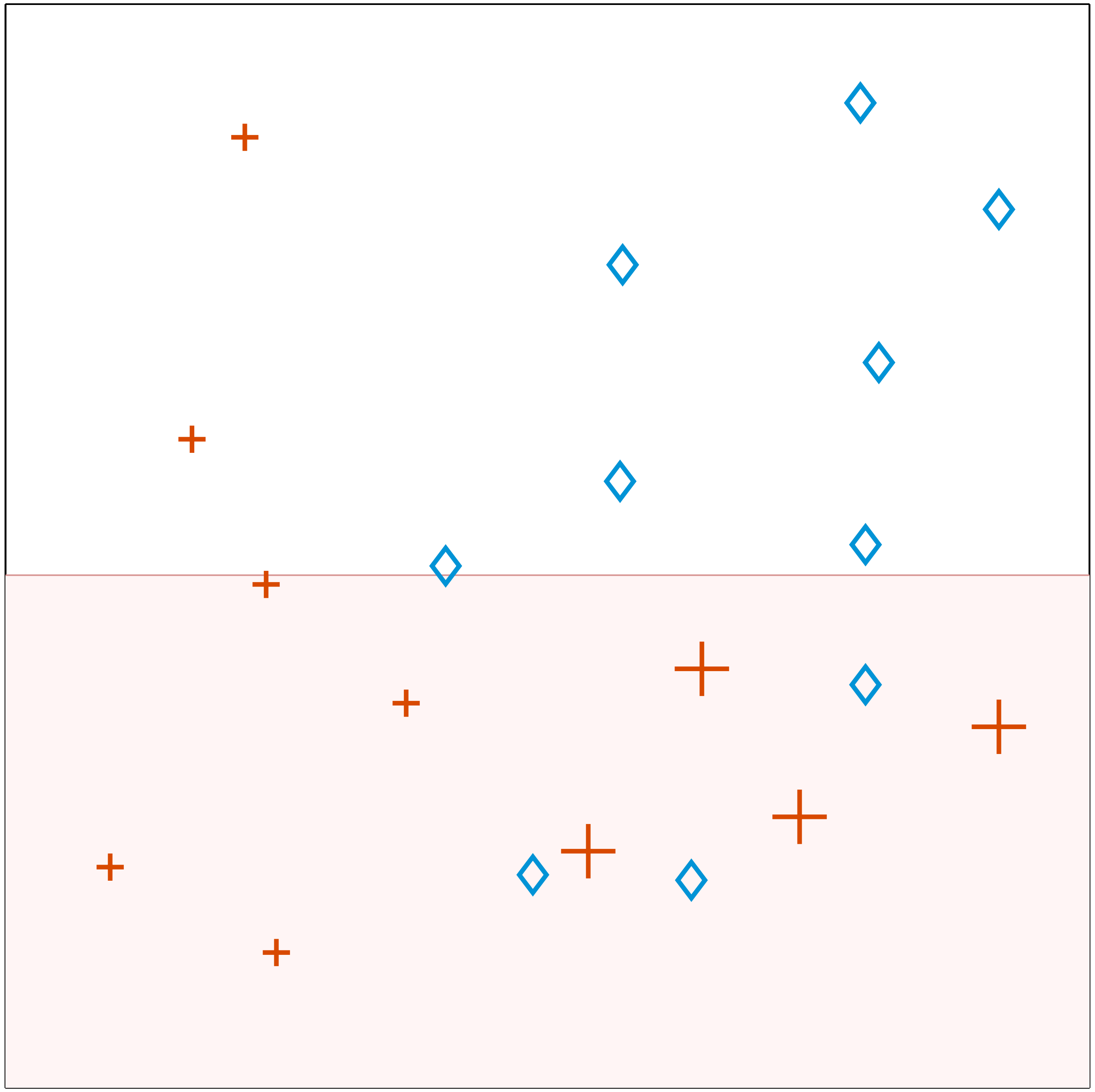
Dimension
$j^* = 1$

Weighted error
$\varepsilon_k = 0.2$

Voting weight
$\alpha_k = 1.39$

Error = 4

Iteration 2: train weak classifier 2



Threshold
$\theta^* = 0.47$

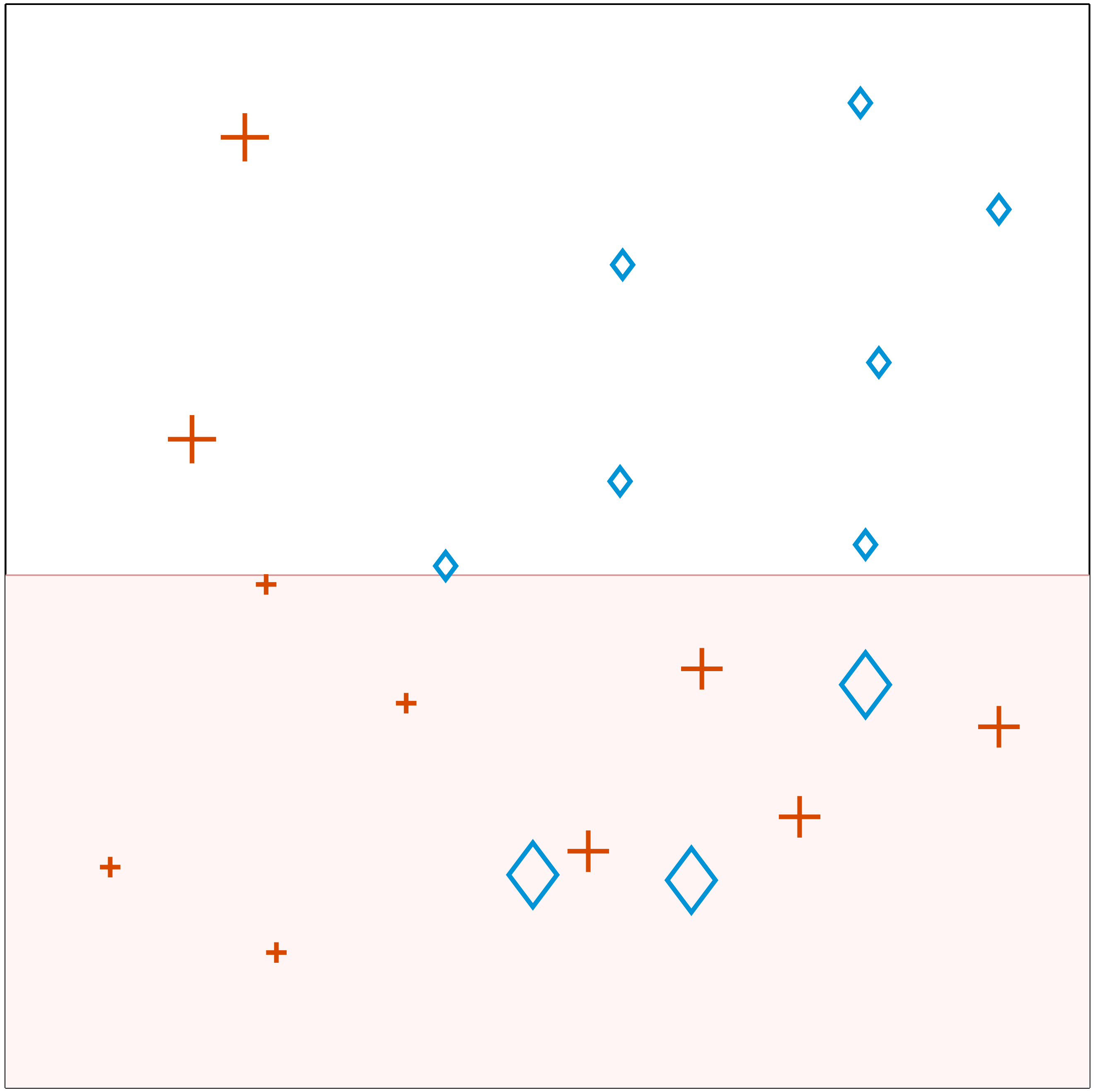
Dimension
$j^* = 2$

Weighted error
$\varepsilon_k = 0.16$

Voting weight
$\alpha_k = 1.69$

Error = 5

Iteration 2: recompute weights



Threshold

$\theta^* = 0.47$

Dimension

$j^* = 2$

Weighted error

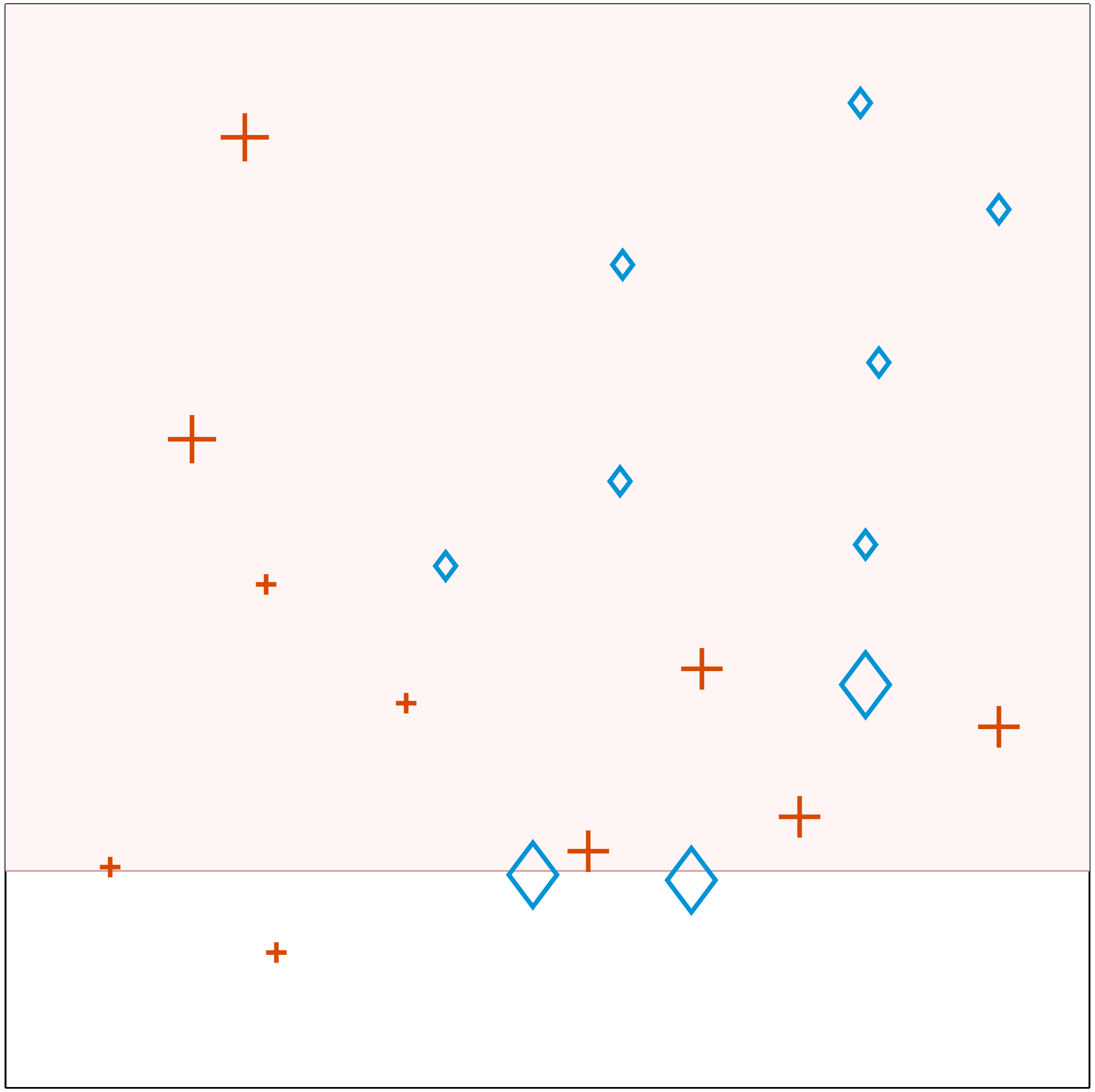
$\varepsilon_k = 0.16$

Voting weight

$\alpha_k = 1.69$

Error = 5

Iteration 3: train weak classifier 3



Threshold
$\theta^* = 0.14$

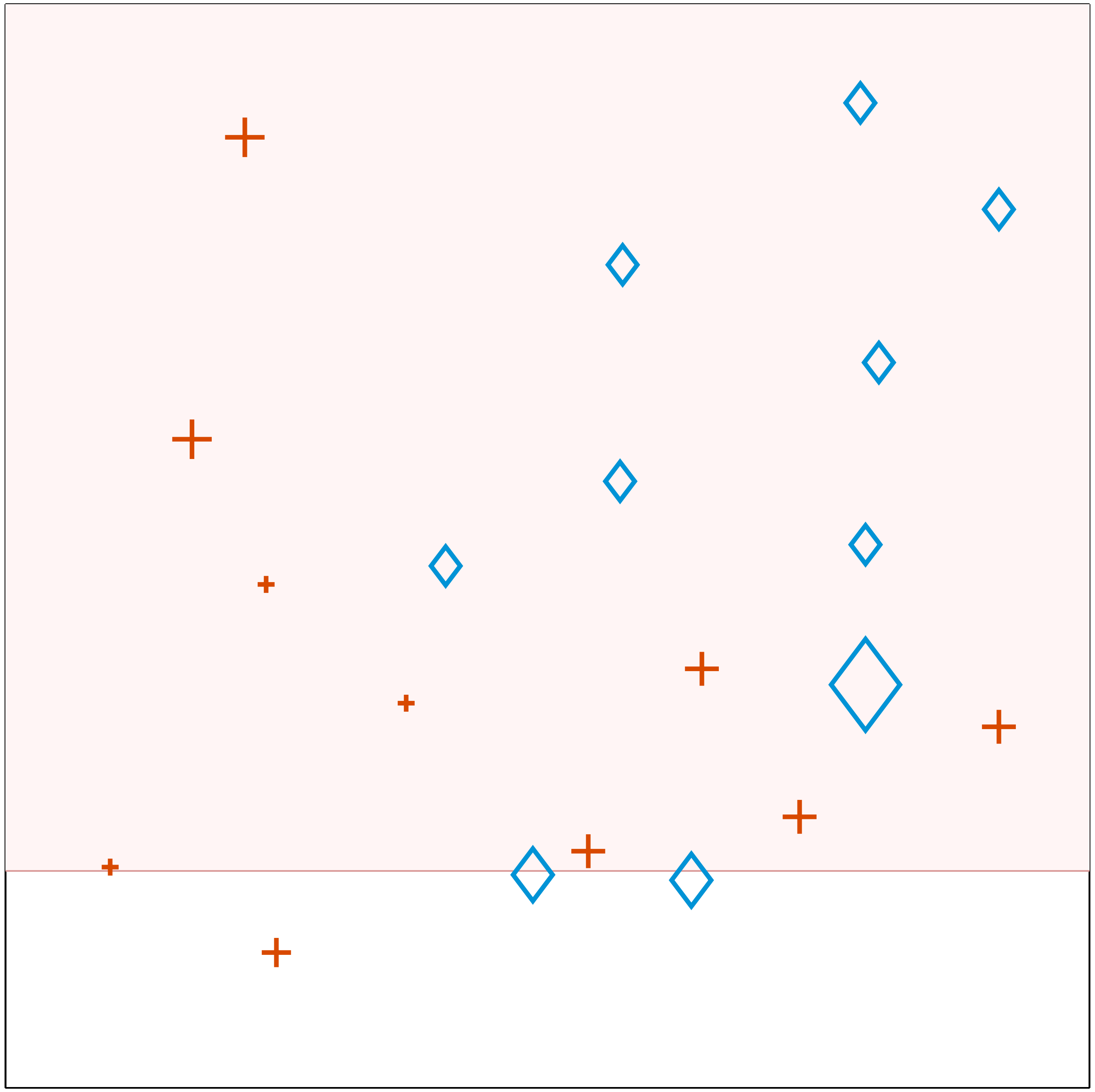
Dimension, sign
$j^* = 2$, neg

Weighted error
$\varepsilon_k = 0.25$

Voting weight
$\alpha_k = 1.11$

Error = 1

Iteration 3: recompute weights



Threshold
$\theta^* = 0.14$

Dimension, sign
$j^* = 2$, neg

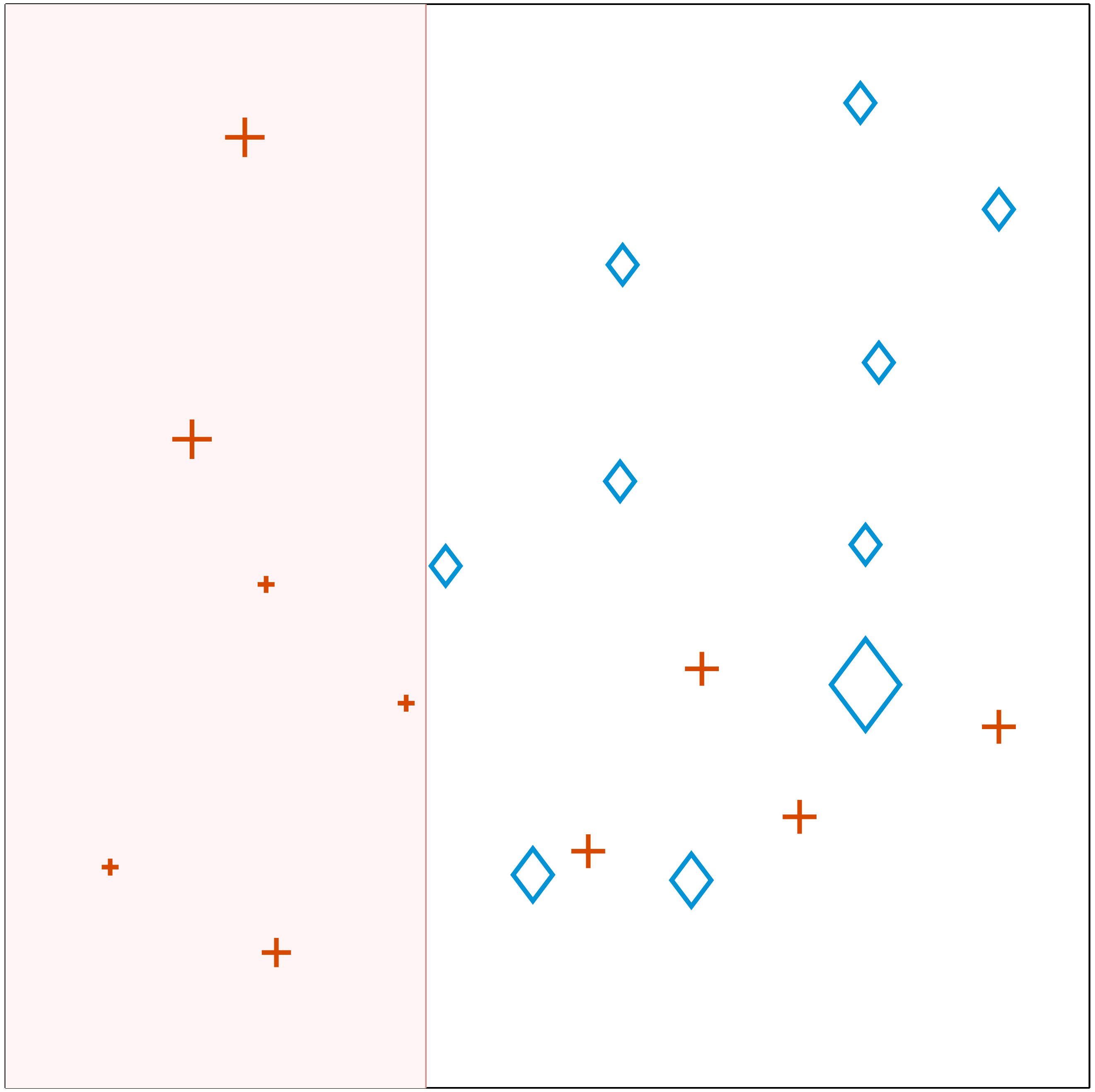
Weighted error
$\varepsilon_k = 0.25$

Voting weight
$\alpha_k = 1.11$

Error = 1

AdaBoost

Iteration 4: train weak classifier 4



Threshold
$\theta^* = 0.37$

Dimension
$j^* = 1$

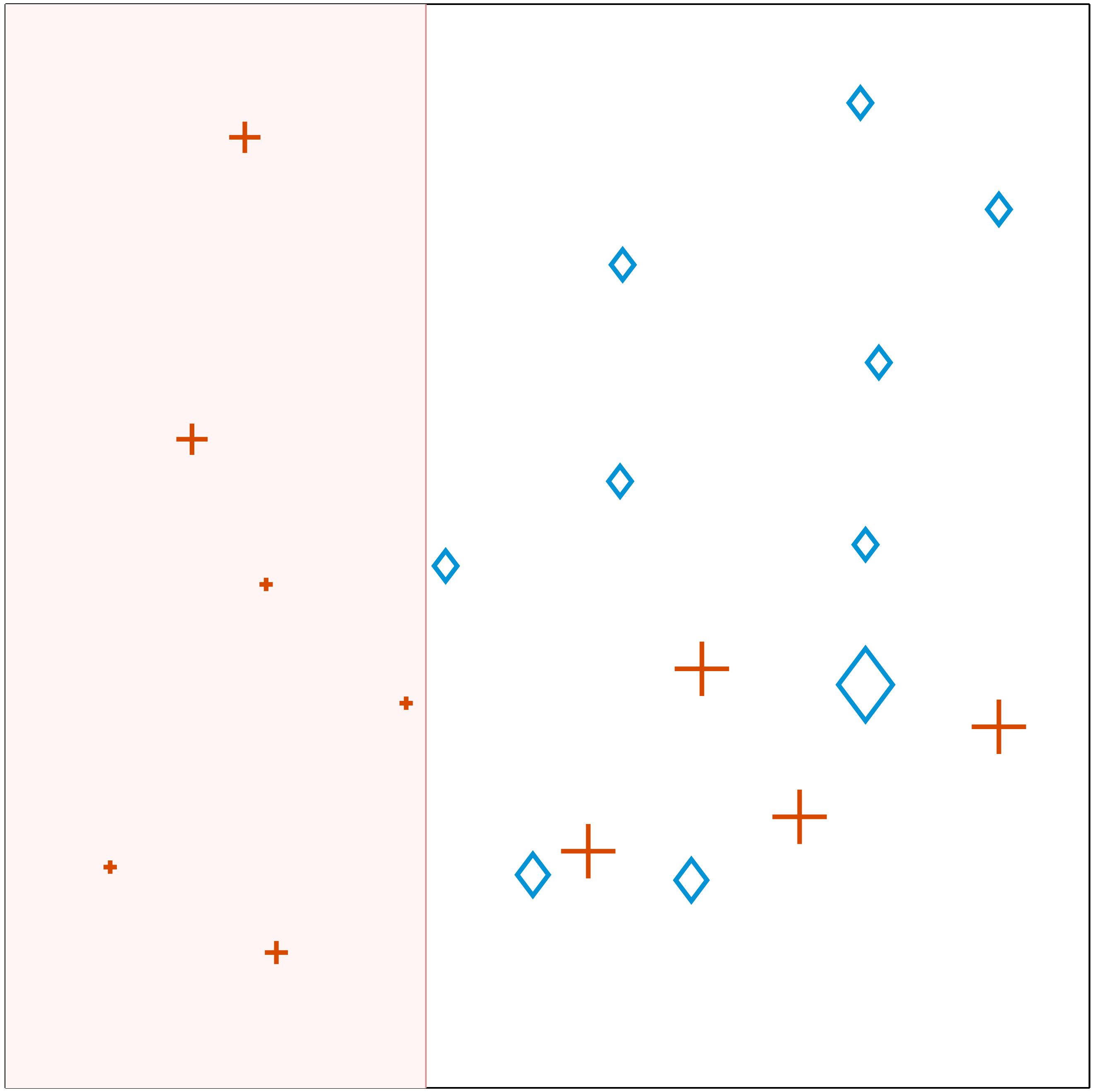
Weighted error
$\varepsilon_k = 0.20$

Voting weight
$\alpha_k = 1.40$

Error = 1

AdaBoost

Iteration 4: recompute weights



Threshold
$\theta^* = 0.37$

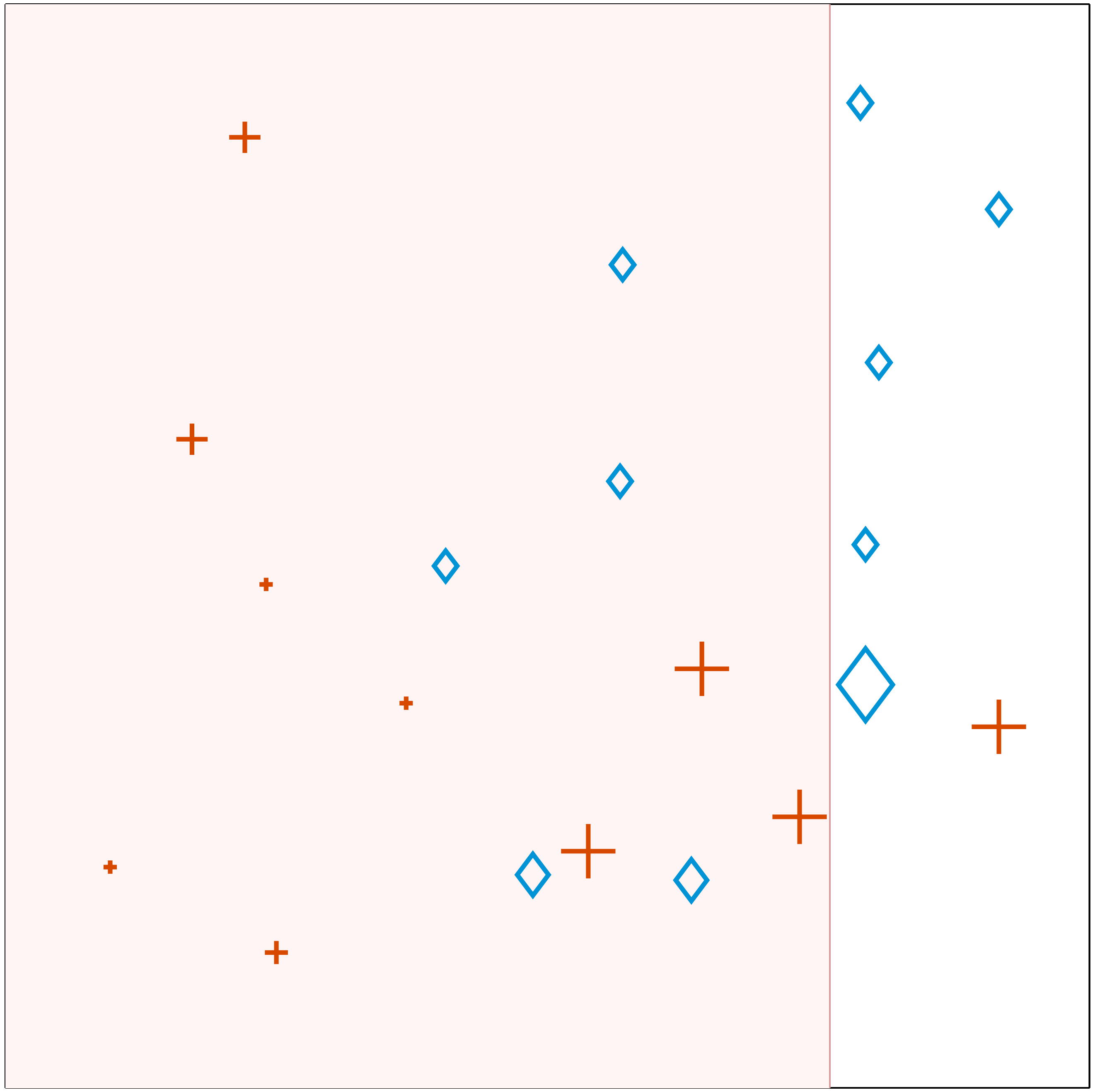
Dimension
$j^* = 1$

Weighted error
$\varepsilon_k = 0.20$

Voting weight
$\alpha_k = 1.40$

Error = 1

Iteration 5: train weak classifier 5



Threshold
$\theta^* = 0.81$

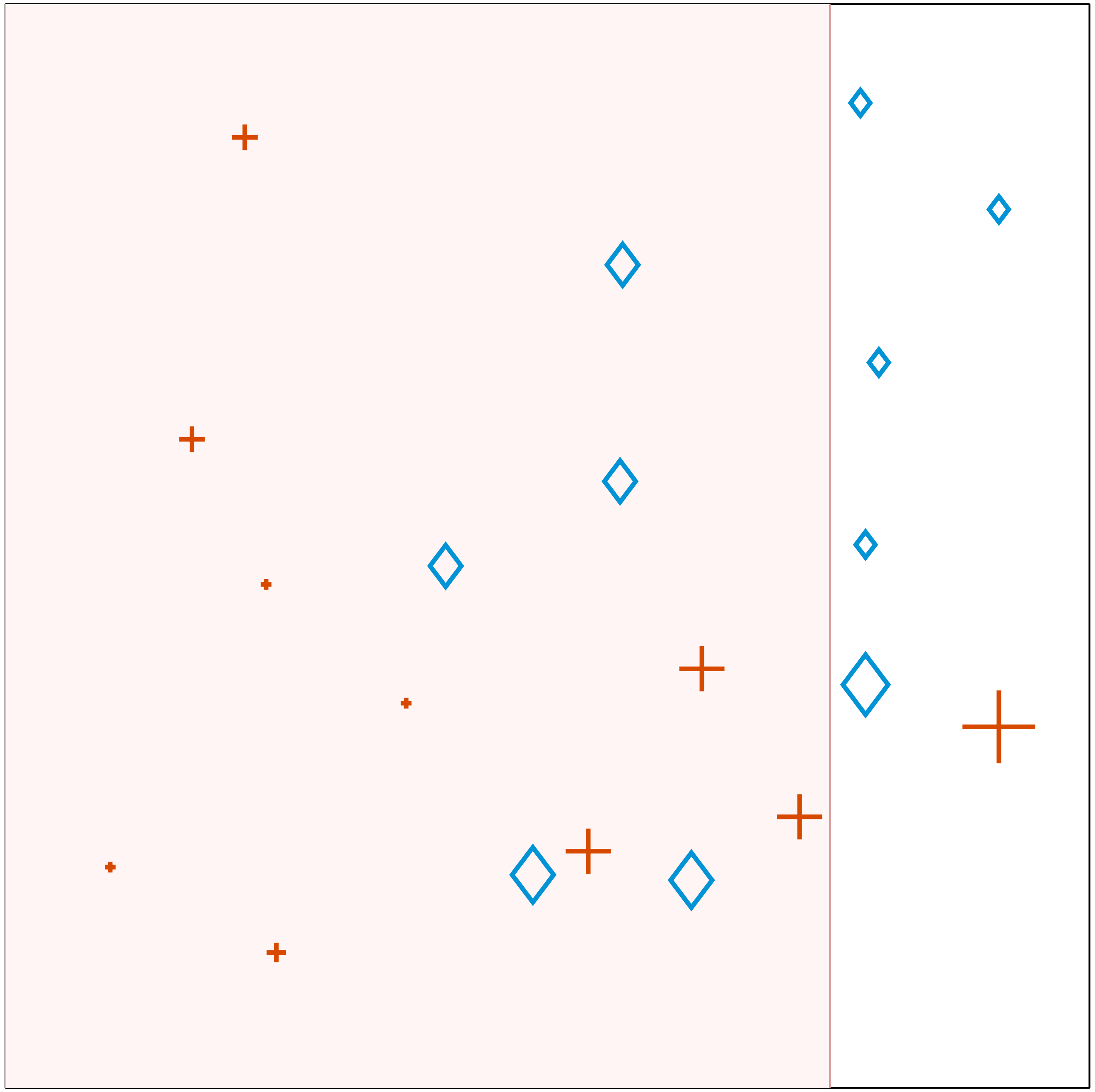
Dimension
$j^* = 1$

Weighted error
$\varepsilon_k = 0.28$

Voting weight
$\alpha_k = 0.96$

Error = 1

Iteration 5: recompute weights



Threshold
$\theta^* = 0.81$

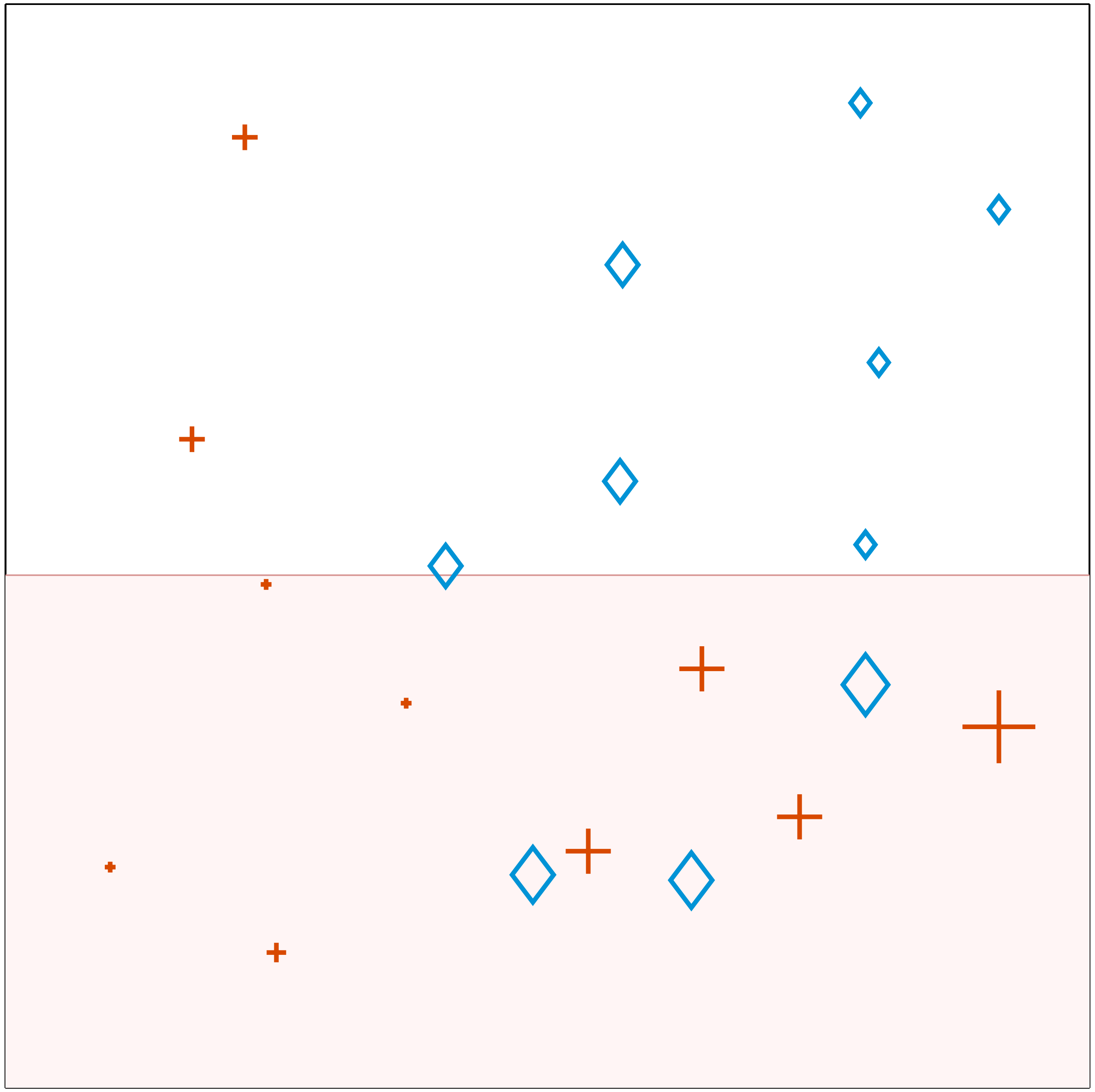
Dimension
$j^* = 1$

Weighted error
$\varepsilon_k = 0.28$

Voting weight
$\alpha_k = 0.96$

Error = 1

Iteration 6: train weak classifier 6



Threshold
$\theta^* = 0.47$

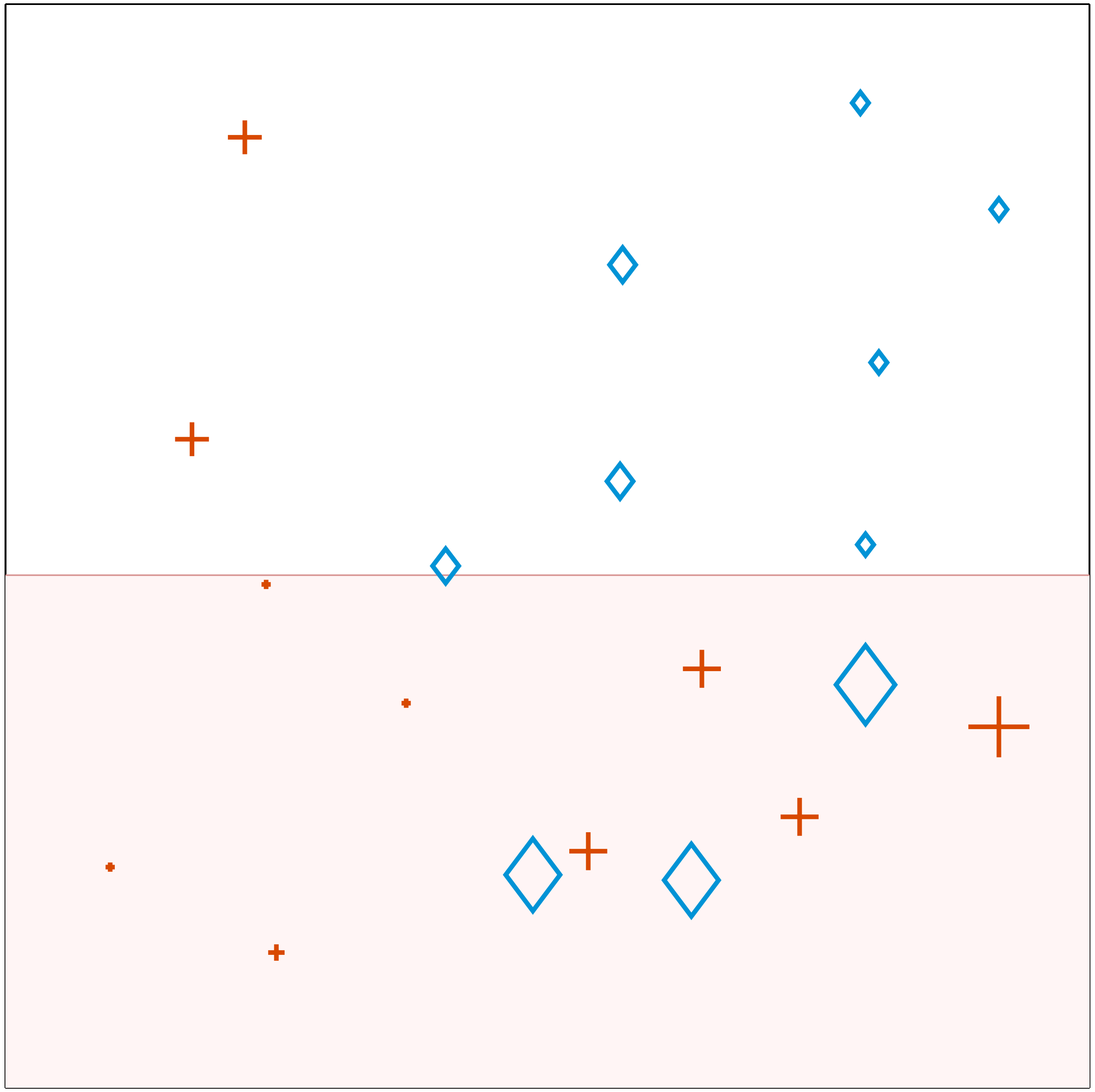
Dimension
$j^* = 2$

Weighted error
$\varepsilon_k = 0.29$

Voting weight
$\alpha_k = 0.88$

Error = 1

Iteration 6: recompute weights



Threshold
$\theta^* = 0.47$

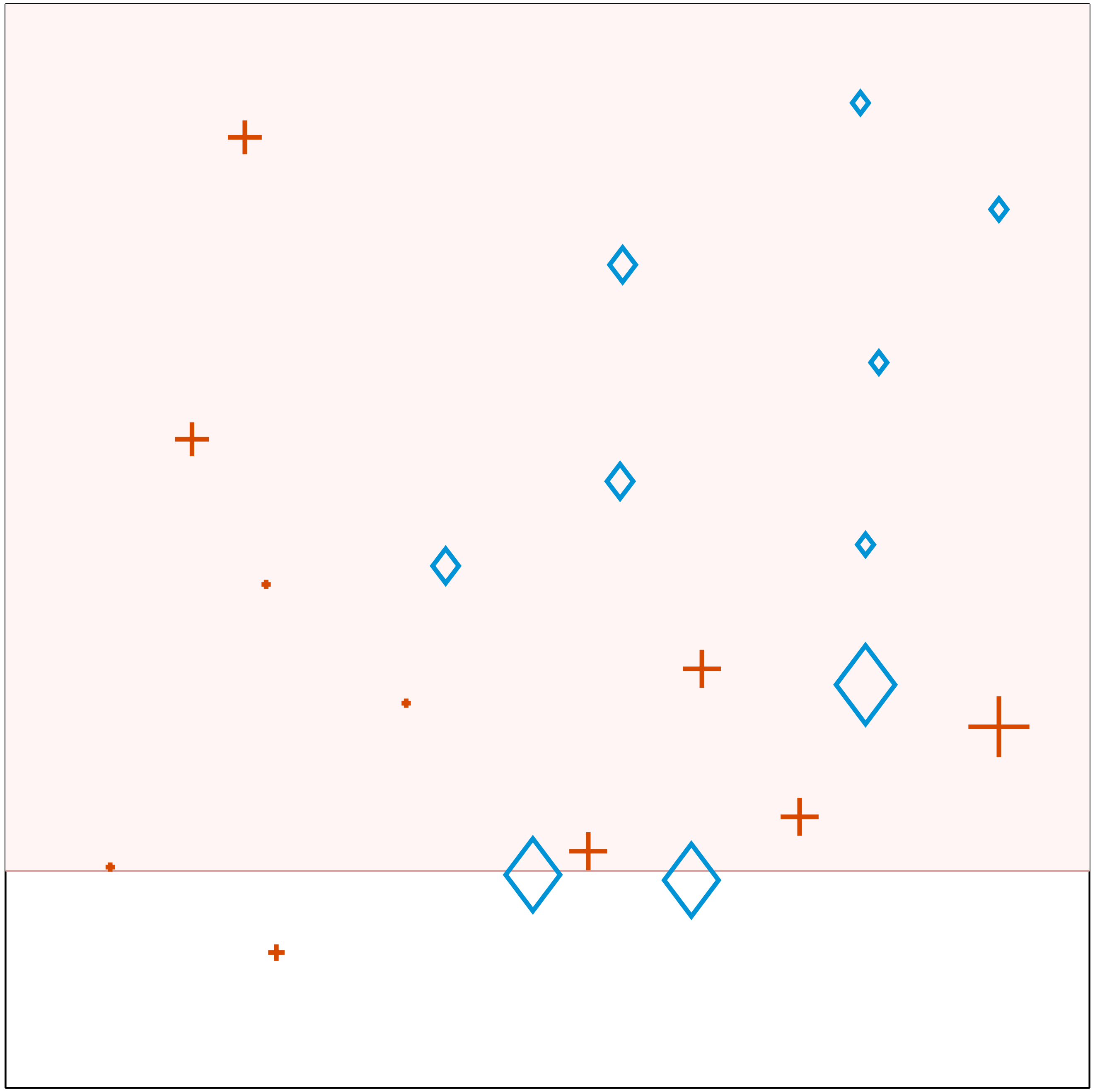
Dimension
$j^* = 2$

Weighted error
$\varepsilon_k = 0.29$

Voting weight
$\alpha_k = 0.88$

Error = 1

Iteration 7: train weak classifier 7



Threshold
$\theta^* = 0.14$

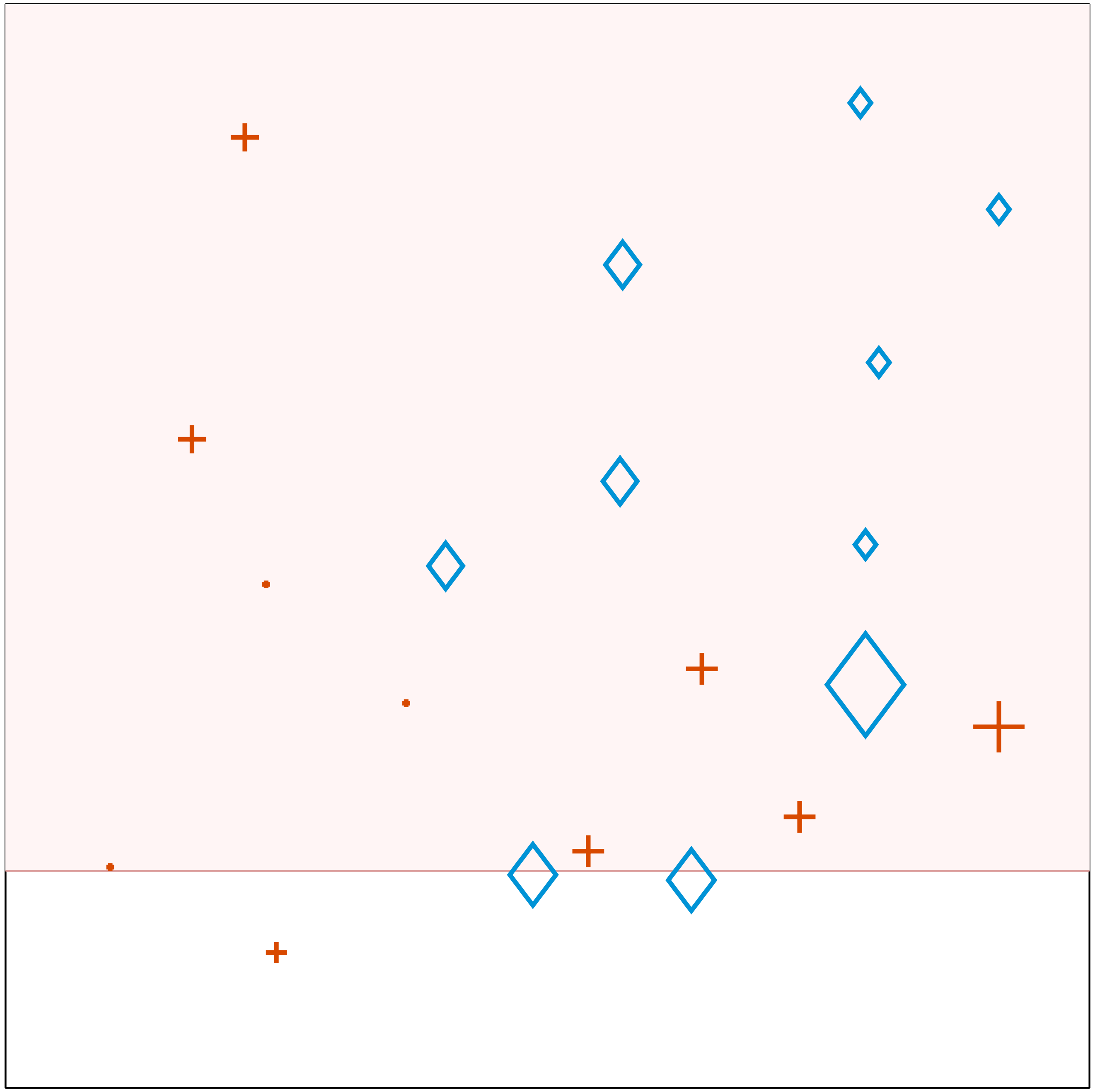
Dimension, sign
$j^* = 2$, neg

Weighted error
$\varepsilon_k = 0.29$

Voting weight
$\alpha_k = 0.88$

Error = 1

Iteration 7: recompute weights



Threshold
$\theta^* = 0.14$

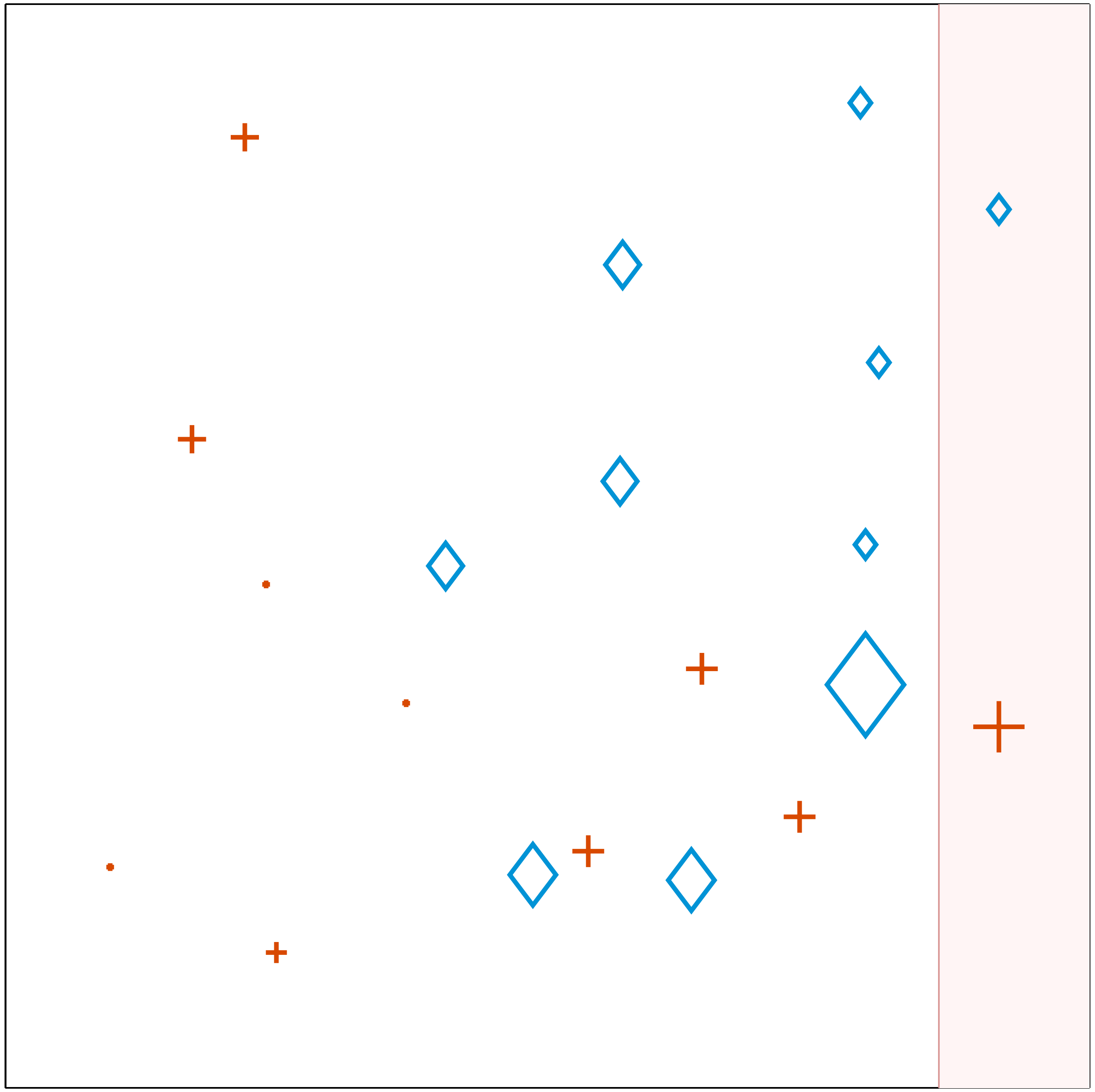
Dimension, sign
$j^* = 2$, neg

Weighted error
$\varepsilon_k = 0.29$

Voting weight
$\alpha_k = 0.88$

Error = 1

Iteration 8: train weak classifier 8



Threshold
$\theta^* = 0.93$

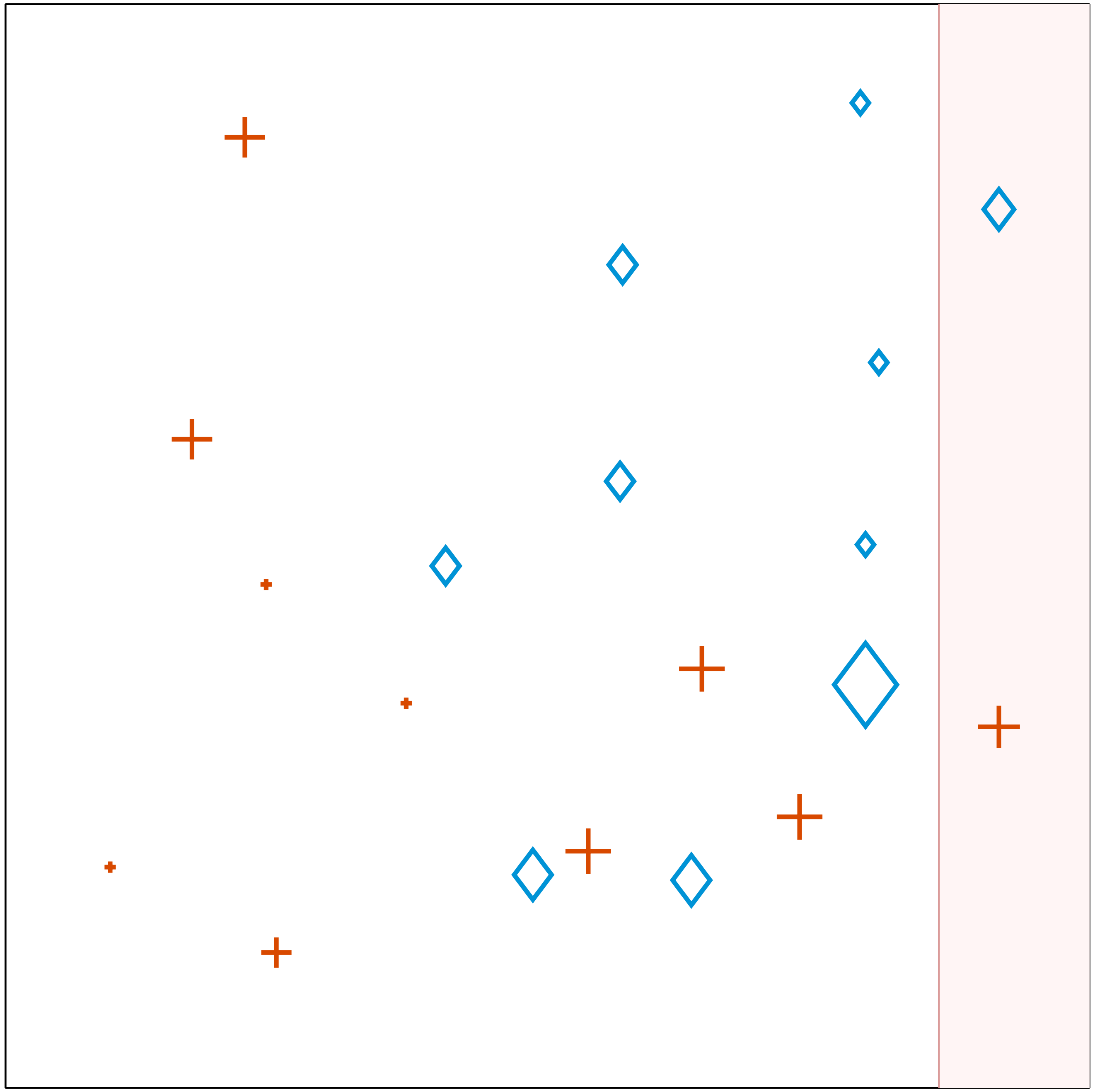
Dimension, sign
$j^* = 1$, neg

Weighted error
$\varepsilon_k = 0.25$

Voting weight
$\alpha_k = 1.12$

Error = 0

Iteration 8: recompute weights



Threshold
$\theta^* = 0.93$

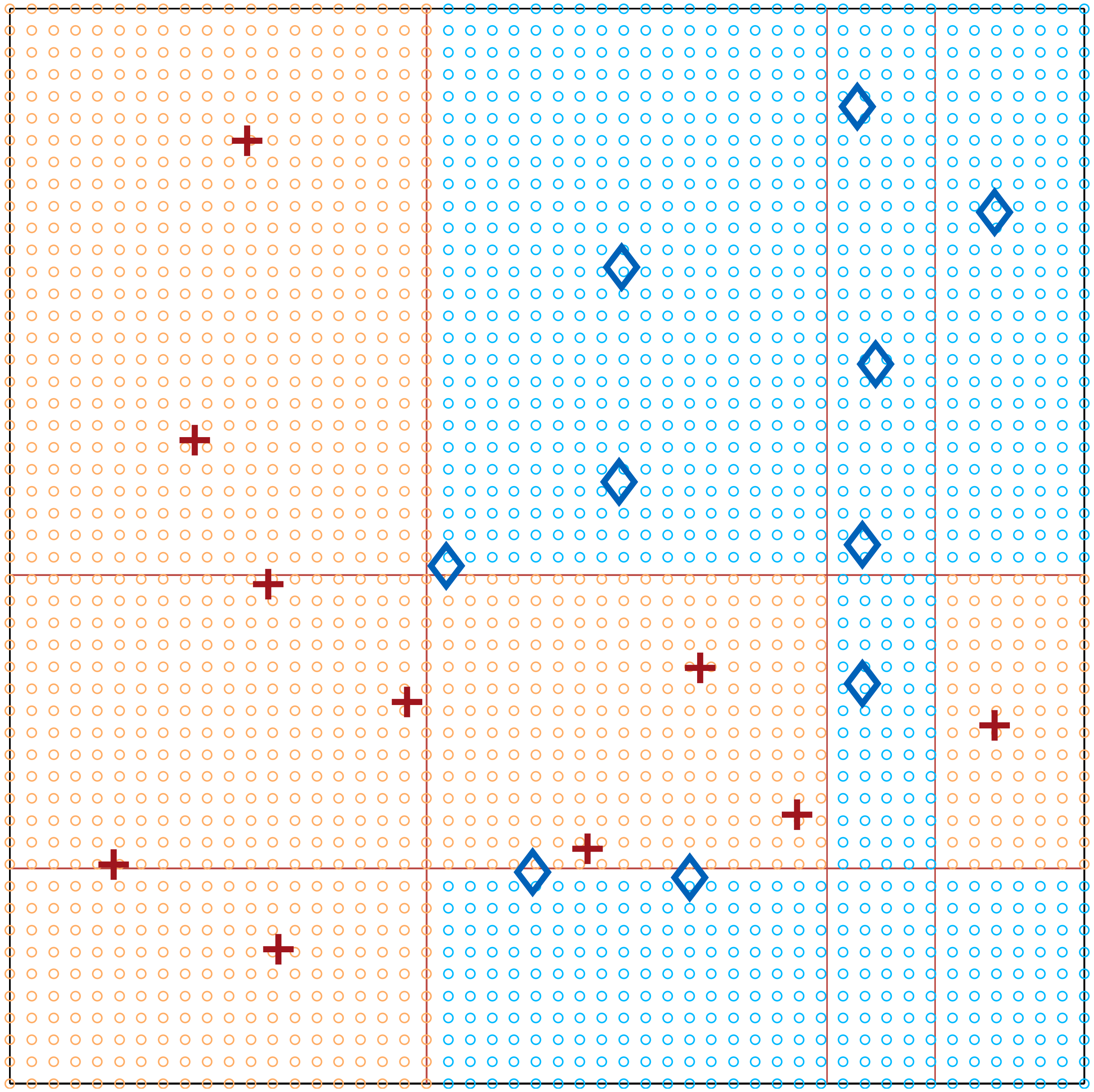
Dimension, sign
$j^* = 1$, neg

Weighted error
$\varepsilon_k = 0.25$

Voting weight
$\alpha_k = 1.12$

Error = 0

Final Strong Classifier



Training error = 0

AdaBoost

Properties

- By increasing the weight of misclassified training pairs, it **focuses on the ″hard″** samples. The next weak classifier is then trained on the **mistakes of the previous one**
- The weight distribution captures all information about previously learned classifiers (a Markov property – loosely speaking)
- AdaBoost is a **non-linear classifier**

- AdaBoost can be seen as a **principled feature selector:** it tells you what the best features are (ranked by the voting weight), what the best thresholds are how to combine them to a classifier
- This makes the learning result **interpretable** and allows for knowledge extraction which can be checked and verified by human experts
- Helps classifier design to be **more science than art**

AdaBoost in Action

Kai Arras
Social Robotics Lab, University of Freiburg

Nov 2009 Social Robotics Laboratory

AdaBoost

Summary AdaBoost

- Ensemble methods such as boosting are **meta algorithms** that improve ("boost") the performance of learning algorithms by combining them
- AdaBoost **minimizes the training error** (an **upper bound** thereof) if each weak classifier performs **better than random guessing** (i.e. has error less than 0.5 for a binary classification problem)
- **Advantages**
  - AdaBoost has **good generalization properties**, it can be proven to **maximize the margin** (proof in literature)
  - Simple to implement
  - **Interpretability** by taking a principled approach to feature selection
- **Drawbacks**
  - **Noise-sensitive** due to hard margin, can overfit under such conditions
  - **Not probabilistic**

Contents

- Introduction and basics
- Bayes Classifier
- Logistic Regression
- Support Vector Machines
- AdaBoost
- k-Nearest Neighbor
- Cross-validation
- Performance measures

K-Nearest Neighbor

Non-Parametric Classifiers

- So far, we have considered classifiers that learn **parametric models** from data:
  - Bayes classifier: distributions for class-conditional densities and priors
  - Logistic Regression: sigmoid mapping of linear activation function
  - Support Vector Machines: hyperplane
  - AdaBoost: set of parametric weak classifiers with associated voting weights
- No matter how much data are thrown at a parametric model, it will not require more parameters
- Learning parametric models may be **costly** for large training sets and subject to **convergence issues** during optimization
- So let us consider **non-parametric models**

K-Nearest Neighbor

Non-Parametric Classifiers

- Non-parametric models are memory-based. They involve **storing the entire training set** in order to make predictions for new data points
- They are not characterized by a bounded set of parameters but **grow** with the number of training pairs
- This approach is called **instance-based learning, memory-based learning** or **lazy learning**
- Very **simple and fast to train** but **slow at making decisions**

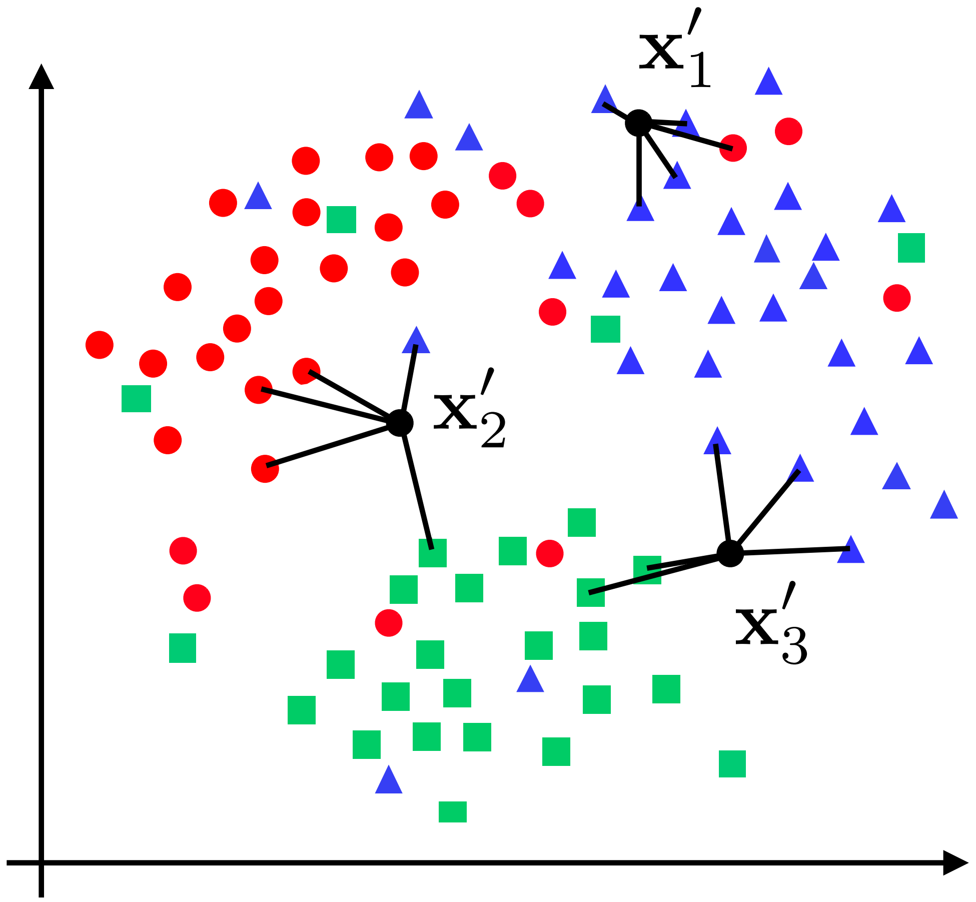
- The most trivial instance-based learning algorithm is **table lookup**: store all training samples in a lookup table, and then when asked for $h(\mathbf{x}')$, see if $\mathbf{x}'$ is in the table. Obviously, this method **generalizes poorly**
- **K-nearest neighbor** classification is only a slight variation of this method

K-Nearest Neighbor

K-Nearest Neighbor Classifier

- Given a new data point $\mathbf{x}'$, the **k-nearest neighbor classifier** (k-NN) finds the k samples that are **nearest** to $\mathbf{x}'$
- For class prediction, the algorithm takes the **majority vote of the neighbors** (plurality vote in the multi-class case)

- **Example:** three classes, $k = 5$ and three query points. Using the Euclidian distance we find varying numbers of neighbors. The plurality votes induce the decision boundaries



K-Nearest Neighbor

Learning

- Learning consists in **storing all training samples**

- For large training sets, **storage requirements** may become very high and there are a number of extensions that address this issue by exploiting redundancies in the training set
- **Condensing methods** decrease the number of stored instances without degrading performance
  - The **Condensed nearest neighbor algorithm** (CNN) removes points in the interior of decision regions. Since samples that define the discriminative function are **located around the decision boundaries**, such points can be safely discarded ("absorbed")
  - **Class-outliers** are easily spotted by running k-NN over the training set and testing if a point's k nearest neighbors include more than r examples of other classes

K-Nearest Neighbor

Inference and Decision

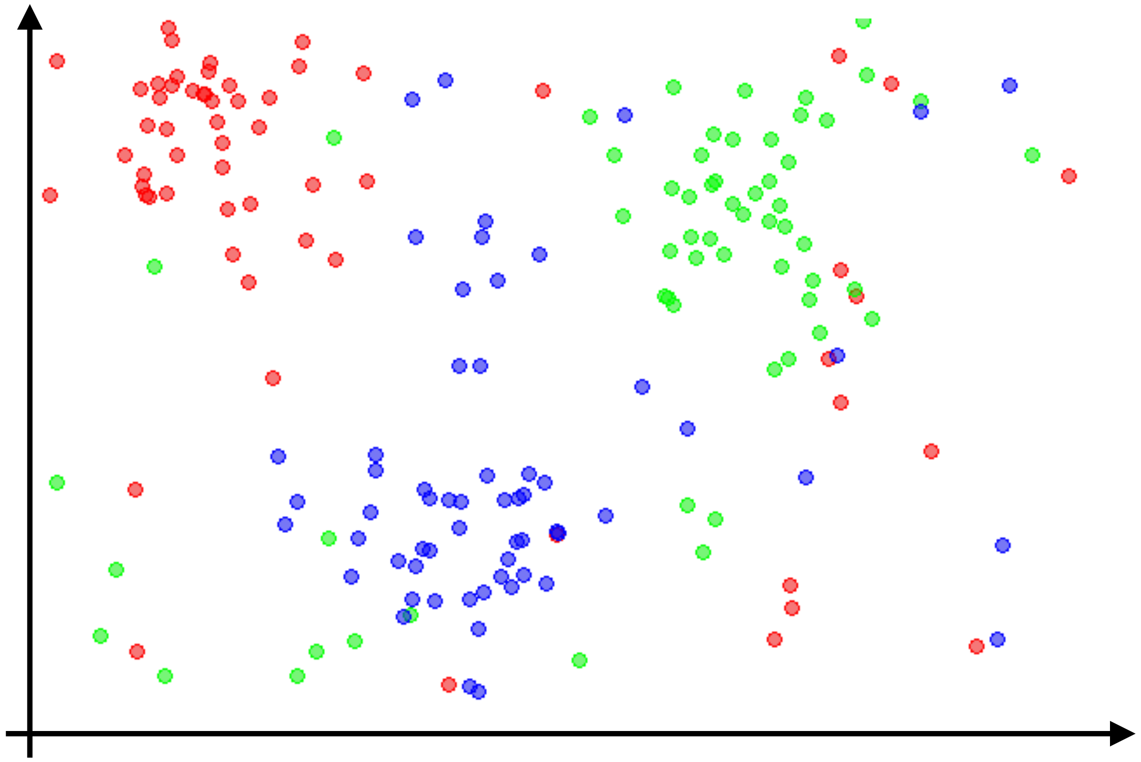
- Class prediction is made by a **majority/plurality vote of the k nearest neighbors**
- If $k = 1$ the new data point is simply assigned to the class of its (single) nearest neighbor
- The k-NN classifier **approximates the discriminant function only locally** as opposed to learning a decision boundary across the entire space

- Naive implementations iterate through all training samples and compute all distances to a new data point. This $O(N)$ approach may be too costly for large N
- Smart implementations use **kd-trees** or **hash tables** (e.g. locality-sensitive hashing) to achieve **sublinear run time**

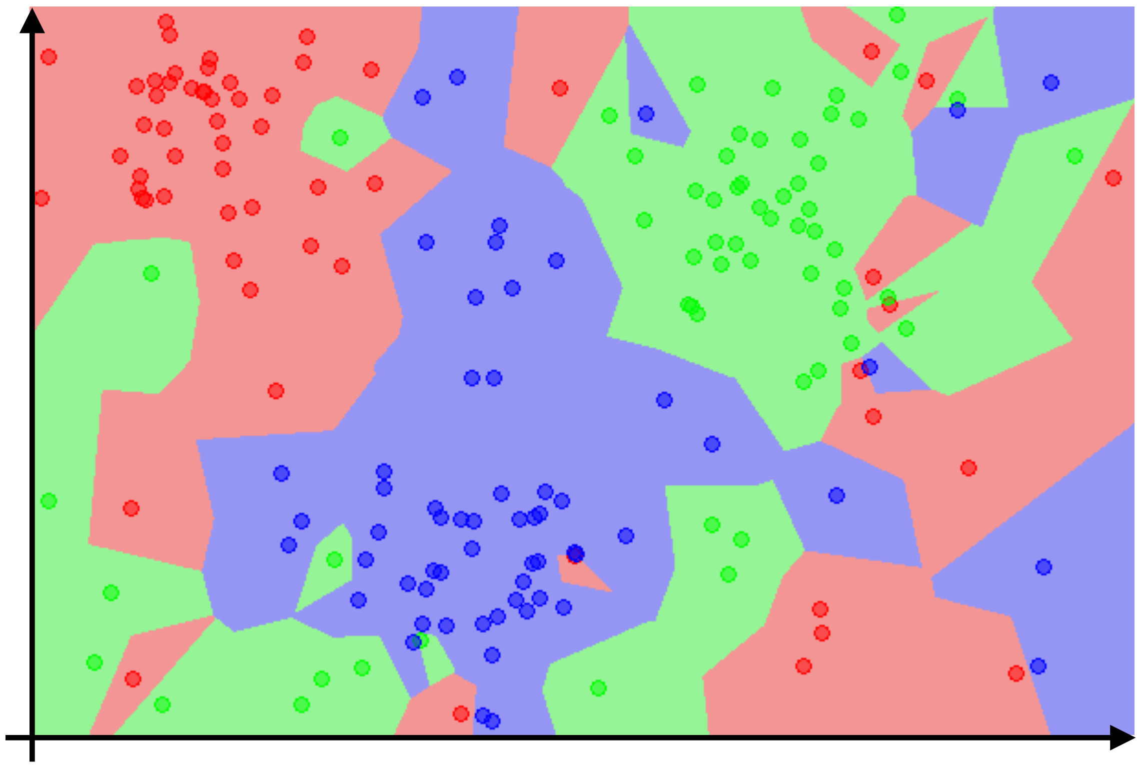
K-Nearest Neighbor

Example

- Three classes, Euclidian distance



Data



1-NN

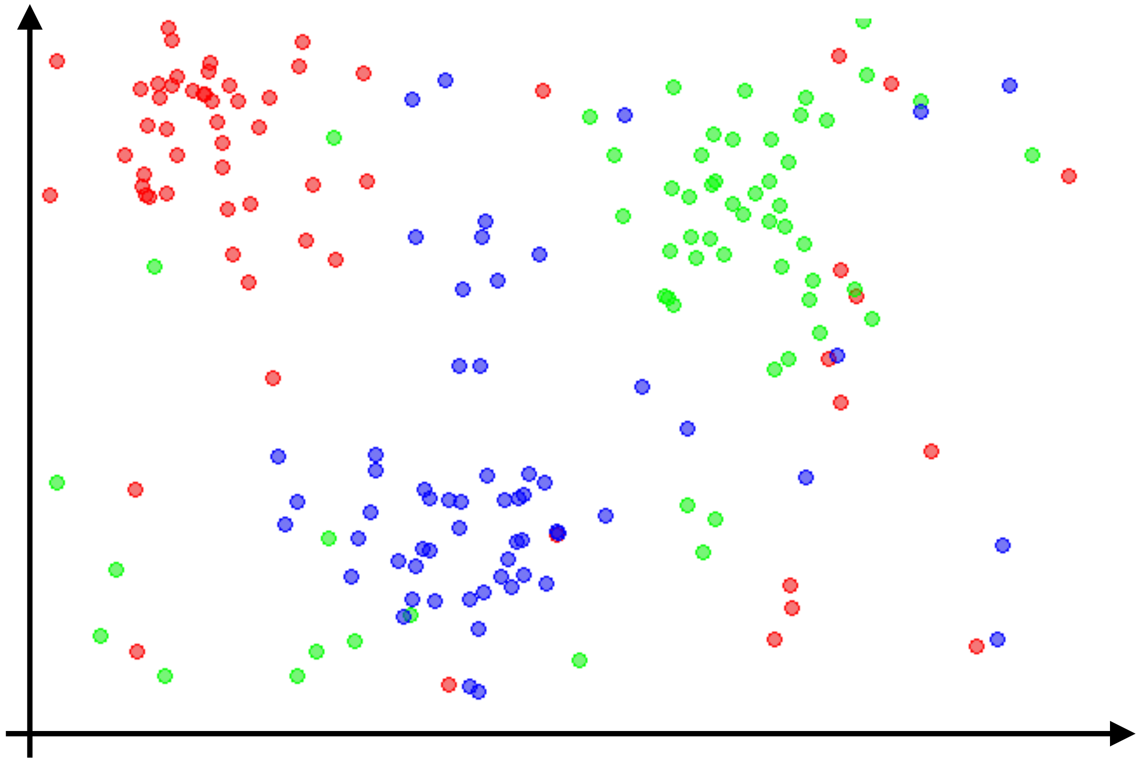
Source [7]

- For 1-NN the discriminant function lies on a Voronoi set

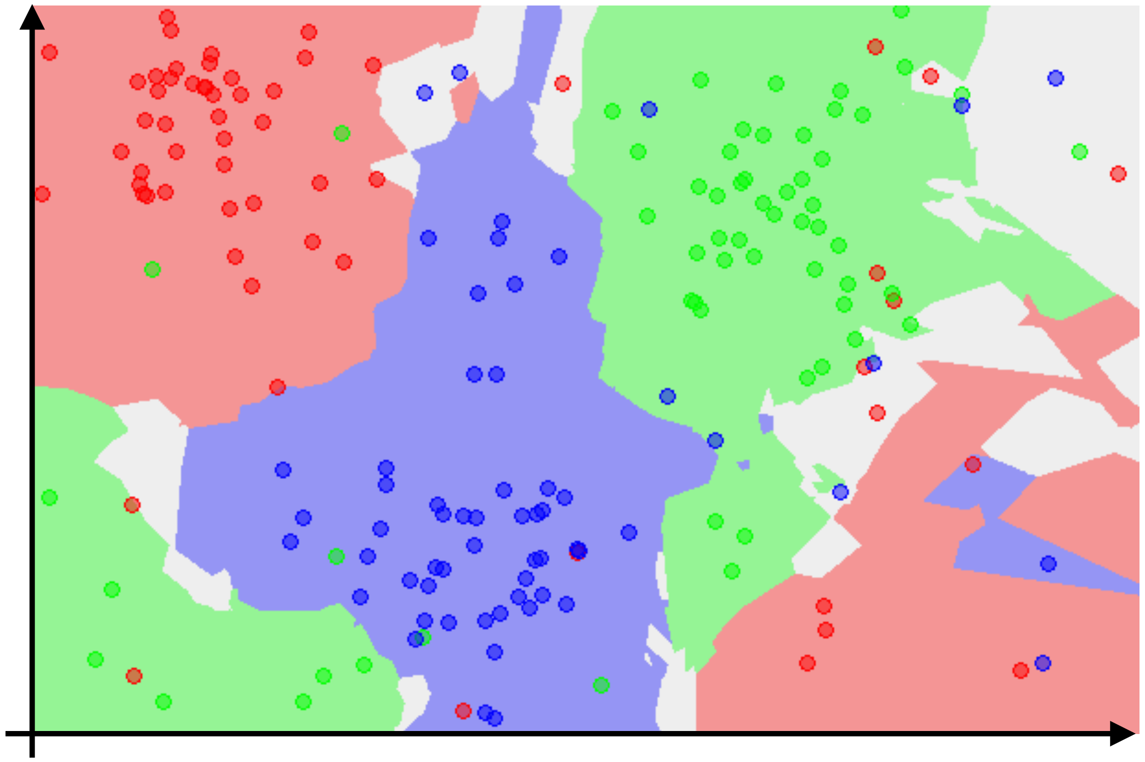
K-Nearest Neighbor

Example

- Three classes, Euclidian distance



Data



5-NN

Source [7]

- White areas correspond to unclassified regions where 5-NN voting is tied

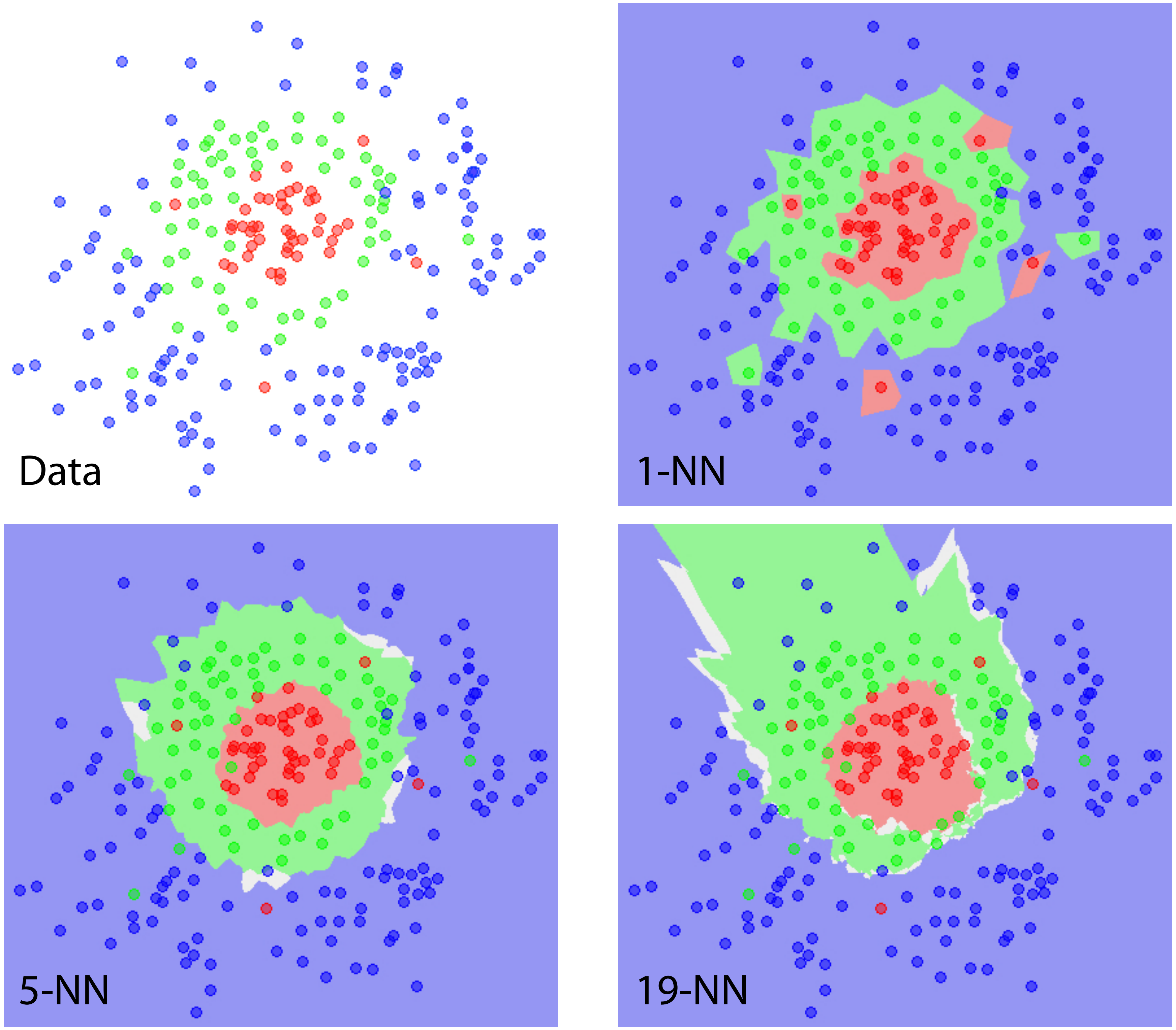
K-Nearest Neighbor

Parameter k

- To **avoid ties**, k should be an odd number or a well chosen number in the multi-class case
- If ties cannot be avoided, classes can be **drawn randomly**
- **Small values** of k may lead to overfitting in the presence of noise
- **Large values** of k have the advantage of **smoother decision boundaries** and more precise information about the **ambiguity of the decision** via the ratio of samples for each class
- However, too large values of k are **detrimental:** it **destroys the locality** of the estimation since farther examples are taken into account
- The proper choice of k **depends on the task** and can be estimated using cross-validation

K-Nearest Neighbor

Parameter k



Source [7]

- **1-NN:** noisy, overfitting
- **19-NN:** poor local approx. of true discriminant function
- **5-NN:** good compromise

K-Nearest Neighbor

Distance Metrics

- K-NN requires a **distance metric** to be defined that measures the similarity of any two vectors in feature space
- Typically, distances are measured with a **Minkowski distance** or L^p-norm

$$L^p(\mathbf{x}, \mathbf{x}') = (\sum_{i=1}^{m} |x_i - x_i'|^p)^{1/p}$$

- With $p = 2$ this is the **Euclidian distance**, with $p = 1$ we have the **Manhattan** (taxicab) **distance.** In the limiting case of p reaching infinity, we obtain the **Chebyshev distance**
- With Boolean feature vectors, the number of attributes/features on which the two points differ is called the **Hamming distance**
- There is research that focusses on **distance metric learning** with the goal to learn from data a function that measure how similar two objects are

K-Nearest Neighbor

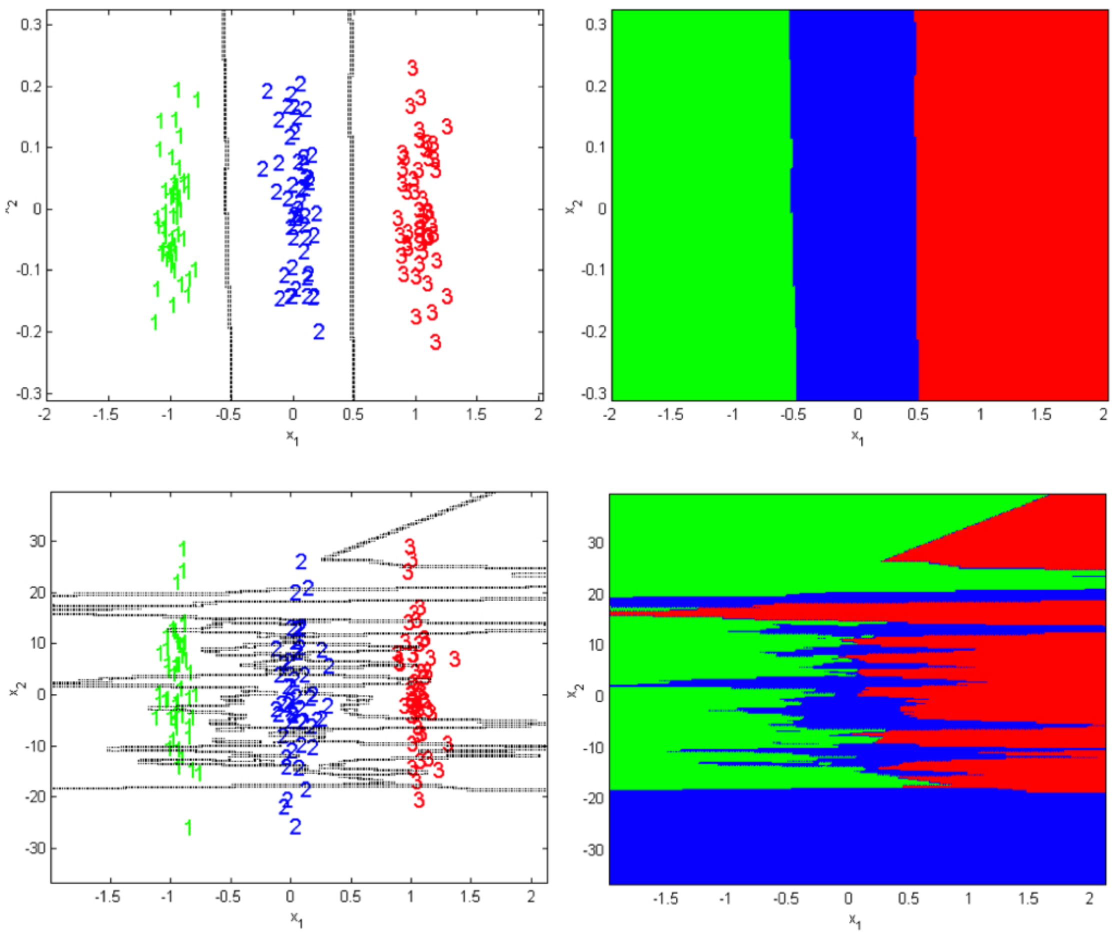
Distance Metrics

- K-NN heavily relies on the choice of the distance metric, particularly in **high-dimensional** feature spaces
- Generally, the concept of distance becomes less precise as the number of dimensions grows. In other words, in high dimensions "nearest" becomes **meaningless**
- With the Euclidean distance in high dimensions, for example, all vectors are **almost equidistant** to the query vector $\mathbf{x}'$
- Strange things happen in high dimensional spaces. The related phenomena are referred to as ”**curse of dimensionality**”
- **Irrelevant features** or **noise** dimensions may also affect k-NN performance
- Other distance metrics may or may not perform better in such cases. The best metric can be found using cross-validation

K-Nearest Neighbor

Feature Scaling

- k-NN is sensitive to **improperly scaled features**, particularly when used with the Euclidian distance
- Example: the x_1-feature contains all the discriminatory information. The x_2-feature is white noise, and does not contain classification information
- Top row: both axes are scaled properly
  - k-NN ($k = 5$) finds decision boundaries fairly close to the optimal
- Bottom row: x_2-feature multiplied by 100
  - The Gaussian distance metric is dominated by the large values of the x_2-feature. k-NN performs very poorly



Source [6]

K-Nearest Neighbor

Feature Scaling

- The general observation is that features with a particularly **broad range** of values **dominate** any distance computation between points
- Methods that employ a **distance function** such as nearest neighbor methods and SVMs are particularly sensitive to this
- Thus, k-NN, SVM as well as many other learning algorithms require that the input features are scaled to similar ranges, typically [0, 1] or [–1, 1]
- Let x is the original value and x' the normalized value, the simplest method to rescale features into a [0, 1] range is

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- Feature scaling (or data normalization) is a generally recommended preprocessing step for almost all learning tasks

K-Nearest Neighbor

Summary K-Nearest Neighbor

- **Non-parametric**, instance-based classifier
- Defined by a **parameter** k and a **distance metric**
- The k-nearest neighbors rule is one of the oldest and simplest methods for pattern recognition. Good baseline classifier in a comparison
- Trivial learning, expensive inference
- **Advantages**
  - Very simple to implement
  - Naturally multi-class
- **Drawbacks**
  - Large storage requirements, computationally intensive inference
  - Susceptible to the curse of dimensionality
  - Noise-sensitive to some extent, not probabilistic

Supervised Learning

Contents

- Introduction and basics
- Bayes Classifier
- Logistic Regression
- Support Vector Machines
- AdaBoost
- k-Nearest Neighbor
- Cross-validation
- Performance measures

Cross-Validation

Motivation

- Given a concrete classification task at hand, how do you find the **best classifier** for the problem? And how do you choose the **best values** of its "extrinsic" parameters?
- "Extrinsic" parameters, often called **hyperparameters**, are parameters that are not learned from data. Examples include: SVM kernel type and kernel parameters, neighborhood size k in k-NN or the number of rounds K in AdaBoost
- This is where **cross-validation** comes into play
- Cross-validation is a **model selection/validation technique** for assessing how a (learned) model will **generalize** to an independent data set
- Can be used for both, comparing different classifiers and comparing different sets of hyperparameter values

Cross-Validation

Validation Set

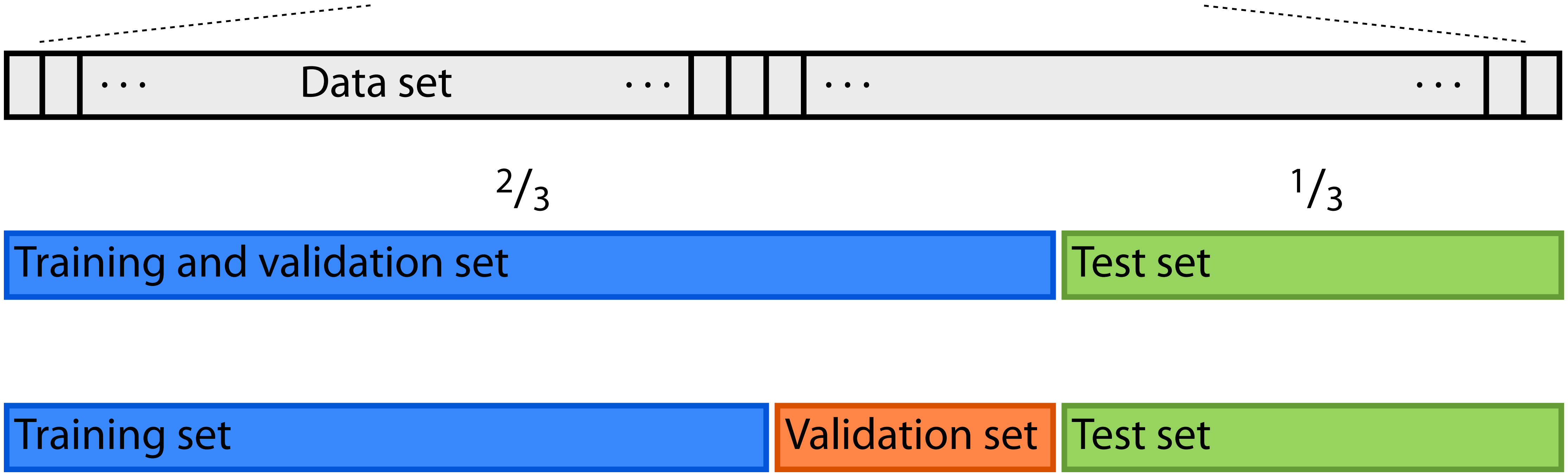
- Can't we just learn several classifiers and compare their **training errors?**
- This does not work for two reasons: **overfitting** to the training data may occur and more **complex models** will almost always give fewer errors than simpler ones. Complex models may generalize poorly (and contradict the principle of Occam's razor)
- We therefore need a data set **different** from the training set. This is called **validation set**
- A **single run** on that validation set might not be enough, in particular when data sets are small (e.g. when data or labels are costly) or when they contain noise and outliers that may mislead learning or validation
- Thus, we want to **average over several runs** and, in addition, **average over several validation sets** in order to avoid overfitting on the data of a single validation set

Cross-Validation

Training, Validation and Test Set

- How does the **test set** relate to training and validation sets? The test set is split from the data set and **kept apart** for **final evaluation.** A ratio of $^2/_3$ (training and validation) and $^1/_3$ (test) is typical
- Let $\mathcal{D}$ be the entire labeled data set

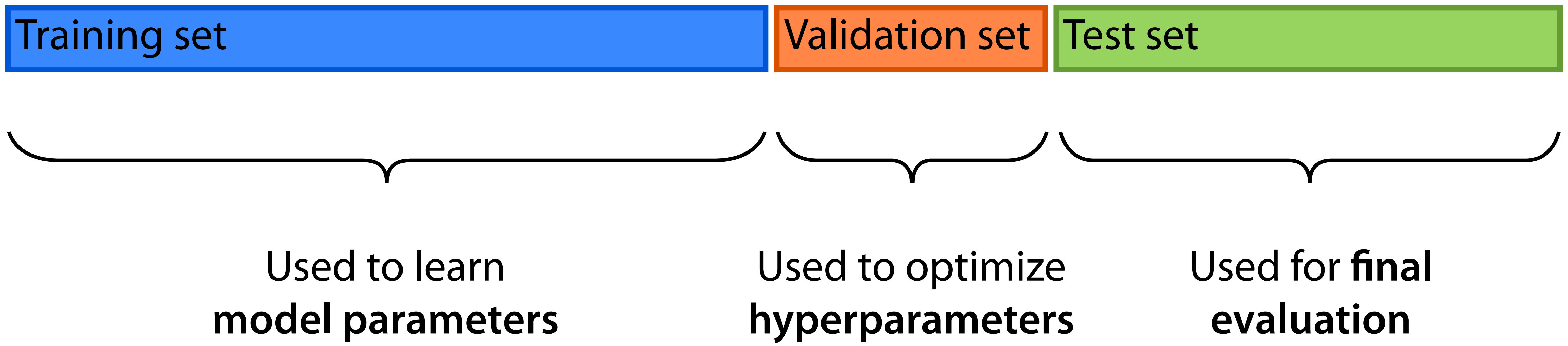
$$\mathcal{D} = \{(\mathbf{x}_1, y_1),\ (\mathbf{x}_2, y_2),\ \cdots\ ,(\mathbf{x}_N, y_N)\}$$



Cross-Validation

Training, Validation and Test Set

- Purposes of the different partitions

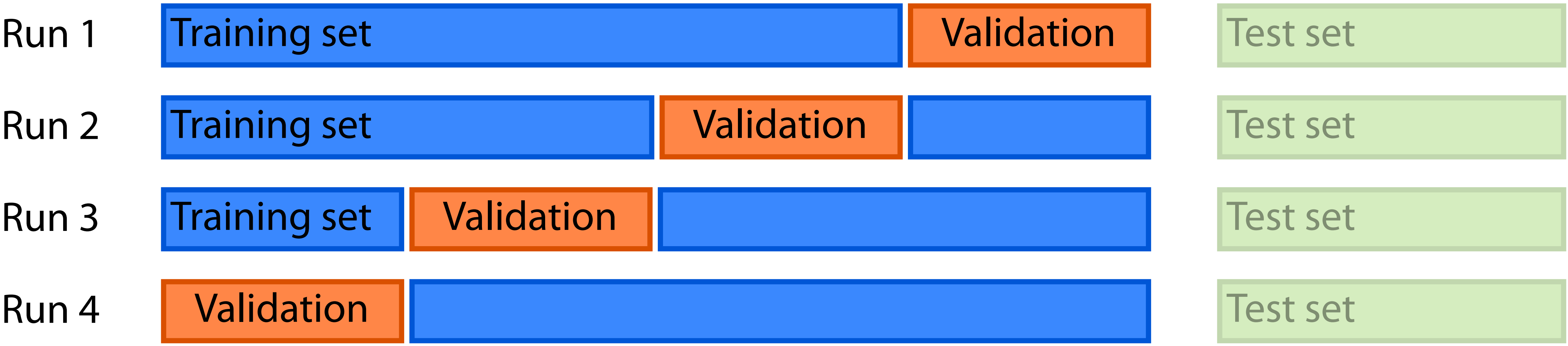

Training set
Validation set
Test set
Used to learn **model parameters**
Used to optimize **hyperparameters**
Used for **final evaluation**

- To be able to average over **several** validation sets, we need to generate K training/validation set pairs. The sets should be **as large as possible** so that error estimates are accurate, while **minimizing mutual overlap**
- Before any splitting is carried out, $\mathcal{D}$ must be **randomly permuted**

Cross-Validation

K-Fold Cross-Validation

- In **K-fold cross-validation**, the data are partitioned into K **folds**. One of the K folds is kept out as the validation set, the remaining K–1 form the training set. This is repeated K times
- K is typically 5, 10 or 30. Figure shows $K = 4$
- As N increases, K can be smaller. If N is small, K should be large to allow large enough training sets



Cross-Validation

Leave-One-Out Cross-Validation

- One extreme case of K-fold cross-validation is **leave-one-out cross-validation**
- Only **one sample** is left out as the validation "set" (a single instance) and training uses N–1 samples. This will require N runs over the set pairs
- This may be costly (we have to learn the classifier N times) but advisable in cases where labeled data are very hard to find such as medical diagnosis

Random Subsampling Cross-Validation

- **Random subsampling cross-validation** divides the data set into a training and validation set by **randomly drawing** samples from $\mathcal{D}$
- This **decouples** the number of runs from the number of folds but has the drawback that some samples may **never** be selected for validation, whereas others may be selected **more than once**

Cross-Validation

Evaluation Procedure

- In each of the K runs we assess the predictive accuracy of model candidate m by computing an error metric $\epsilon_m(k)$ over the respective validation set
- All K validation results are then **averaged** to get ϵ_m for model m
- This procedure is repeated for all model candidates m (i.e. different classifiers or classifiers with different hyperparameters) to find m* as the model with the smallest averaged error ϵ_m
- The **final evaluation** on the **test set** quantifies the performance of the best model m* using relevant metrics. **This step is always carried out**, also if cross-validation is skipped
- Of course, once the best classifier or best set of hyperparameters for an application is found, we **retrain the classifier** on **all labeled data**
- Hyperparameter optimization can be computationally very expensive

Performance Metrics

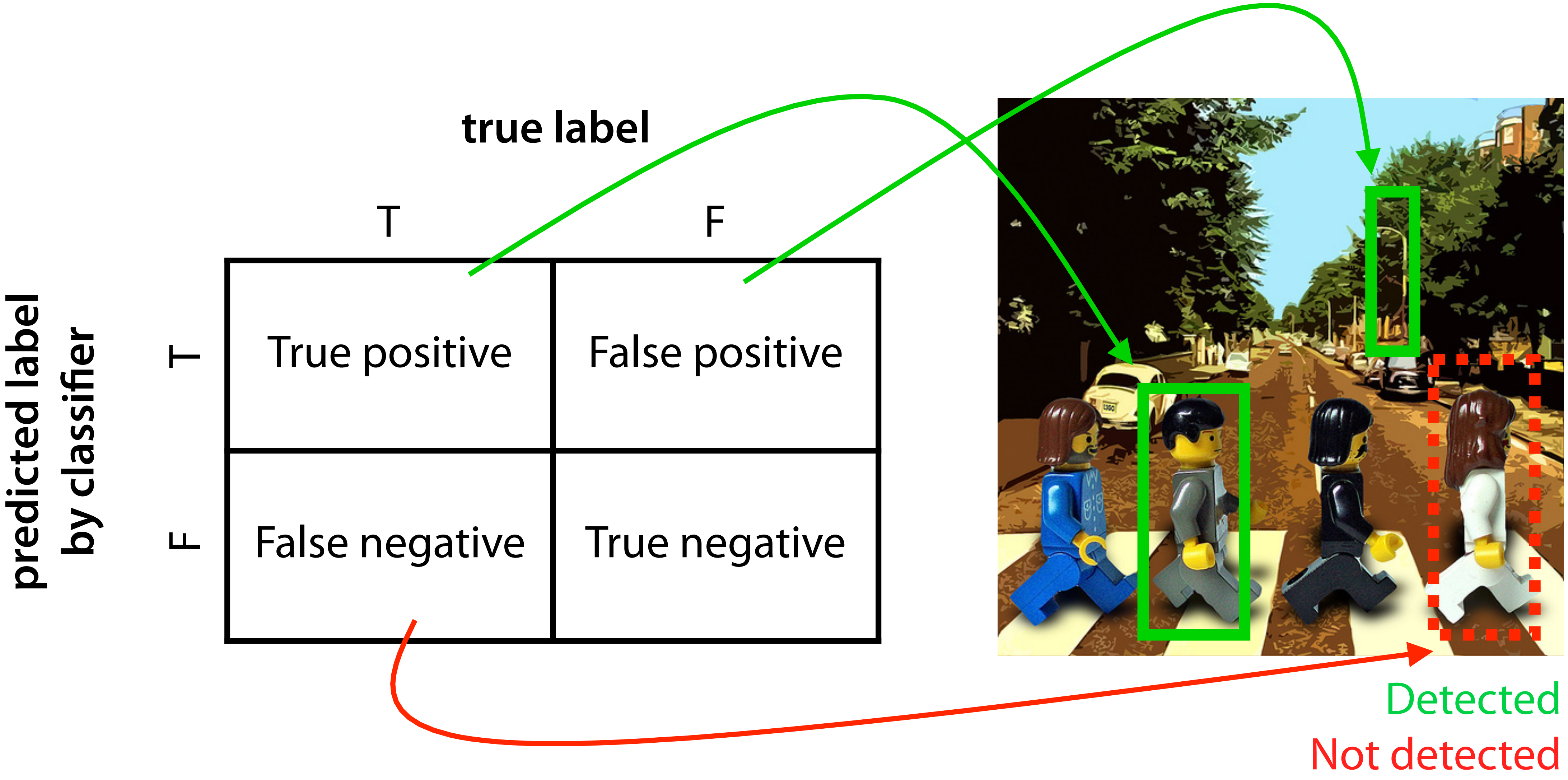
Motivation

- Once a classifier is learned, we want to measure its performance. So far, we have not been very specific on how to do that in terms of **performance measures**
- While learning and validation is done on the training and validation sets, performance is evaluated on an **independent test set**. This is done by iterating over the samples in the test set and comparing the **predicted labels** with the **true labels**
- Doing so we count four numbers in a binary classification problem: the number of **true positives** (TP), **false positives** (FP), **false negatives** (FN), and **true negatives** (TN)
- All measures of classification performance are based on **these four numbers**
- Note that $TP + FP + FN + TN = N$

Performance Metrics

Error Types

- The four numbers can be arranged into a 2 x 2 **confusion matrix** or **contingency table** (s x s for s classes)



		true label	
predicted label by classifier		T	F
	T	True positive	False positive
	F	False negative	True negative

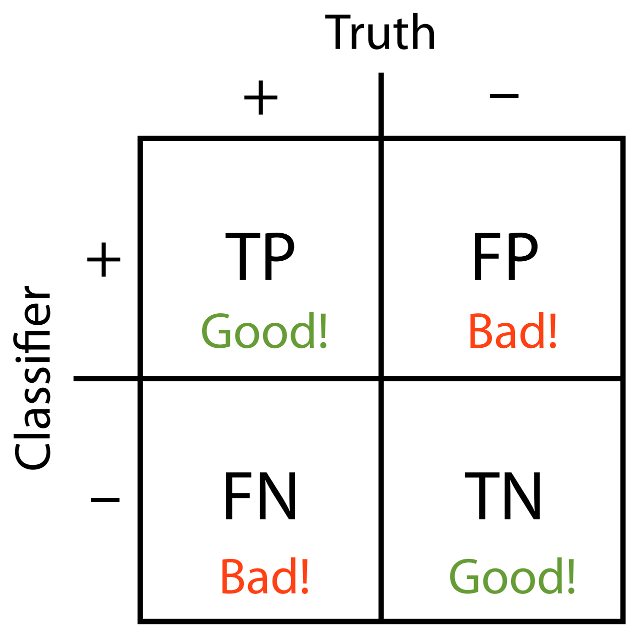
Performance Metrics

Error Types

- True positives (TP) and true negatives (TN) correspond to **correct** classifier predictions
- False positives (FP) are like ”**wrong alarms**” or ”**hallucinations**” (a.k.a. Type I errors)
- False negatives (FN) are like ”**missed detections**” (a.k.a. Type II errors)
- **Different combinations** of ratios have been given **various names**. All vary between 0 and 1

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

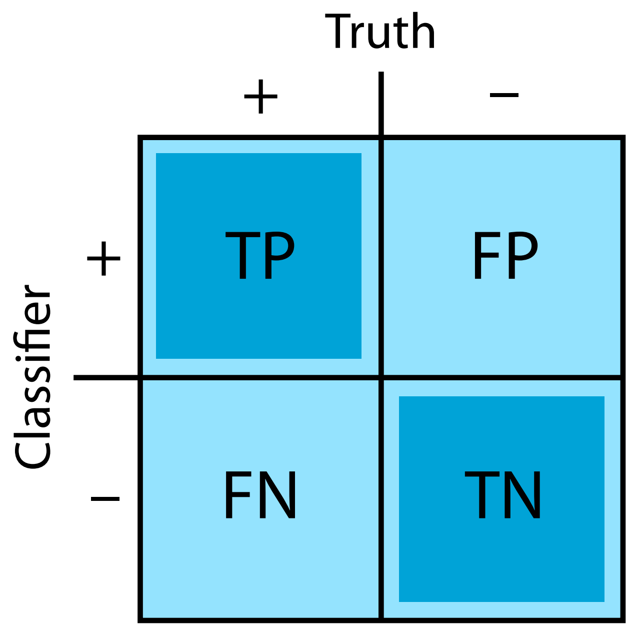
- Dark color is numerator, light color is denominator



Truth

Classifier

	+	−
+	TP Good!	FP Bad!
−	FN Bad!	TN Good!



Truth

Classifier

	+	−
+	TP	FP
−	FN	TN

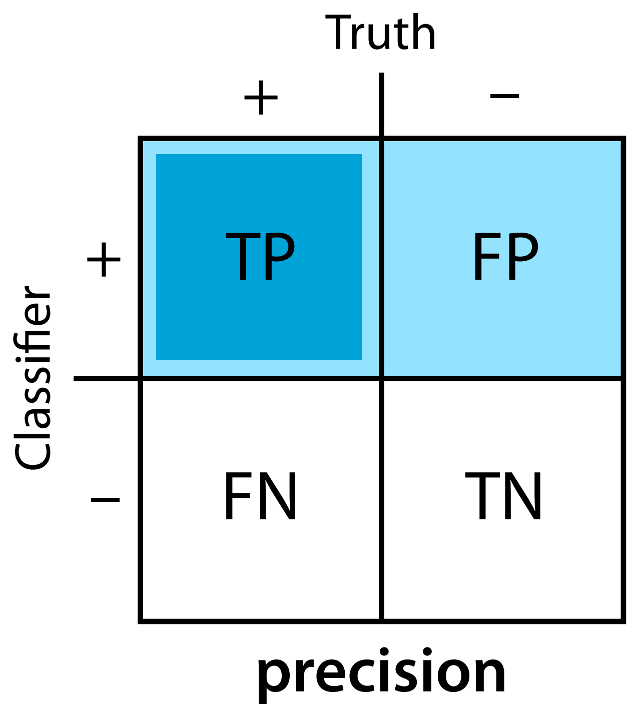
accuracy

Performance Metrics

Precision and Recall

- **Precision** is the fraction of detections (first row) that are truly relevant

$$\text{Precision} = \frac{TP}{TP + FP}$$



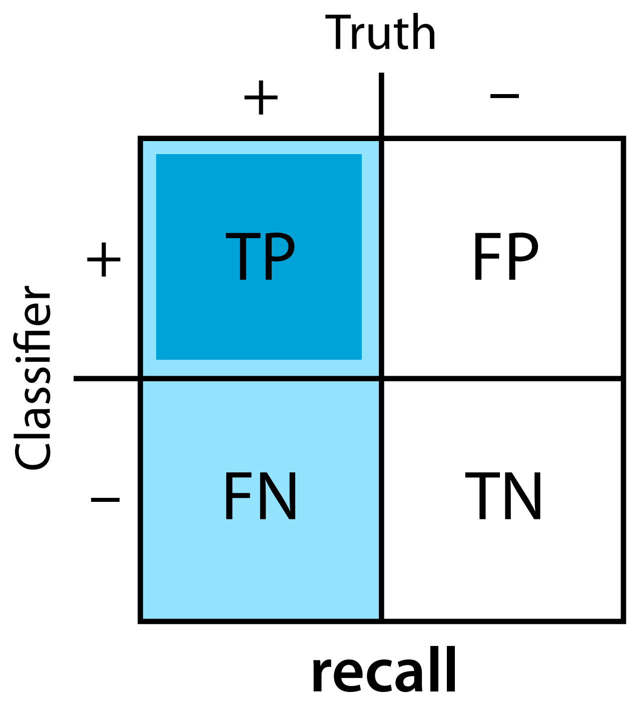
	+	−
+	TP	FP
−	FN	TN

- A conservative/"careful" classifier has high precision
- A precision score of 1.0 for a class C means that every item labeled as belonging to class C does indeed belong to class C
- But nothing is said about the (true) number of items from class C that were not labeled correctly (FN)
- Precision is also known as **positive predictive value** (PPV)

Performance Metrics

Precision and Recall

- **Recall** is the fraction of truly relevant instances (first column) that are correctly detected



$$\text{Recall} = \frac{TP}{TP + FN}$$

- A liberal/"loose" classifier has high recall
- A recall of 1.0 means that every item from class $\mathcal{C}$ was labeled as belonging to class $\mathcal{C}$
- But nothing is said about how many other items were incorrectly also labeled as belonging to class $\mathcal{C}$.
- Recall is also known as **true positive rate** (TPR) or **sensitivity**

Performance Metrics

F-Measure

- Precision or recall alone cannot fully measure a classifier's performance. The insight is that three of the counts in a confusion matrix can vary independently (the forth one follows from $TP + FP + FN + TN = N$)
- Hence, **no single number**, and **no pair of numbers**, can **characterize completely** the performance of a classifier
- Precision and recall are typically considered **jointly**: either by specifying one measure for a fixed level at the other measure (e.g. precision at recall of 0.75), by combing them into a single measure, or by plotting PR-curves
- Popular **single** performance measures are **accuracy** (see above) and **F-measure.** The F-measure takes the harmonic mean of precision and recall

$$\mathrm{F} = 2 \cdot \frac{\mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}$$

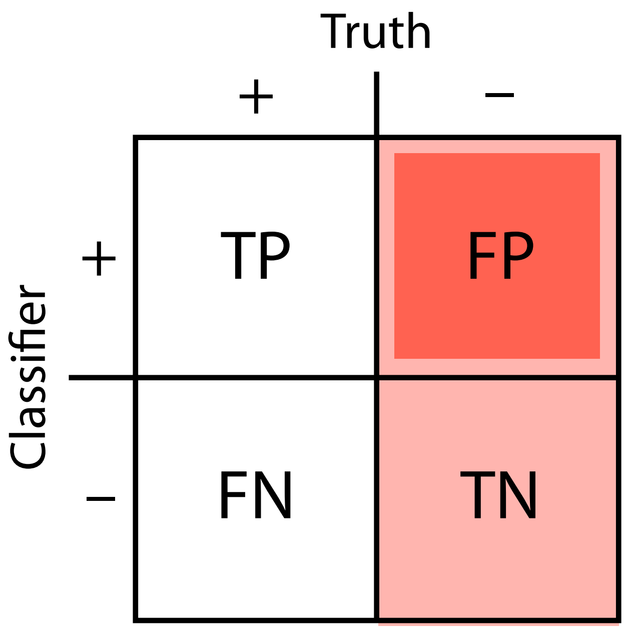
Performance Metrics

ROC Curves

- **Receiver Operating Characteristics** (ROC) are often used when evaluating binary classification problems. They offer a **more complete picture** of the performance of a classifier and provide a principled mechanism to explore operating point trade-offs
- A ROC curve shows how the number of correctly classified positive examples ("benefits") varies with the number of incorrectly classified negative examples ("costs")
- We define the **false positive rate (FPR)** as

$$\mathrm{FPR} = \frac{FP}{TN + FP}$$

- The false positive rate is also known as **false alarm rate** or **fall-out**

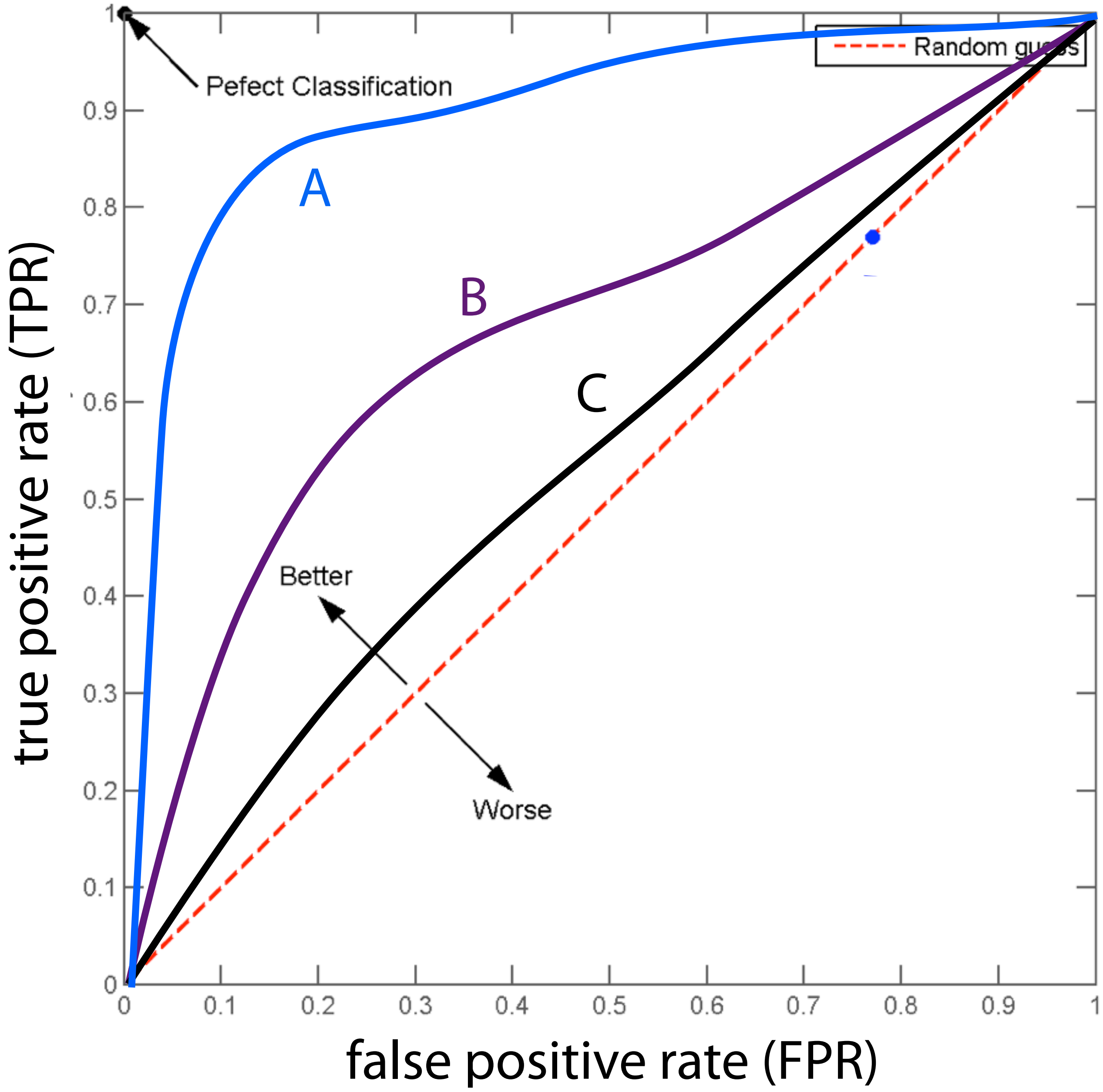


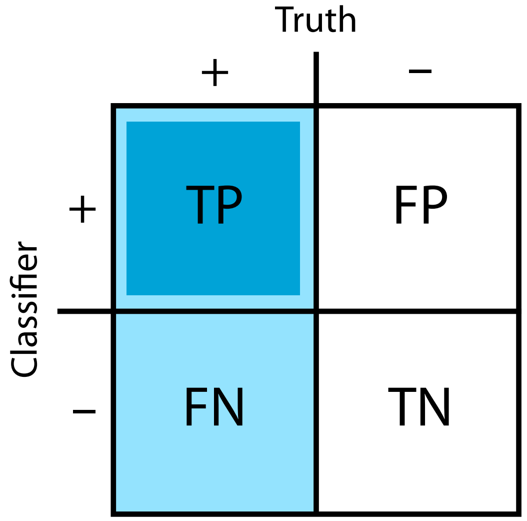
false positive rate

Performance Metrics

ROC Curves

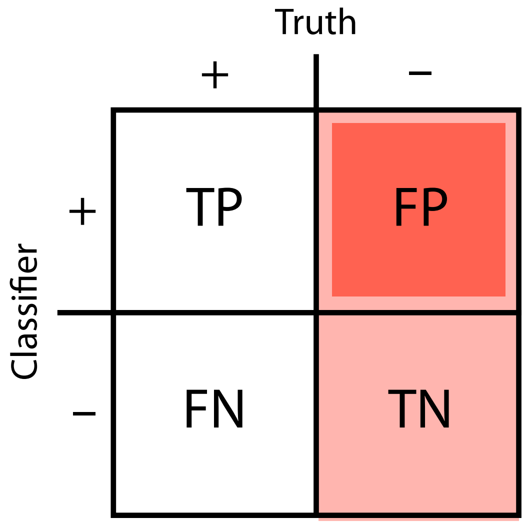
ROC Space





	+	–
+	TP	FP
–	FN	TN

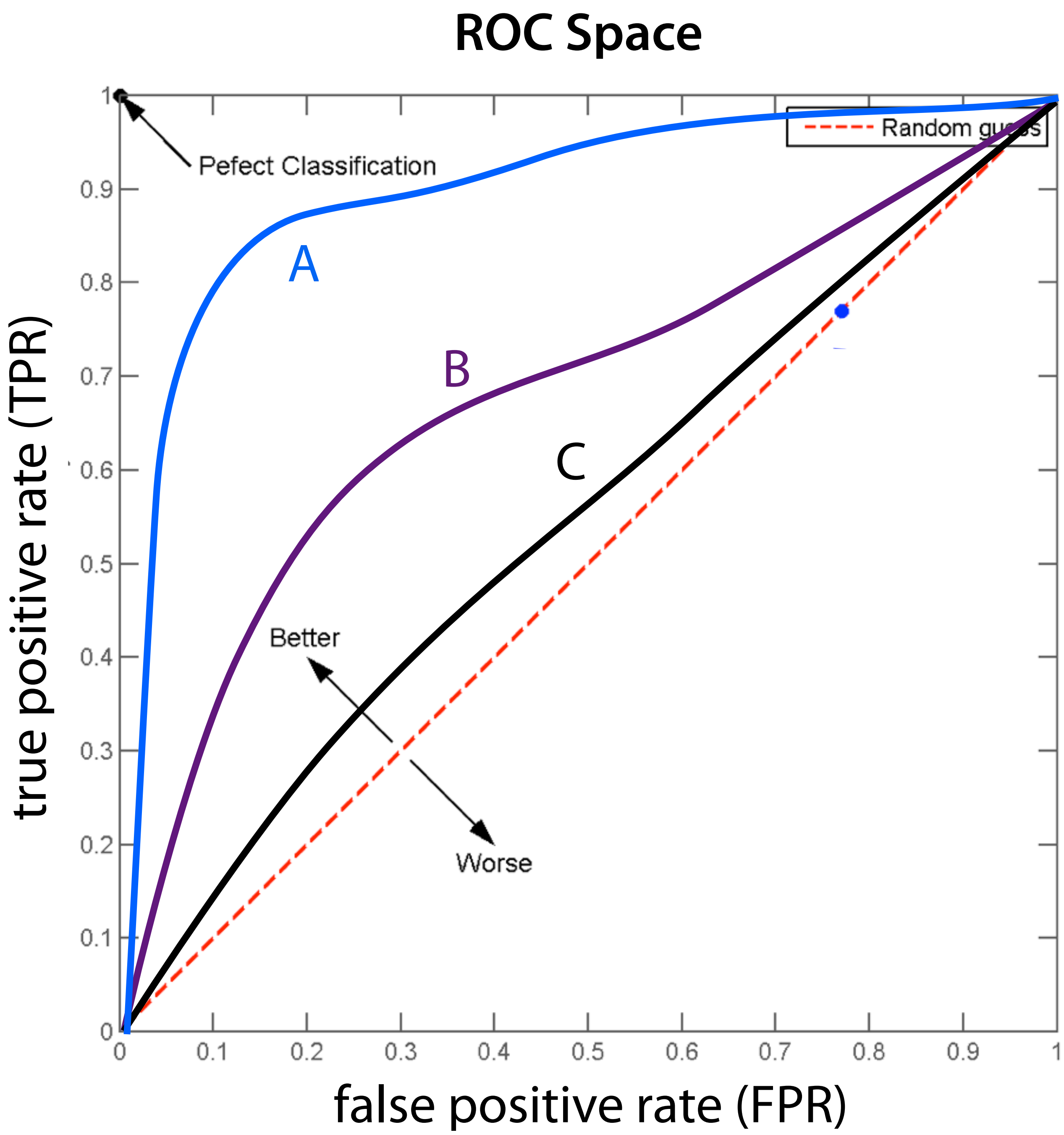
recall (TPR)



	+	–
+	TP	FP
–	FN	TN

false positive rate (FPR)

ROC Curves



- ROC curves plot recall/TPR versus FPR as the classifier goes from **“conservative”** to **“liberal”**
- Classifier C is close to **random guessing**
- Classifier B is better than classifier C
- Classifier A is better than classifier B

Performance Metrics

ROC Curves

- How to generate a ROC curve? Every point on the curve is a FPR/TPR-pair produced by the classifier at a given **discrimination threshold**
- Most classifiers naturally yield either a **probability** or a **score** that represents the degree to which a sample is a member of a class
  - Examples: the class probability in probabilistic classifiers, the confidence in AdaBoost or the y-value in SVMs
- Such classifiers then **threshold** this probability/score to predict the class
  - Examples: The sign(.) function in AdaBoost and SVM implies a fix discrimination threshold of 0 on the confidence or the y-value, respectively. For (binary) probabilistic classifiers the posterior class probability ratio is thresholded at a value of 1
- Now, instead of a fix value, the discrimination threshold is **varied** and the classifier is **re-evaluated** at every new threshold value. This method produces the points for the ROC curve

Performance Metrics

AUC and PR-Curves

- To compare classifiers we may want to reduce ROC performance to a **single** performance measure. A common method is to calculate the **area under the ROC curve**, abbreviated **AUC**
- Then, $AUC(h_1) > AUC(h_2)$ means that classifier h_1 has better average performance than classifier h_2
- However, ROC curves can present an overly optimistic view of an classifier's performance if there is a large skew/imbalance in the class distribution (very unequal numbers of sample for the positive/negative class)
- **Precision-Recall (PR) curves** are an alternative to ROC curves for tasks with imbalanced data. They can expose differences between classifiers that are not apparent in ROC space
- PR curves plot **precision versus recall** and are obtained in the same way than ROC curves

References

Sources and Further Reading

The AdaBoost section contains material by Matas and Sochman [1] and Grabner [2]. Small bits are also taken from Russell and Norvig [3] (chapter 18) and Bischop [4] (chapter 14). The k-NN section follows partly chapter 18.8 in [3] and contains material from the lecture notes of Gutierrez-Osuna [5]. See also the Wikipedia article on k-NN [6]. The Java applet on k-NN by Mirkes proved very useful to produce some of the picture [7]. The cross-validation section is based on Alpaydin's book [8] and the video lecture of mathematicalmonk [9]. The performance measure section uses material from Press' lecture notes [10|.

[1] J. Matas, J. Sochman, "AdaBoost", Lecture Notes, Centre for Machine Perception, Czech Technical University, Prague, 2010

[2] H. Grabner, "AdaBoost", 2008. Online: http://www.icg.tugraz.at/courses/lv710.084/BoostingProof.pdf/at_download/file (Dec 2013)

[3] S. Russell, P. Norvig, "Artificial Intelligence: A Modern Approach", 3rd edition, Prentice Hall, 2009. See http://aima.cs.berkeley.edu

[4] C.M. Bischop, "Pattern Recognition and Machine Learning", Springer, 2nd ed., 2007. See http://research.microsoft.com/en-us/um/people/cmbishop/prml

References

Sources and Further Reading

[5] R. Gutierrez-Osuna, "Pattern Recognition, Lecture 8: Nearest Neighbors", Lecture Notes, Texas A&M University, 2011

[6] Wikipedia, article "k-nearest neighbor algorithm", Online: http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm (Dec 2013)

[7] E.M. Mirkes, "KNN and Potential Energy: Applet", University of Leicester, 2011. Online: http://www.math.le.ac.uk/people/ag153/homepage/KNN/KNN3.html (Dec 2013)

[8] E. Alpaydin, "Introduction to Machine Learning", The MIT Press, 2009. See http://www.cmpe.boun.edu.tr/~ethem/i2ml2e

[9] mathematicalmonk, "(ML 12.5-12.7) Cross-validation", mathematicalmonk YouTube channel. Online: http://www.youtube.com/user/mathematicalmonk (Dec 2013)

[10] W.H. Press, "Unit 17: Classifier Performance: ROC, Precision-Recall, and All That", Lecture notes CS 395T, University of Texas at Austin, 2008