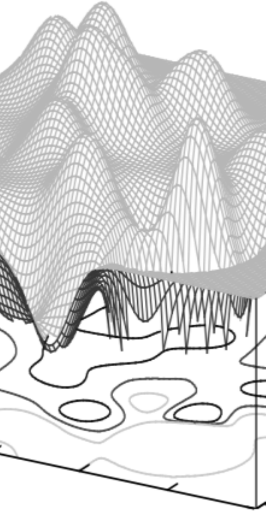


Machine Learning

The Breadth of ML methods

Marc Toussaint
University of Stuttgart
Summer 2019

Local learning & Ensemble learning

- “Simpler is Better”

Local learning & Ensemble learning

- "Simpler is Better"
  - We've learned about [kernel] ridge — logistic regression
  - We've learned about high-capacity NN training
  - Sometimes one should consider also much simpler methods as baseline

- Content:
  - 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

Local & lazy 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:

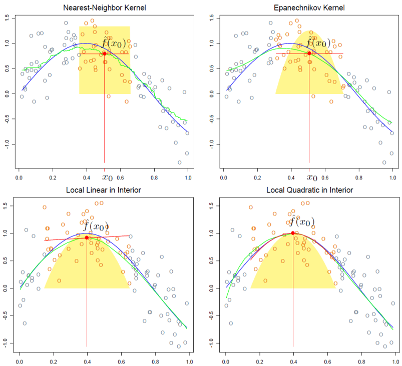
$$\hat{\beta} = (X^\top W X + \lambda I)^{\text{-}1} X^\top W y \ , \quad W = \text{diag}(K(x^*, x_{1:n}))$$

Regression example

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

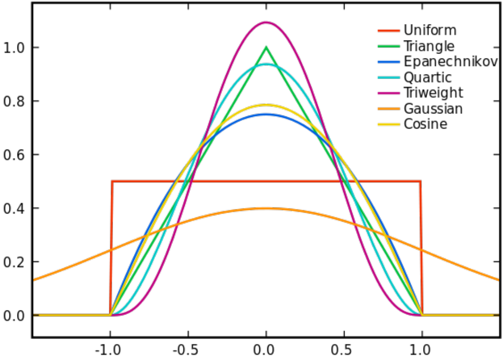
Epanechnikov quadratic smoothing kernel:

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



(Hastie, Sec 6.3)

Smoothing Kernels



from Wikipedia

Which metric to use for NN?

- This is *the* crutial question? The fundamental question of generalization.
  - Given a query x^*, which data points x_i would you consider as being "related", so that the label of x_i is correlated to the correct label of x^*?

Which metric to use for NN?

- This is *the* crutial question? The fundamental question of generalization.
  - Given a query x^*, which data points x_i would you consider as being "related", so that the label of x_i is correlated to the correct label of x^*?

- Possible answers beyond naive Euclidean distance $|x^* - x_i|$
  - Some other kernel function $k(x^*, x_i)$
  - First encode x into a "meaningful" latent representation z; then use Euclidean distance there

Which metric to use for NN?

- This is *the* crutial question? The fundamental question of generalization.
  - Given a query x^*, which data points x_i would you consider as being "related", so that the label of x_i is correlated to the correct label of x^*?

- Possible answers beyond naive Euclidean distance $|x^* - x_i|$
  - Some other kernel function $k(x^*, x_i)$
  - First encode x into a "meaningful" latent representation z; then use Euclidean distance there
  - Take some off-the-shelf pretrained image NN, chop of the head, use this internal representation

kd-trees

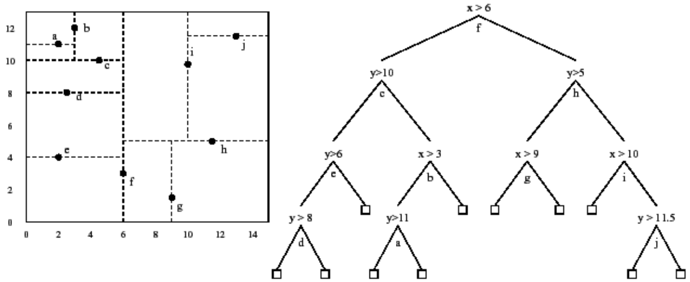
- For local & lazy learning it is essential to efficiently retrieve the kNN

  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 nets; 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? (You should know that!)

Bootstrap & Bagging

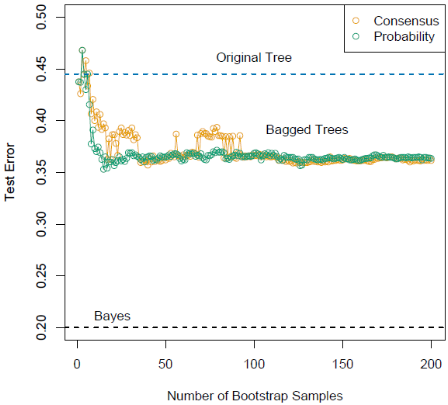
- **Bootstrap:**
  - Data set D of size n
  - Generate M data sets D_m by resampling D *with replacement*
  - Each D_m is also of size n (some samples doubled or missing)

  - 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)

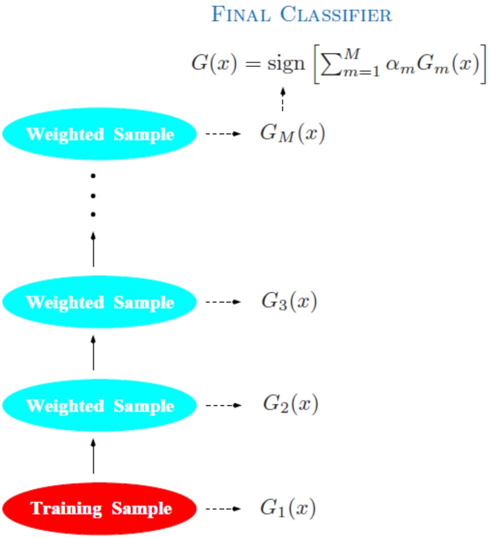
- Binary classification problem with data $D = \{(x_i, y_i)\}_{i=1}^n$, $y_i \in \{-1, +1\}$
- We know how to train discriminative functions $f(x)$; let

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

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

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

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

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

Classical examples for boosting

Decision Trees

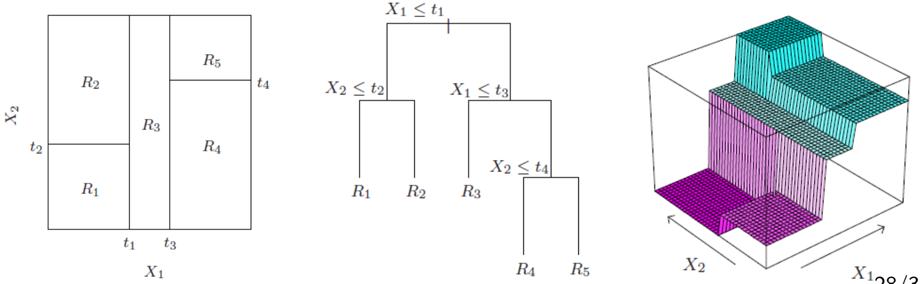
- Decision trees are particularly used in Bagging and Boosting contexts
- Decision trees are “linear in features”, but the features are the terminal regions of a tree, which are constructed depending on the data
- We’ll learn about
  - Boosted decision trees & stumps
  - Random Forests

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

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

  – 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 most 5 data points live in each region)

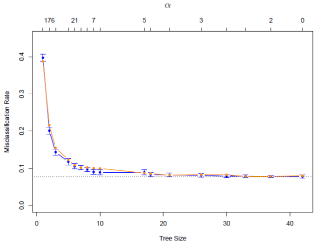
- Then we rank all nodes using “weakest link pruning”:
  Iteratively remove the node that least increases

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

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

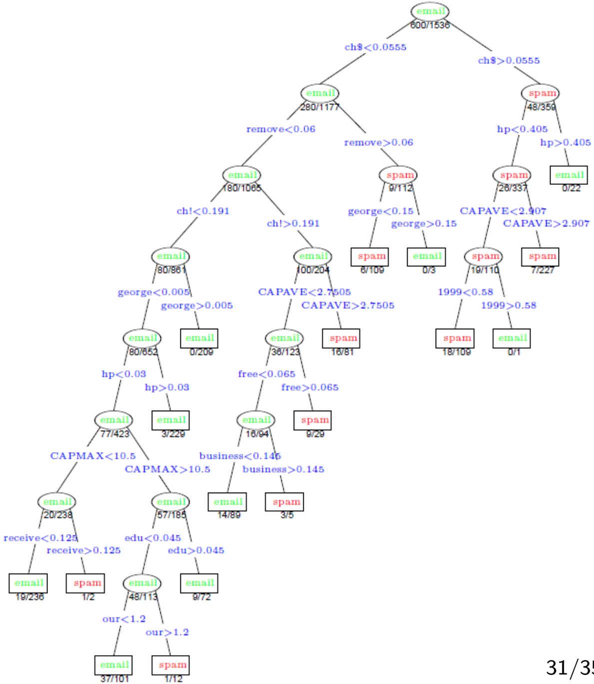
This is equivalent to choosing a regularization parameter λ for
$$L(T) = \sum_{i=1}^{n}(y_i - f(x_i))^2 \ + \ \lambda|T|$$
where the regularization $|T|$ is the tree size

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%

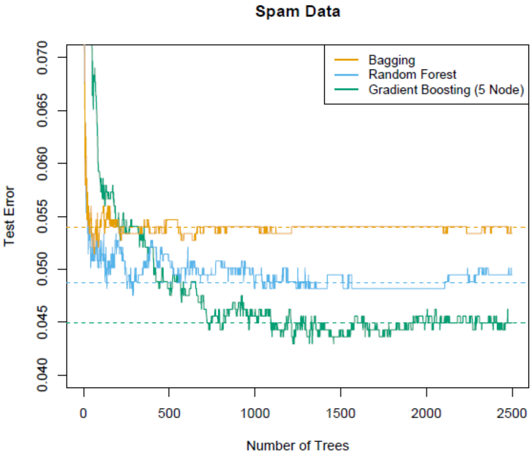
Random Forests: Bootstrapping & randomized splits

- 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

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

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

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

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