STAT 479: Machine Learning
Lecture Notes

Sebastian Raschka
Department of Statistics
University of Wisconsin–Madison

http://stat.wisc.edu/~sraschka/teaching/stat479-fs2019/

Fall 2019

Contents

STAT 479: Machine Learning
Lecture Notes

Sebastian Raschka
Department of Statistics
University of Wisconsin–Madison

http://stat.wisc.edu/~sraschka/teaching/stat479-fs2019/

Fall 2019

7 Ensemble Methods

7.1 Introduction

- In broad terms, using ensemble methods is about combining models to an ensemble such that the ensemble has a better performance than an individual model on average.
- The main categories of ensemble methods involve voting schemes among high-variance models to prevent "outlier" predictions and overfitting, and the other involves boosting "weak learners" to become "strong learners."

7.2 Majority Voting

- We will use the term "majority" throughout this lecture in the context of voting to refer to both majority and plurality voting.[1]
- Plurality: mode, the class that receives the most votes; for binary classification, majority and plurality are the same

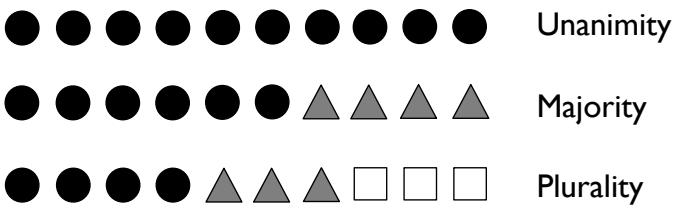

Figure 1: Illustration of unanimity, majority, and plurality voting

[1] In the UK, people distinguish between majority and plurality voting via the terms "absolute" and "relative" majority, respectively, https://en.wikipedia.org/wiki/Plurality_(voting).

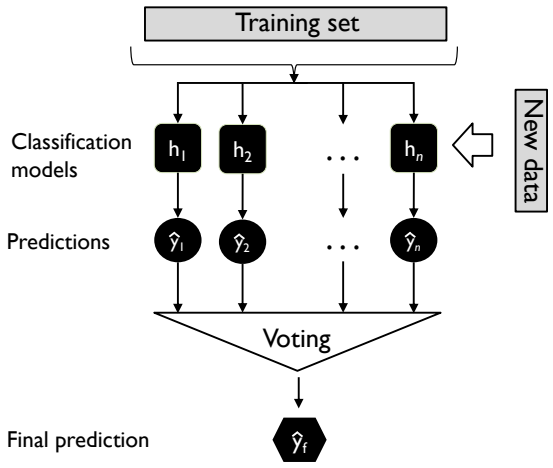

Figure 2: Illustration of the majority voting concept. Here, assume n different classifiers, $\{h_1, h_2, ..., h_n\}$ where $h_i(\mathbf{x}) = \hat{y}_i$.

In lecture 2, (*Nearest Neighbor Methods*) we learned that the majority (or plurality) voting can simply be expressed as the *mode*:

$$\hat{y}_f = mode\{h_1(\mathbf{x}), h_2(\mathbf{x}), ...h_n(\mathbf{x})\}. \tag{1}$$

The following illustration demonstrates why majority voting can be effective (under certain assumptions).

- Given are n independent classifiers $(h_1, ..., h_n)$ with a base error rate ϵ;
- Here, independent means that the errors are uncorrelated;
- Assume a binary classification task (there are two unique class labels).

Assuming the error rate is better than random guessing (i.e., lower than 0.5 for binary classification),

$$\forall \epsilon_i \in \{\epsilon_1, \epsilon_2, ..., \epsilon_n\}, \epsilon_i < 0.5, \tag{2}$$

the error of the ensemble can be computed using a binomial probability distribution since the ensemble makes a wrong prediction if more than 50% of the n classifiers make a wrong prediction.

The probability that we make a wrong prediction via the ensemble if k classifiers predict the same class label (where $k > \lceil n/2 \rceil$ because of majority voting) is then

$$P(k) = \binom{n}{k} \epsilon^k (1 - \epsilon)^{n-k}, \tag{3}$$

where $\binom{n}{k}$ is the binomial coefficient

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}. \tag{4}$$

However, we need to consider all cases $k \in \{\lceil n/2 \rceil, ..., n\}$ (cumulative prob. distribution) to compute the ensemble error

$$\epsilon_{ens} = \sum_{k}^{n} \binom{n}{k} \epsilon^k (1-\epsilon)^{n-k}. \tag{5}$$

Consider the following example with n=11 and $\epsilon = 0.25$, where the ensemble error decreases substantially compared to the error rate of the individual models:

$$\epsilon_{ens} = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1-0.25)^{11-k} = 0.034. \tag{6}$$

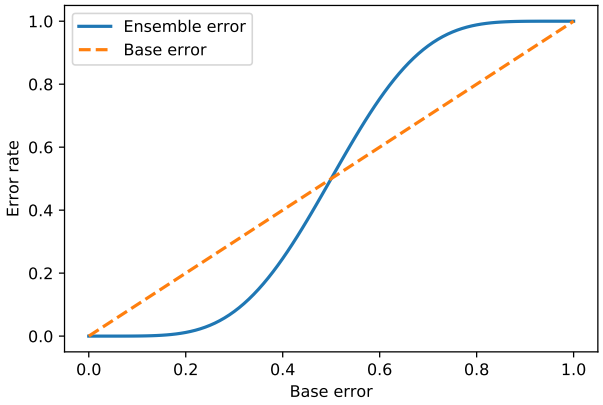

Figure 3: error-rate

7.3 Soft Majority Voting

For well calibrated classifiers[2] we can also use the predicted class membership probabilities to infer the class label,

$$\hat{y} = \arg\max_{j} \sum_{i=1}^{n} w_i p_{i,j}, \tag{7}$$

where $p_{i,j}$ is the predicted class membership probability for class label j by the ith classifier. Here w_i is an optional weighting parameter. If we set

$w_i = 1/n, \forall w_i \in \{w_1, ..., w_n\}$,

then all probabilities are weighted uniformly.

To illustrate this, let us assume we have a binary classification problem with class labels $j \in \{0, 1\}$ and an ensemble of three classifiers $h_i (i \in \{1, 2, 3\})$:

$$h_1(\mathbf{x}) \rightarrow [0.9, 0.1], h_2(\mathbf{x}) \rightarrow [0.8, 0.2], h_3(\mathbf{x}) \rightarrow [0.4, 0.6]. \tag{8}$$

[2]For more information about calibrating classifiers, see https://scikit-learn.org/stable/auto_examples/calibration/plot_calibration.html.

We can then calculate the individual class membership probabilities as follows:

$$p(j=0|\mathbf{x}) = 0.2 \cdot 0.9 + 0.2 \cdot 0.8 + 0.6 \cdot 0.4 = 0.58, p(j=1|\mathbf{x}) = 0.2 \cdot 0.1 + 0.2 \cdot 0.2 + 0.6 \cdot 0.6 = 0.42. \tag{9}$$

The predicted class label is then

$$\hat{y} = \arg\max_j \Big\{ p(j=0|\mathbf{x}),\ p(j=1|\mathbf{x}) \Big\} = 0. \tag{10}$$

7.4 Bagging

- Bagging relies on a concept similar to majority voting but uses the same learning algorithm (typically a decision tree algorithm) to fit models on different subsets of the training data (bootstrap samples).
- In a nutshell, a boostrap sample is a sample of size n drawn with replacement from an original training set $\mathcal{D}$ with $|D| = n$. Consequently, some training examples are duplicated in each bootstrap sample, and some other training examples do not appear in a given bootstrap sample at all (usually, we refer to these examples as "out-of-bag sample".). This is illustrated in Figure 5. We will revisit bootstrapping later in the model evaluation lectures.
- In the limit, there are approx. 63.2% unique examples in a given boostrap sample. Consequently, 37.2% examples from the original training set won't appear in a given boostrap sample at all. The justification is illustrated below in Eq. 11 to Eq. 13.
- Bagging can improve the accuracy of unstable models that tend to overfit[3].

Algorithm 1 Bagging

1: Let n be the number of bootstrap samples
2:
3: **for** i=1 to n **do**
4: Draw bootstrap sample of size m, $\mathcal{D}_i$
5: Train base classifier h_i on $\mathcal{D}_i$
6: $\hat{y} = mode\{h_1(\mathbf{x}), ..., h_n(\mathbf{x})\}$

Figure 4: The bagging algorithm.

Figure 5: Illustration of bootstrap sampling

[3] Leo Breiman. "Bagging predictors". In: *Machine learning* 24.2 (1996), pp. 123–140.

- If we sample from a uniform distribution, we can compute the probability that a given example from a dataset of size n is *not* drawn as a bootstrap sample as

$$P(\text{not chosen}) = \left(1 - \frac{1}{n}\right)^n, \tag{11}$$

which is asymptotically equivalent to

$$\frac{1}{e} \approx 0.368 \quad \text{as} \quad n \to \infty. \tag{12}$$

Vice versa, we can then compute the probability that a sample is chosen as

$$P(\text{chosen}) = 1 - \left(1 - \frac{1}{n}\right)^n \approx 0.632. \tag{13}$$

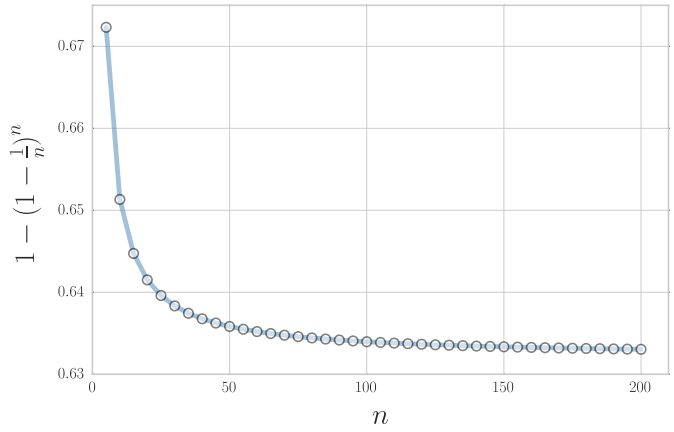

Figure 6: Proportion of unique training examples in a bootstrap sample.

Training example indices	Bagging round 1	Bagging round 2	…
1	2	7	…
2	2	3	…
3	1	2	…
4	3	1	…
5	7	1	…
6	2	7	…
7	4	7	…
	h_1	h_2	h_n

Figure 7: Illustration of bootstrapping in the context of bagging.

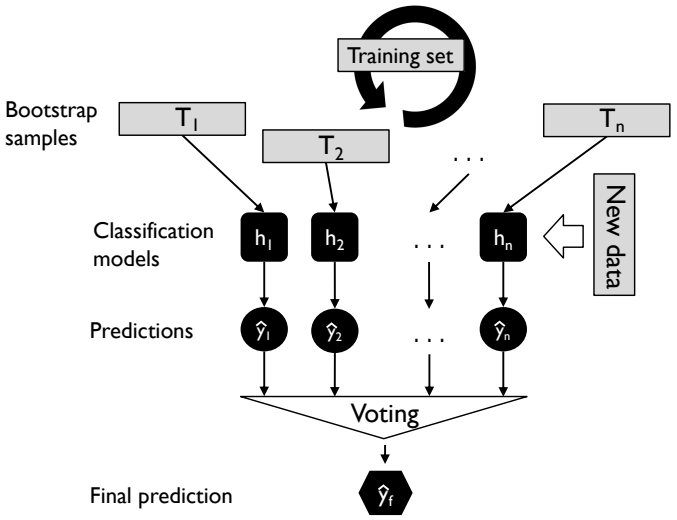

Figure 8: The concept of bagging. Here, assume n different classifiers, $\{h_1, h_2, ..., h_n\}$ where $h_i(\mathbf{x}) = \hat{y}_i$.

7.5 Bias and Variance Intuition

- "Bias and variance" will be discussed in more detail in the next lecture, where we will decompose loss functions into their variance and bias components and see how it relates to overfitting and underfitting.

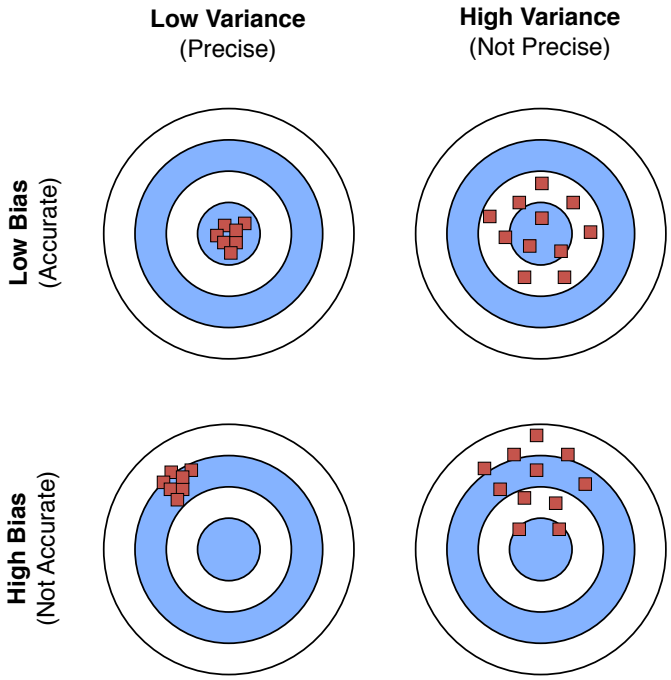

Figure 9: Bias and variance intuition.

- One can say that individual, unpruned decision tree "have high variance" (in this context, the individual decision trees "tend to overfit"); a bagging model has a lower variance than the individual trees and is less prone to overfitting – again, bias and variance decomposition will be discussed in more detail next lecture.

7.6 Boosting

- There are two broad categories of boosting: Adaptive boosting and gradient boosting.
- Adaptive and gradient boosting rely on the same concept of boosting "weak learners" (such as decision tree stumps) to "strong learners."
- Here, a decision tree stump refers to a decision tree that only has one level, i.e., $h(\mathbf{x}) = s\big(\mathbf{1}(x_k \geq t)\big)$, where $s(x) \in \{-1, 1\}$ and $k \in \{1, ..., K\}$ (K is the number of features).
- Boosting is an iterative process, where the training set is reweighted, at each iteration, based on mistakes a weak leaner made (i.e., misclassifications); the two approaches, adaptive and gradient boosting, differ mainly regarding how the weights are updated and how the classifiers are combined.

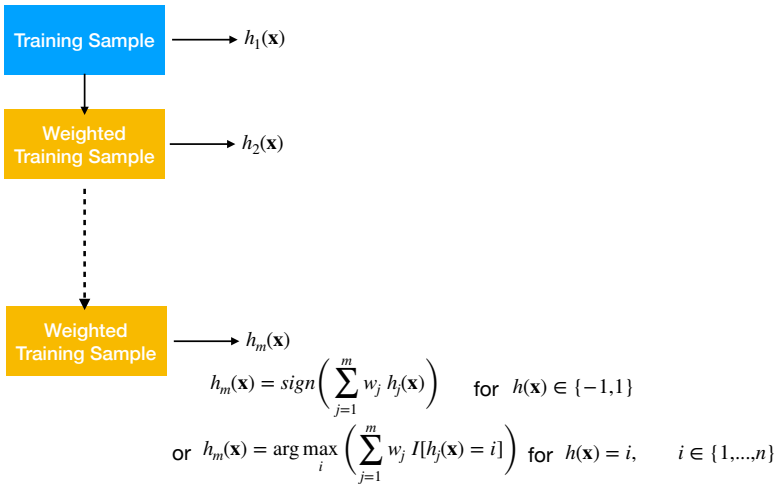

Figure 10: A general outline of the boosting procedure for n iterations.

Intuitively, we can outline the general boosting procedure for AdaBoost is as follows:

- Initialize a weight vector with uniform weights
- Loop:
  - Apply *weak learner* to weighted training examples (instead of orig. training set, may draw bootstrap samples with weighted probability)
  - Increase weight for misclassified examples
- (Weighted) majority voting on trained classifiers

7.6.1 AdaBoost (Adaptive Boosting)

- AdaBoost (short for "adaptive boosting") was initially described by Freund and Schapire in 1997[4].

[4]Yoav Freund and Robert E Schapire. "A decision-theoretic generalization of on-line learning and an application to boosting". In: *Journal of computer and system sciences* 55.1 (1997), pp. 119–139.

Algorithm 1 AdaBoost

Initialize k: the number of AdaBoost rounds
Initialize $\mathcal{D}$: the training dataset, $\mathcal{D} = \{\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, ..., \mathbf{x}^{[n]}, y^{[n]} \rangle\}$
Initialize $w_1(i) = 1/n, \quad i = 1, ..., n, \ \mathbf{w}_1 \in \mathbb{R}^n$

for r=1 to k **do**
  For all $i : \mathbf{w}_r(i) := w_r(i) / \sum_i w_r(i)$ [normalize weights]
  $h_r := FitWeakLearner(\mathcal{D}, \mathbf{w}_r)$
  $\epsilon_r := \sum_i w_r(i) \ \mathbf{1}(h_r(i) \neq y_i)$ [compute error]
  if $\epsilon_r > 1/2$ then stop
  $\alpha_r := \frac{1}{2} \log[(1 - \epsilon_r)/\epsilon_r]$ [small if error is large and vice versa]
  $w_{r+1}(i) := w_r(i) \times \begin{cases} e^{-\alpha_r} & \text{if } h_r(\mathbf{x}^{[i]}) = y^{[i]} \\ e^{\alpha_r} & \text{if } h_r(\mathbf{x}^{[i]}) \neq y^{[i]} \end{cases}$
Predict: $h_k(\mathbf{x}) = \arg\max_j \sum_r \alpha_r \mathbf{1}[h_r(\mathbf{x}) = j]$

Figure 11: AdaBoost algorithm.

In Figure 11, $\mathbf{1}\left(h_r(i) \neq y_i\right)$ refers to the 0/1 loss,

$$\mathbf{1}\left(h_r(i) \neq y_i\right) = \begin{cases} 0 & \text{if } h_r(i) = y_i \\ 1 & \text{if } h_r(i) \neq y_i \end{cases}.$$

The early stopping criterion if $\epsilon_r > 1/2$ then stop in Figure 11 assumes a binary classification setting. Otherwise, it can be generatlized to $\epsilon_r > 1/c$, where c is the number of unique class labels in the training set.

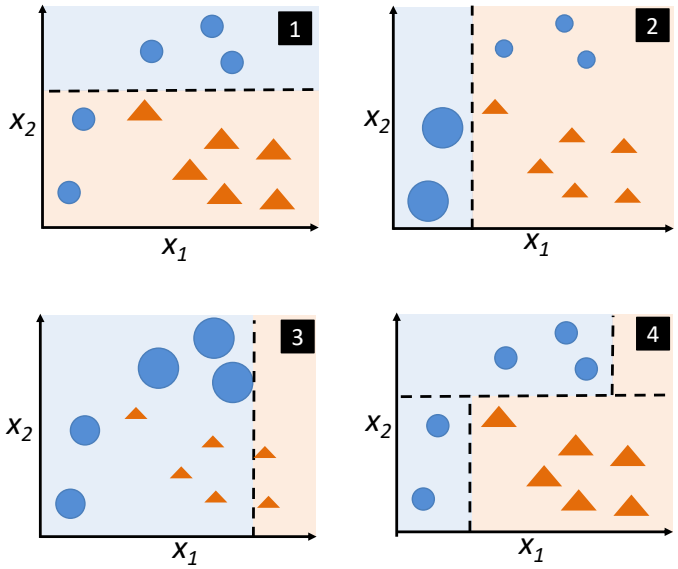

Figure 12: Illustration of the AdaBoost algorithms for three iterations on a toy dataset. The size of the symbols (circles shall represent training examples from one class, and triangles shall represent training examples from another class) is proportional to the weighting of the training examples at each round. The 4th subpanel shows a combination/ensemble of the hypotheses from subpanels 1-3.

7.6.2 Gradient Boosting

Similar to AdaBoost, gradient boosting[5] is an ensemble method that combines multiple weak learners (such as decision tree stumps) into a strong learner (here, a weak learner is an algorithm that produces models with a low predictive performance; vice versa, a strong learner is a learning algorithm that produces models with a high predictive performance). Also, similar to AdaBoost (and in contrast to Bagging), gradient boosting is a sequential (rather than parallel) algorithm – it is powerful but rather expensive to train[6].

In contrast to AdaBoost, we do not adjust the weights for the training examples that have been either misclassified or correctly classified. Also, we do not compute a weight for each model in the sequential gradient boosting ensemble. Instead, in gradient boosting, we optimize a differentiable loss function (e.g., mean-squared-error for regression or negative log-likelihood for classification) of a weak learner via consecutive rounds of boosting. The output of gradient boosting is an additive models of multiple weak learners (in contrast to AdaBoost, we do not apply majority voting to the ensemble of models).

Note that in gradient boosting, we may use decision trees (the most common base learning algorithm for gradient boosting is in fact *a* decision tree algorithm), but we do not necessarily use decision tree stumps. The first model in gradient boosting is basically just a decision tree's root node (on which we do majority voting in classification or averaging in regression). The consequent models in gradient boosting are deeper decision trees. The depth is usually determined by the practitioner. Values like 8 or 32 are common, depending on the complexity of the dataset/difficulty of the task – the max. tree depth is a hyperparameter to be tuned.

Conceptually, the main idea behind gradient boosting can be summarized via the following three steps:

1. Construct a base tree (just the root node)
2. Build next tree based on errors of the previous tree
3. Combine tree from step 1 with trees from step 2. Go to step 2.

For the sake of simplicity, the following paragraphs will illustrate gradient boosting using a regression (not classification) example. At the end of this section, we will briefly look at how we can apply gradient boosting to decision tree classifiers – we won't go into too much detail, about classification specifics here, because we haven't covered (log-)likelihood maximization and logistic regression, yet.

Suppose you have the following dataset (Table 1):

x_1# Rooms	x_2=City	x_3=Age	y=Price (in million US Dollars)
5	Boston	30	1.5
10	Madison	20	0.5
6	Lansing	20	0.25
5	Waunakee	10	0.1

Table 1: Example dataset for illustrating gradient boosting.

Step 1. The first step is to construct the root node. As we remember from the decision tree lecture, making a prediction based on all training examples at a given node in a decision tree

[5] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine". In: *Annals of statistics* (2001), pp. 1189–1232.

[6] Many efficient implementations have been developed in recent years, including XGBoost and LightGBM.

regressor is basically just computing the average target value. Hence, step 1 is computing the prediction

$$\hat{y_1} = \frac{1}{n}\sum_{i=1}^{n} y^{(i)} = 0.5875$$

given the dataset in Table 1.

Step 2. For the second step, we first compute the so-called *pseudo residuals*. These pseudo residuals are basically the difference between the true target value and the predicted target value. For the regression case, the residual based on the output from step 1 is simply

$$r_1 = y_1 - \hat{y_1}.$$

Note that we use the term "pseudo" residual, because there are different ways to compute these differences and the term "residual" has a specific meaning in regression analysis.

Now, after computing the residuals for all training examples, we can add a new column to our dataset, which we refer to as r_1. This is shown in Table 2.

x_1# Rooms	x_2=City	x_3=Age	y=Price	r_1=Res
5	Boston	30	1.5	1.5 - 0.5875 = 0.9125
10	Madison	20	0.5	0.5 - 0.5875 = -0.0875
6	Lansing	20	0.25	0.25 - 0.5875 = -0.3375
5	Waunakee	10	0.1	0.1 - 0.5875 = -0.4875

Table 2: Example dataset for illustrating gradient boosting, including the residuals after Step 1.

The next part of step 2 is to fit a *new* decision tree to the residuals[7]. Note that we usually limit the depth of this decision tree (e.g., setting a max depth, which is a hyperparameter). An example decision tree fit to these residuals may look like as shown in Figure 13.

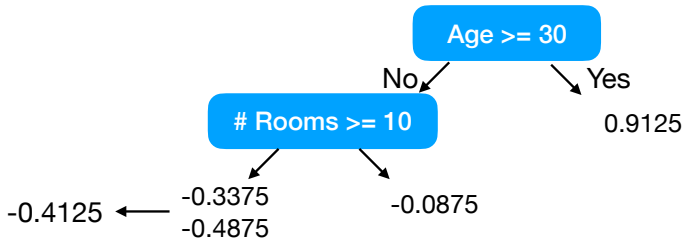

Figure 13: A decision tree fit to the residuals r_1. Note that in the left-most node, the residuals are averaged as it is typical in regression trees where we have multiple training examples at a given node.

Step 3. In this third step, we combine the tree from step 1 (the root node) and the tree from step 2. For example, if we were to make a prediction for "Lansing" based on the dataset in Table 1, the prediction would be

$$\hat{y}_{\text{Lansing}} = 0.5875 + \alpha \times (-0.4125).$$

[7]Basically, the residuals r_1 are now our target labels – instead of fitting a decision tree to class labels y as in AdaBoost, we fit the decision tree to predict r_1.

Here, 0.5875 is the value predicted by the root node from step 1. And −0.4125 is the residual (r_1 value) from step 2. The coefficient α is a learning rate or step size parameter in the range [0, 1] – it helps to not add the full residual but a rescaled version (a smaller "step"). A typical value for α would be 0.01 or 0.1, but this is also a hyperparameter that needs to be tuned manually. Empirically, choosing an *alpha* value that is too large will likely result in a gradient boosting model with high variance (a model that tends to overfit).

After step 3, we would now go back to step 2 and repeat the procedure (step 2 and step 3) T times. In each round, we take the predictions from step 3 and compute a new set of residuals for fitting the next tree. For example, for the "Lansing" example above with $alpha = 0.1$, the new residual r_2 would be

$$r_{2,\text{Lansing}} = y_{\text{Lansing}} - \hat{y}_{\text{Lansing}} = 0.25 - 0.5875 + 0.1 \times (-0.4125) = -0.37875.$$

We would do this for all training examples and add a new column r_2 to Table 2. Then, we would fit a tree on these residuals, and so forth.

The general algorithm is summarized below.

Algorithm 1 Gradient Boosting

1: Initialize T: the number of trees for gradient boosting rounds
2: Initialize $\mathcal{D}$: the training dataset, $\{\langle \mathbf{x}^{(i)}, y^{(i)} \rangle\}_{i=1}^{n}$
3: Choose $L\big(y^{(i)}, h(\mathbf{x}^{(i)})\big)$, a differentiable loss function
4: **Step 1**: Initialize model $h_0(\mathbf{x}) = \underset{\hat{y}}{\text{argmin}} \sum_{i=1}^{n} L\big(y^{(i)}, \hat{y}\big)$ [root node]
5: **Step 2**:
6: **for** t=1 to T **do**
7: **A.** Compute pseudo residual $r_{i,t} = -\left[\frac{\partial L(y^{(i)}, h(\mathbf{x}^{(i)}))}{\partial h(\mathbf{x}^{(i)})}\right]_{h(\mathbf{x})=h_{t-1}(\mathbf{x})}$, for $i = 1$ to n
8: **B.** Fit tree to r_{i_t} values, and create terminal nodes $R_{j,t}$ for $j = 1, ..., J_t$.
9: **C.**
10: **for** j=1 to J_t **do**
11: $\hat{y}_{j,t} = \underset{\hat{y}}{\text{argmin}} \sum_{\mathbf{x}^{(i)} \in R_{i,j}} L\big(y^{(i)}, h_{t-1}(\mathbf{x}^{(i)}) + \hat{y}\big)$
12: **D.** Update $h_t(\mathbf{x}) = h_{t-1}(\mathbf{x}) + \alpha \sum_{j=1}^{J_t} \hat{y}_{j,t}\, \mathbb{I}\big(\mathbf{x} \in R_{j,t}\big)$
13: **Step 3**: Return $h_t(\mathbf{x})$

Here, $r_{i,t}$ is the pseudo residual of the t-th tree and i-th example.

In a regression case, the differentiable loss function in Line 3 could be the squared error:

$$SSE' = \frac{1}{2}\big(y^{(i)} - h(\mathbf{x}^{(i)})\big)^2.$$

(We add a $\frac{1}{2}$ scaling factor because it cancels nicely in the partial derivative). Via the chain rule, we can differentiate it as follows, with respect to the prediction:

$$\frac{\partial}{\partial h(\mathbf{x}^{(i)})} \frac{1}{2}\big(y^{(i)} - h(\mathbf{x}^{(i)})\big)^2 = 2 \times \frac{1}{2}\big(y^{(i)} - h(\mathbf{x}^{(i)})\big) \times (0 - 1) \tag{14}$$

$$= -\big(y^{(i)} - h(\mathbf{x}^{(i)})\big). \tag{15}$$

Here, the expression $-\big(y^{(i)} - h(\mathbf{x}^{(i)})\big)$ is the negative residual.

Note that for the final prediction, we combine all T trees:

$$h_0(\mathbf{x}) + \alpha\ \hat{y}_{j,t=1} + ... + \alpha\ \hat{y}_{j,T}.$$

We can apply a similar concept for fitting a gradient boosting model for classification. The main algorithm is the same, except that we use a different loss function (minimizing the negative log-likelihood). Also, there are minor details for scaling the "pseudo residuals." However, since we haven't talked about logistic regression and negative log-likelihood optimization, this is out-of-scope at this point.

7.7 Random Forests

7.7.1 Overview

- Random forests are among the most widely used machine learning algorithm, probably due to their relatively good performance "out of the box" and ease of use (not much tuning required to get good results).
- In the context of bagging, random forests are relatively easy to understand conceptually: the random forest algorithm can be understood as bagging with decision trees, but instead of growing the decision trees by basing the splitting criterion on the complete feature set, we use random feature subsets.
- To summarize, in random forests, we fit decision trees on different bootstrap samples, and in addition, for each decision tree, we select a random subset of features at each node to decide upon the optimal split; while the size of the feature subset to consider at each node is a hyperparameter that we can tune, a "rule-of-thumb" suggestion is to use NumFeatures $= \log_2 m + 1$.

7.7.2 Does random forest select a subset of features for every tree or every node?

Earlier random decision forests by Tin Kam Ho[8] used the "random subspace method," where each tree got a random subset of features.

> "The essence of the method is to build multiple trees in randomly selected subspaces of the feature space." – Tin Kam Ho

However, a few years later, Leo Breiman described the procedure of selecting different subsets of features for each node (while a tree was given the full set of features) — Leo Breiman's formulation has become the "trademark" random forest algorithm that we typically refer to these days when we speak of "random forest[9]:"

> "... random forest with random features is formed by selecting at random, at each node, a small group of input variables to split on."

7.7.3 Generalization Error

- The reason why random forests may work better in practice than a regular bagging model, for example, may be explained by the additional randomization that further diversifies the individual trees (i.e., decorrelates them).

[8] Tin Kam Ho. "Random decision forests". In: *Document analysis and recognition, 1995., proceedings of the third international conference on.* Vol. 1. IEEE. 1995, pp. 278–282.

[9] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

- In Breiman's random forest paper, the upper bound of the generalization error is given as

$$\text{PE} \leq \frac{\bar{\rho} \cdot (1 - s^2)}{s^2}, \tag{16}$$

where $\bar{\rho}$ is the average correlation among trees and s measures the strength of the trees as classifiers. I.e., the average predictive performance concerning the classifiers' margin. We do not need to get into the details of how $\bar{\rho}$ and s are calculated to get an intuition for their relationship. I.e., the lower correlation, the lower the error. Similarly, the higher the strength of the ensemble or trees, the lower the error. So, randomization of the feature subspaces may decrease the "strength" of the individual trees, but at the same time, it reduces the correlation of the trees. Then, compared to bagging, random forests may be at the sweet spot where the correlation and strength decreases result in a better net result.

7.7.4 Feature Importance via Random Forests

While random forests are naturally less interpretable than individual decision trees, where we can trace a decision via a rule sets, it is possible (and common) to compute the so-called "feature importance" of the inputs – that means, we can infer how important a feature is for the overall prediction. However, this is a topic that will be discussed later in the "Feature Selection" lecture.

7.7.5 Extremely Randomized Trees (ExtraTrees)

- A few years after random forests were developed, an even "more random" procedure was developed called *Extremely Randomized Trees*[10].
- Compared to regular random forests, the ExtraTrees algorithm selects a random feature at each decision tree nodes for splitting; hence, it is very fast because there is no information gain computation and feature comparison step.
- Intuitively, one might say that ExtraTrees have another "random component" (compared to random forests) to further reduce the correlation among trees – however, it might decrease the strength of the individual trees (if you think back of the generalization error bound discussed in the previous section on random forests).

7.8 Stacking

7.8.1 Overview

- Stacking[11] is a special case of ensembling where we combine an ensemble of models through a so-called meta-classifier.
- In general, in stacking, we have "base learners" that learn from the initial training set, and the resulting models then make predictions that serve as input features to a "meta-learner."

[10] **geurts2006extremel**.
[11] **wolpert1992stacked**.

7.8.2 Naive Stacking

Algorithm 1 "Naive" Stacking

Input: Training set $\mathcal{D} = \{\langle \mathbf{x}^{[1]}, \mathbf{y}^{[1]} \rangle, ..., \langle \mathbf{x}^{[n]}, \mathbf{y}^{[n]} \rangle\}$
Output: Ensemble classifier h_E

Step 1: Learn base-classifiers
for $t \leftarrow 1$ to T **do**
    Fit base model h_t on $\mathcal{D}$
Step 2: construct new dataset $\mathcal{D}'$ from $\mathcal{D}$
for $i \leftarrow 1$ to n **do**
    add $\langle \mathbf{x'}^{[i]}, \mathbf{y}^{[i]} \rangle$ to new dataset, where $\mathbf{x'}^{[i]} = \{h_1(\mathbf{x}^{[i]}), ..., h_T(\mathbf{x}^{[i]})\}$
Step 3: learn meta-classifier h_E
return $h_E(\mathcal{D}')$

Figure 14: Outline of the naive stacking algorithm.

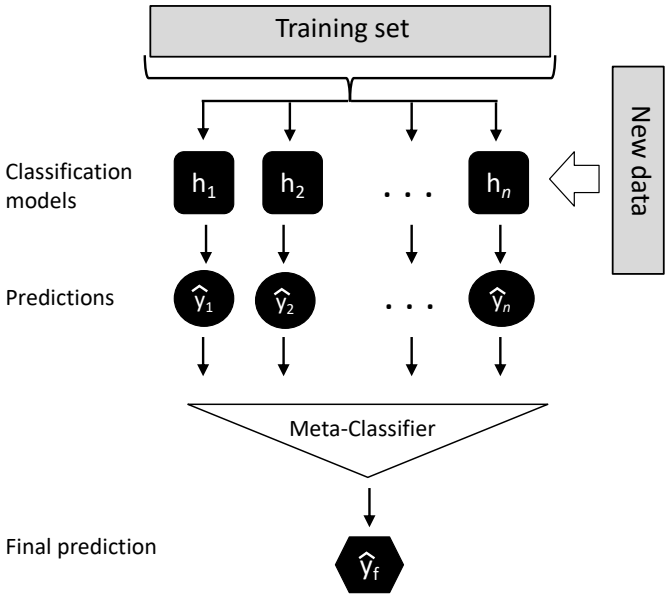

Figure 15: This figure illustrates basic concept of stacking, which is relatively similar to the voting classifier at the beginning of this lecture. Note that here, in contrast to majority voting, we have a meta-classifier that takes the predictions of the models produced by the base learners ($h_1 ... h_n$) as inputs.

The problem with the naive stacking algorithm outlined in Fig. 14 and Fig. 15 is that it has a high tendency to suffer from extensive overfitting. The reason for a potentially high degree of overfitting is that if the base learners overfit, then the meta-classifier heavily relies on these predictions made by the base-classifiers. A better alternative would be to use stacking with k-fold cross-validation or leave-one-out cross-validation.

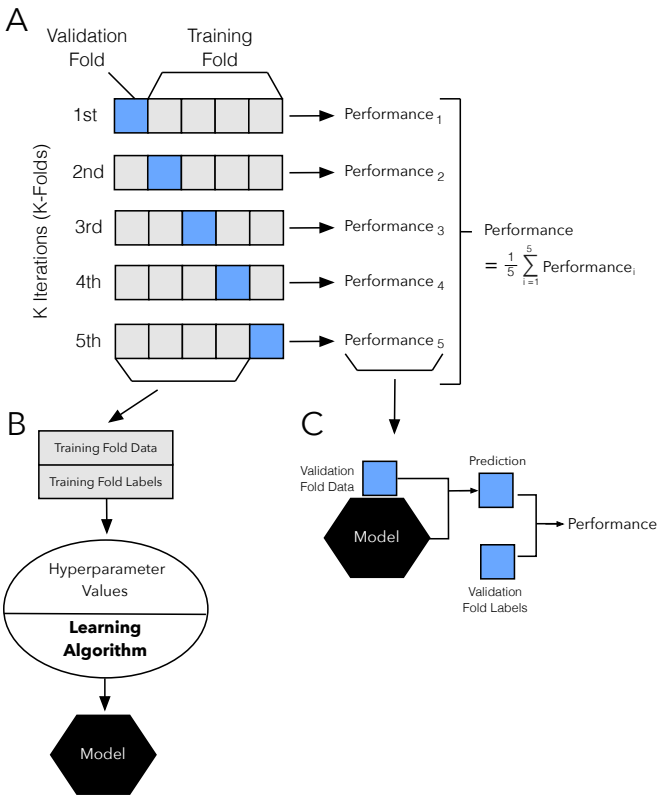

Figure 16: Illustration of k-fold cross-validation

Figure 17: Illustration of leave-one-out cross-validation, which is a special case of k-fold cross-validation, where $k = n$ (where n is the number of examples in the training set).

7.8.3 Stacking with Cross-Validation

- The use of cross-validation (or leave-one-out cross-validation) is highly recommended for performing stacking, to avoid overfitting.

Algorithm 1 Stacking with cross-validation

Input: Training set $\mathcal{D} = \{\langle \mathbf{x}^{[1]}, \mathbf{y}^{[1]} \rangle, ..., \langle \mathbf{x}^{[n]}, \mathbf{y}^{[n]} \rangle\}$
Output: Ensemble classifier h_E

Step 1: Learn base-classifiers
Construct new dataset $\mathcal{D}' = \{\}$
Randomly split $\mathcal{D}$ into k equal-size subsets: $\mathcal{D} = \{\mathcal{D}_1, ..., \mathcal{D}_k\}$
for $j \leftarrow 1$ to k **do**
    for $t \leftarrow 1$ to T **do**
        Fit base model h_t on $\mathcal{D} \setminus \mathcal{D}_k$
    for $i \leftarrow 1$ to $n \in |\mathcal{D} \setminus \mathcal{D}_k|$ **do**
        Add $\langle \mathbf{x'}^{[i]}, \mathbf{y}^{[i]} \rangle$ to new dataset $\mathcal{D}'$, where $\mathbf{x'}^{[i]} = \{h_1(\mathbf{x}^{[i]}), ..., h_T(\mathbf{x}^{[i]})\}$
Step 3: learn meta-classifier h_E
return $h_E(\mathcal{D}')$

Figure 18: Stacking with k-fold cross-validation

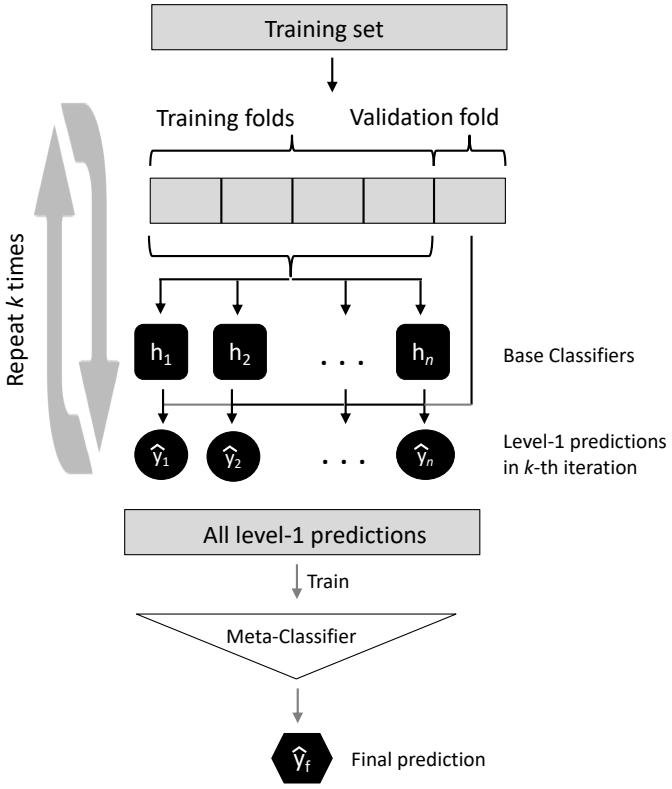

Figure 19: Illustration of stacking with cross-validation.

7.9 Resources

7.9.1 Assigned Reading

- Python Machine Learning, 2nd Ed., Chapter 7

7.9.2 Further Reading

Listed below are optional reading materials for students interested in more in-depth coverage of the ensemble methods we discussed (not required for homework or the exam).

- Breiman, L. (1996). Bagging predictors. *Machine learning*, *24*(2), 123-140.
- Wolpert, D. H. (1992). Stacked generalization. *Neural networks*, *5*(2), 241-259.
- Breiman, L. (2001). Random forests. *Machine learning*, *45*(1), 5-32.
- Freund, Y., Schapire, R., & Abe, N. (1999). A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, *14*(771-780), 1612.
- Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 1189-1232.
- Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM sigkdd international conference on knowledge discovery and data mining* (pp. 785-794). ACM.