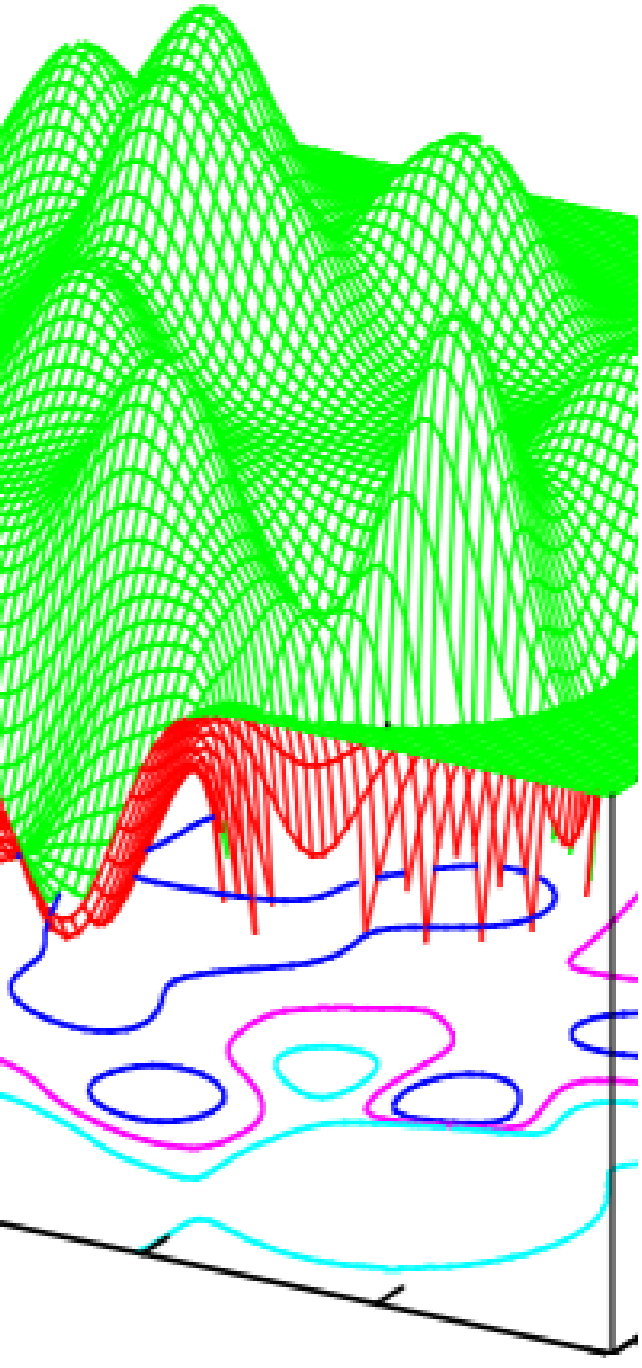


Machine Learning

Part 2: The Breadth of ML ideas

Marc Toussaint
U Stuttgart

The Breadth of ML ideas

- A. Ideas about features & data preprocessing
  – centering & whitening
  – PCA
  – PLS (for classification?)
- B. Ideas about local learners
  – local & lazy learning
  – kNN
  – kd-trees
- C. Ideas about combining weak or randomized learners
  – Bootstrap, bagging, and model averaging
  – Boosting
  – (Boosted) decision trees & stumps
  – Random forests
- D. Ideas about other loss functions
  – hinge loss, linear programming, SVMs
- E. Ideas about deep learners

Centering & Whitening

- Some researchers 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

  $$x \leftarrow \frac{1}{\sqrt{\mathrm{Var}(x)}}\, x \ , \quad y \leftarrow \frac{1}{\sqrt{\mathrm{Var}(y)}}\, y$$

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

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

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

Principle Component Analysis (PCA)

Principle Component Analysis (PCA)

- Image $x \in \mathbb{R}^d$ with really large d
  $\rightarrow$ high computational costs
  $\rightarrow$ many parameters, overfitting

  Are really all input dimensions necessary?

- Idea: We only select a subset of input dimensions

Principle Component Analysis (PCA)

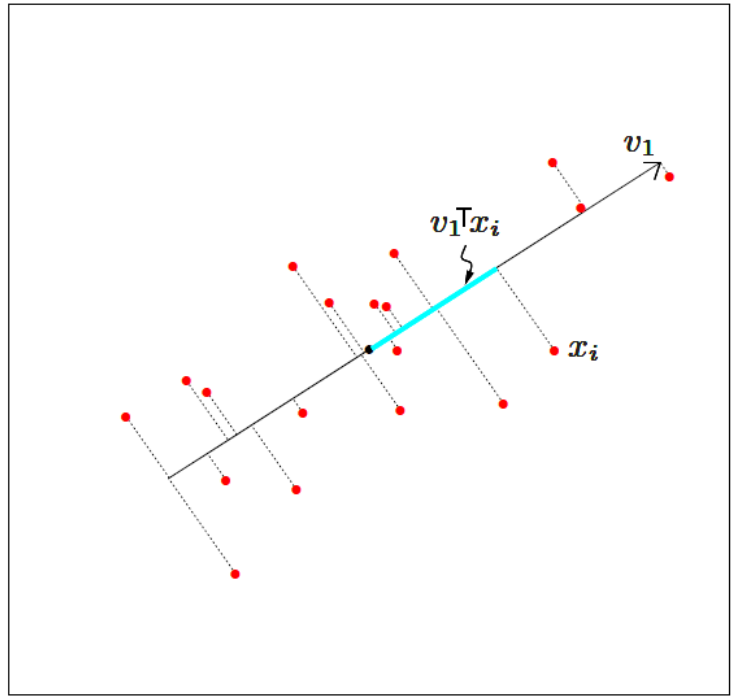
- Heuristic: Pick the directions of **largest variance**

- Given data in $\mathbb{R}^d$ we can compute
  – the mean $\mu = \langle x \rangle = \frac{1}{n} \sum_i x_i$
  – the variance $A = \mathrm{Var}(x) = \left\langle x x^\top \right\rangle - \mu\mu^\top = \frac{1}{n} \boldsymbol{X}^\top \boldsymbol{X} - \mu\mu^\top$
- The *principle components* are the those directions that have largest variance
  The **singular value decomposition** of a symmetric matrix is

$$A = V D V^\top$$

  with $D = \mathrm{diag}(\sigma_1^2, .., \sigma_d^2)$, $\sigma_1 \geq ... \geq \sigma_d$
  and $V = (v_1, .., v_d)$ with *orthonormal* row vectors v_i
  The sub-matrix $V_p = (v_1, .., v_p) \in \mathbb{R}^{d \times p}$ defines an orthogonal "p-dimensional coordinate system"
- $z = V_p^\top x$ is the projection of an input into these p dimensions

Principle Component Analysis (PCA)



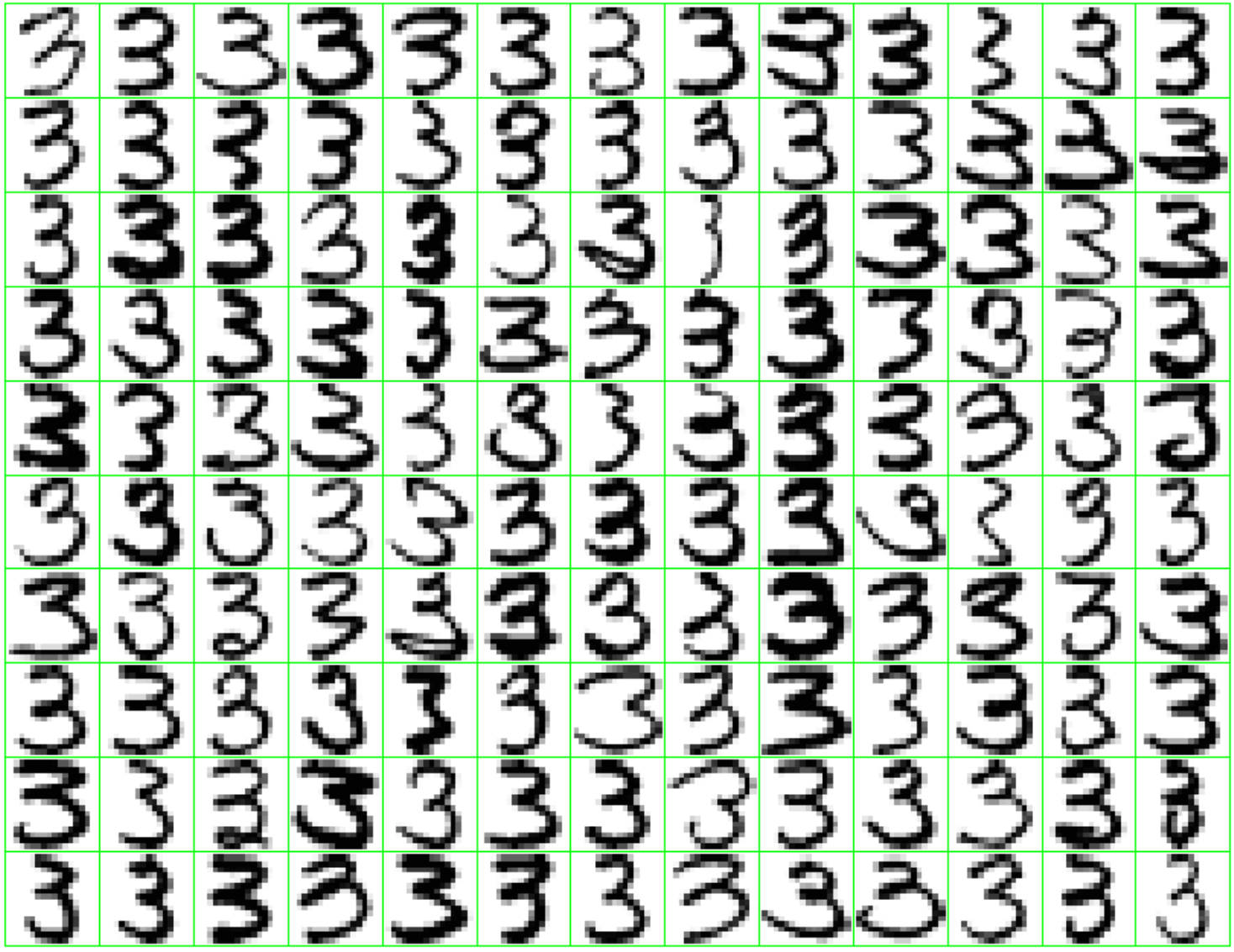
$V_p^\top$ is the matrix that projects to the largest variance directions of $\boldsymbol{X}^\top \boldsymbol{X}$

$$x \mapsto z = V_p^\top x$$

$$\boldsymbol{X} \mapsto \boldsymbol{Z} = \boldsymbol{X} V_p$$

- Heuristic: Apply ML method on top of $\boldsymbol{Z}$ instead of $\boldsymbol{X}$

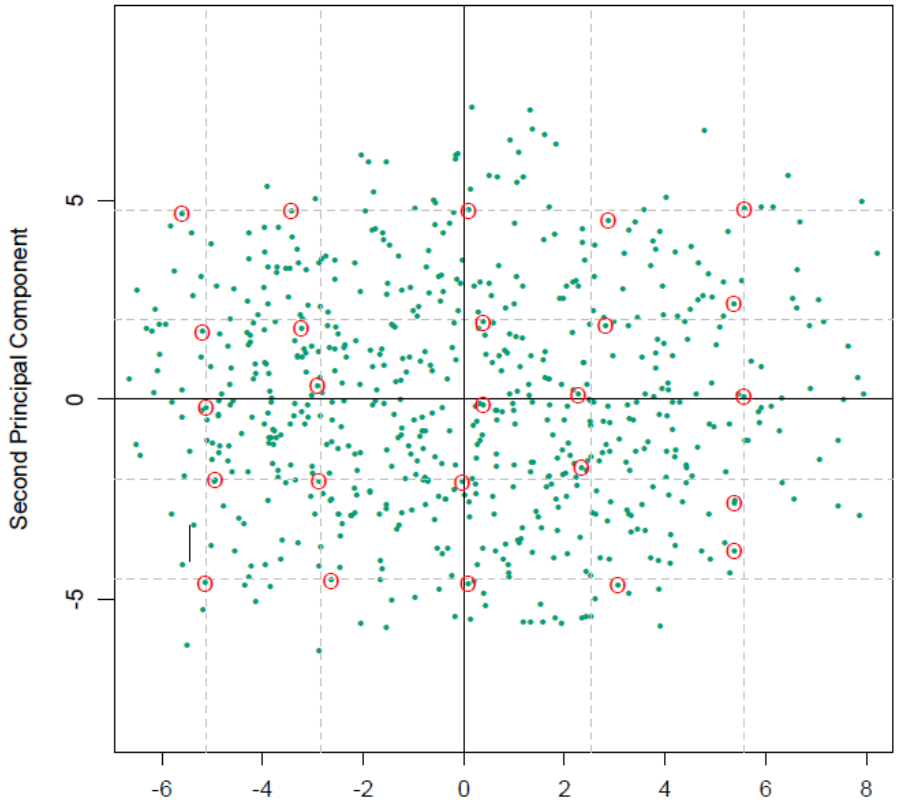
Example: Digits

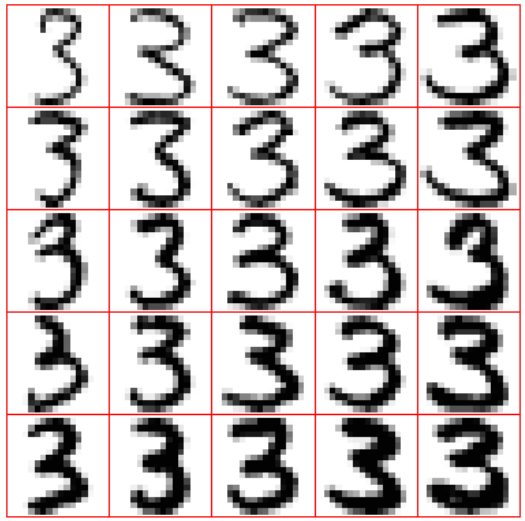


Example: Digits

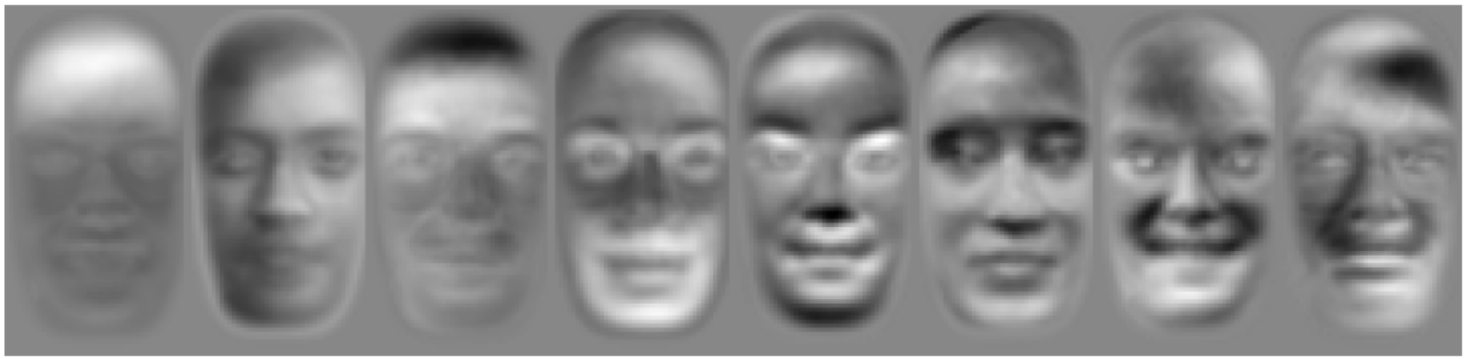
- The "basis vectors" in V_p are also **eigenvectors**
  Every data point can be expressed in these eigenvectors

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





Example: Eigenfaces



(Viola & Jones)

"Feature PCA" & Kernel PCA (for reference only)

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

- 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 $\boldsymbol{X}^\top \boldsymbol{X} = \sum_i \tilde{\phi}(x_i)\tilde{\phi}(x_i)^\top$. We can rewrite this as

$$\boldsymbol{X}^\top \boldsymbol{X} v_j = \lambda v_j$$

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

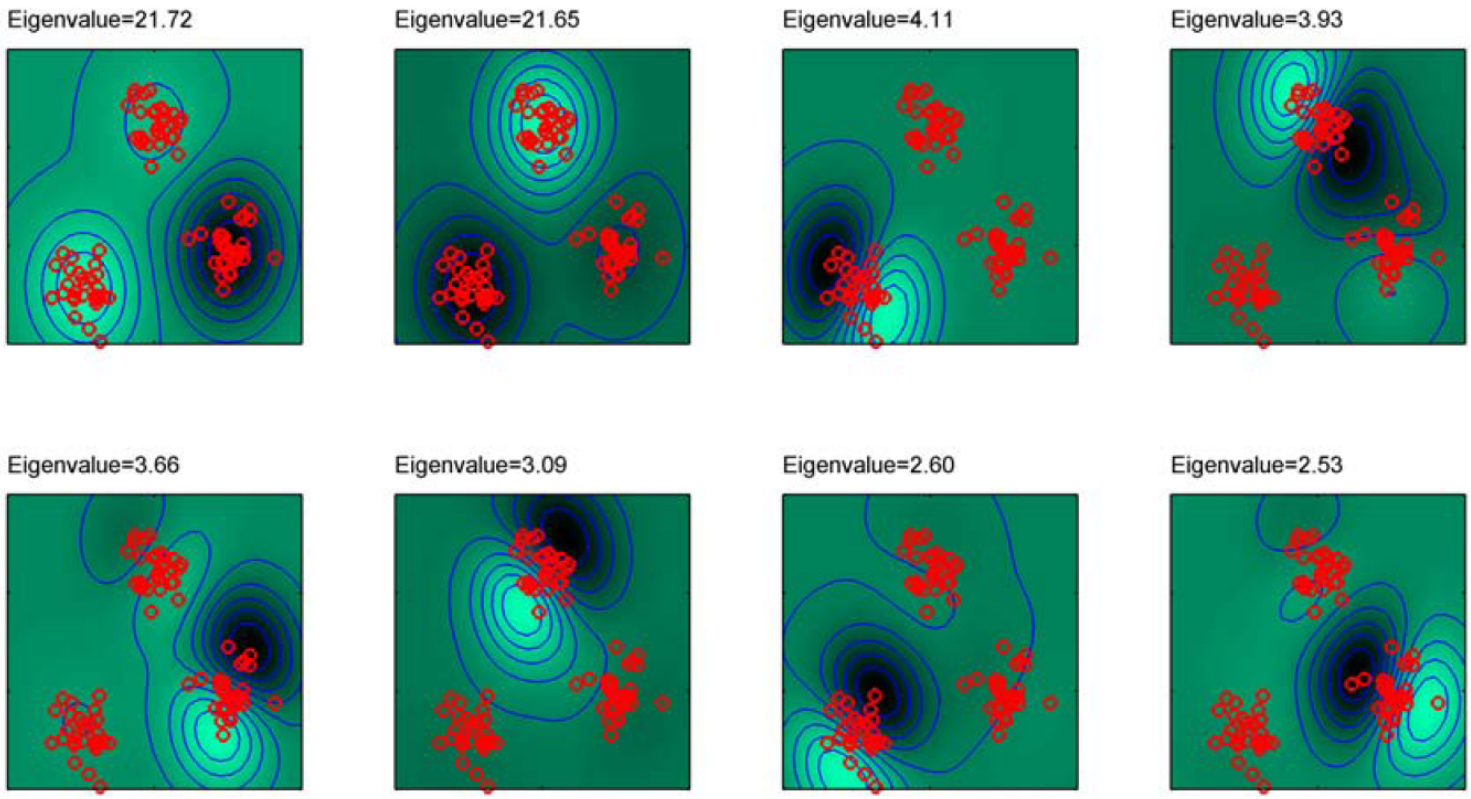
$$\boldsymbol{K}\alpha_j = \lambda \alpha_j$$

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

Kernel PCA

red points: data

green shading: eigenvector $\boldsymbol{\alpha}_j$ represented as $z_j(x) = \sum_i \alpha_{ji} k(x_j, x)$



Kernel PCA “coordinates” allow us to discriminate clusters!

Kernel PCA (for reference only)

- Kernel PCA uncovers quite surprising structure:

  While PCA is "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)

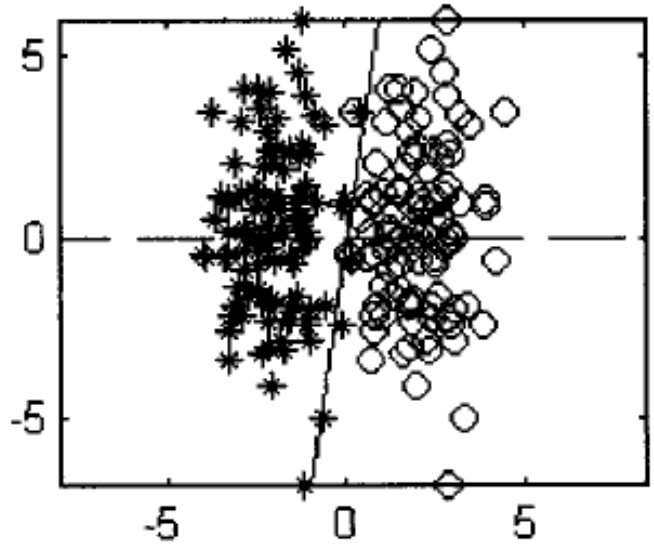
  see "Dimensionality Reduction: A Short Tutorial" by Ali Ghodsi

Partial Least Squares (PLS)

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

1: initialize the *predicted output*: $\hat{\boldsymbol{y}} = \langle y \rangle \, \mathbf{1}_n$
2: initialize the *remaining input dimensions*: $\hat{\boldsymbol{X}} = \boldsymbol{X}$
3: **for** $i = 1, .., p$ **do**
4: $\quad i$-coordinate for all data points: $\boldsymbol{z}_i = \hat{\boldsymbol{X}} \hat{\boldsymbol{X}}^\top \boldsymbol{y}$
5: $\quad$ update prediction: $\hat{\boldsymbol{y}} \leftarrow \hat{\boldsymbol{y}} + \frac{\boldsymbol{z}_i^\top \boldsymbol{y}}{\boldsymbol{z}_i^\top \boldsymbol{z}_i} \, \boldsymbol{z}_i$
6: $\quad$ remove "used" input dimensions: $\hat{\boldsymbol{x}}_j \leftarrow \hat{\boldsymbol{x}}_j - \frac{\boldsymbol{z}_i^\top \hat{\boldsymbol{x}}_j}{\boldsymbol{z}_i^\top \boldsymbol{z}_i} \, \boldsymbol{z}_i$
$\quad$ where $\hat{\boldsymbol{x}}_j$ is the jth column of $\hat{\boldsymbol{X}}$
7: **end for**

(Hastie, page 81)

Line 4 identifies a new "coordinate" as the maximal correlation between the remaning input dimensions and $\boldsymbol{y}$. All z_i are orthogonal.
Line 5 updates the prediction that is possible using z_i
Line 6 removes the "used" dimension from $\hat{\boldsymbol{X}}$ to ensure orthgonality in line 4

PLS for classification

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

B. Ideas about local learners

– local & lazy learning
– kNN
– kd-trees

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 points 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 points 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 points 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:

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

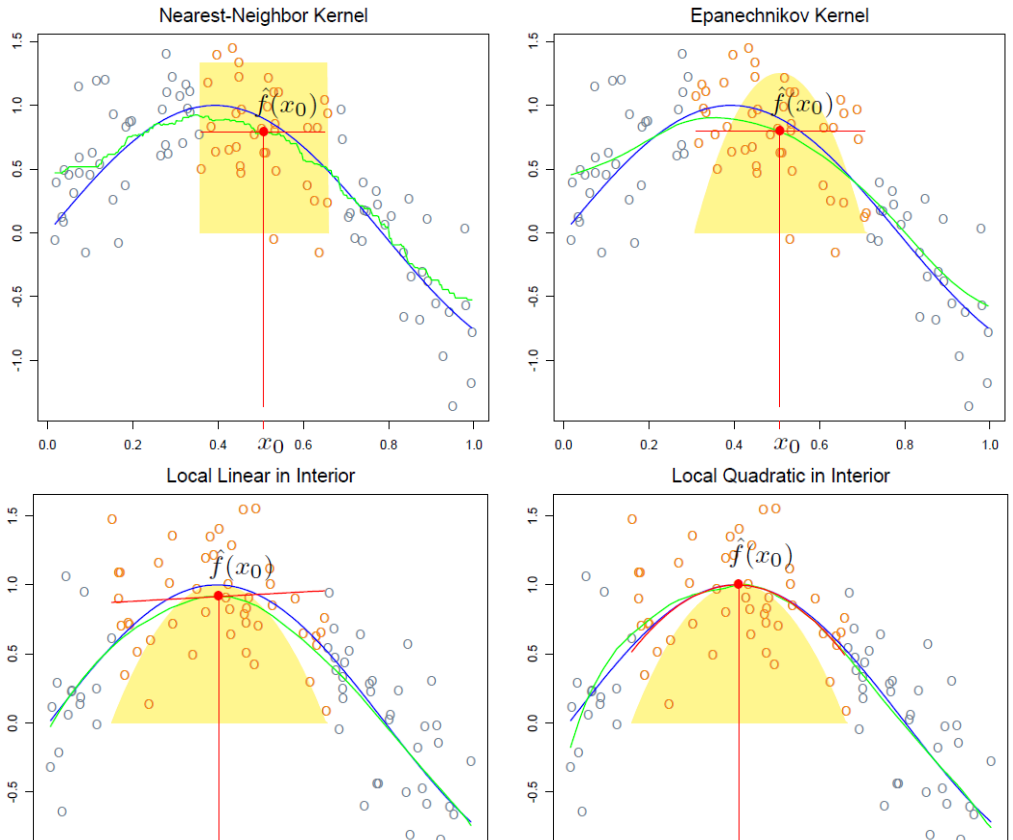
Local & lazy learning

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)



kd-trees

kd-trees

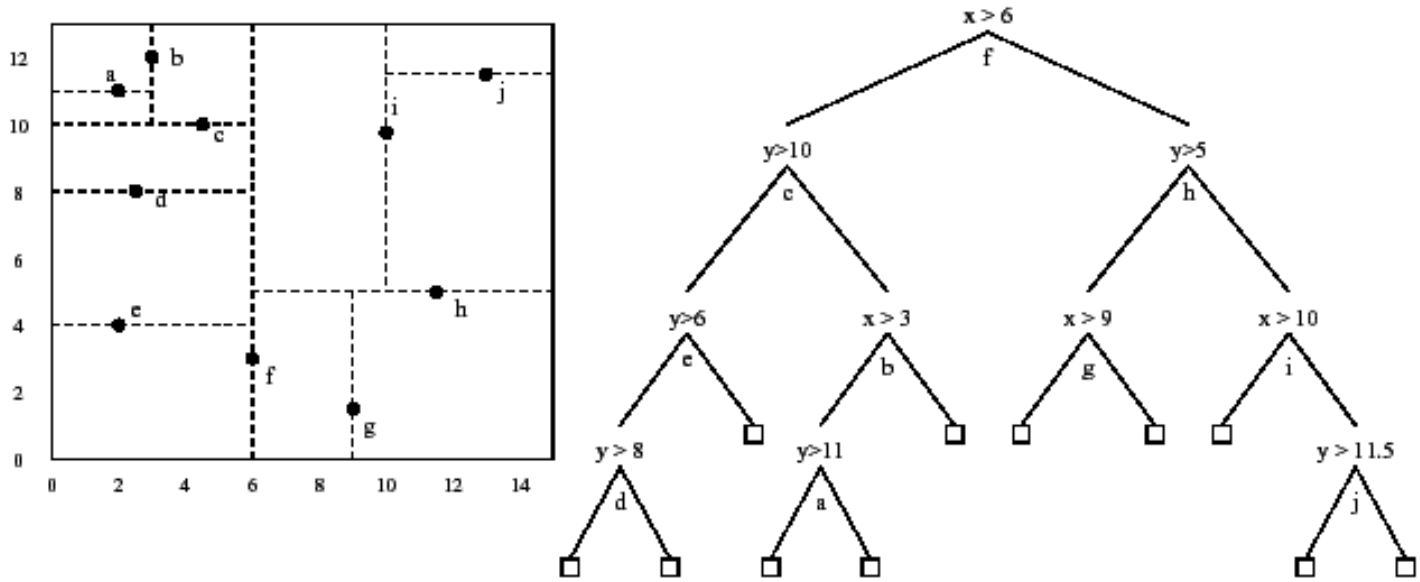
- For local & lazy learning it is essential to efficiently retrieve the kNN

  Problem: Given a data set 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..)

C. Ideas about combining weak or randomized learners

- Bootstrap, bagging, and model averaging
- Boosting
- (Boosted) decision trees & stumps
- Random forests

Combining learners

- The general idea is:
  – Given a 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; whatever)

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

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

- Concerning their combination, the interesting question is

  How to choose the α_m?

Bootstrap & Bagging

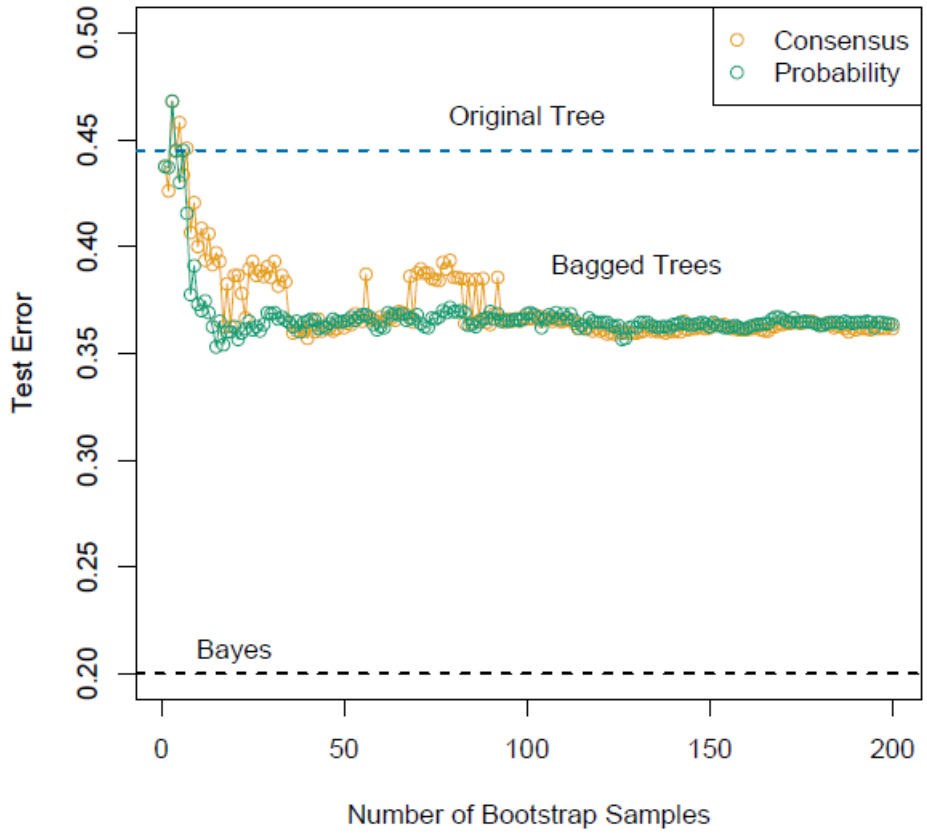
- **Bootstrap:**
  - Data set D of size n
  - Generate M data sets D_m by resampling D *with replacement*
  - Each D_m is also of size n (some samples doubled or missing)

  - Distribution over data sets $\leftrightarrow$ distribution over β (compare slide 02-14)
  - 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 model
  – Equal weighting would not be clever
  – More confident models (less variance, less parameters, high 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

Boosting

- In Bagging and Model Averaging, the models are trained on the same data, or unbiased randomized versions

- 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)

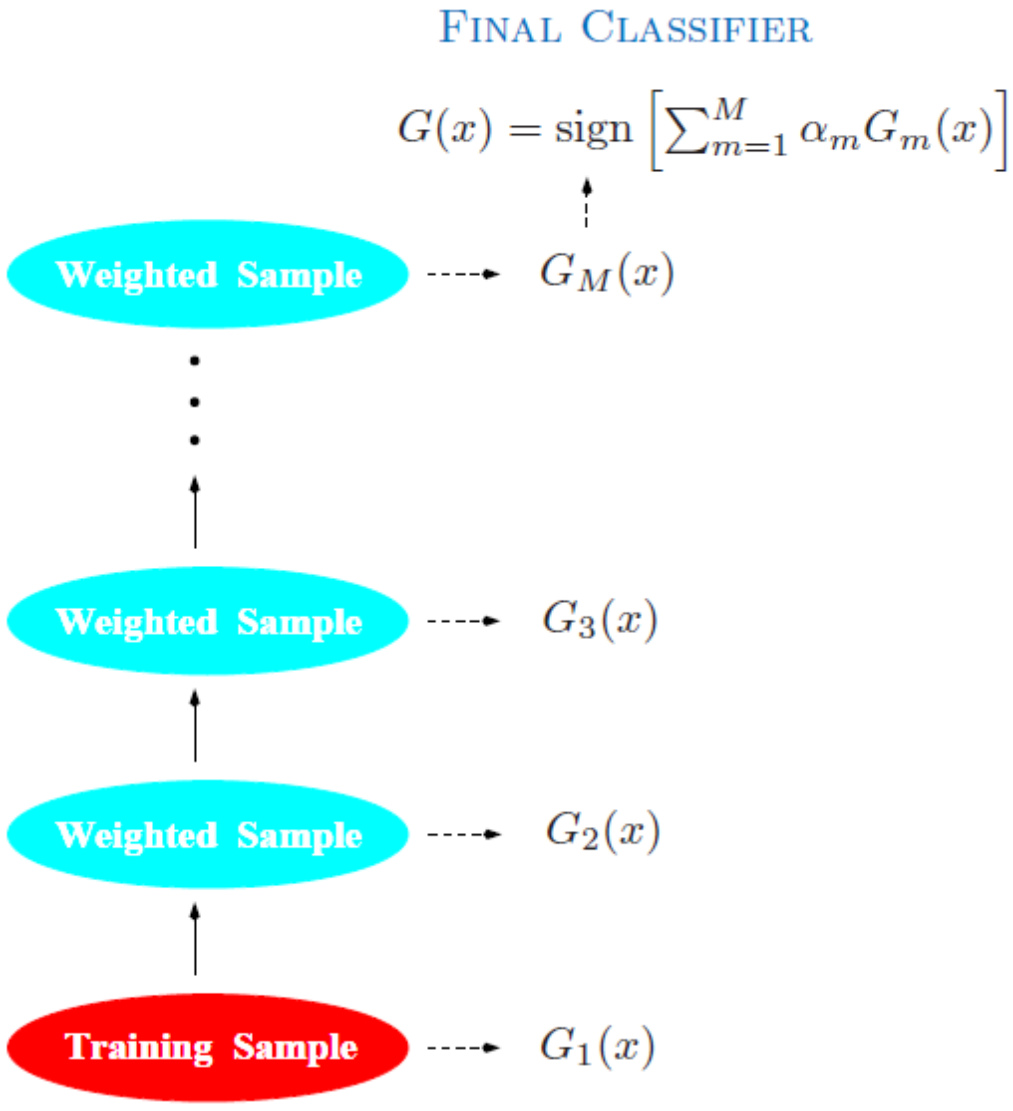
- Binary classification problem with data $D = \{(x_i, y_i)\}_{i=1}^n$, $y_i \in \{-1, +1\}$
- We know how to train discriminative functions $f(x)$; let

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

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

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

- 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
– Can use linear regression to choose α_m (instead of line search)

Gradient Boosting

- Hastie's book quite "likes" gradient boosting
  – Can be applied to any loss function
  – No matter if regression or classification (or CRFs)
  – Very good performance

  – Simpler, more general, better than AdaBoost

Decision Trees

Decision Trees

- So far we always referred to our “core learners” (Part I of this lecture)
- Decision trees are particularly used in Bagging and Boosting contexts

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

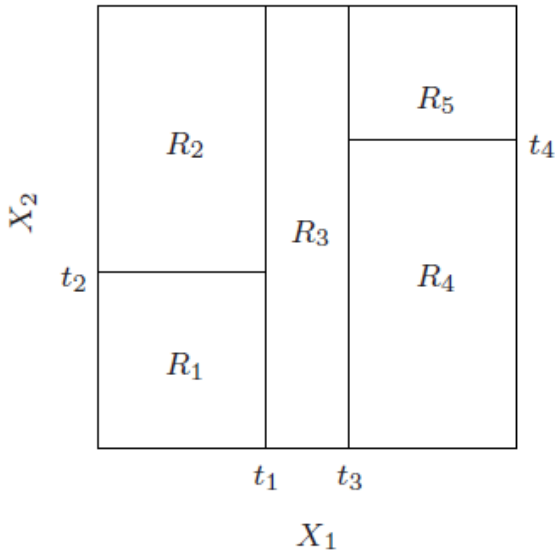
Decision Trees

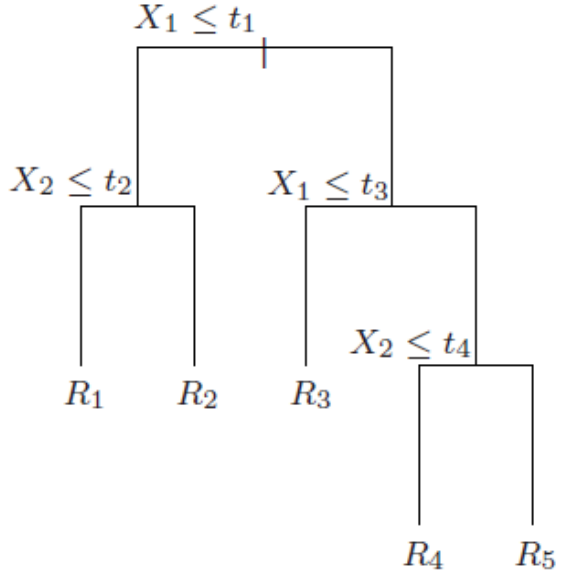
- We describe CART (classification and regression tree)
- Decision trees are linear in features:

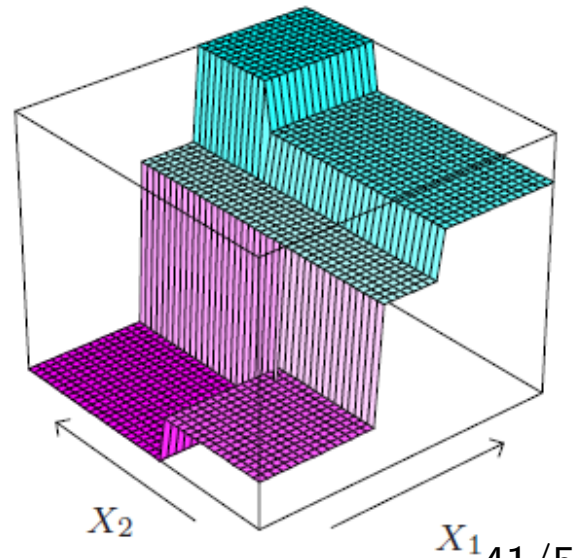
$$f(x) = \sum_{j=1}^{k} c_j \, I(x \in R_j)$$

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 $c_j = \frac{\sum_i y_i \ I(x_i \in R_j)}{\sum_i I(x_i \in R_j)}$
- Each split $x_a > t$ is defined by a choice of feature (input dimension) $a \in \{1, .., d\}$ and a threshold t
- Given a yet unsplit region R_j, we split is 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 feature (input dimension) a

Growing the decision tree

- We first grow a very large tree (e.g. until a minimum region size of 5)

- 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

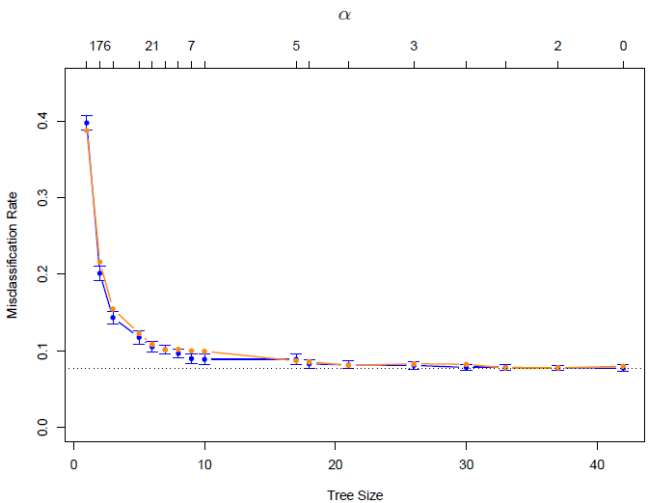
$$L(T) = \sum_{i=1}^{n}(y_i - f(x_i))^2 \; + \; \lambda|T|$$

  where the regularization $|T|$ is the tree size

Example:

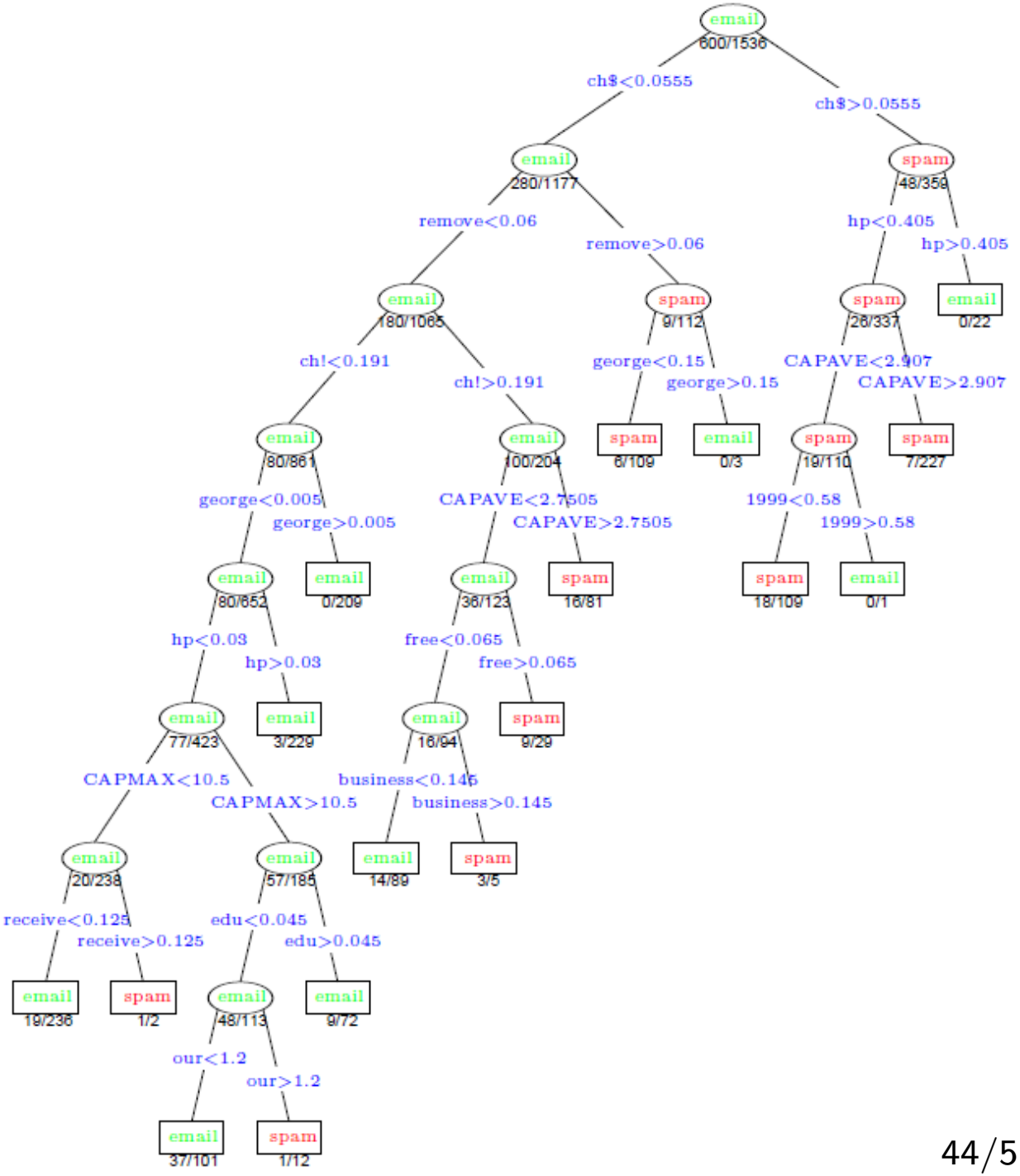
CART on the Spam data set

(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 just one split
- Gradient boosting of decision trees 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%

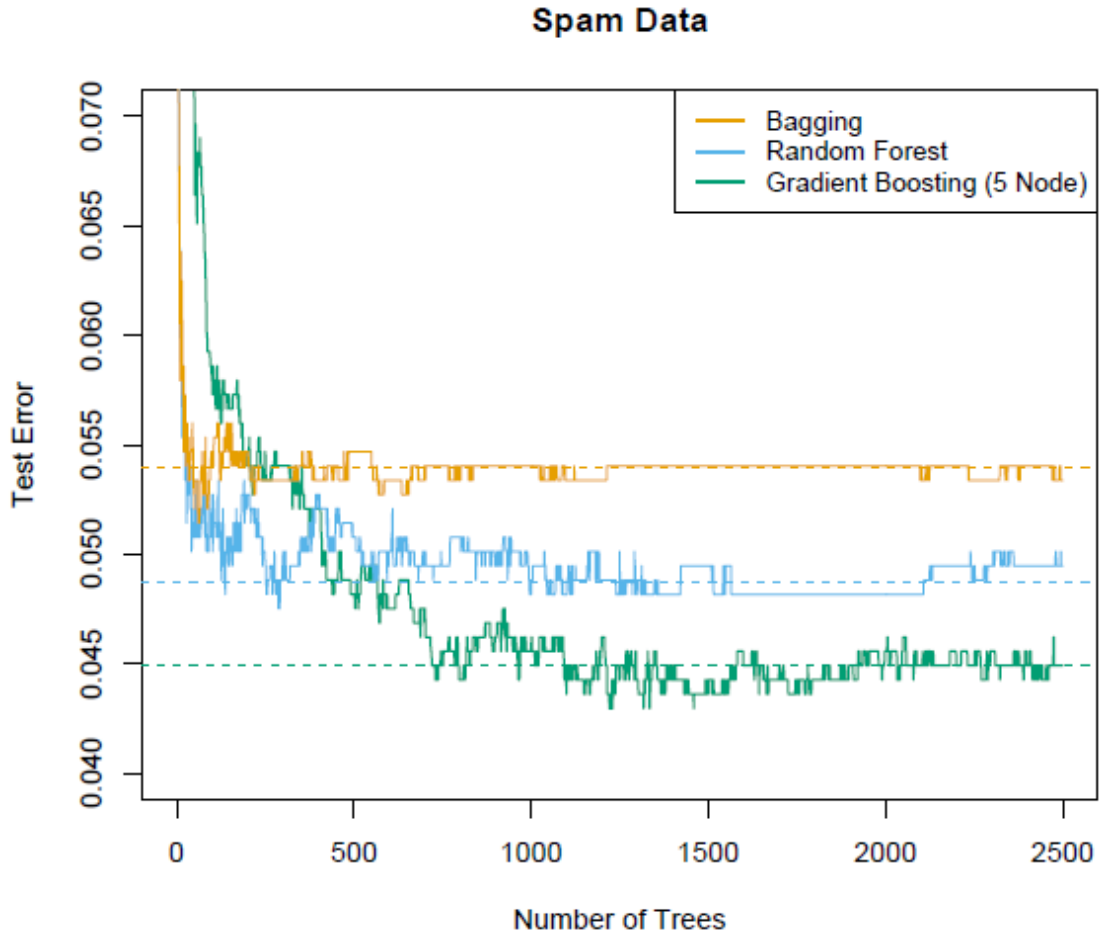
Random Forests: Bagging & randomized splits

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

- Random Forests do Bagging, but additionally randomize the trees:

  – When growing a new split, choose the feature (input dimension) a only from a *random subset* m features
  – m is often very small; even $m = 1$ or $m = 3$

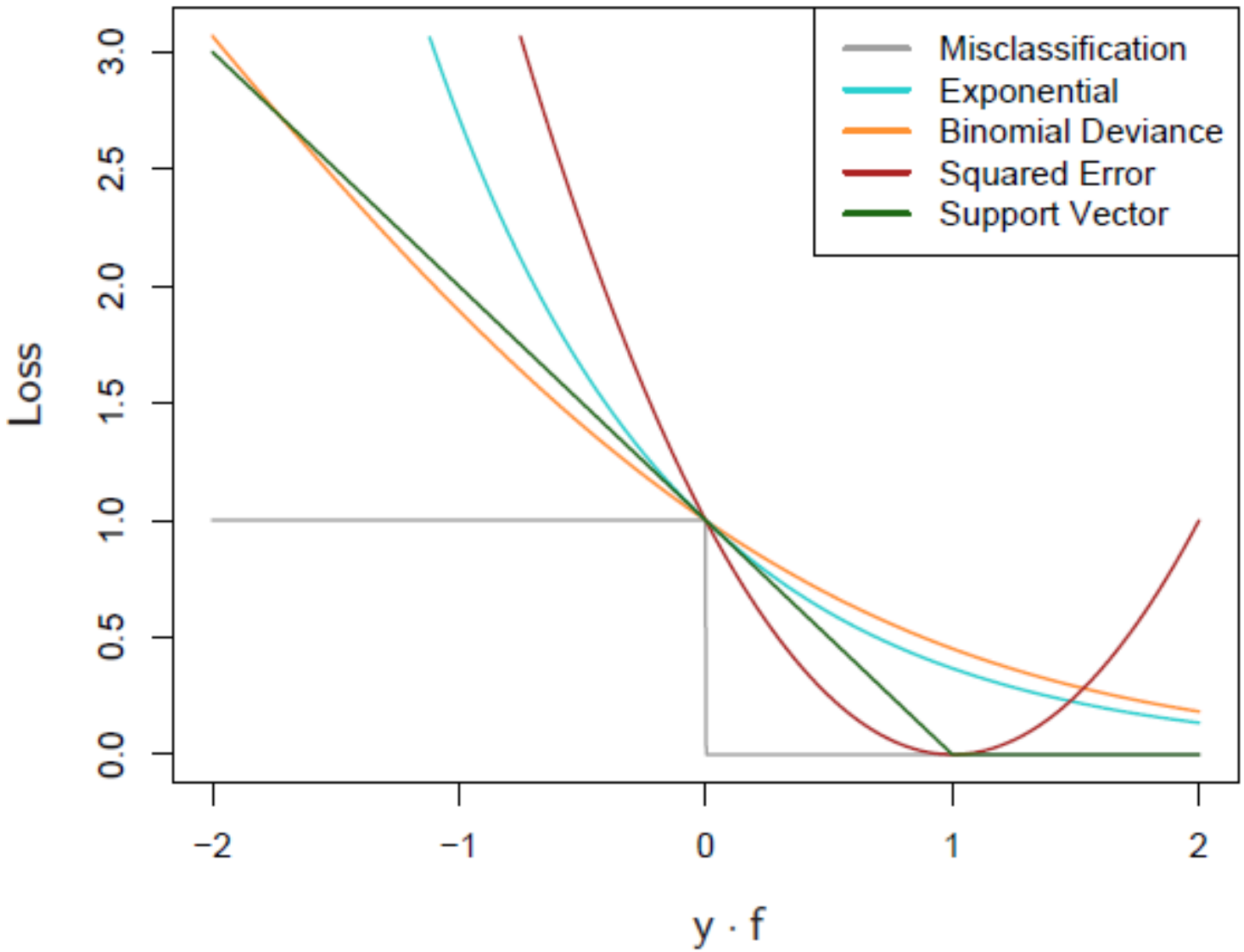
Random Forests vs. gradient boosted trees

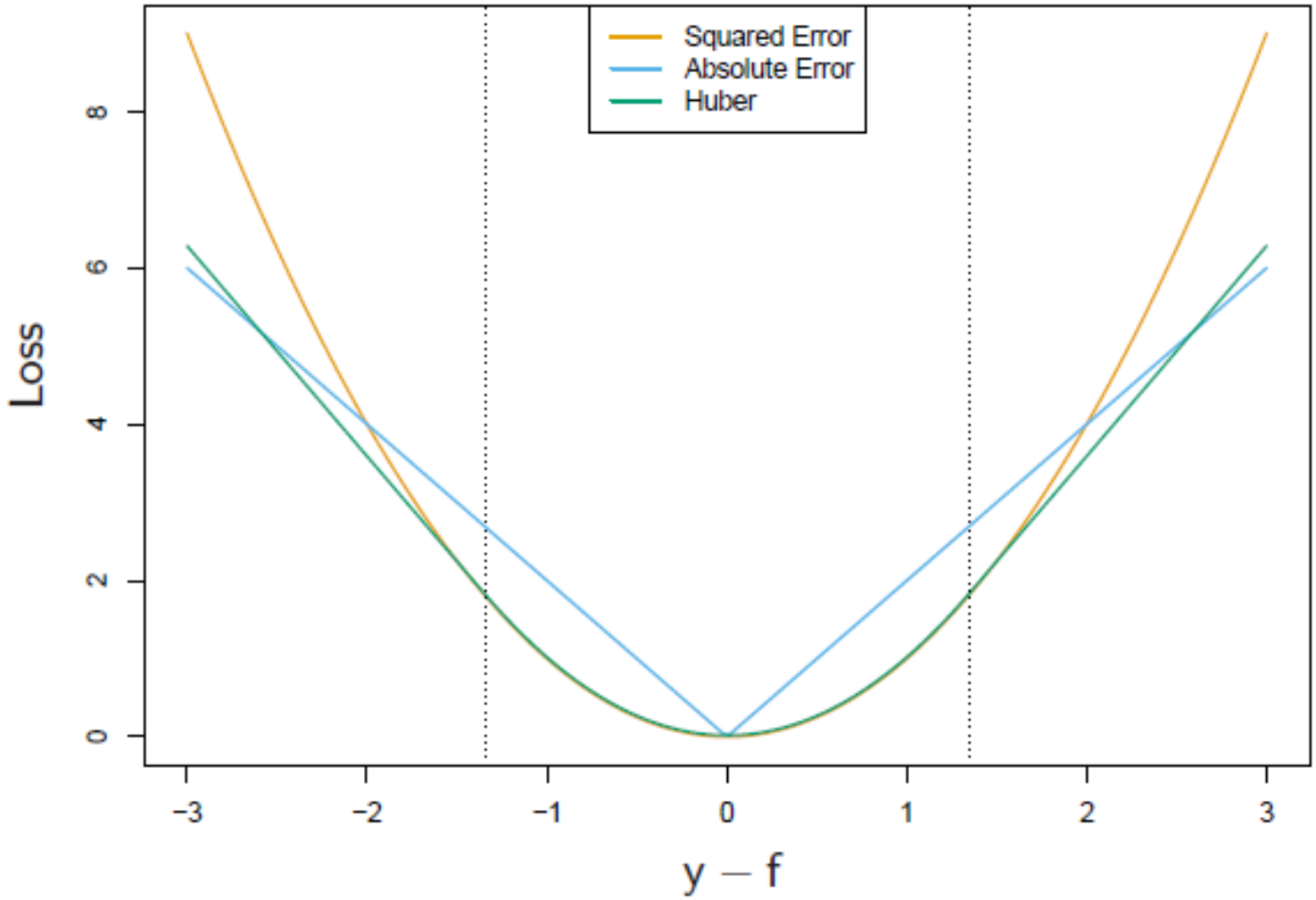


(Hastie, Fig 15.1)

D. Ideas about other loss functions

– hinge loss, linear programming, SVMs


Misclassification
Exponential
Binomial Deviance
Squared Error
Support Vector
Loss
0.0
0.5
1.0
1.5
2.0
2.5
3.0
−2
−1
0
1
2
y · f


Squared Error
Absolute Error
Huber
Loss
0
2
4
6
8
−3
−2
−1
0
1
2
3
y − f