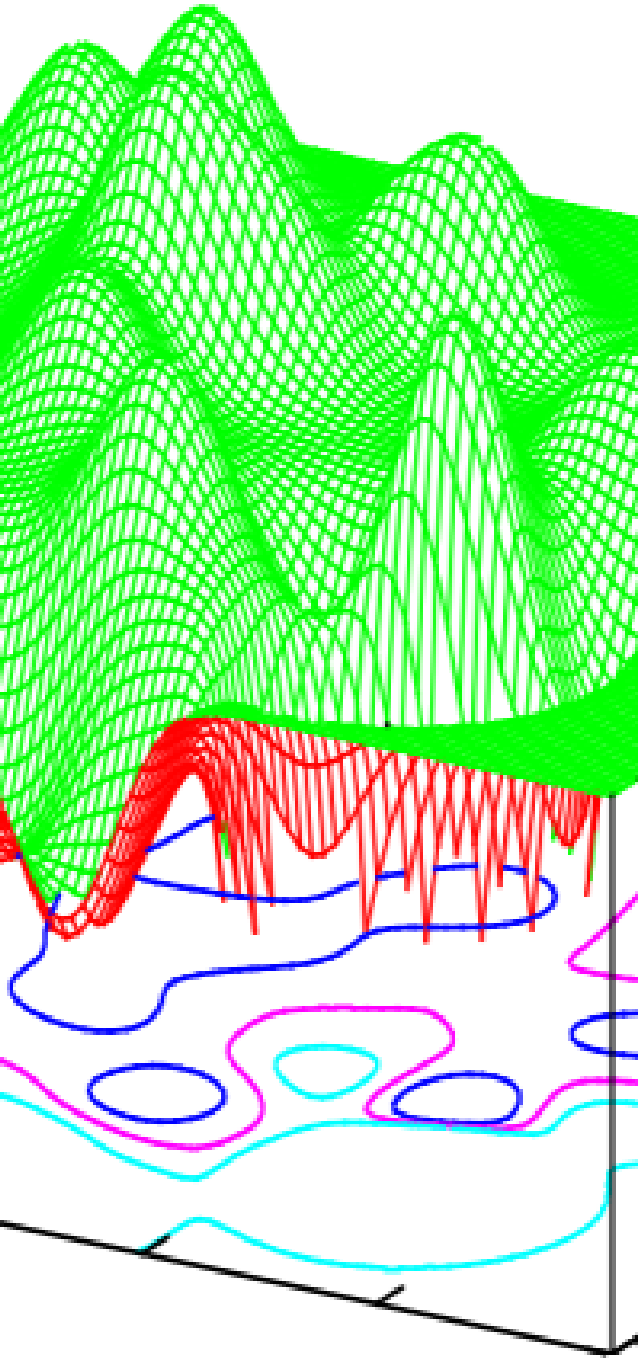


Machine Learning

The Breadth of ML methods

Marc Toussaint
University of Stuttgart
Summer 2016

- Other loss functions
  - Robust error functions (esp. Forsyth)
  - Hinge loss & SVMs
- Neural Networks & Deep learning
  - Basic equations
  - Deep Learning & regularizing the representation, autoencoders
- Unsupervised Learning
  - PCA, kernel PCA Spectral Clustering, Multidimensional Scaling, ISOMAP
  - Non-negative Matrix Factorization*, Factor Analysis*, ICA*, PLS*
- Clustering
  - k-means, Gaussian Mixture model
  - Agglomerative Hierarchical Clustering
- Local learners
  - local & lazy learning, kNN, Smoothing Kernel, kd-trees
- Combining weak or randomized learners
  - Bootstrap, bagging, and model averaging
  - Boosting
  - (Boosted) decision trees & stumps, random forests

Other loss functions

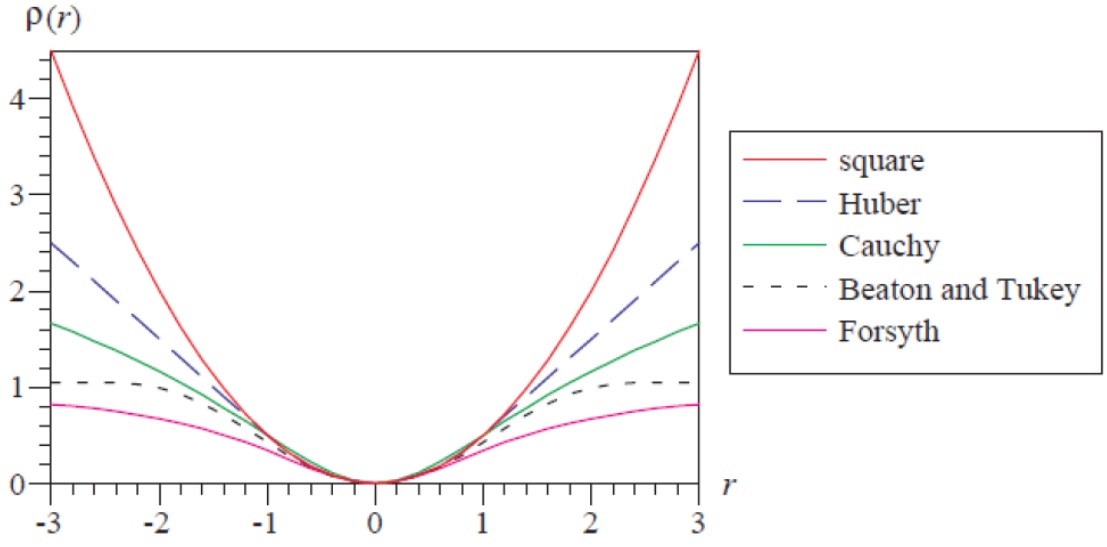
Loss functions

- So far we've talked about
  - **least squares** $L^{\text{ls}}(f) = \sum_{i=1}^{n} (y_i - f(x_i))^2$ for regression
  - **neg-log-likelihood** $L^{\text{nll}}(f) = -\sum_{i=1}^{n} \log p(y_i \mid x_i)$ for classification

- There are reasons to consider other loss functions:
  - Outliers are too important in least squares
  - In classification, maximizing a *margin* is intuitive
  - Intuitively, small errors should be penalized little or not at all
  - Some loss functions lead to sparse sets of **support vectors** $\rightarrow$ conciseness of function representation, good generalization

Robust error functions

- There is (at least) two approaches to handle outliers:
  - Analyze/label them explicitly and discount them in the error
  - Choose cost functions that are more “robust” to outliers



(Kheng’s lecture CVPR)

$r = y_i - f(x_i)$ is the *residual* (=model error)

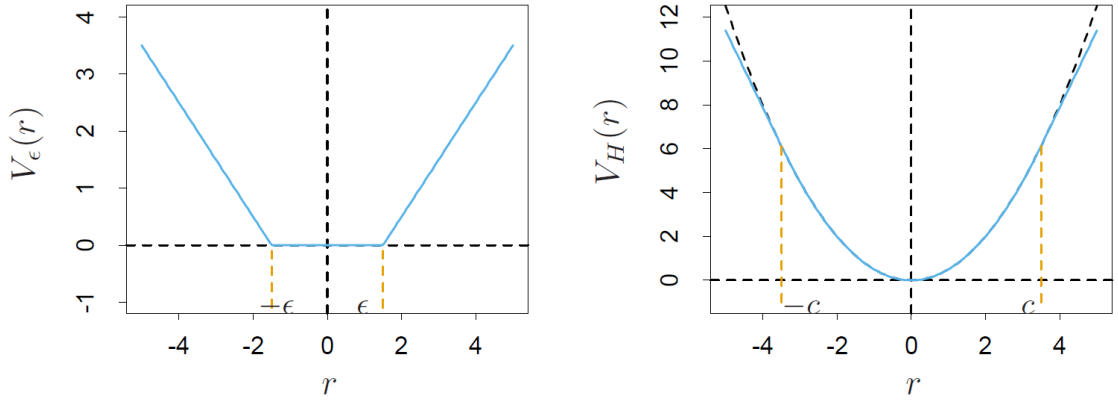
Robust error functions

- standard least squares: $L^{\text{ls}}(r) = r^2$
- **Forsyth**: $L^{\text{ls}}(r) = \frac{r^2}{a^2+r^2}$
- **Huber's robust regression** (1964): $L^{\text{H}}(r) = \begin{cases} r^2/2 & \text{if } |r| \leq c \\ c|r| - c^2/2 & \text{else} \end{cases}$
- Less used:
  - Cauchy: $L(r) = a^2/2 \log(1 + r^2/a^2)$
  - Beaton & Turkey: const for $|r| > a$, cubic inside

"Outliers have small/no gradient"

By this we mean: $\frac{\partial \beta^*(D)}{\partial y_i}$ is small: *Varying an outlier data point y_i does not have much influence on the optimal parameter β^** $\leftrightarrow$ robustness
Cf. our discussion of the estimator variance Var$\{\beta^*\}$

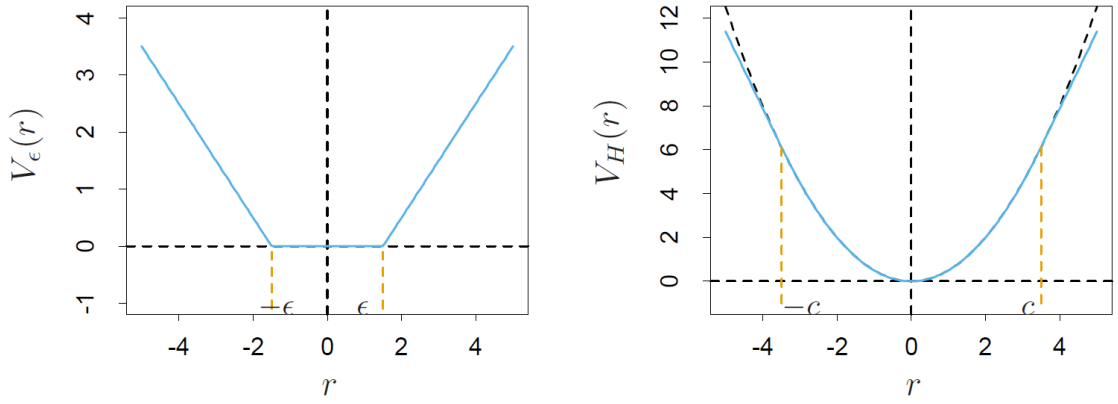
SVM regression: “Inliers have no gradient”*



(Hastie, page 435)

- **SVM regression (ϵ-insensitive)**: $L^\epsilon(r) = [|r| - \epsilon]_+$
  (sub-script $+$ indicates the positive part)

SVM regression: “Inliers have no gradient”*

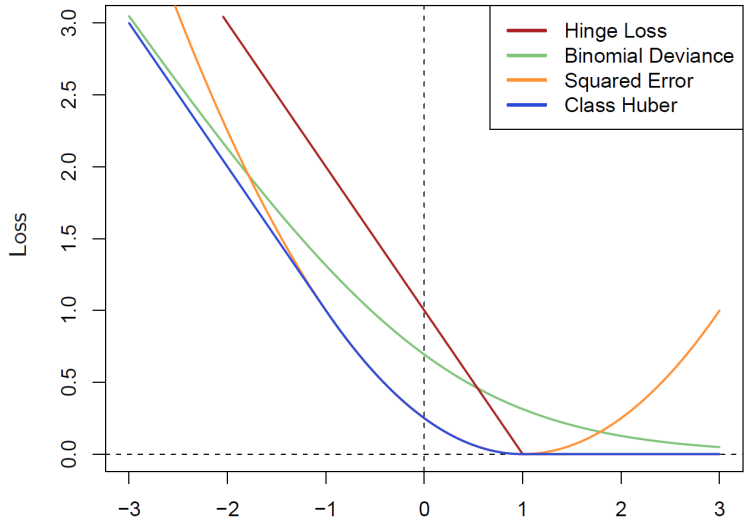


(Hastie, page 435)

- **SVM regression (ϵ-insensitive)**: $L^\epsilon(r) = [|r| - \epsilon]_+$
  (sub-script $+$ indicates the positive part)

- When optimizing for the SVM regression loss, the optimal regression (function f^*, or parameters β^*) does *not* vary inlier data points. It only depends on *critical* data points. These are called support vectors.
- In the kernel case $f(x) = \kappa(x)^\top \alpha$, every α_i relates to exactly one training point (cf. RBF features). Clearly, $\alpha_i = 0$ for those data points inside the margin! α is sparse; only support vectors have $\alpha \neq 0$.
- I skip the details for SVM regression and reiterate this for classification

Loss functions for classification



(Hastie, page 426)

- The "x-axis" denotes $y_i f(x_i)$, assuming $y_i \in \{-1, +1\}$
  - **neg-log-likelihood** (green curve above)
$$L^{\text{nll}}(y, f) = -\log p(y_i \mid x_i) = -\log \left[\sigma(f(x_i))^y + [1 - \sigma(f(x_i))]^{1-y}\right]$$
  - **hinge loss** $L^{\text{hinge}}(y, f) = [1 - yf(x)]_+ = \max\{0,\ 1 - yf(x)\}$
  - **class Huber's robust regression**
$$L^{\text{HC}}(r) = \begin{cases} -4yf(x) & \text{if } yf(x) \le -1 \\ [1 - yf(x)]_+^2 & \text{else} \end{cases}$$

- Neg-log-likelihood ("Binomial Deviance") and Huber have same asymtotes as the hinge loss, but rounded in the interior

Hinge loss

- Consider the hinge loss with L_2 regularization, minimizing

$$L^{\text{hinge}}(f) = \sum_{i=1}^{n} [1 - y_i f(x_i)]_+ + \lambda \|\beta\|^2$$

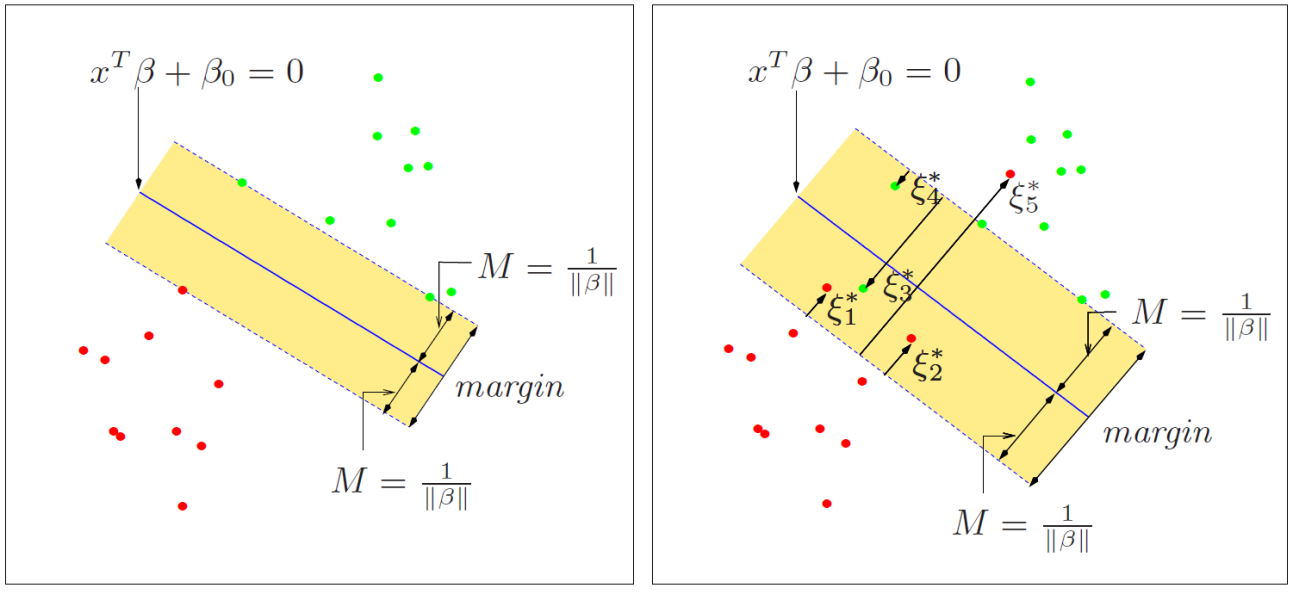
- It is easy to show that the following are equivalent

$$\min_{\beta} \|\beta\|^2 + C \sum_{i=1}^{n} \max\{0,\ 1 - y_i f(x_i)\} \tag{1}$$

$$\min_{\beta,\xi} \|\beta\|^2 + C \sum_{i=1}^{n} \xi_i \quad \text{s.t.} \quad y_i f(x_i) \geq 1 - \xi_i\ , \quad \xi_i \geq 0 \tag{2}$$

(where $C = 1/\lambda > 0$ and $y_i \in \{-1, +1\}$)

- The optimal model (optimal β) will "not depend" on data points for which $\xi_i = 0$. Here "not depend" is meant in the variational sense: If you would vary the data point (x_i) slightly, β^* would not be changed.



(Hastie, page 418)

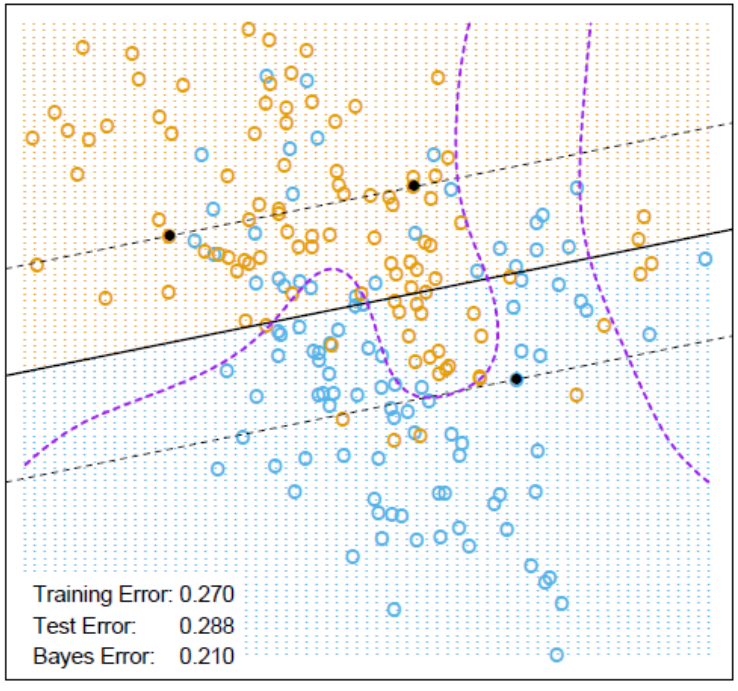
- Right:

$$\min_{\beta,\xi} \|\beta\|^2 + C\sum_{i=1}^{n} \xi_i \quad \text{s.t.} \quad y_i f(x_i) \geq 1 - \xi_i \ , \quad \xi_i \geq 0$$

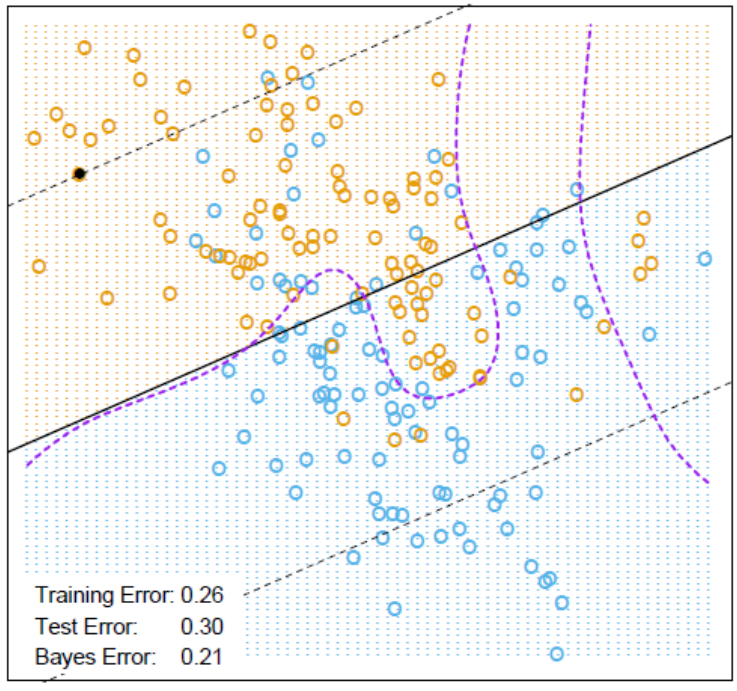
- Left: assuming $\forall_i : \ \xi_i = 0$ we just maximize the margin:

$$\min_{\beta} \|\beta\|^2 \quad \text{s.t.} \quad y_i f(x_i) \geq 1$$

- Right: We see that β^* does not depend directly (variationally) on x_i for data points outside the margin!
  Data points inside the margin or wrong classified: **support vectors**

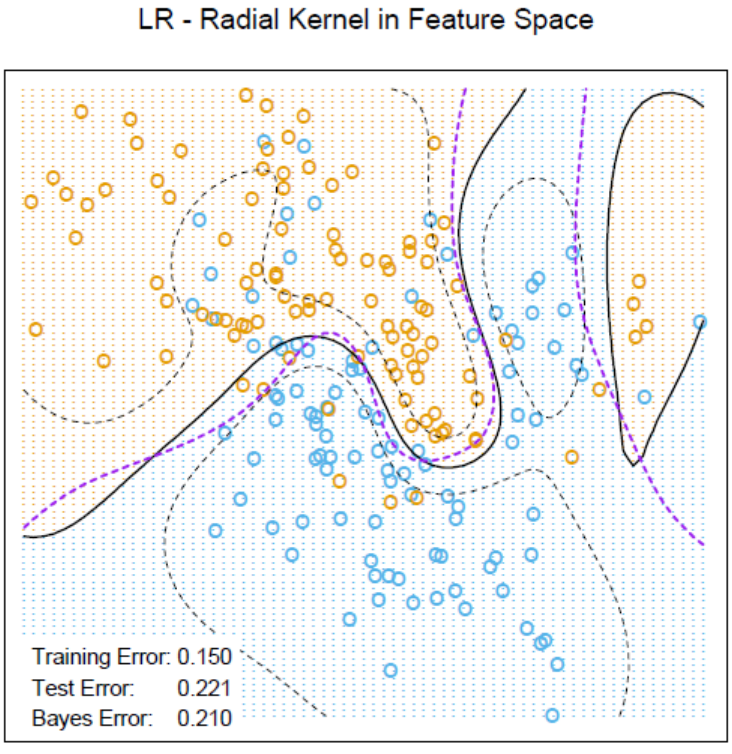


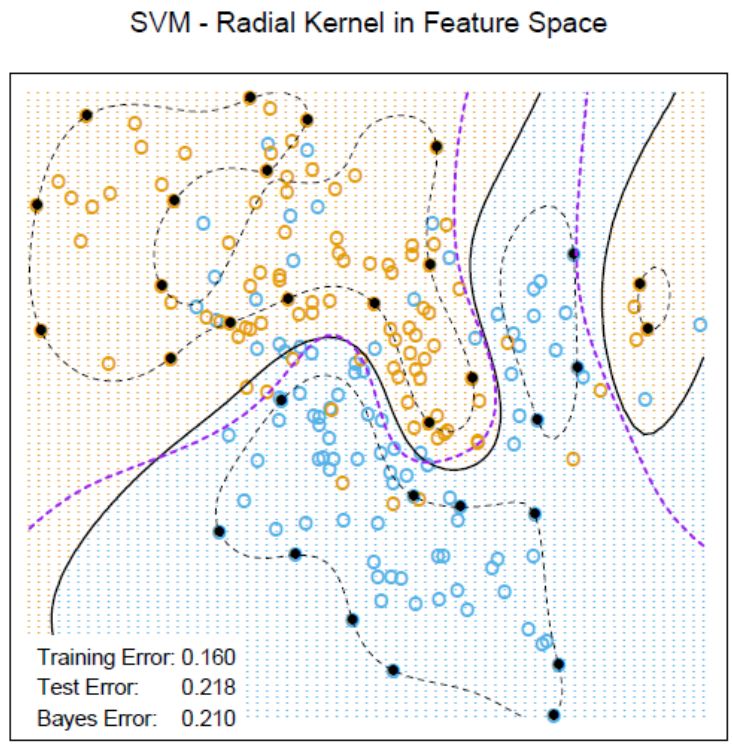
$C = 10000$



$C = 0.01$

- A linear SVM (on non-linear classification problem) for different regularization
  Black points: exactly on the margin ($\xi_i = 0, \alpha_i > 0$)
  Support vectors ($\alpha_i > 0$): inside of margin or wrong side of margin





- Kernel Logistic Regression (using neg-log-likelihood) vs. Kernel SVM (using hinge)

  [My view: these are equally powerful. The core difference is in how they can be trained, and how "sparse" the optimal α is]

Finding the optimal discriminative function

- The constrained problem

$$\min_{\beta,\xi} \|\beta\|^2 + C\sum_{i=1}^{n} \xi_i \quad \text{s.t.} \quad y_i(x_i^\top \beta) \geq 1 - \xi_i \ , \quad \xi_i \geq 0$$

  is a **linear program** and can be reformulated as the dual problem, with dual parameters α_i that indicate whether the constraint $y_i(x_i^\top \beta) \geq 1 - \xi_i$ is active. The dual problem is **convex**. SVM libraries use, e.g., CPLEX to solve this.

- For all inactive constraints ($y_i(x_i^\top \beta) \geq 1$) the data point (x_i, y_i) does not directly influence the solution β^*. Active points are support vectors.

- Let (x, ξ) be the primal variables, (α, μ) the dual, we derive the dual problem:

$$\min_{\beta,\xi} \|\beta\|^2 + C\sum_{i=1}^n \xi_i \quad \text{s.t.} \quad y_i(x_i^\top \beta) \geq 1 - \xi_i \ , \quad \xi_i \geq 0 \tag{3}$$

$$L(\beta, \xi, \alpha, \mu) = \|\beta\|^2 + C\sum_{i=1}^n \xi_i - \sum_{i=1}^n \alpha_i[y_i(x_i^\top \beta) - (1 - \xi_i)] - \sum_{i=1}^n \mu_i \xi_i \tag{4}$$

$$\partial_\beta L \stackrel{!}{=} 0 \quad \Rightarrow \quad 2\beta = \sum_{i=1}^n \alpha_i y_i x_i \tag{5}$$

$$\partial_\xi L \stackrel{!}{=} 0 \quad \Rightarrow \quad \forall_i : \ \alpha_i = C - \mu_i \tag{6}$$

$$l(\alpha, \mu) = \min_{\beta,\xi} L(\beta, \xi, \alpha, \mu) = -\tfrac{1}{4}\sum_{i=1}^n \sum_{i'=1}^n \alpha_i \alpha_{i'} y_i y_{i'} \hat{x}_i^\top \hat{x}_{i'} + \sum_{i=1}^n \alpha_i \tag{7}$$

$$\max_{\alpha,\mu} l(\alpha, \mu) \quad \text{s.t.} \quad 0 \leq \alpha_i \leq C \tag{8}$$

- 4: Lagrangian (with negative Lagrange terms because of $\geq$ instead of $\leq$)
- 5: the optimal β^* depends only on $x_i y_i$ for which $\alpha_i > 0 \rightarrow$ support vectors
- 7: This assumes that $x_i = (1, \hat{x}_i)$ includes the constant feature 1 (so that the statistics become centered)
- 8: This is the dual problem. $\mu_i \leq 0$ implies $\alpha_i \leq C$
- Note: the dual problem only refers to $\hat{x}_i^\top \hat{x}_i \ \rightarrow$ **kernelization**

Conclusions on the hinge loss

- The hinge implies an important structural aspect of optimal solutions: support vectors
- A **core benefit of the hinge loss/SVMs**: These optimization problems are structurally different to the (dense) Newton methods for logistic regression. They can scale well to **large data sets** using specialized constrained optimization methods. (However, there also exist incremental, well-scaling methods for kernel ridge regression/GPs.)
  (I personally think: the benefit is *not* in terms of the predictive performance, which is just different, not better or worse.)

- The hinge loss can equally be applied to conditional random fields (including multi-class classification), which is called **structured-output SVM**

Comments on comparing classifiers*

- Given two classification algorithm that optimize different loss functions, what are objective ways to compare them?

- Often one only measures the accuracy (rate of correct classification). A test (e.g. paired t-test) can check the *significance* of difference between the algorithms.
  On comparing classifiers: Pitfalls to avoid and a recommended approach SL Salzberg - Data Mining and knowledge discovery, 1997 - Springer
  Approximate statistical tests for comparing supervised classification learning algorithms TG Dietterich - Neural computation, 1998 - MIT Press

- Critical view:
  Provost, Fawcett, Kohavi: *The case against accuracy estimation for comparing induction algorithms.* ICML, 1998

Precision, Recall & ROC curves*

- Measures: data size n, *false positives* (FP), *true positives* (TP), data positives (DP=TP+FN), etc:

$$\textit{accuracy} = \frac{\textit{TP+TN}}{n}$$
$$\textit{precision} = \frac{\textit{TP}}{\textit{TP+FP}}$$
$$\textit{recall (TP-rate)} = \frac{\textit{TP}}{\textit{TP+FN=DP}}$$
$$\textit{FP-rate} = \frac{\textit{FP}}{\textit{FP+TN=DN}}$$

- Binary classifiers typically define a discriminative function $f(x) \in \mathbb{R}$. (Or $f(y=1, x) = f(x), f(y=0, x) = 0$.) The "decision threshold" typically is zero:

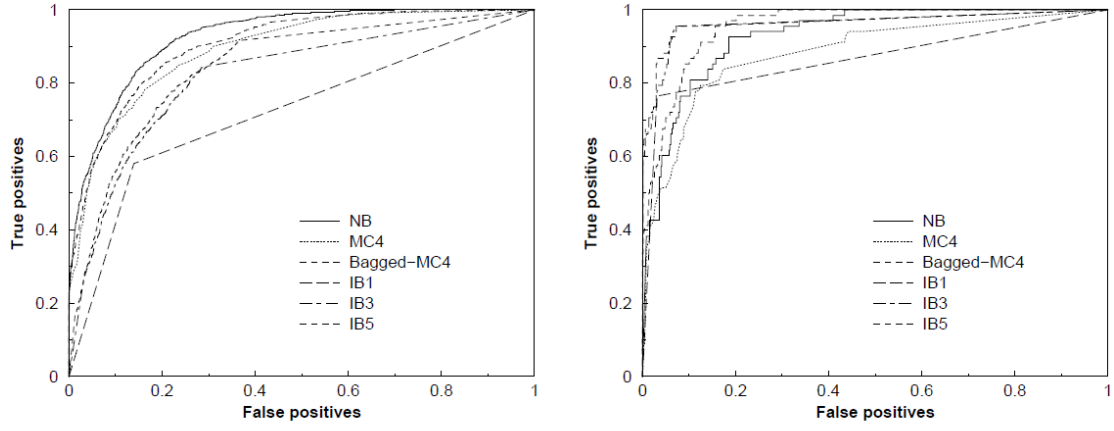
$$y^* = \operatorname*{argmax}_y f(y, x) = [f(x) > 0]$$

  But in some applications one might want to tune the threshold *after* training.

- decreasing threshold: TPR$\to 1$, increasing threshold: FPR$\to 1$

ROC curves*

- ROC curves (Receiver Operating Characteristic) plot TPR vs. FPR for all possible choices of decision thresholds:



(a) Adult (b) Satimage

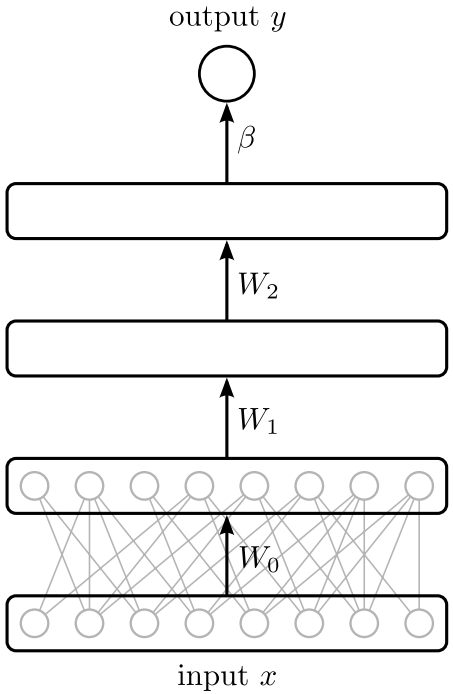
- Provost, Fawcett, Kohavi empirically demonstrate, that rarely an algorithm dominates another algorithm over the *whole* ROC curve $\rightarrow$ there always exists decision thresholds (cost assignments to TPR/FPR) for which the other algorithm is better.

- Their conclusion: Show the ROC curve as a whole
- Some report on the area under the ROC curve – which is intuitive but meaningless...

Neural Networks

Neural Networks

- Consider a regression problem with input $x \in \mathbb{R}^d$ and output $y \in \mathbb{R}$



  - Linear function: ($\beta \in \mathbb{R}^d$)
$$f(x) = \beta^\top x$$
  - 1-layer Neural Network function: ($W_0 \in \mathbb{R}^{h_1 \times d}$)
$$f(x) = \beta^\top \sigma(W_0 x)$$
  - 2-layer Neural Network function:
$$f(x) = \beta^\top \sigma(W_1 \sigma(W_0 x))$$

- Neural Networks are a special function model $y = f(x, w)$, i.e. a special way to parameterize non-linear functions

Neural Networks: basic equations

- Consider L hidden layers, each h_l-dimensional
  - let $z_l = W_{l\text{-}1} x_{l\text{-}1}$ be the inputs to all neurons in layer l
  - let $x_l = \sigma(z_l)$ the **activation** of all neurons in layer l
  - redundantly, we denote by $x_0 \equiv x$ the activation of the input layer, and by $\phi(x) \equiv x_L$ the activation of the last hidden layer
- **Forward propagation:** An L-layer NN recursively computes, for $l = 1, .., L$,

$$\forall_{l=1,..,L} : \; z_l = W_{l\text{-}1} x_{l\text{-}1} \; , \quad x_l = \sigma(z_l)$$

  and then computes the output $f \equiv z_{L+1} = W_L x_L$
- **Backpropagation:** Given some loss $\ell(f)$, let $\delta_{L+1} \triangleq \frac{\partial \ell}{\partial f}$. We can recursivly compute the loss-gradient w.r.t. the *inputs* of layer l:

$$\forall_{l=L,..,1} : \; \delta_l \triangleq \frac{d\ell}{dz_l} = \frac{d\ell}{dz_{l+1}} \frac{\partial z_{l+1}}{\partial x_l} \frac{\partial x_l}{\partial z_l} = [\delta_{l+1} \; W_l] \circ [x_l \circ (1 - x_l)]^\top$$

  where $\circ$ is an *element-wise product*. The gradient w.r.t. weights is:

$$\frac{d\ell}{dW_{l,ij}} = \frac{d\ell}{dz_{l+1,i}} \frac{\partial z_{l+1,i}}{\partial W_{1,ij}} = \delta_{l+1,i} \; x_{l,j} \quad \text{or} \quad \frac{d\ell}{dW_l} = \delta_{l+1}^\top x_l^\top$$

NN regression & regularization

- In the standard regression case, $y \in \mathbb{R}$, we typically assume a squared error loss $\ell(f) = \sum_i (f(x_i, w) - y_i)^2$. We have

$$\delta_{L+1} = \sum_i 2(f(x_i, w) - y_i)^\top$$

- Regularization: Add a L_2 or L_1 regularization. First compute all gradients as before, then add $\lambda W_{l,ij}$ (for L_2), or $\lambda \operatorname{sign} W_{l,ij}$ (for L_1) to the gradient. Historically, this is called **weight decay**, as the additional gradient leads to a step decaying the weighs.
- The optimal output weights are as for standard regression

$$W_L^* = (X^\top X + \lambda I)^{-1} X^\top y$$

where X is the data matrix of activations $x_L \equiv \phi(x)$

NN classification

- Consider the multi-class case $y \in \{1, .., M\}$. Then we have M output neurons to represent the **discriminative function**

$$f(x, y, w) = (W_L^\top z_L)_y \ , \quad W \in \mathbb{R}^{M \times h_L}$$

- Choosing neg-log-likelihood objective $\leftrightarrow$ logistic regression
- Choosing hinge loss objective $\leftrightarrow$ "NN + SVM"
  - For given x, let y^* be the correct class. The **one-vs-all** hinge loss is $\sum_{y \neq y^*} [1 - (f_{y^*} - f_y)]_+$
  - For output neuron $y \neq y^*$ this implies a gradient $\delta_y = [f_{y^*} < f_y + 1]$
  - For output neuron y^* this implies a gradient $\delta_{y^*} = -\sum_{y \neq y^*} [f_{y^*} < f_y + 1]$ Only data points inside the margin induce an error (and gradient).
  - This is also called **Perceptron Algorithm**

NN model selection

- NN have many **hyper parameters**: The choice of λ as well as L, and h_l

$$\rightarrow \text{ cross validation}$$

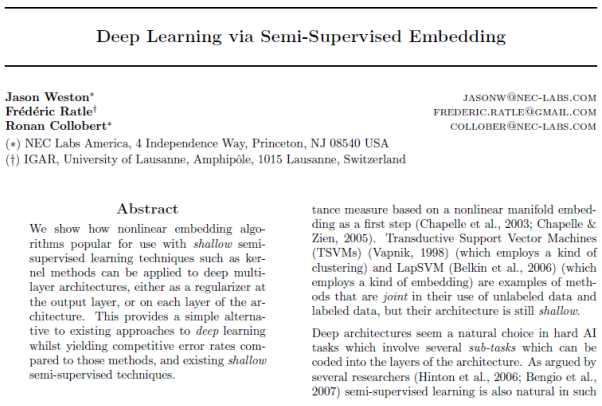
Deep Learning

- Discussion papers:
  - Thomas G. Dietterich et al.: *Structured machine learning: the next ten years*
  - Bengio, Yoshua and LeCun, Yann: *Scaling learning algorithms towards AI*
  - Rodney Douglas et al.: *Future Challenges for the Science and Engineering of Learning*
  - Pat Langley: *The changing science of machine learning*
  - Glorot, Bengio: *Understanding the difficulty of training deep feedforward neural networks*

- There are open issues w.r.t. **representations** (e.g. discovering/developing abstractions, hierarchies, etc).
  Deep Learning and the revival of Neural Networks was driven by this

Deep Learning

- Must read:



Deep Learning via Semi-Supervised Embedding

Jason Weston[*] JASONW@NEC-LABS.COM
Frédéric Ratle[†] FREDERIC.RATLE@GMAIL.COM
Ronan Collobert[*] COLLOBER@NEC-LABS.COM
(*) NEC Labs America, 4 Independence Way, Princeton, NJ 08540 USA
(†) IGAR, University of Lausanne, Amphipôle, 1015 Lausanne, Switzerland

Abstract

We show how nonlinear embedding algorithms popular for use with *shallow* semi-supervised learning techniques such as kernel methods can be applied to deep multi-layer architectures, either as a regularizer at the output layer, or on each layer of the architecture. This provides a simple alternative to existing approaches to *deep* learning whilst yielding competitive error rates compared to those methods, and existing *shallow* semi-supervised techniques.

tance measure based on a nonlinear manifold embedding as a first step (Chapelle et al., 2003; Chapelle & Zien, 2005). Transductive Support Vector Machines (TSVMs) (Vapnik, 1998) (which employs a kind of clustering) and LapSVM (Belkin et al., 2006) (which employs a kind of embedding) are examples of methods that are *joint* in their use of unlabeled data and labeled data, but their architecture is still *shallow*.

Deep architectures seem a natural choice in hard AI tasks which involve several *sub-tasks* which can be coded into the layers of the architecture. As argued by several researchers (Hinton et al., 2006; Bengio et al., 2007) semi-supervised learning is also natural in such

Jason Weston et al.: *Deep Learning via Semi-Supervised Embedding* (ICML 2008)

`www.thespermwhale.com/jaseweston/papers/deep_embed.pdf`

- Rough ideas:
  - Multi-layer NNs are a “deep” function model
  - Only training on least squares gives too little “pressure” on inner representations
  - Mixing least squares cost with representational costs leads to better representations & generalization

Deep Learning

- In this perspective,
  1) Deep Learning considers deep function models (NNs usually)
  2) Deep Learning makes explicit statements about what internal representations should capture

- In Weston et al.'s case

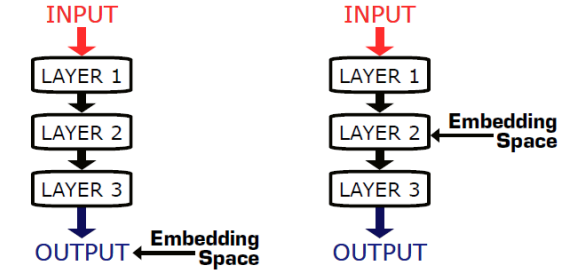
$$L(\beta) = \underbrace{\sum_{i=1}^{n}(y_i - f(x_i))^2}_{\text{squared error}} + \lambda \sum_i \underbrace{\sum_{jk}(\|z_{ij} - z_{ik}\|^2 - d_{jk})^2}_{\text{Multidimensional Scaling regularization}}$$

  where $z_i = \sigma(W_{i\text{-}1} z_{i\text{-}1})$ are the activations in the ith layer

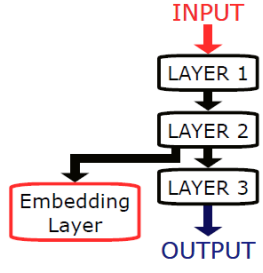
  This mixes the squared error with a representational cost for each layer

Deep Learning

- Results from Weston et al. (ICML, 2008)



(a) Output (b) Internal



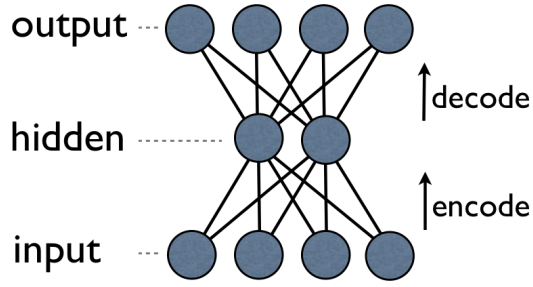
(c) Auxiliary

Mnist1h dataset, deep NNs of 2, 6, 8, 10 and 15 layers; each hidden layer 50 hidden units

	2	4	6	8	10	15
NN	26.0	26.1	27.2	28.3	34.2	47.7
$Embed^{O}$NN	19.7	15.1	15.1	15.0	13.7	11.8
$Embed^{ALL}$NN	18.2	12.6	7.9	8.5	6.3	9.3

Similar idea: Stacking Autoencoders

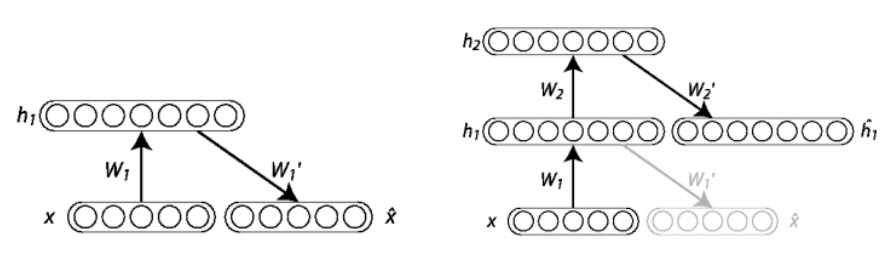
- An autoencoder typically is a NN of the type



  which is trained to reproduce the input: $\min_i \|y(x_i) - x_i\|^2$
  The hidden layer ("bottleneck") needs to find a good representation/compression.

- Similar to the PCA objective, but nonlinear

- Stacking autoencoders:



Deep Learning – further reading

- Weston, Ratle & Collobert: *Deep Learning via Semi-Supervised Embedding*, ICML 2008.
- Hinton & Salakhutdinov: *Reducing the Dimensionality of Data with Neural Networks*, Science 313, pp. 504-507, 2006.
- Bengio & LeCun: *Scaling Learning Algorithms Towards AI*. In Bottou et al. (Eds) “Large-Scale Kernel Machines”, MIT Press 2007.
- Hadsell, Chopra & LeCun: *Dimensionality Reduction by Learning an Invariant Mapping*, CVPR 2006.
- Glorot, Bengio: *Understanding the difficulty of training deep feedforward neural networks*, AISTATS’10.

... and newer papers citing those

Unsupervised learning

Unsupervised learning

- What does that mean? Generally: modelling $P(x)$
- Instances:
  - Finding lower-dimensional spaces
  - Clustering
  - Density estimation
  - Fitting a graphical model
- “Supervised Learning as special case”...

Principle Component Analysis (PCA)

- Assume we have data $D = \{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$.

  Intuitively: "We belief that there is an **underlying lower-dimensional space** explaining this data".

- How can we formalize this?

PCA: minimizing projection error

- For each $x_i \in \mathbb{R}^d$ we postulate a lower-dimensional latent variable $z_i \in \mathbb{R}^p$

$$x_i \approx V_p z_i + \mu$$

- Optimality:

  Find V_p, μ and values z_i that minimize $\sum_{i=1}^{n} \|x_i - (V_p z_i + \mu)\|^2$

Optimal V_p

$$\hat{\mu}, \hat{z}_{1:n} = \underset{\mu, z_{1:n}}{\operatorname{argmin}} \sum_{i=1}^{n} \|x_i - V_p z_i - \mu\|^2$$

$$\Rightarrow \hat{\mu} = \langle x_i \rangle = \tfrac{1}{n} \textstyle\sum_{i=1}^{n} x_i \ , \quad \hat{z}_i = V_p^\top (x_i - \mu)$$

Optimal V_p

$$\hat{\mu}, \hat{z}_{1:n} = \underset{\mu, z_{1:n}}{\operatorname{argmin}} \sum_{i=1}^{n} \|x_i - V_p z_i - \mu\|^2$$

$$\Rightarrow \hat{\mu} = \langle x_i \rangle = \tfrac{1}{n} \textstyle\sum_{i=1}^{n} x_i \ , \quad \hat{z}_i = V_p^\top (x_i - \mu)$$

- Center the data $\tilde{x}_i = x_i - \hat{\mu}$. Then

$$\hat{V}_p = \underset{V_p}{\operatorname{argmin}} \sum_{i=1}^{n} \|\tilde{x}_i - V_p V_p^\top \tilde{x}_i\|^2$$

Optimal V_p

$$\hat{\mu}, \hat{z}_{1:n} = \operatorname*{argmin}_{\mu, z_{1:n}} \sum_{i=1}^{n} \|x_i - V_p z_i - \mu\|^2$$

$$\Rightarrow \hat{\mu} = \langle x_i \rangle = \tfrac{1}{n} \textstyle\sum_{i=1}^{n} x_i \ , \quad \hat{z}_i = V_p^\top (x_i - \mu)$$

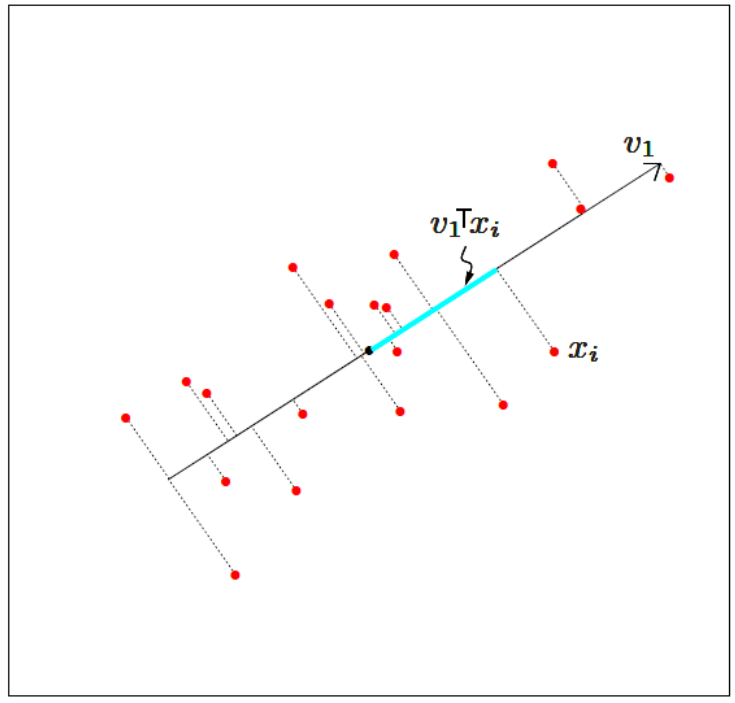
- Center the data $\tilde{x}_i = x_i - \hat{\mu}$. Then

$$\hat{V}_p = \operatorname*{argmin}_{V_p} \sum_{i=1}^{n} \|\tilde{x}_i - V_p V_p^\top \tilde{x}_i\|^2$$

- Solution via Singular Value Decomposition
  - Let $X \in \mathbb{R}^{n \times d}$ be the centered data matrix containing all $\tilde{x}_i$
  - We compute a sorted Singular Value Decomposition $X^\top X = V D V^\top$
    D is diagonal with sorted singular values $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$
    $V = (v_1 \ v_2 \ \cdots \ v_d)$ contains largest eigenvectors v_i as columns

$$V_p := V_{1:d,1:p} = (v_1 \ v_2 \ \cdots \ v_p)$$

Principle Component Analysis (PCA)



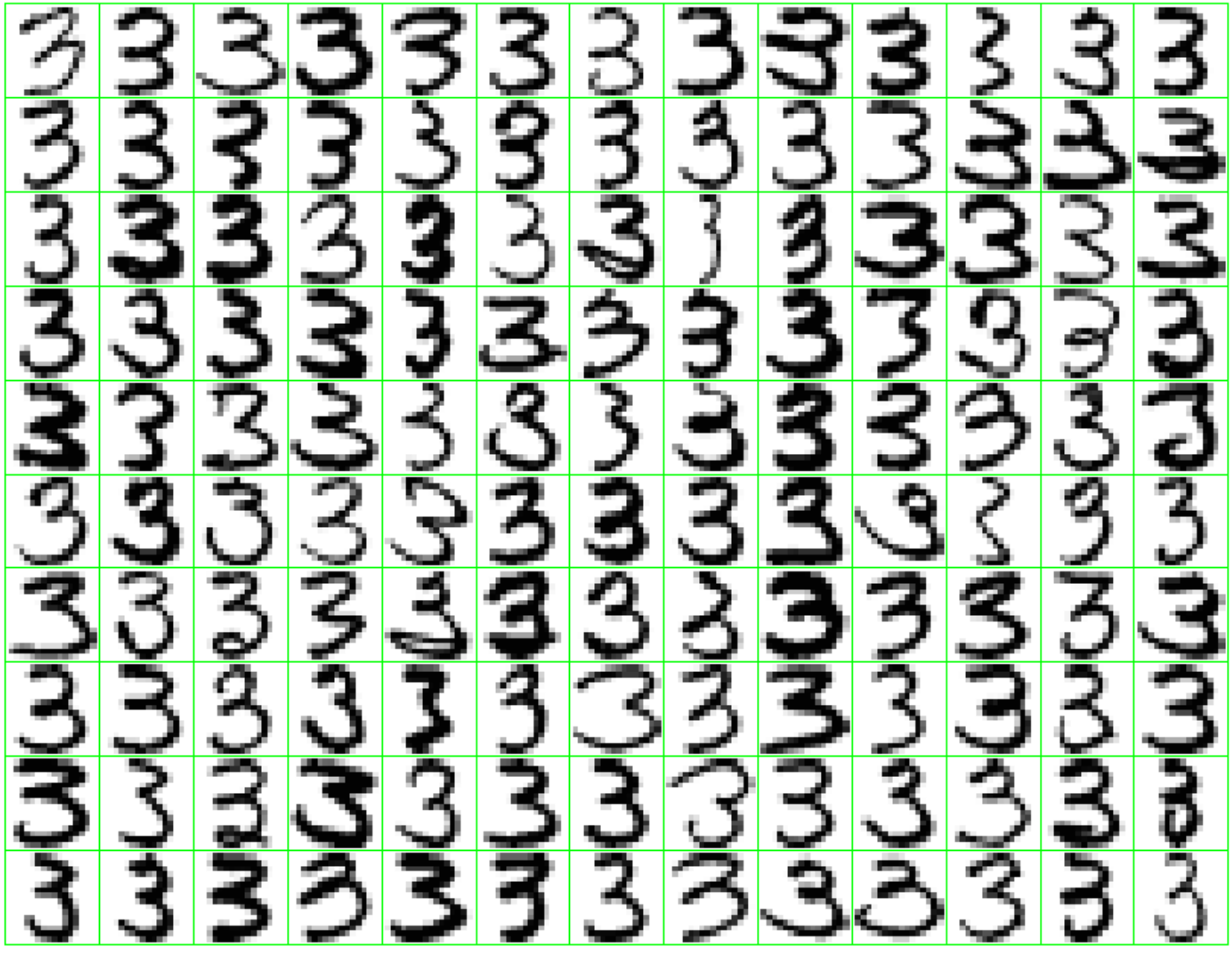
$V_p^\top$ is the matrix that projects to the largest variance directions of $X^\top X$

$$z_i = V_p^\top (x_i - \mu) \ , \quad Z = XV_p$$

- In non-centered case: Compute SVD of variance

$$A = \mathsf{Var}\{x\} = \left\langle xx^\top \right\rangle - \mu\mu^\top = \frac{1}{n} X^\top X - \mu\mu^\top$$

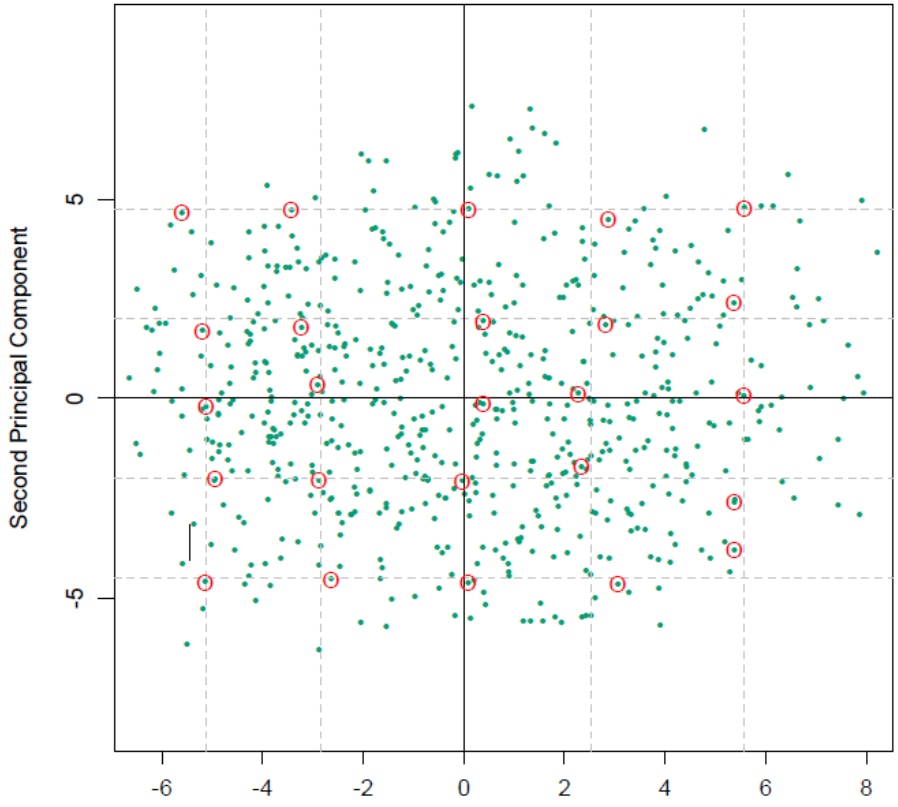
Example: Digits

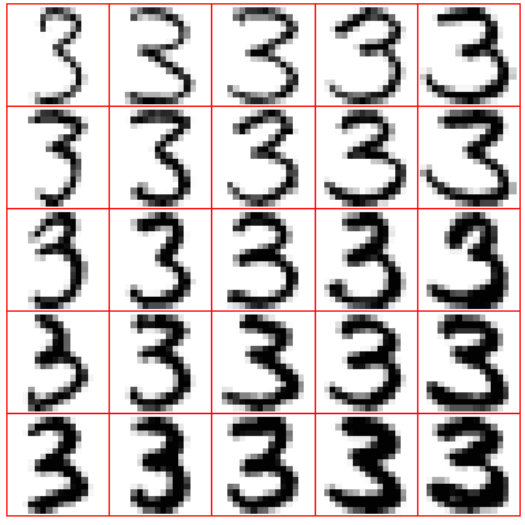


Example: Digits

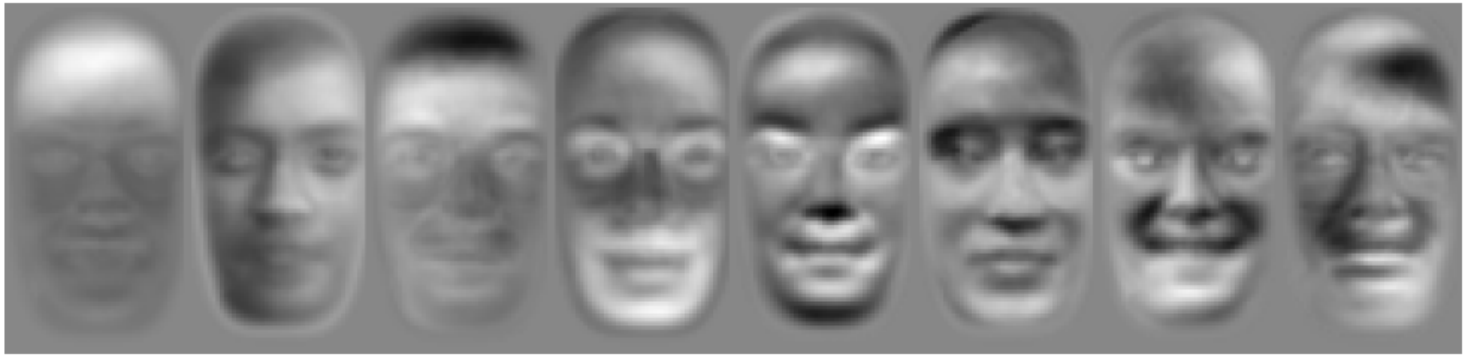
- The "basis vectors" in V_p are also **eigenvectors**
  Every data point can be expressed in these eigenvectors

$$x \approx \mu + V_p z$$
$$= \mu + z_1 v_1 + z_2 v_2 + \dots$$
$$= \text{  } + z_1 \cdot \text{  } + z_2 \cdot \text{  } + \cdots$$





Example: Eigenfaces



(Viola & Jones)

“Feature PCA” & Kernel PCA

- The *feature* trick: $X = \begin{pmatrix} \phi(x_1)^\top \\ \vdots \\ \phi(x_n)^\top \end{pmatrix} \in \mathbb{R}^{n \times k}$

- The *kernel* trick: rewrite all necessary equations such that they only involve scalar products $\phi(x)^\top \phi(x') = k(x, x')$:

We want to compute eigenvectors of $X^\top X = \sum_i \phi(x_i)\phi(x_i)^\top$. We can rewrite this as

$$X^\top X v_j = \lambda v_j$$

$$\underbrace{XX^\top}_{K} \underbrace{X v_j}_{K\alpha_j} = \lambda \underbrace{X v_j}_{K\alpha_j} , \quad v_j = \sum_i \alpha_{ji} \phi(x_i)$$

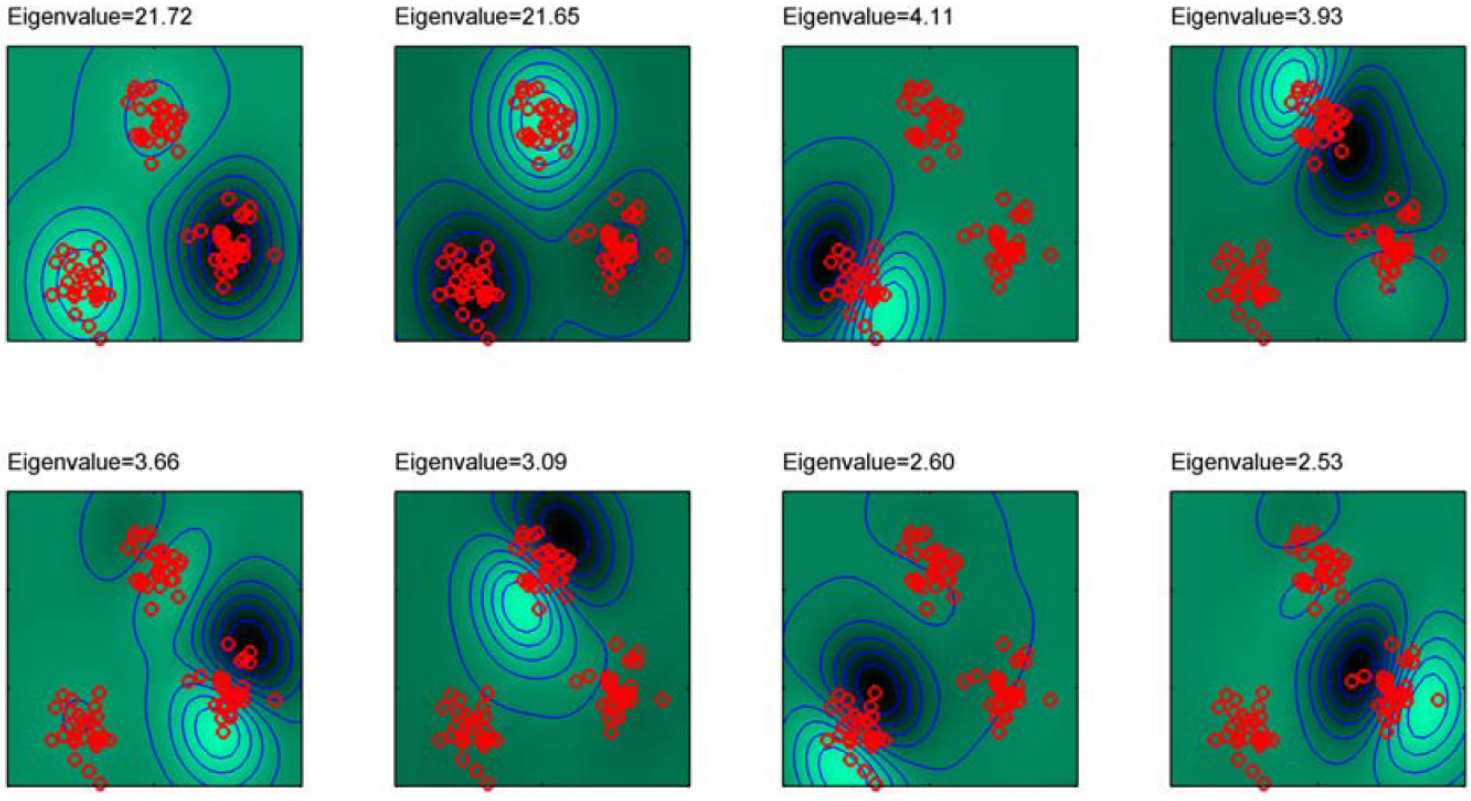
$$K\alpha_j = \lambda \alpha_j$$

Where $K = XX^\top$ with entries $K_{ij} = \phi(x_i)^\top \phi(x_j)$.
$\rightarrow$ We compute SVD of the kernel matrix $K \rightarrow$ gives eigenvectors $\alpha_j \in \mathbb{R}^n$.
Projection: $x \mapsto z = V_p^\top \phi(x) = \sum_i \alpha_{1:p,i} \phi(x_i)^\top \phi(x) = A\kappa(x)$
(with matrix $A \in \mathbb{R}^{p \times n}$, $A_{ji} = \alpha_{ji}$ and vector $\kappa(x) \in \mathbb{R}^n$, $\kappa_i(x) = k(x_i, x)$)
Since we cannot *center the features* $\phi(x)$ we actually need “the double centered kernel matrix” $\widetilde{K} = (\mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}^\top) K (\mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}^\top)$, where $K_{ij} = \phi(x_i)^\top \phi(x_j)$ is uncentered.

Kernel PCA

red points: data

green shading: eigenvector α_j represented as functions $\sum_i \alpha_{ji} k(x_j, x)$



Kernel PCA “coordinates” allow us to discriminate clusters!

Kernel PCA

- Kernel PCA uncovers quite surprising structure:

  While PCA "merely" picks high-variance dimensions
  Kernel PCA picks high variance *features*—where features correspond to basis functions (RKHS elements) over x

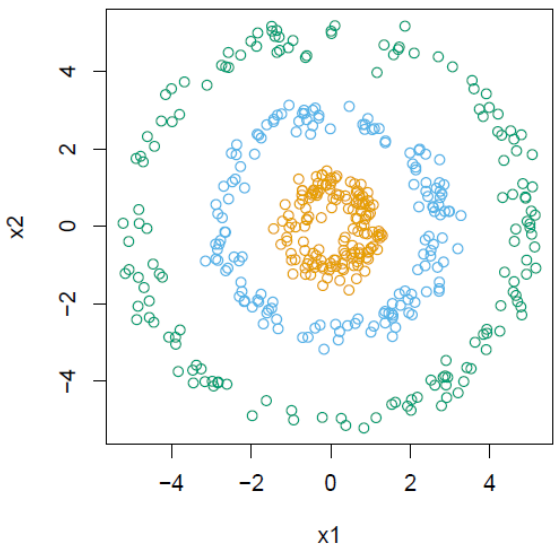
- Kernel PCA may map data x_i to latent coordinates z_i where *clustering* is much easier

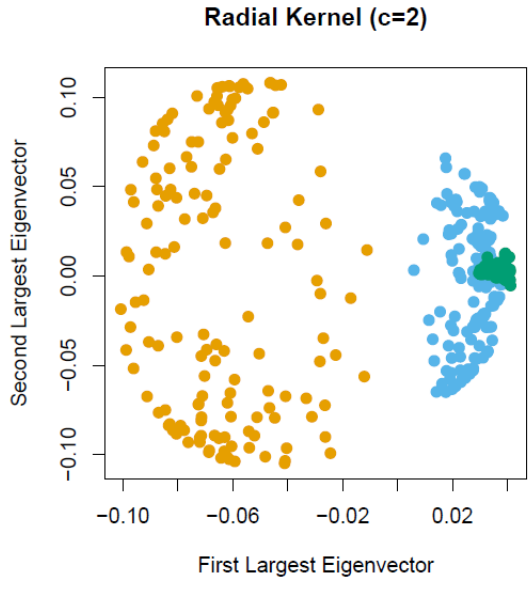
- All of the following can be represented as kernel PCA:
  – Local Linear Embedding
  – Metric Multidimensional Scaling
  – Laplacian Eigenmaps (Spectral Clustering)

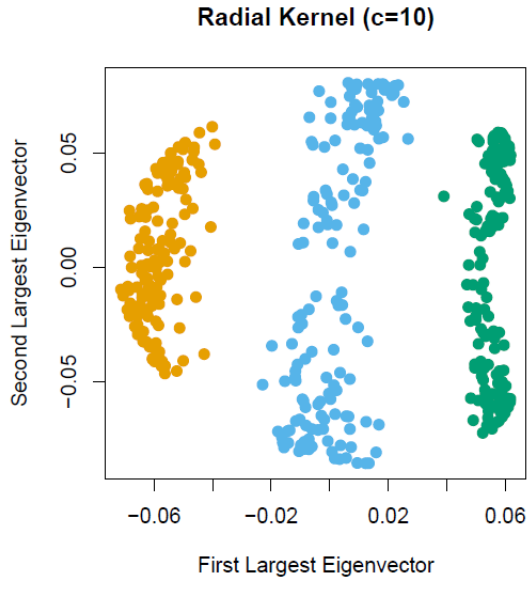
  see "Dimensionality Reduction: A Short Tutorial" by Ali Ghodsi

Kernel PCA clustering

- Using a kernel function $k(x, x') = e^{-\|x-x'\|^2/c}$:







- Gaussian mixtures or k-means will easily cluster this

Spectral Clustering

Spectral Clustering is very similar to kernel PCA:

- Instead of the kernel matrix K with entries $k_{ij} = k(x_i, x_j)$ we construct a weighted *adjacency matrix*, e.g.,

$$w_{ij} = \begin{cases} 0 & \text{if } x_i \text{ are not a } k\text{NN of } x_j \\ e^{-\|x_i - x_j\|^2/c} & \text{otherwise} \end{cases}$$

  w_{ij} is the weight of the *edge* between data point x_i and x_j.

- Instead of computing *maximal* eigenvectors of $\widetilde{K}$, compute *minimal* eigenvectors of

$$L = \mathbf{I} - \widetilde{W}\ , \quad \widetilde{W} = \mathrm{diag}(\textstyle\sum_j w_{ij})^{-1} W$$

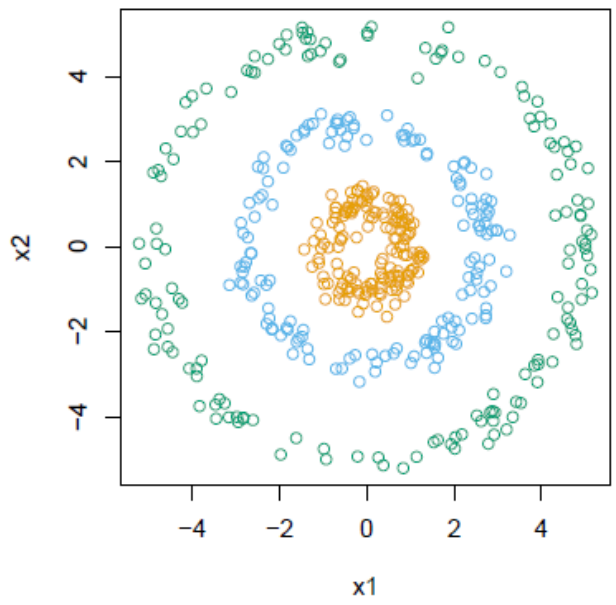
  ($\sum_j w_{ij}$ is called *degree of node* i, $\widetilde{W}$ is the normalized weighted adjacency matrix)

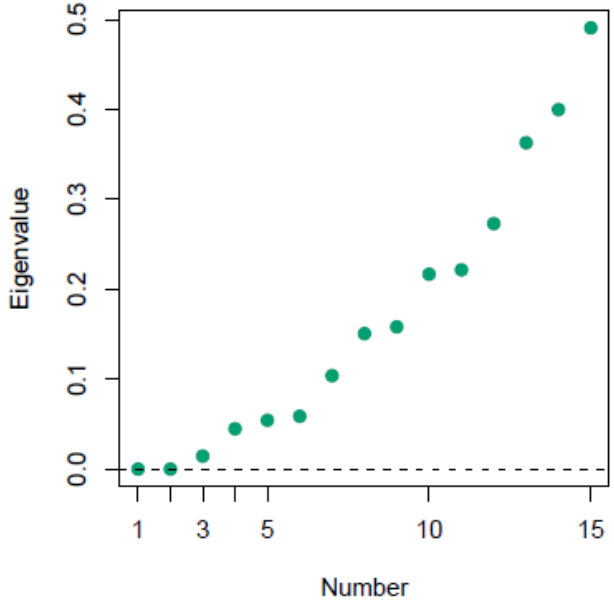
- Given $L = UDV^\top$, we pick the p smallest eigenvectors $V_p = V_{1:n,1:p}$

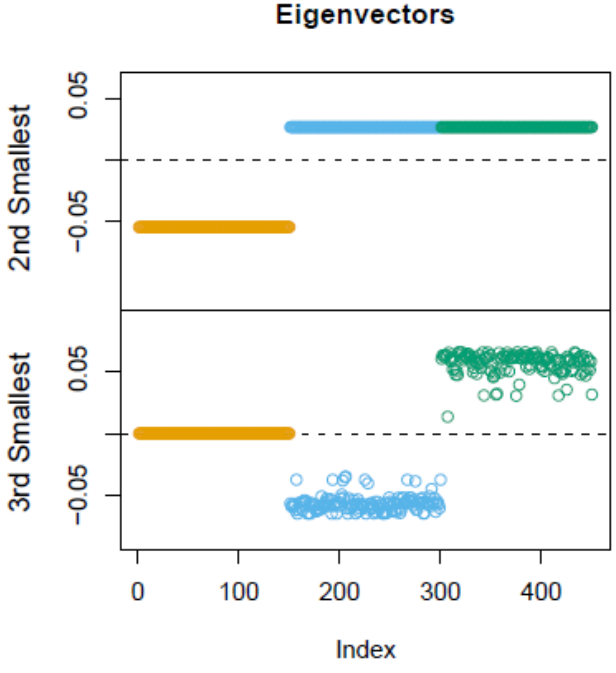
- The latent coordinates for x_i are $z_i = V_{i,1:p}$

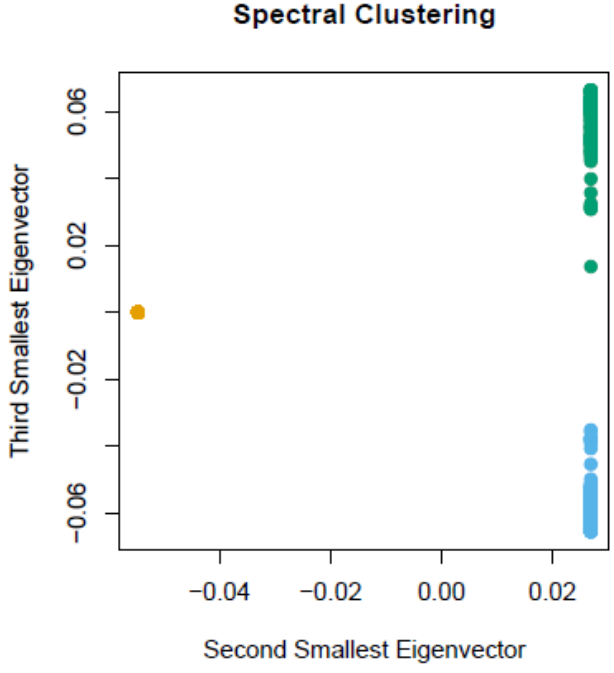
- Spectral Clustering provides a method to compute latent low-dimensional coordinates $z_i = V_{i,1:p}$ for each high-dimensional $x_i \in \mathbb{R}^d$ input.

- This is then followed by a standard clustering, e.g., Gaussian Mixture or k-means


x2
x1
4
2
0
−2
−4
−4
−2
0
2
4


Eigenvalue
Number
0.0
0.1
0.2
0.3
0.4
0.5
1
3
5
10
15


Eigenvectors
2nd Smallest
3rd Smallest
0.05
−0.05
0.05
−0.05
0
100
200
300
400
Index


Spectral Clustering
Third Smallest Eigenvector
Second Smallest Eigenvector
0.06
0.02
−0.02
−0.06
−0.04
−0.02
0.00
0.02

- Spectral Clustering is similar to kernel PCA:
  - The kernel matrix K usually represents similarity
    The weighted adjacency matrix W represents proximity & similarity
  - High Eigenvectors of K are similar to low EV of $L = \mathbf{I} - W$

- Original interpretation of Spectral Clustering:
  - $L = \mathbf{I} - W$ (weighted graph Laplacian) describes a diffusion process:
    The diffusion rate W_{ij} is high if i and j are close and similar
  - Eigenvectors of L correspond to stationary solutions

Metric Multidimensional Scaling

- Assume we have data $D = \{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$.
  As before we want to indentify latent lower-dimensional representations $z_i \in \mathbb{R}^p$ for this data.

- A simple idea: Minimize the stress

$$S_C(z_{1:n}) = \sum_{i \neq j} (d_{ij}^2 - \|z_i - z_j\|^2)^2$$

  We want distances in high-dimensional space to be equal to distances in low-dimensional space.

Metric Multidimensional Scaling = (kernel) PCA

- Note the relation:

$$d_{ij}^2 = \|x_i - x_j\|^2 = \|x_i - \bar{x}\|^2 + \|x_j - \bar{x}\|^2 - 2(x_i - \bar{x})^\top (x_j - \bar{x})$$

This translates a distance into a (centered) scalar product

Metric Multidimensional Scaling = (kernel) PCA

- Note the relation:

$$d_{ij}^2 = \|x_i - x_j\|^2 = \|x_i - \bar{x}\|^2 + \|x_j - \bar{x}\|^2 - 2(x_i - \bar{x})^\top (x_j - \bar{x})$$

This translates a distance into a (centered) scalar product

- If may we define

$$\widetilde{K} = (\mathbf{I} - \tfrac{1}{n}\mathbf{1}\mathbf{1}^\top) D (\mathbf{I} - \tfrac{1}{n}\mathbf{1}\mathbf{1}^\top) \;, \quad D_{ij} = -d_{ij}^2/2$$

then $\widetilde{K_{ij}} = (x_i - \bar{x})^\top (x_j - \bar{x})$ is the normal covariance matrix and MDS is equivalent to kernel PCA

Non-metric Multidimensional Scaling

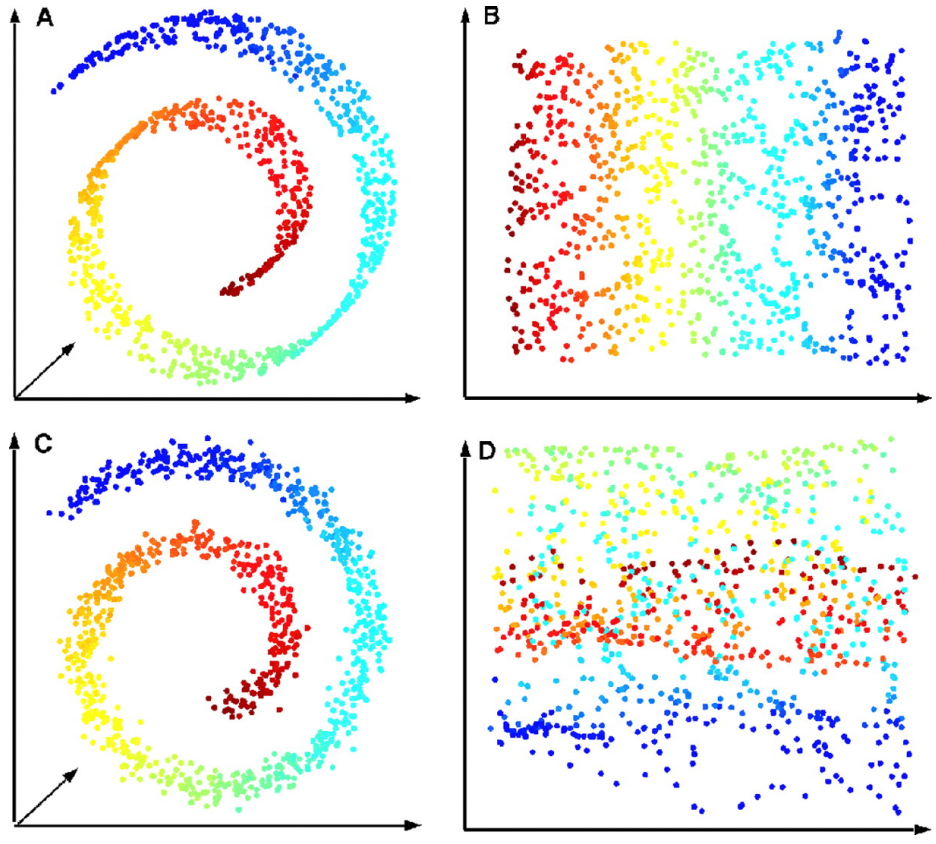
- We can do this for any data (also non-vectorial or not $\in \mathbb{R}^d$) as long as we have a data set of comparative dissimilarities d_{ij}

$$S(z_{1:n}) = \sum_{i \neq j} (d_{ij}^2 - |z_i - z_j|^2)^2$$

- Minimize $S(z_{1:n})$ w.r.t. $z_{1:n}$ *without any further constraints*!

Example for Non-Metric MDS: ISOMAP

- Construct kNN graph and label edges with Euclidean distance
  - Between any two x_i and x_j, compute "geodescic" distance d_{ij} (shortest path along the graph)
  - Then apply MDS



by Tenenbaum et al.

The zoo of dimensionality reduction methods

- PCA family:
  – kernel PCA, non-neg. Matrix Factorization, Factor Analysis

- All of the following can be represented as kernel PCA:
  – Local Linear Embedding
  – Metric Multidimensional Scaling
  – Laplacian Eigenmaps (Spectral Clustering)

They all use different notions of distance/correlation as input to kernel PCA

see “Dimensionality Reduction: A Short Tutorial” by Ali Ghodsi

PCA variants*

PCA variant: Non-negative Matrix Factorization*

- Assume we have data $D = \{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$.
  As for PCA (where we had $x_i \approx V_p z_i + \mu$) we search for a lower-dimensional space with linear relation to x_i

- In NMF we require everything is **non-negative**: the data x_i, the projection W, and latent variables z_i
  Find $W \in \mathbb{R}^{p \times d}$ (the tansposed projection) and $Z \in \mathbb{R}^{n \times p}$ (the latent variables z_i) such that

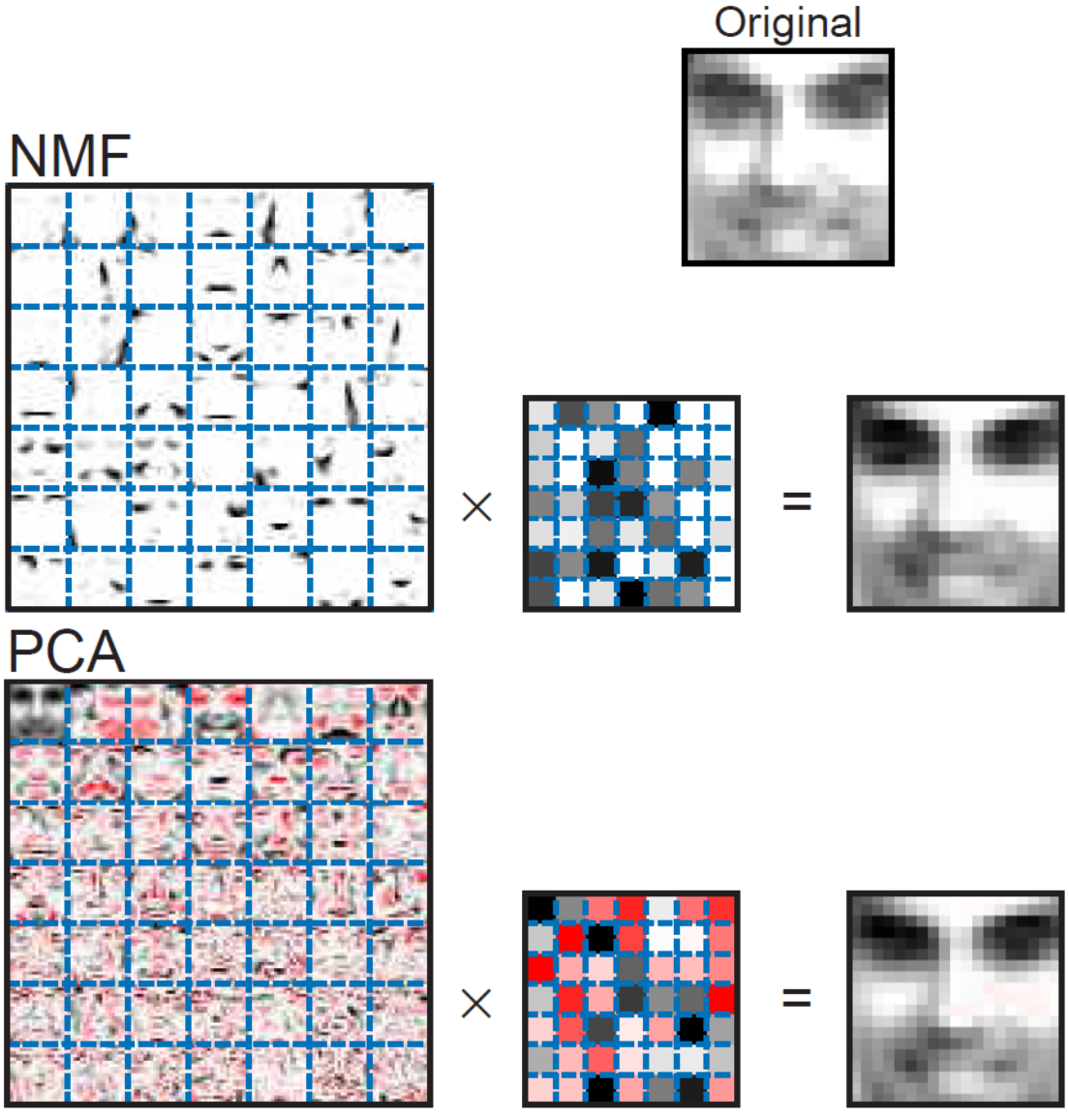
$$X \approx ZW$$

- Iterative solution: (E-step and M-step like...)

$$z_{ik} \leftarrow z_{ik} \frac{\sum_{j=1}^d w_{kj} x_{ij} / (ZW)_{ij}}{\sum_{j=1}^d w_{kj}}$$

$$w_{kj} \leftarrow w_{kj} \frac{\sum_{i=1}^N z_{ik} x_{ij} / (ZW)_{ij}}{\sum_{i=1}^N z_{ik}}$$

PCA variant: Non-negative Matrix Factorization*



(from Hastie 14.6)

PCA variant: Factor Analysis*

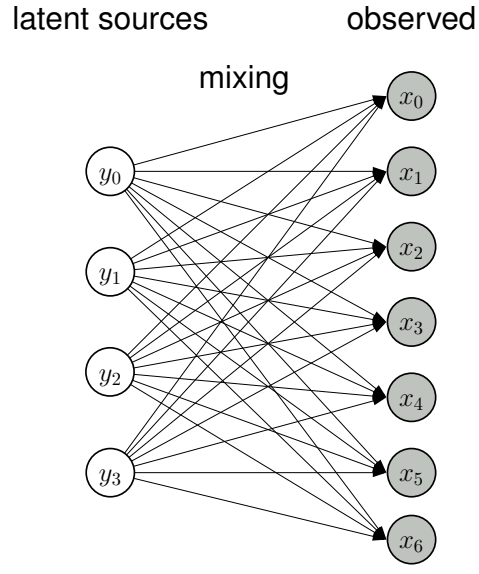
Another variant of PCA: (Bishop 12.64)

Allows for different noise in each dimension

$P(x_i \mid z_i) = \mathcal{N}(x_i \mid V_p z_i + \mu, \Sigma)$ (with Σ diagonal)

Independent Component Analysis*

- Assume we have data $D = \{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$.
  PCA: $P(x_i \mid z_i) = \mathcal{N}(x_i \mid W z_i + \mu, \mathbf{I}) \;, \quad P(z_i) = \mathcal{N}(z_i \mid 0, \mathbf{I})$
  Factor Analysis: $P(x_i \mid z_i) = \mathcal{N}(x_i \mid W z_i + \mu, \Sigma) \;, \quad P(z_i) = \mathcal{N}(z_i \mid 0, \mathbf{I})$
  ICA: $P(x_i \mid z_i) = \mathcal{N}(x_i \mid W z_i + \mu, \epsilon\mathbf{I}) \;, \quad P(z_i) = \prod_{j=1}^d P(z_{ij})$



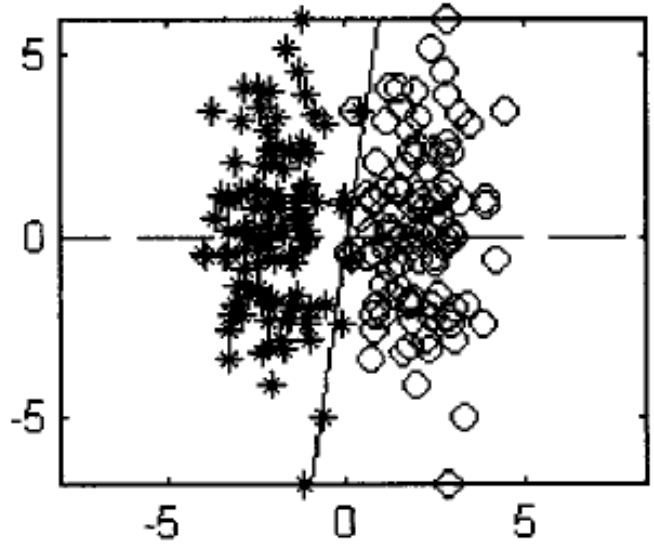
- In ICA
  1) We have (usually) as many latent variables as observed $\dim(x_i) = \dim(z_i)$
  2) We require all latent variables to be **independent**
  3) We allow for latent variables to be **non-Gaussian**

  Note: without point (3) ICA would be sense-less!

Partial least squares (PLS)*

- Is it really a good idea to just pick the p-higest variance components??

  Why should that be a good idea?



PLS*

- Idea: The first dimension to pick should be the one **most correlated with the OUTPUT**, not with itself!

Input: data $X \in \mathbb{R}^{n \times d}$, $y \in \mathbb{R}^n$
Output: predictions $\hat{y} \in \mathbb{R}^n$

1: initialize the *predicted output*: $\hat{y} = \langle y \rangle \, 1_n$
2: initialize the *remaining input dimensions*: $\hat{X} = X$
3: **for** $i = 1, .., p$ **do**
4: $\quad$ i-th input 'basis vector': $z_i = \hat{X}\hat{X}^\top y$
5: $\quad$ update prediction: $\hat{y} \leftarrow \hat{y} + Z_i y \quad$ where $Z_i = \frac{z_i z_i^\top}{z_i^\top z_i}$
6: $\quad$ remove "used" input dimensions: $\hat{X} \leftarrow \hat{X}(\mathbf{I} - Z_i)$
7: **end for**

(Hastie, page 81)

Line 4 identifies a new input "coordinate" via maximal correlation between the remaning input dimensions and y.
Line 5 updates the prediction to include the project of y onto z_i
Line 6 removes the projection of input data $\hat{X}$ along z_i. All z_i will be orthogonal.

PLS for classification*

- Not obvious.
- We'll try to invent one in the exercises :-)

Clustering

- Clustering often involves two steps:
- First map the data to some embedding that emphasizes clusters
  - (Feature) PCA
  - Spectral Clustering
  - Kernel PCA
  - ISOMAP
- Then explicitly analyze clusters
  - k-means clustering
  - Gaussian Mixture Model
  - Agglomerative Clustering

k-means Clustering

- Given data $D = \{x_i\}_{i=1}^n$, find K centers μ_k, and a data assignment $c : i \mapsto k$ to minimize

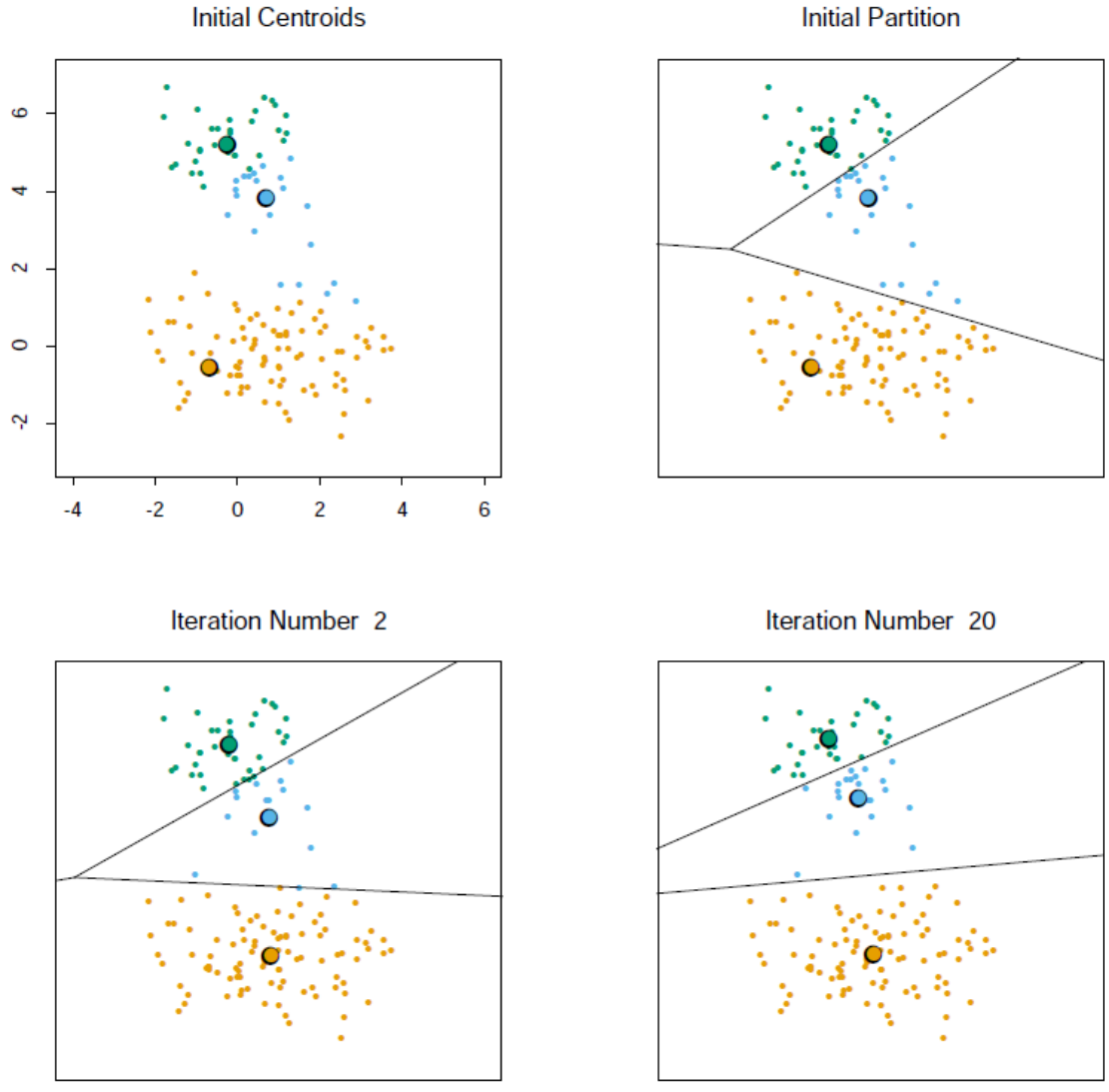
$$\min_{c,\mu} \sum_i (x_i - \mu_{c(i)})^2$$

- k-means clustering:
  - Pick K data points randomly to initialize the centers μ_k
  - Iterate adapting the assignments $c(i)$ and the centers μ_k:

$$\forall_i : \; c(i) \leftarrow \operatorname*{argmin}_{c(i)} \sum_i (x_i - \mu_{c(i)})^2 = \operatorname*{argmin}_k (x_i - \mu_k)^2$$

$$\forall_k : \; \mu_k \leftarrow \operatorname*{argmin}_{\mu_k} \sum_i (x_i - \mu_{c(i)})^2 = \frac{1}{|c^{-1}(k)|} \sum_{i \in c^{-1}(k)} x_i$$

k-means Clustering

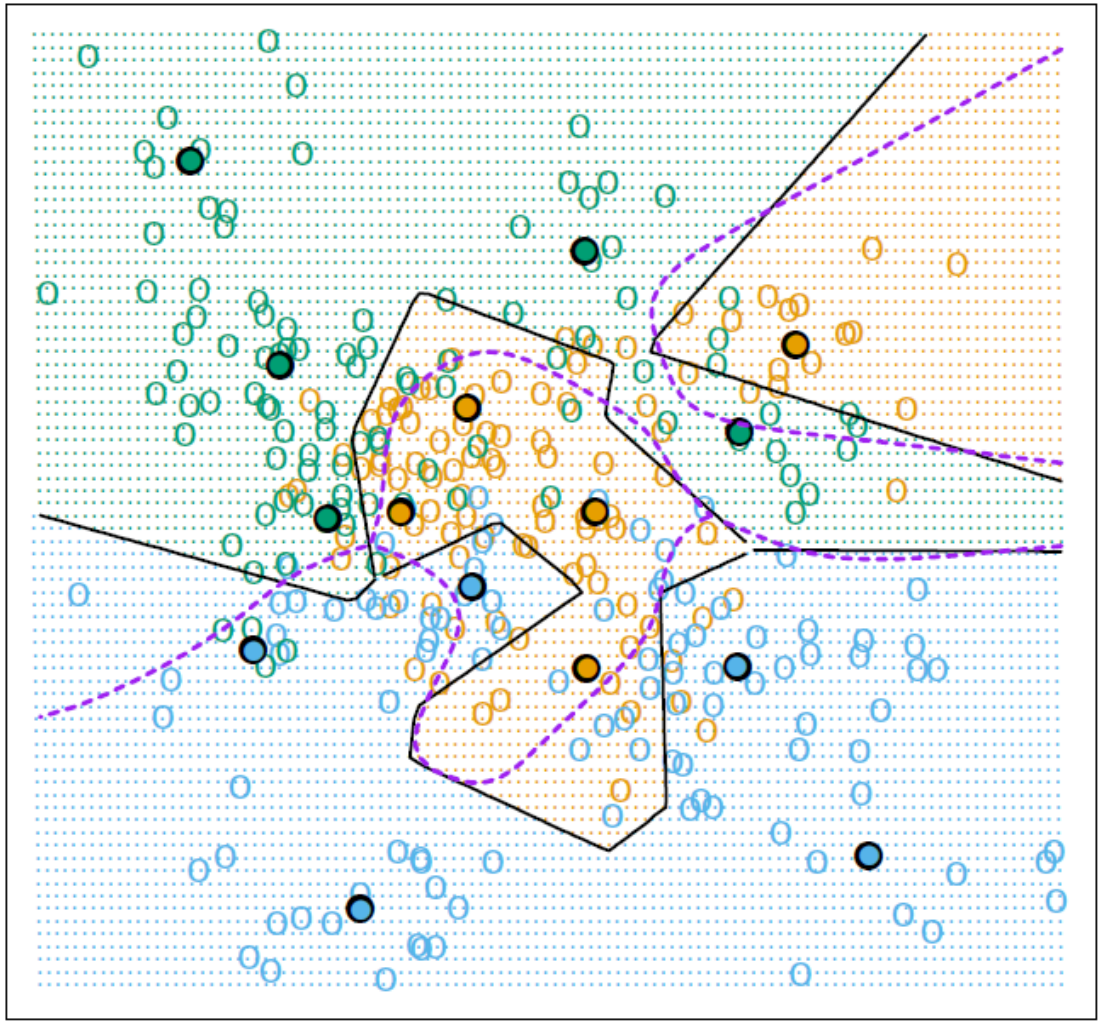


from Hastie

k-means Clustering

- Converges to local minimum $\rightarrow$ **many restarts**
- Choosing k? Plot $L(k) = \min_{c,\mu} \sum_i (x_i - \mu_{c(i)})^2$ for different k – choose a tradeoff between model complexity (large k) and data fit (small loss $L(k)$)

k-means Clustering for Classification



from Hastie

Gaussian Mixture Model for Clustering

- GMMs can/should be introduced as *generative* probabilistic model of the data:
  - K different Gaussians with parmameters μ_k, Σ_k
  - Assignment RANDOM VARIABLE $c_i \in \{1, .., K\}$ with $P(c_i = k) = \pi_k$
  - The observed data point x_i with $P(x_i \mid c_i = k; \mu_k, \Sigma_k) = \mathcal{N}(x_i \mid \mu_k, \Sigma_k)$
- EM-Algorithm described as a kind of soft-assignment version of k-means
  - Initialize the centers $\mu_{1:K}$ randomly from the data; all covariances $\Sigma_{1:K}$ to unit and all π_k uniformly.
  - **E-step:** (probabilistic/soft assignment) Compute

$$q(c_i = k) = P(c_i = k \mid x_i, \mu_{1:K}, \Sigma_{1:K}) \propto \mathcal{N}(x_i \mid \mu_k, \Sigma_k)\ \pi_k$$

  - **M-step:** Update parameters (centers AND covariances)

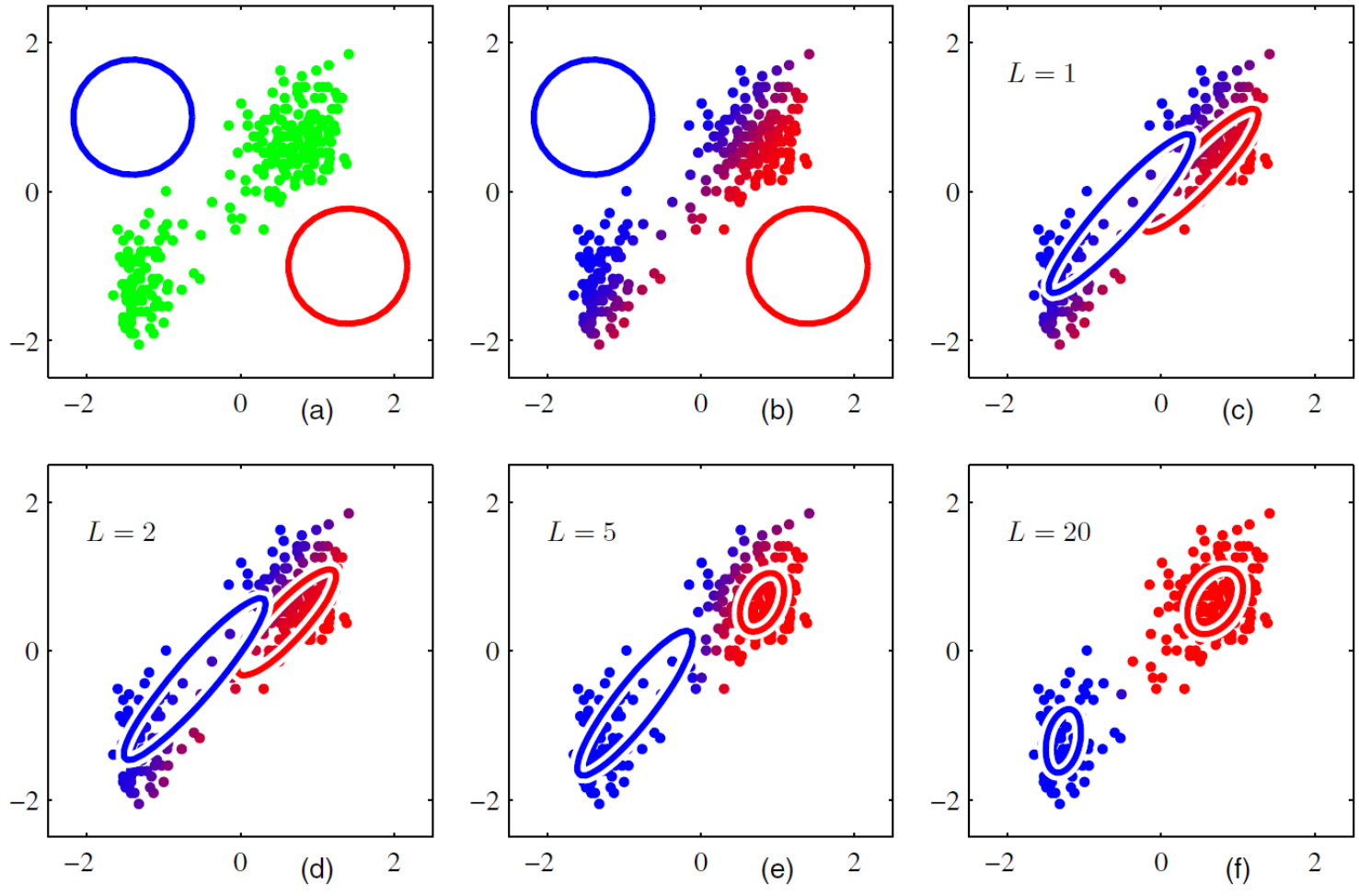
$$\pi_k = \frac{1}{n} \sum_i q(c_i = k)$$

$$\mu_k = \frac{1}{n\pi_k} \sum_i q(c_i = k)\ x_i$$

$$\Sigma_k = \frac{1}{n\pi_k} \sum_i q(c_i = k)\ x_i x_i^\top - \mu_k \mu_k^\top$$

Gaussian Mixture Model

EM iterations for Gaussian Mixture model:

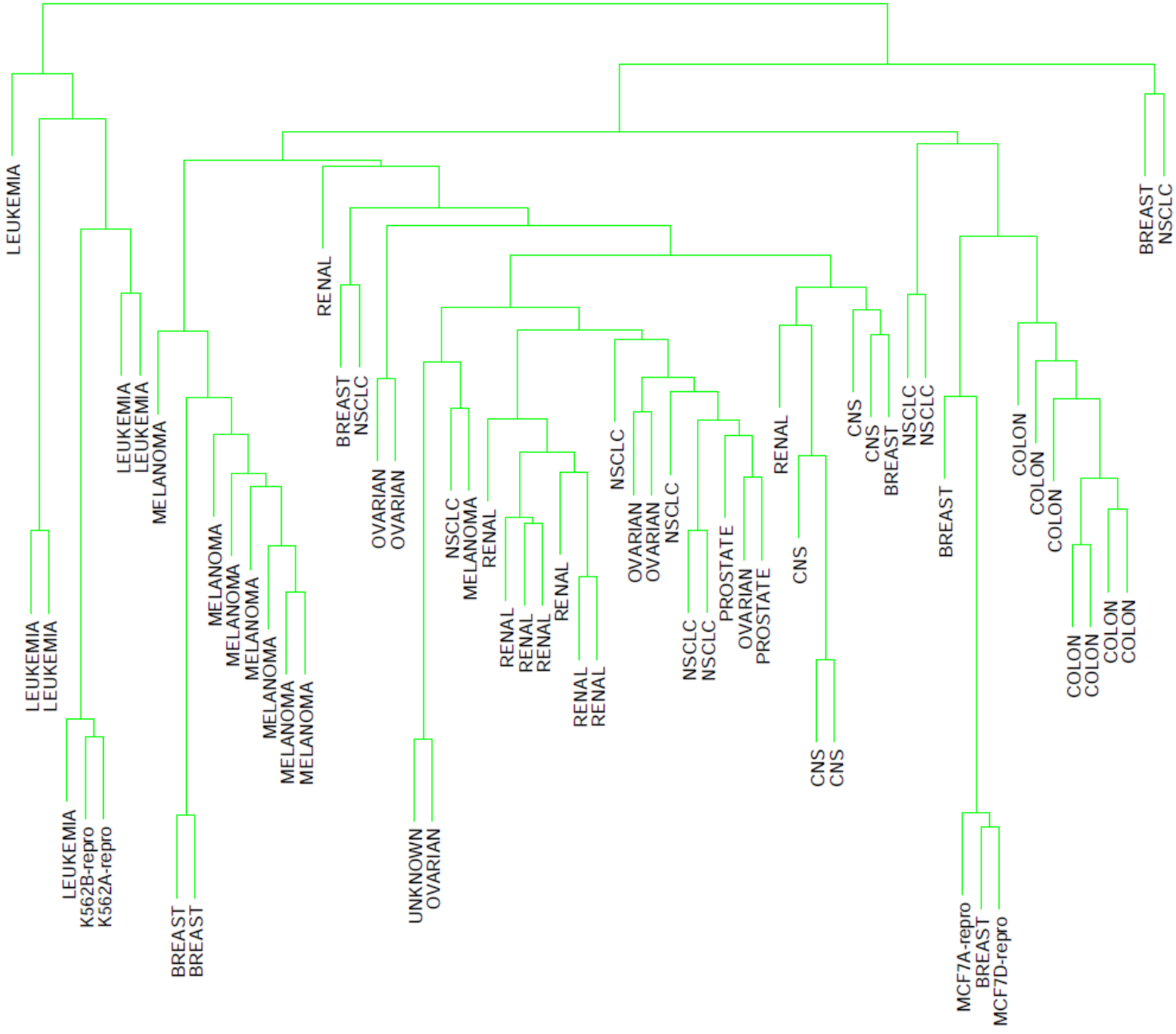


from Bishop

Agglomerative Hierarchical Clustering

- *agglomerative* = bottom-up, *divisive* = top-down
- Merge the two groups with the smallest intergroup dissimilarity
- Dissimilarity of two groups G, H can be measures as
  - Nearest Neighbor (or "single linkage"): $d(G, H) = \min_{i \in G, j \in H} d(x_i, x_j)$
  - Furthest Neighbor (or "complete linkage"): $d(G, H) = \max_{i \in G, j \in H} d(x_i, x_j)$
  - Group Average: $d(G, H) = \frac{1}{|G||H|} \sum_{i \in G} \sum_{j \in H} d(x_i, x_j)$

Agglomerative Hierarchical Clustering



Local learning

Local & lazy learning

- Idea of local (or “lazy”) learning:
  Do not try to build one global model $f(x)$ from the data. Instead, whenever we have a query point x^*, we build a specific local model in the neighborhood of x^*.

Local & lazy learning

- Idea of local (or "lazy") learning:
  Do not try to build one global model $f(x)$ from the data. Instead, whenever we have a query point x^*, we build a specific local model in the neighborhood of x^*.

- Typical approach:
  – Given a query point x^*, find all kNN in the data $D = \{(x_i, y_i)\}_{i=1}^N$
  – Fit a local model f_{x^*} only to these kNNs, perhaps weighted
  – Use the local model f_{x^*} to predict $x^* \mapsto \hat{y}_0$

Local & lazy learning

- Idea of local (or "lazy") learning:
  Do not try to build one global model $f(x)$ from the data. Instead, whenever we have a query point x^*, we build a specific local model in the neighborhood of x^*.

- Typical approach:
  – Given a query point x^*, find all kNN in the data $D = \{(x_i, y_i)\}_{i=1}^N$
  – Fit a local model f_{x^*} only to these kNNs, perhaps weighted
  – Use the local model f_{x^*} to predict $x^* \mapsto \hat{y}_0$

- **Weighted local least squares:**

$$L^{\text{local}}(\beta, x^*) = \sum_{i=1}^n K(x^*, x_i)(y_i - f(x_i))^2 + \lambda\|\beta\|^2$$

  where $K(x^*, x)$ is called **smoothing kernel**. The optimum is:

$$\hat{\beta} = (X^\top W X + \lambda I)^{-1} X^\top W y \ , \quad W = \text{diag}(K(x^*, x_{1:n}))$$

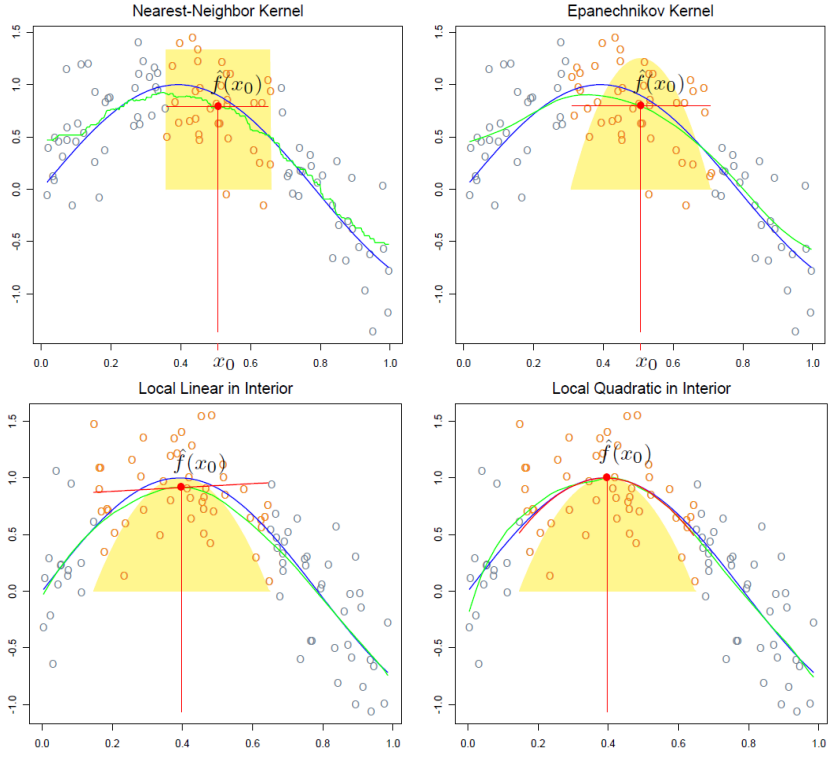
Regression example

kNN smoothing kernel: $K(x^*, x_i) = \begin{cases} 1 & \text{if } x_i \in \text{kNN}(x^*) \\ 0 & \text{otherwise} \end{cases}$

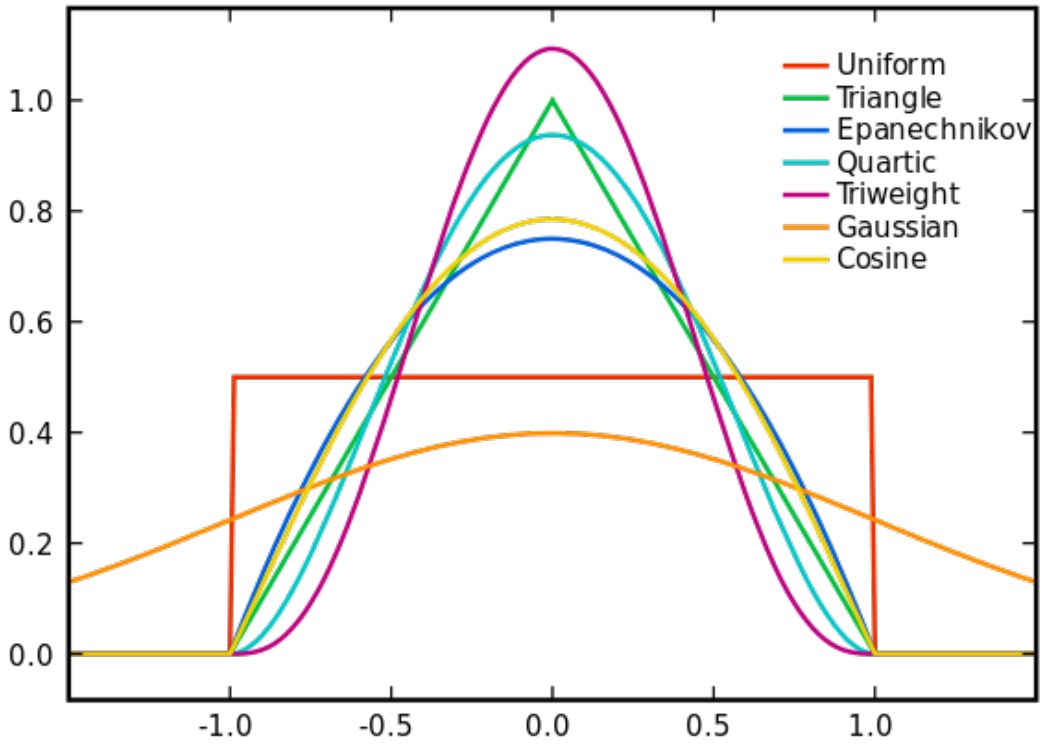
Epanechnikov quadratic smoothing kernel:

$$K_\lambda(x^*, x) = D(|x^* - x|/\lambda)\ , \quad D(s) = \begin{cases} \frac{3}{4}(1 - s^2) & \text{if } s \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

(Hastie, Sec 6.3)



Smoothing Kernels



from Wikipedia

kd-trees

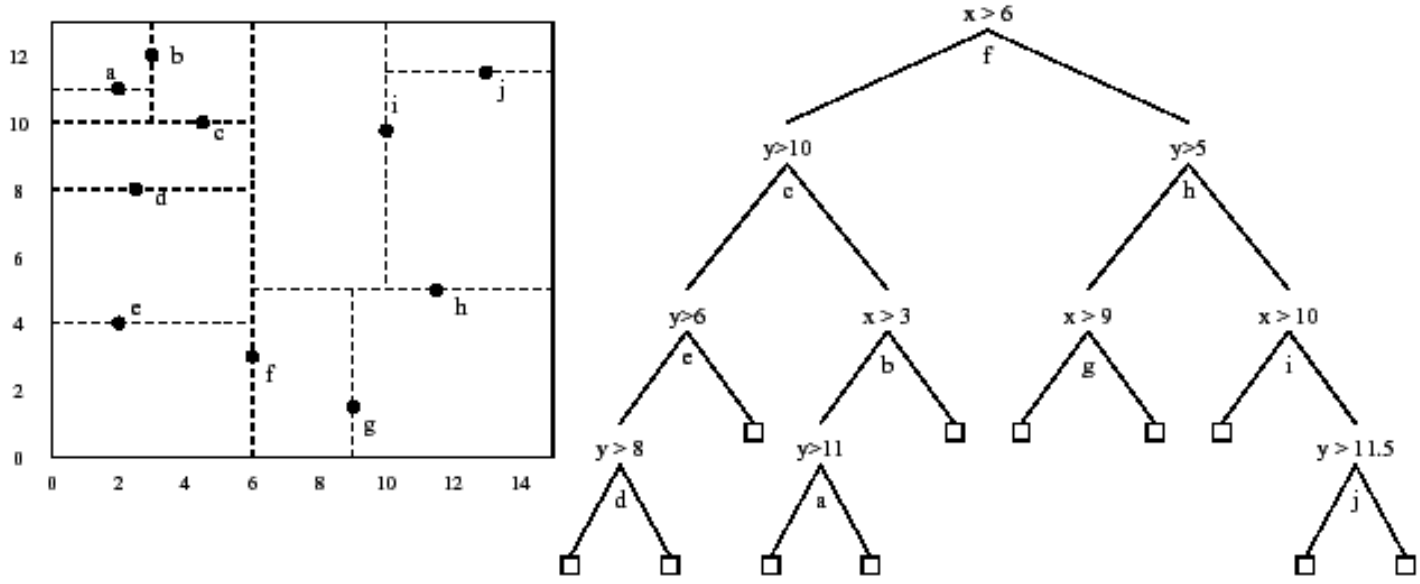
- For local & lazy learning it is essential to efficiently retrieve the kNN

  Problem: Given data X, a query x^*, identify the kNNs of x^* in X.

- Linear time (stepping through all of X) is far too slow.

  A kd-tree pre-structures the data into a binary tree, allowing $O(\log n)$ retrieval of kNNs.

kd-trees



(There are “typos” in this figure... Exercise to find them.)

- Every node plays two roles:
  – it defines a hyperplane that separates the data along *one* coordinate
  – it hosts a data point, which lives exactly on the hyperplane (defines the division)

kd-trees

- Simplest (non-efficient) way to construct a kd-tree:
  – hyperplanes divide alternatingly along 1st, 2nd, ... coordinate
  – choose random point, use it to define hyperplane, divide data, iterate

- Nearest neighbor search:
  – descent to a leave node and take this as initial nearest point
  – ascent and check at each branching the possibility that a nearer point exists on the other side of the hyperplane

- Approximate Nearest Neighbor (libann on Debian..)

Combining weak and randomized learners

Combining learners

- The general idea is:
  – Given data D, let us learn *various models* $f_1, .., f_M$
  – Our prediction is then some combination of these, e.g.

$$f(x) = \sum_{m=1}^{M} \alpha_m f_m(x)$$

- *Various models* could be:

  Model averaging: Fully different types of models (using different (e.g. limited) feature sets; neural net; decision trees; hyperparameters)

  Bootstrap: Models of same type, trained on randomized versions of D

  Boosting: Models of same type, trained on cleverly designed modifications/reweightings of D

- How can we choose the α_m?

Bootstrap & Bagging

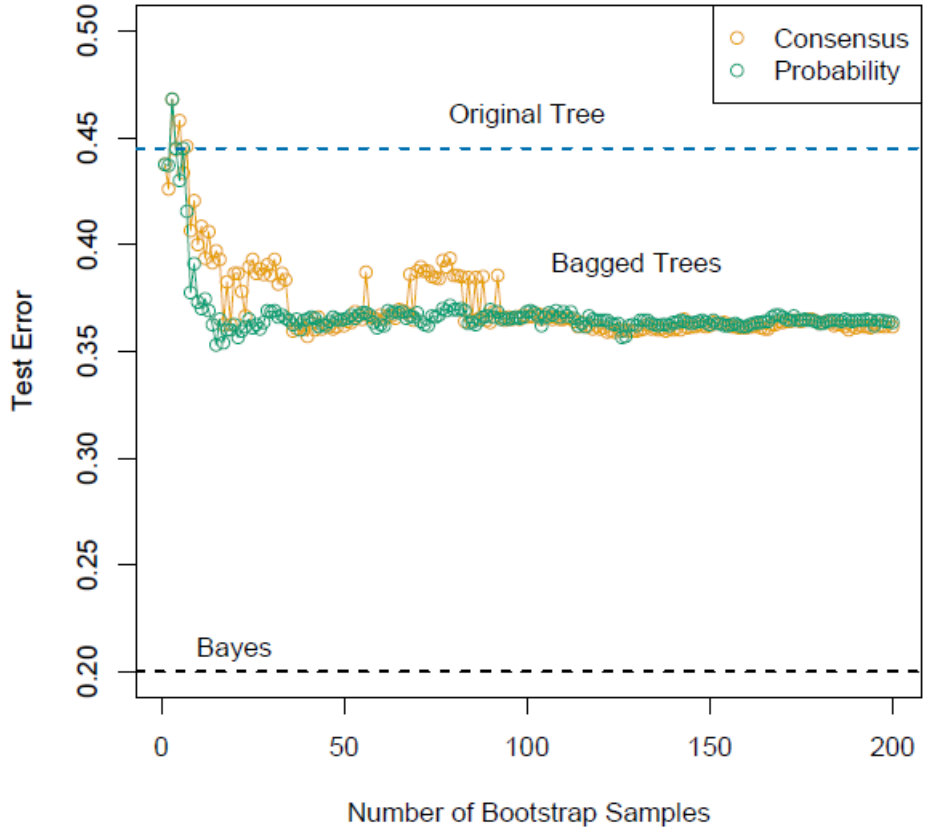
- **Bootstrap:**
  – Data set D of size n
  – Generate M data sets D_m by resampling D *with replacement*
  – Each D_m is also of size n (some samples doubled or missing)

  – Distribution over data sets $\leftrightarrow$ distribution over β (compare slide 02-13)
  – The ensemble $\{f_1, .., f_M\}$ is similar to cross-validation
  – Mean and variance of $\{f_1, .., f_M\}$ can be used for model assessment

- **Bagging:** ("bootstrap aggregation")

$$f(x) = \frac{1}{M} \sum_{m=1}^{M} f_m(x)$$

- Bagging has similar effect to regularization:



(Hastie, Sec 8.7)

Bayesian Model Averaging

- If $f_1, .., f_M$ are very different models
  – Equal weighting would not be clever
  – More confident models (less variance, less parameters, higher likelihood)
  $\rightarrow$ higher weight

- Bayesian Averaging

$$P(y|x) = \sum_{m=1}^{M} P(y|x, f_m, D)\ P(f_m|D)$$

The term $P(f_m|D)$ is the weighting α_m: it is high, when the model is likely under the data ($\leftrightarrow$ the data is likely under the model & the model has “fewer parameters”).

The basis function view: Models are features!

- Compare model averaging $f(x) = \sum_{m=1}^{M} \alpha_m f_m(x)$ with regression:

$$f(x) = \sum_{j=1}^{k} \phi_j(x)\, \beta_j = \phi(x)^\top \beta$$

- We can think of the M models f_m as **features** ϕ_j for linear regression!
  – We know how to find optimal parameters α
  – But beware overfitting!

Boosting

- In Bagging and Model Averaging, the models are trained on the “same data” (unbiased randomized versions of the same data)

- Boosting tries to be cleverer:
  – It adapts the data for each learner
  – It assigns each learner a differently *weighted* version of the data

- With this, boosing can
  – *Combine many “weak” classifiers to produce a powerful “committee”*
  – A weak learner only needs to be somewhat better than random

AdaBoost

(Freund & Schapire, 1997)
(classical Algo; use Gradient Boosting instead in practice)

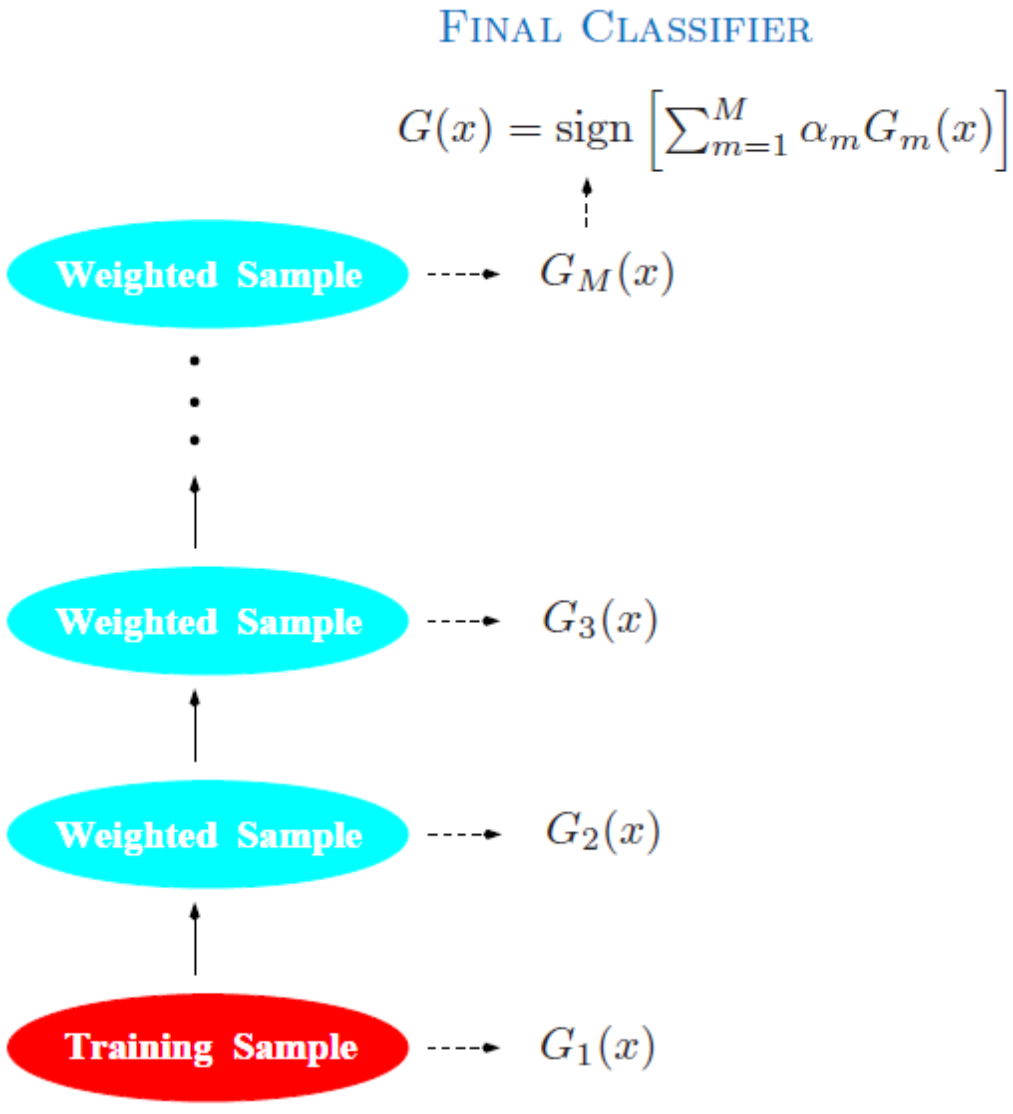
- Binary classification problem with data $D = \{(x_i, y_i)\}_{i=1}^n$, $y_i \in \{-1, +1\}$
- We know how to train discriminative functions $f(x)$; let

$$G(x) = \text{sign}\, f(x) \quad \in \{-1, +1\}$$

- We will train a sequence of classificers $G_1, .., G_M$, each on differently weighted data, to yield a classifier

$$G(x) = \text{sign} \sum_{m=1}^{M} \alpha_m G_m(x)$$

AdaBoost



(Hastie, Sec 10.1)

AdaBoost

Input: data $D = \{(x_i, y_i)\}_{i=1}^n$
Output: family of M classifiers G_m and weights α_m

1: initialize $\forall_i : \; w_i = 1/n$
2: **for** $m = 1, .., M$ **do**
3: Fit classifier G_m to the training data weighted by w_i
4: $\text{err}_m = \frac{\sum_{i=1}^n w_i \; [y_i \neq G_m(x_i)]}{\sum_{i=1}^n w_i}$
5: $\alpha_m = \log[\frac{1-\text{err}_m}{\text{err}_m}]$
6: $\forall_i : \; w_i \leftarrow w_i \; \exp\{\alpha_m \; [y_i \neq G_m(x_i)]\}$
7: **end for**

(Hastie, sec 10.1)

Weights unchanged for correctly classified points
Multiply weights with $\frac{1-\text{err}_m}{\text{err}_m} > 1$ for mis-classified data points

- *Real AdaBoost:* A variant exists that combines probabilistic classifiers $\sigma(f(x)) \in [0, 1]$ instead of discrete $G(x) \in \{-1, +1\}$

The basis function view

- In AdaBoost, each model G_m depends on the data weights w_m
  We could write this as

$$f(x) = \sum_{m=1}^{M} \alpha_m f_m(x, w_m)$$

  The "features" $f_m(x, w_m)$ now have additional parameters w_m
  We'd like to optimize

$$\min_{\alpha, w_1, .., w_M} L(f)$$

  w.r.t. α and all the feature parameters w_m.

- In general this is hard.
  But assuming $\alpha_{\hat{m}}$ and $w_{\hat{m}}$ fixed, optimizing for α_m and w_m is efficient.

- AdaBoost does exactly this, choosing w_m so that the "feature" f_m will best reduce the loss (cf. PLS)

  (Literally, AdaBoost uses exponential loss or neg-log-likelihood; Hastie sec 10.4 & 10.5)

Gradient Boosting

- AdaBoost generates a series of basis functions by using different data weightings w_m depending on so-far classification errors
- We can also generate a series of basis functions f_m by fitting them to the gradient of the so-far loss

Gradient Boosting

- Assume we want to miminize some loss function

$$\min_f L(f) = \sum_{i=1}^{n} L(y_i, f(x_i))$$

We can solve this using gradient descent

$$f^* = f_0 + \alpha_1 \underbrace{\frac{\partial L(f_0)}{\partial f}}_{\approx f_1} + \alpha_2 \underbrace{\frac{\partial L(f_0 + \alpha_1 f_1)}{\partial f}}_{\approx f_2} + \alpha_3 \underbrace{\frac{\partial L(f_0 + \alpha_1 f_1 + \alpha_2 f_2)}{\partial f}}_{\approx f_3} + \cdots$$

– Each f_m aproximates the so-far loss gradient
– We use linear regression to choose α_m (instead of line search)

- More intuitively: $\frac{\partial L(f)}{\partial f}$ "points into the direction of the error/redisual of f". It shows how f could be improved.
Gradient boosting uses the next lerner $f_k \approx \frac{\partial L(f_{\text{so far}})}{\partial f}$ to approximate how f can be improved.
Optimizing α's does the improvement.

Gradient Boosting

Input: function class $\mathcal{F}$ (e.g., of discriminative functions), data $D = \{(x_i, y_i)\}_{i=1}^n$, an arbitrary loss function $L(y, \hat{y})$

Output: function $\hat{f}$ to minimize $\sum_{i=1}^n L(y_i, f(x_i))$

1: Initialize a constant $\hat{f} = f_0 = \text{argmin}_{f \in \mathcal{F}} \sum_{i=1}^n L(y_i, f(x_i))$

2: **for** $m = 1 : M$ **do**

3: For each data point $i = 1 : n$ compute $r_{im} = -\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\Big|_{f=\hat{f}}$

4: Fit a **regression** $f_m \in \mathcal{F}$ to the targets r_{im}, minimizing squared error

5: Find optimal coefficients (e.g., via feature logistic regression)

$$\alpha = \text{argmin}_\alpha \sum_i L(y_i, \sum_{j=0}^m \alpha_m f_m(x_i))$$

(often: fix $\alpha_{0:m\text{-}1}$ and only optimize over α_m)

6: Update $\hat{f} = \sum_{j=0}^m \alpha_m f_m$

7: **end for**

- If $\mathcal{F}$ is the set of regression/decision trees, then step 5 usually re-optimizes the terminal constants of all leave nodes of the regression tree f_m. (Step 4 only determines the terminal regions.)

Gradient boosting is the preferred method

- Hastie's book quite "likes" gradient boosting
  - Can be applied to any loss function
  - No matter if regression or classification
  - Very good performance
  - Simpler, more general, better than AdaBoost

Decision Trees

- Decision trees are particularly used in Bagging and Boosting contexts

- Decision trees are “linear in features”, but the features are the terminal regions of a tree, which are constructed depending on the data

- We’ll learn about
  – Boosted decision trees & stumps
  – Random Forests

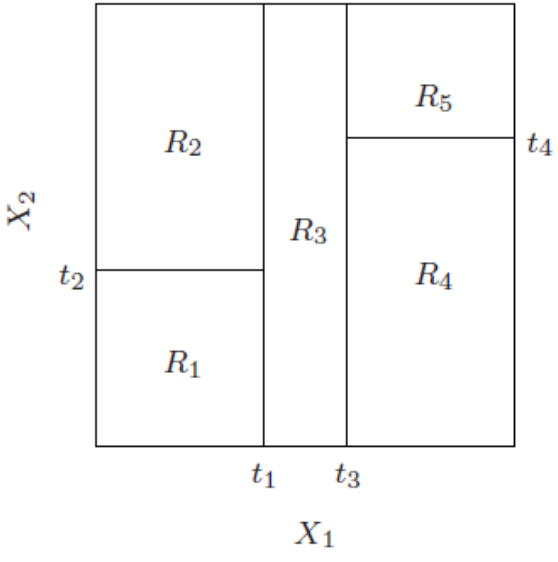
Decision Trees

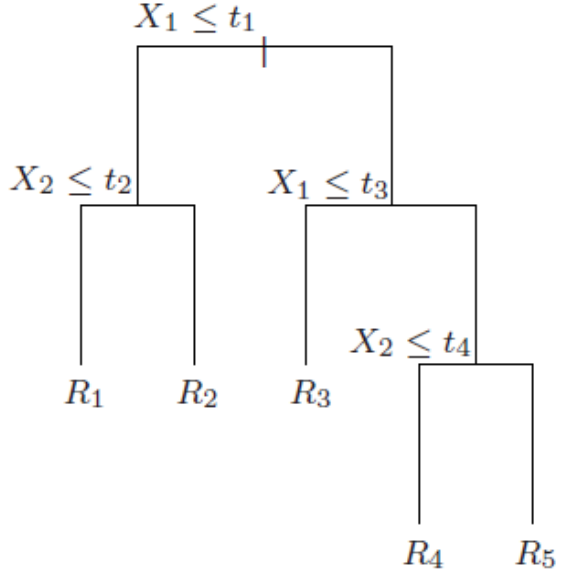
- We describe CART (classification and regression tree)
- Decision trees are linear in features:

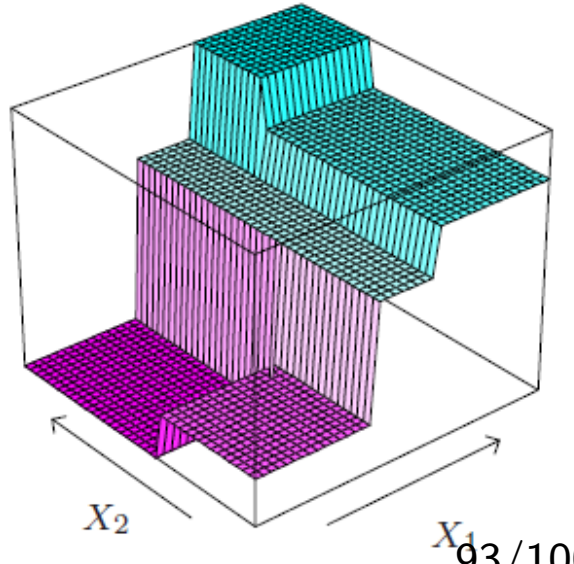
$$f(x) = \sum_{j=1}^{k} c_j \, [x \in R_j]$$

where R_j are disjoint rectangular regions and c_j the constant prediction in a region

- The regions are defined by a binary decision tree







Growing the decision tree

- The constants are the region averages $c_j = \frac{\sum_i y_i \ [x_i \in R_j]}{\sum_i [x_i \in R_j]}$
- Each split $x_a > t$ is defined by a choice of input dimension $a \in \{1, .., d\}$ and a threshold t
- Given a yet unsplit region R_j, we split it by choosing

$$\min_{a,t} \left[\min_{c_1} \sum_{i:x_i \in R_j \wedge x_a \leq t} (y_i - c_1)^2 + \min_{c_2} \sum_{i:x_i \in R_j \wedge x_a > t} (y_i - c_2)^2 \right]$$

  - Finding the threshold t is really quick (slide along)
  - We do this for every input dimension a

Deciding on the depth (if not pre-fixed)

- We first grow a very large tree (e.g. until at least 5 data points live in each region)

- Then we rank all nodes using "weakest link pruning":
  Iteratively remove the node that least increases

$$\sum_{i=1}^{n}(y_i - f(x_i))^2$$

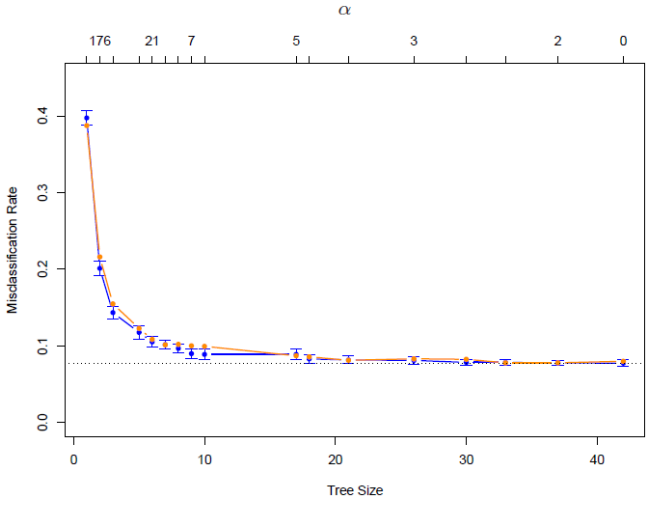
- Use cross-validation to choose the eventual level of pruning

  This is equivalent to choosing a regularization parameter λ for
  $$L(T) = \sum_{i=1}^{n}(y_i - f(x_i))^2 \ + \ \lambda|T|$$
  where the regularization $|T|$ is the tree size

Example:

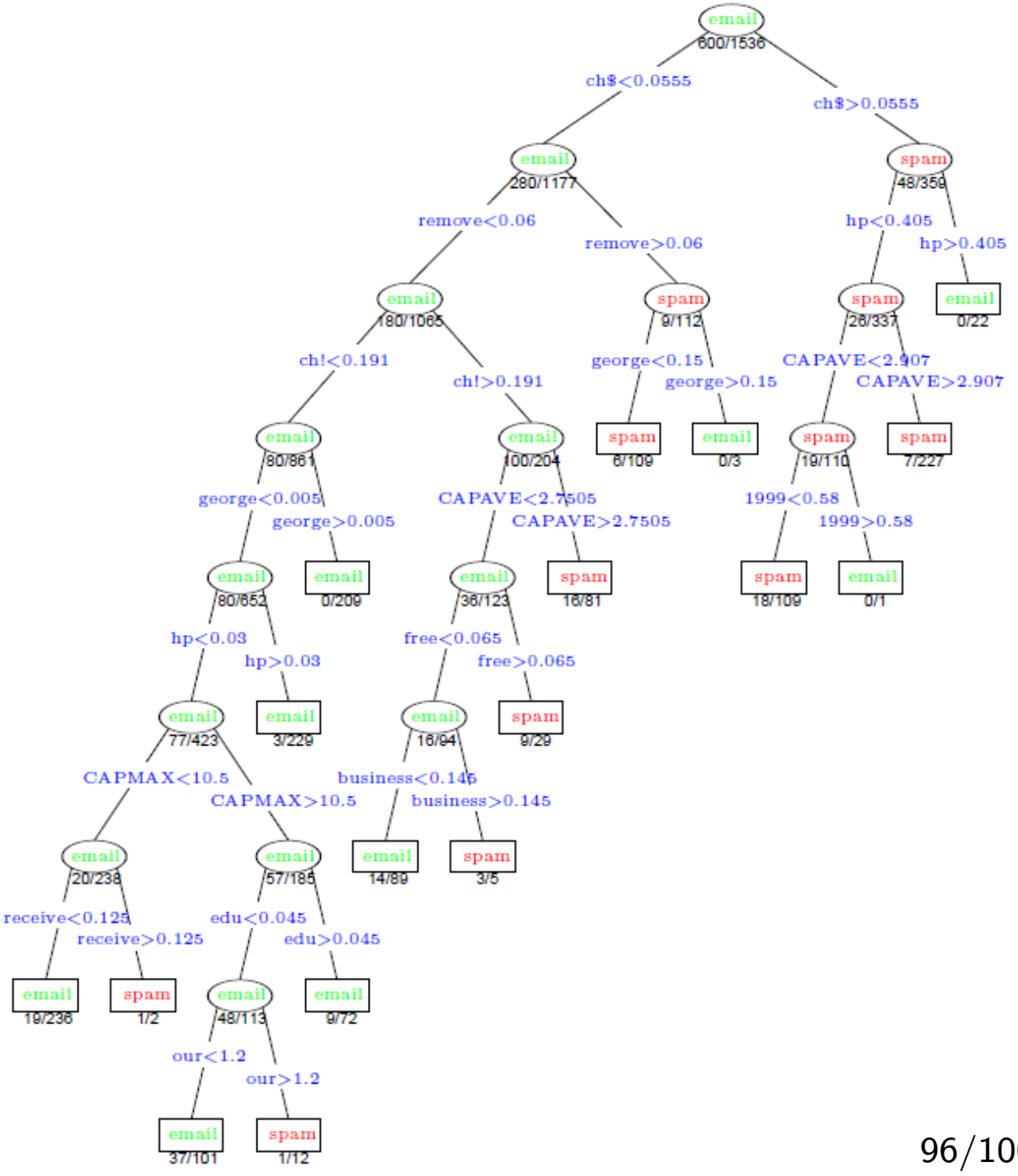
CART on the Spam data set

(details: Hastie, p 320)



	Predicted	
True	email	spam
email	57.3%	4.0%
spam	5.3%	33.4%

Test error rate: 8.7%



Boosting trees & stumps

- A **decision stump** is a decision tree with fixed depth 1 (just one split)

- Gradient boosting of decision trees (of fixed depth J) and stumps is very effective

  Test error rates on Spam data set:

full decision tree	8.7%
boosted decision stumps	4.7%
boosted decision trees with $J = 5$	4.5%

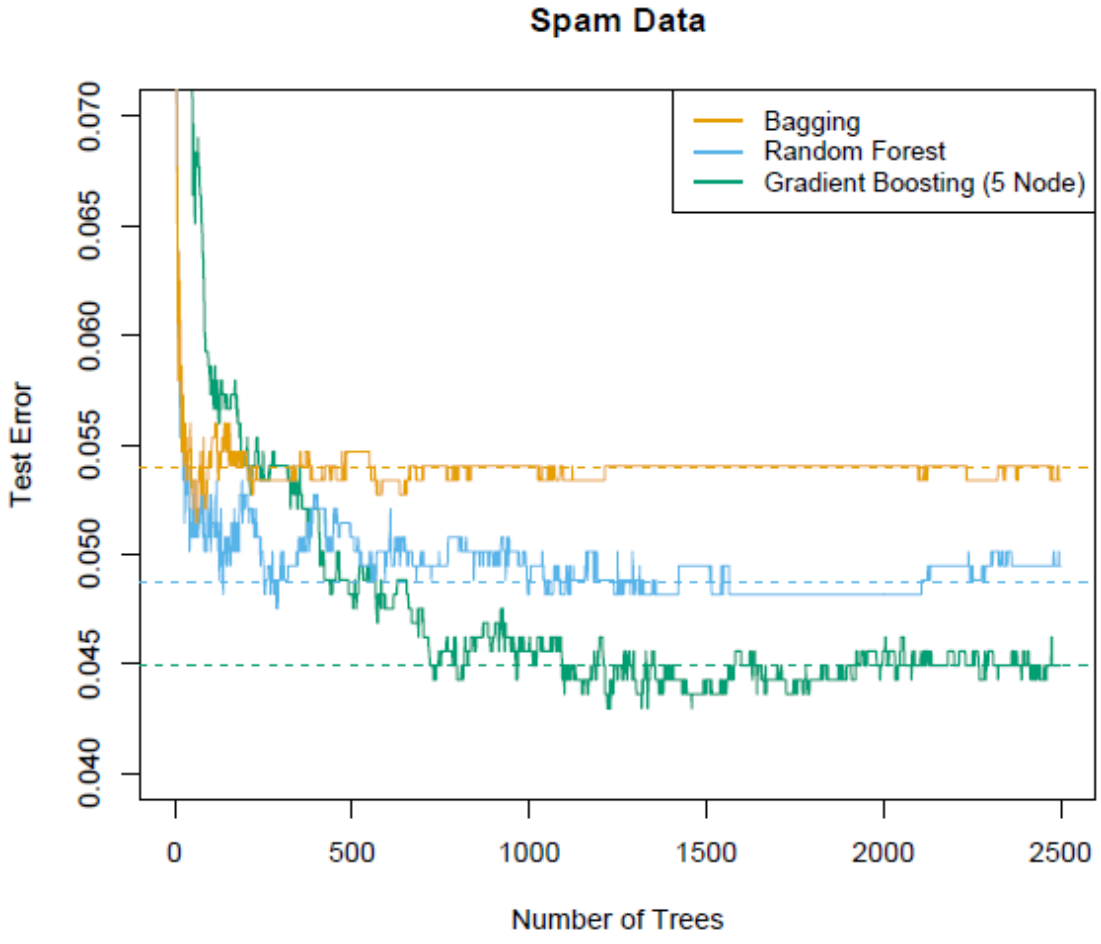
Random Forests: Bootstrapping & randomized splits

- Recall that Bagging averages models $f_1, .., f_M$ where each f_m was trained on a bootstrap resample D_m of the data D
  This randomizes the models and avoids over-generalization

- Random Forests do Bagging, but additionally randomize the trees:
  - When growing a new split, choose the input dimension a only from a *random subset* m features
  - m is often very small; even $m = 1$ or $m = 3$

- Random Forests are the prime example for "creating many randomized weak learners from the same data D"

Random Forests vs. gradient boosted trees



(Hastie, Fig 15.1)

Appendix: Centering & Whitening

- Some prefer to *center* (shift to zero mean) the data before applying methods:

  $$x \leftarrow x - \langle x \rangle \ , \quad y \leftarrow y - \langle y \rangle$$

  this spares augmenting the bias feature 1 to the data.

- More interesting: The loss and the best choice of λ depends on the *scaling* of the data. If we always scale the data in the same range, we may have better priors about choice of λ and interpretation of the loss

  $$x \leftarrow \frac{1}{\sqrt{\mathsf{Var}\{x\}}}\, x \ , \quad y \leftarrow \frac{1}{\sqrt{\mathsf{Var}\{y\}}}\, y$$

- **Whitening:** Transform the data to remove all correlations and varances.

  Let $A = \mathsf{Var}\{x\} = \frac{1}{n} X^\top X - \mu\mu^\top$ with Cholesy decomposition $A = MM^\top$.

  $$x \leftarrow M^{-1} x \ , \quad \text{with } \mathsf{Var}\{M^{-1}x\} = \mathbf{I}_d$$