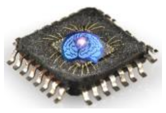
EECE-6822 Machine Learning



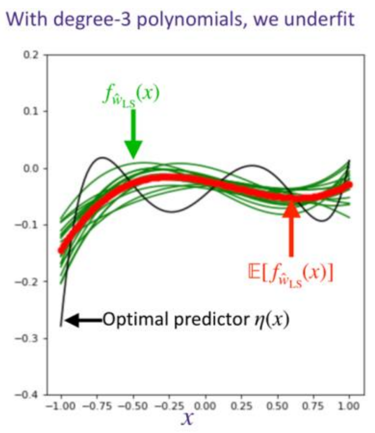
Combining Learners

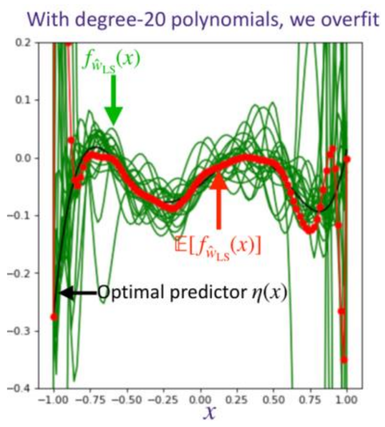
Cris Ababei
Dept. of Electrical and Computer Engineering
MARQUETTE UNIVERSITY
BE THE DIFFERENCE.

PART 1
Combining Learners

Recall Bias-Variance Tradeoff

- Choice of hypothesis class introduces learning bias
  - More complex class → less bias, more variance





3

3

Fighting the bias-variance tradeoff

- **Simple (a.k.a. weak) learners**
  - e.g., logistic regression, naive Bayes, …
  - **Good:** Low variance, do not usually overfit
  - **Bad:** High bias, cannot solve hard learning problems
- **Sophisticated learners**
  - Kernel SVMs, Deep Neural Nets, …
  - **Good:** Low bias, have the potential to learn with Big Data
  - **Bad:** High variance, difficult to generalize
- Can we combine these properties?
  - **In general, no**
  - **But often, yes**

4

4

Ensembles are powerful

- For binary classification, if you have ***N*** weak classifiers, but each one is slightly better than random chance (gets right answer with $p > 0.5$), what happens if you take the majority vote?
- As ***N→∞*** what is the probability that the **majority vote** gets the right answer?

- Marquis de Condorcet, "Essay on the Application of Analysis to the Probability of Majority Decisions" (1785). Known as the Condorcet Jury Theorem
  - https://en.wikipedia.org/wiki/Condorcet's_jury_theorem
- Schapire, "The Strength of Weak Learnability" (1990)
  - https://link.springer.com/article/10.1007/BF00116037

Voting Classifiers

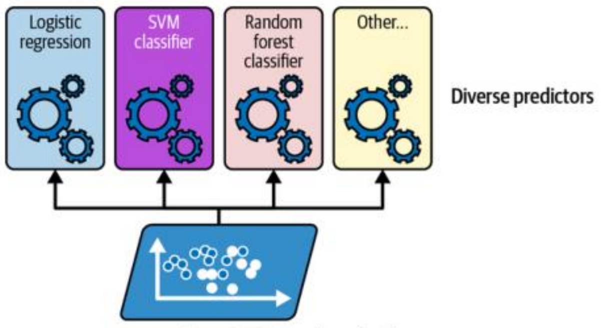


Figure 7-1. Training diverse classifiers

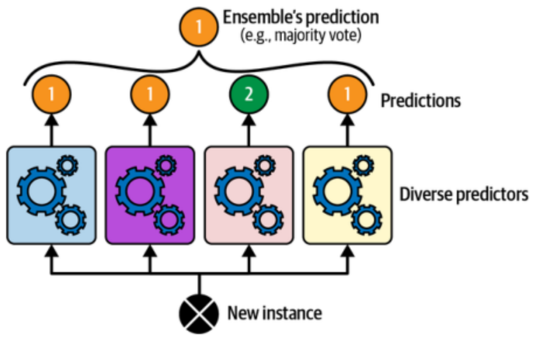


Figure 7-2. Hard voting classifier predictions

Somewhat surprisingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a *weak learner* (meaning it does only slightly better than random guessing), the ensemble can still be a *strong learner* (achieving high accuracy), provided there are a sufficient number of weak learners in the ensemble and they are sufficiently diverse.

[***B3-Geron**] Aurelien Geron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, O'Reilly, 2022.

How is this possible? The following analogy can help shed some light on this mystery. Suppose you have a slightly biased coin that has a 51% chance of coming up heads and 49% chance of coming up tails. If you toss it 1,000 times, you will generally get more or less 510 heads and 490 tails, and hence a majority of heads. If you do the math, you will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%. The more you toss the coin, the higher the probability (e.g., with 10,000 tosses, the probability climbs over 97%). This is due to the *law of large numbers*: as you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%). Figure 7-3 shows 10 series of biased coin tosses. You can see that as the number of tosses increases, the ratio of heads approaches 51%. Eventually all 10 series end up so close to 51% that they are consistently above 50%.

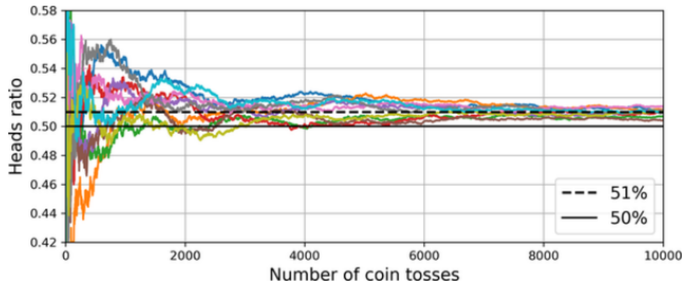


Figure 7-3. The law of large numbers

Similarly, suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing). If you predict the majority voted class, you can hope for up to 75% accuracy! However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case because they are trained on the same data. They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.

Scikit-Learn - `VotingClassifier` class

```
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

voting_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)
```

```
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```

Voting (Ensemble Methods)

- Instead of learning a single classifier, learn **many classifiers**
- **Output class:** (Weighted) vote of each classifier
  - Classifiers that are most "sure" will vote with more conviction
- With sophisticated learners
  - Uncorrelated errors → expected error goes down
  - On average, do better than single classifier!

  (1) Bagging = bootstrap averaging
- With weak learners
  - Each one good at different parts of the input space
  - On average, do better than single classifier!

  (2) Boosting

(1) Bagging and Pasting

One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed. Another approach is to use the same training algorithm for every predictor but train them on different random subsets of the training set. When sampling is performed *with* replacement,[1] this method is called *bagging*[2] (short for *bootstrap aggregating*[3]). When sampling is performed *without* replacement, it is called *pasting*.[4]

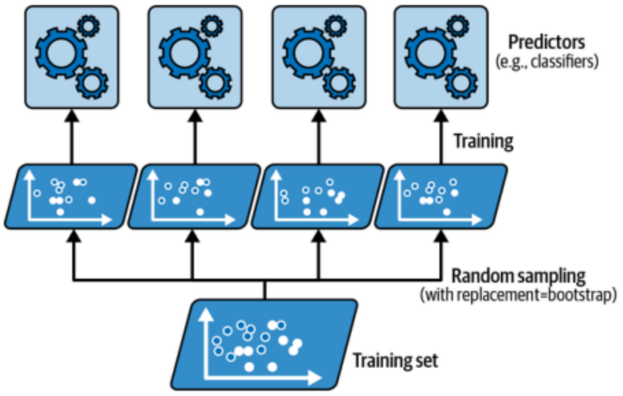


Figure 7-4. Bagging and pasting involve training several predictors on different random samples of the training set

Bagging in a nutshell

To provide a more concrete example of how the bootstrap aggregating of a bagging classifier works, let's consider the example shown in *Figure 7.7*. Here, we have seven different training instances (denoted as indices 1-7) that are sampled randomly with replacement in each round of bagging. Each bootstrap sample is then used to fit a classifier, C_j, which is most typically an unpruned decision tree:

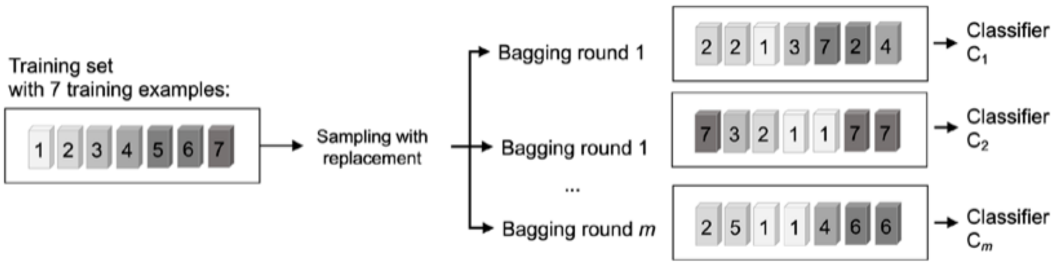


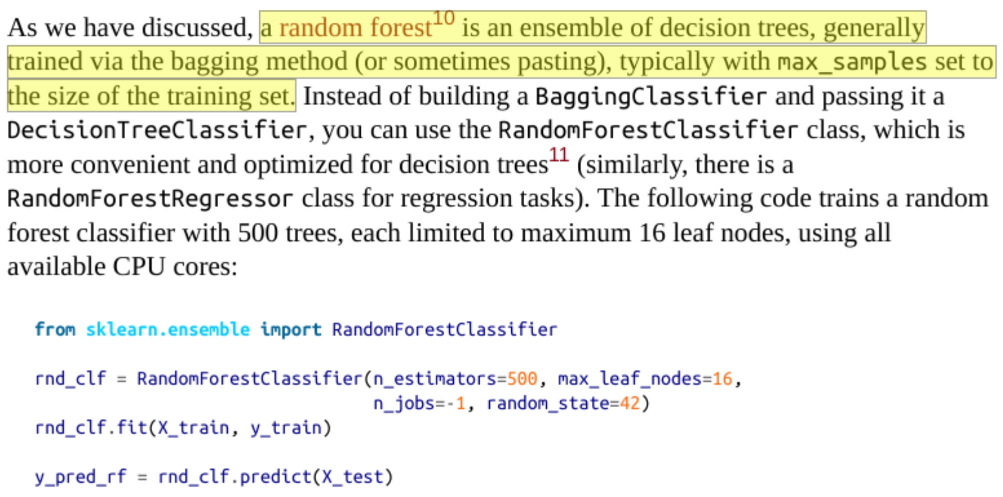
Figure 7.7: An example of bagging

[***B3-Raschka**] Sebastian Raschka, Yuxi Liu, and Vahid Mirjalili, Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python, Packt Publishing, 2022.

Bagging and Pasting

- Each individual predictor has a higher bias than if it were trained on the original training set, but, aggregation reduces both bias and variance.
- Generally, the net result is that the ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.

Random Forests See Appendix A



As we have discussed, a random forest[10] is an ensemble of decision trees, generally trained via the bagging method (or sometimes pasting), typically with `max_samples` set to the size of the training set. Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can use the `RandomForestClassifier` class, which is more convenient and optimized for decision trees[11] (similarly, there is a `RandomForestRegressor` class for regression tasks). The following code trains a random forest classifier with 500 trees, each limited to maximum 16 leaf nodes, using all available CPU cores:

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                 n_jobs=-1, random_state=42)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

Feature Importance

Yet another great quality of random forests is that they make it easy to measure the relative importance of each feature. Scikit-Learn measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average, across all trees in the forest. More precisely, it is a weighted average, where each node's weight is equal to the number of training samples that are associated with it (see Chapter 6).

Similarly, if you train a random forest classifier on the MNIST dataset (introduced in Chapter 3) and plot each pixel's importance, you get the image represented in Figure 7-6.

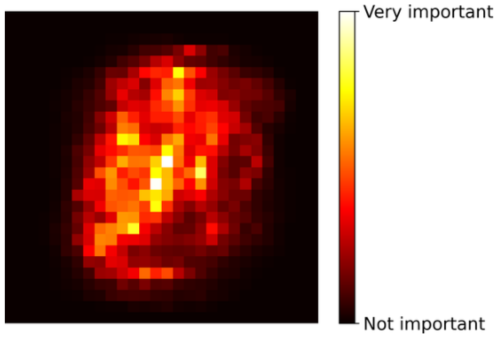


Figure 7-6. MNIST pixel importance (according to a random forest classifier)

(2) **Boosting** - a brief history

- 1988 Kearns and Valiant: "Can weak learners be combined to create a strong learner?"
- 1990 Schapire: "Yup, in theory"
- 1995 Schapire and Freund: "Practical for 0/1 loss" -> AdaBoost
- 2001 Friedman: "Practical for arbitrary losses"
- 2014 Tianqi Chen: "Scale it up!" -> XGBoost

- https://en.wikipedia.org/wiki/Boosting_(machine_learning)

Boosting and additive models

Instead of ensembling bootstrapped models, can we:
- Keep the idea of ensembling/combining simpler models, but
- Not necessarily have the models be identically distributed?

Key idea:
- Given a current collection of models, add a new model that focuses on what the previous models got wrong

Boosting (originally called *hypothesis boosting*) refers to any ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. There are many boosting methods available, but by far the most popular are *AdaBoost*[13] (short for *adaptive boosting*) and *gradient boosting*. Let's start with AdaBoost.

17

17

(A) AdaBoost

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost.

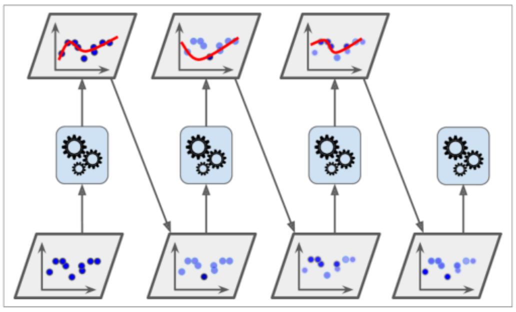


Figure 7-7. AdaBoost sequential training with instance weight updates

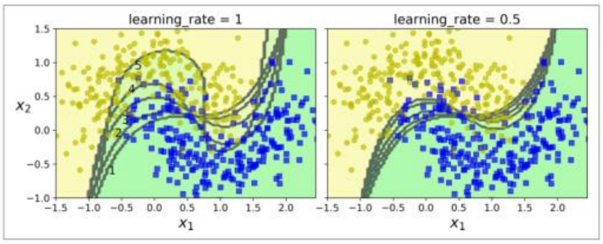


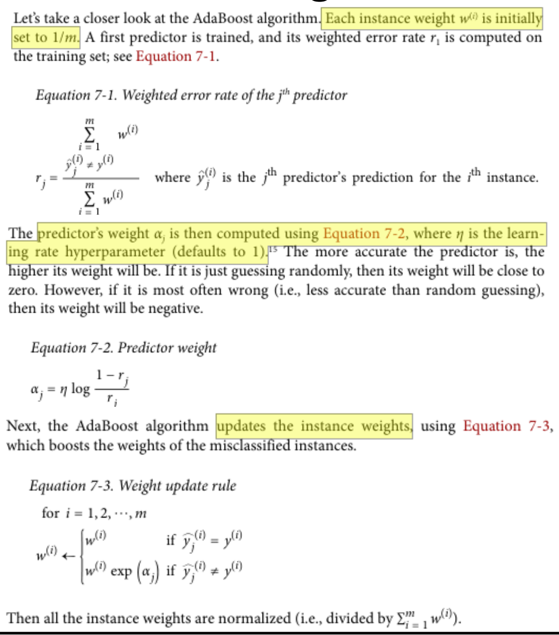
Figure 7-8. Decision boundaries of consecutive predictors

Here, each predictor is a highly regularized SVM classifier with an RBF kernel

18

18

AdaBoost Algorithm


Let's take a closer look at the AdaBoost algorithm. Each instance weight $w^{(i)}$ is initially set to $1/m$. A first predictor is trained, and its weighted error rate r_1 is computed on the training set; see Equation 7-1.

Equation 7-1. Weighted error rate of the jth predictor

$$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}} \quad \text{where } \hat{y}_j^{(i)} \text{ is the } j^{\text{th}} \text{ predictor's prediction for the } i^{\text{th}} \text{ instance.}$$

The predictor's weight α_j is then computed using Equation 7-2, where η is the learning rate hyperparameter (defaults to 1).[15] The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be close to zero. However, if it is most often wrong (i.e., less accurate than random guessing), then its weight will be negative.

Equation 7-2. Predictor weight

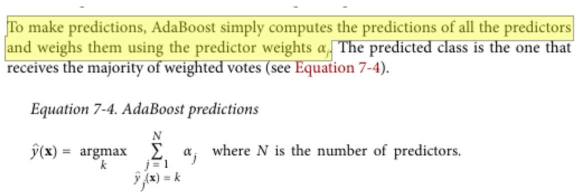
$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Next, the AdaBoost algorithm updates the instance weights, using Equation 7-3, which boosts the weights of the misclassified instances.

Equation 7-3. Weight update rule

$$\text{for } i = 1, 2, \cdots, m$$
$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp\left(\alpha_j\right) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

Then all the instance weights are normalized (i.e., divided by $\sum_{i=1}^{m} w^{(i)}$).


To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights α_j. The predicted class is the one that receives the majority of weighted votes (see Equation 7-4).

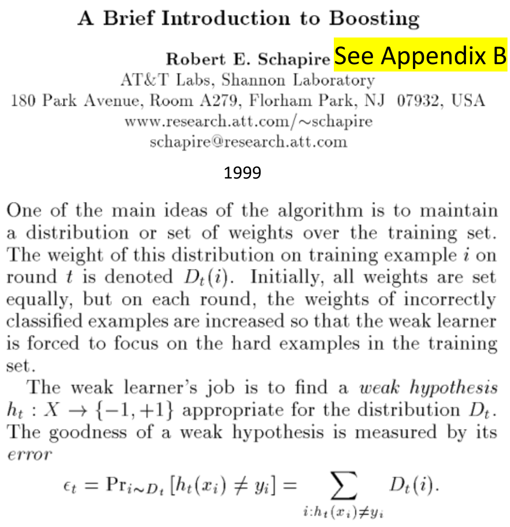
Equation 7-4. AdaBoost predictions

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x}) = k}}^{N} \alpha_j \quad \text{where } N \text{ is the number of predictors.}$$

19

19

AdaBoost **Pseudocode**:


A Brief Introduction to Boosting

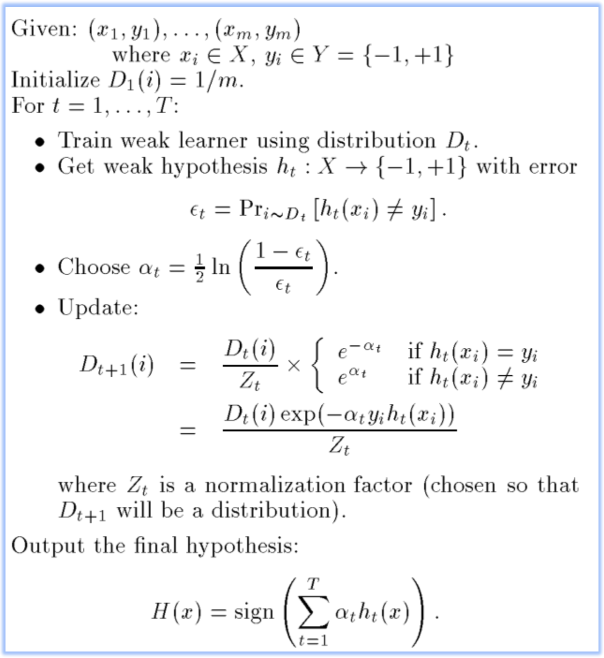
Robert E. Schapire See Appendix B
AT&T Labs, Shannon Laboratory
180 Park Avenue, Room A279, Florham Park, NJ 07932, USA
www.research.att.com/~schapire
schapire@research.att.com

1999

One of the main ideas of the algorithm is to maintain a distribution or set of weights over the training set. The weight of this distribution on training example i on round t is denoted $D_t(i)$. Initially, all weights are set equally, but on each round, the weights of incorrectly classified examples are increased so that the weak learner is forced to focus on the hard examples in the training set.

The weak learner's job is to find a *weak hypothesis* $h_t : X \to \{-1, +1\}$ appropriate for the distribution D_t. The goodness of a weak hypothesis is measured by its *error*

$$\epsilon_t = \Pr_{i \sim D_t}\left[h_t(x_i) \neq y_i\right] = \sum_{i : h_t(x_i) \neq y_i} D_t(i).$$


```
Given: $(x_1, y_1), \ldots, (x_m, y_m)$
        where $x_i \in X$, $y_i \in Y = \{-1, +1\}$
Initialize $D_1(i) = 1/m$.
For $t = 1, \ldots, T$:
  • Train weak learner using distribution $D_t$.
  • Get weak hypothesis $h_t : X \to \{-1, +1\}$ with error
        $\epsilon_t = \Pr_{i \sim D_t}\left[h_t(x_i) \neq y_i\right]$.
  • Choose $\alpha_t = \frac{1}{2} \ln\left(\frac{1 - \epsilon_t}{\epsilon_t}\right)$.
  • Update:
        $D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\ e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i \end{cases}$
                    $= \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$
    where $Z_t$ is a normalization factor (chosen so that
    $D_{t+1}$ will be a distribution).
Output the final hypothesis:
        $H(x) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$.
```

20

(B) Gradient Boosting

Another very popular boosting algorithm is *gradient boosting*.[17] Just like AdaBoost, gradient boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Next, we'll train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

Then we train a third regressor on the residual errors made by the second predictor:

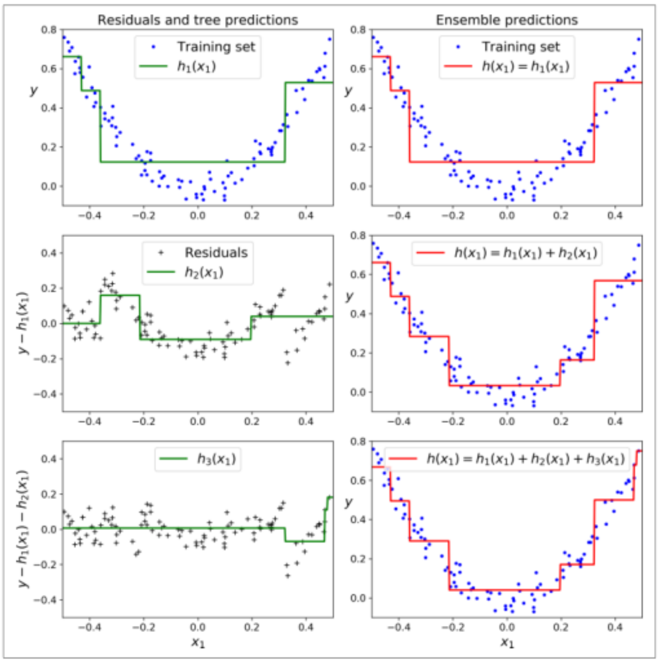
```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

21

21



In this depiction of **Gradient Boosting**, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

22

22

Takeaways

- Single trees: low bias, high variance
- Ensembles: low bias, (relatively) low variance
- **Bagging** averages many lightly dependent models to reduce variance
- Random forests: same but with random subset of features
- **Boosting** learns a linear combination of high bias, highly dependent classifiers to reduce error

23

23

PART 2
Code Time!

24

24

Code Time

- See demonstration and discussion in class.
- See also links in the “Code Examples” for this lecture assignment.

25

25

Conclusion

Takeaways

26

26

- **Ensemble methods** combine different classification models to cancel out their individual weaknesses, which often results in stable and well-performing models that are very attractive for industrial applications as well as machine learning competitions.
- **AdaBoost** Summary:

Concept	Expression
Hypothesis space	$\mathcal{H} = \{h_1, \ldots, h_M\}$
Weak learner	returns $h_t(x) \in \{-1, +1\}$ with error $\epsilon_t < 0.5$
Distribution update	$D_{t+1}(i) \propto D_t(i) e^{-\alpha_t y_i h_t(x_i)}$
Weight of hypothesis	$\alpha_t = \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$
Final strong classifier	$\boxed{H(x) = \text{sign}\left(\sum_{t-1}^{T} \alpha_t h_t(x)\right)}$

References and Credits

Many of the teaching materials for this course have been adapted from various sources. We are very grateful and thank the following professors, researchers, and practitioners for sharing their teaching materials (in no particular order):

- Yaser S. Abu-Mostafa, Malik Magdon-Ismail and Hsuan-Tien Lin. https://amlbook.com/slides.html
- Ethem Alpaydin. https://www.cmpe.boun.edu.tr/~ethem/i2ml3e/
- Natasha Jaques. https://courses.cs.washington.edu/courses/cse446/25sp/
- Lyle Ungar. https://alliance.seas.upenn.edu/~cis520/dynamic/2022/wiki/index.php?n=Lectures.Lectures
- Aurelien Geron. https://github.com/ageron/handson-ml3
- Sebastian Raschka. https://github.com/rasbt/machine-learning-book
- Trevor Hastie. https://www.statlearning.com/resources-python
- Andrew Ng. https://www.youtube.com/playlist?list=PLoROMvodv4rMiGQp3WXShtMGgzqpfVfbU
- Richard Povineli. https://www.richard.povinelli.org/teaching
- … and many others.

Appendix A: Decision Trees

(1)Decision Trees: Core Idea

A **decision tree** recursively partitions the feature space into regions where the target variable is approximately constant (for regression) or pure (for classification).

At each internal node, the algorithm selects:

- a **feature** j
- and a **split threshold** s

to divide the data $D = \{(x_i, y_i)\}$ into two subsets:

$$R_1(j,s) = \{x \mid x_j \le s\},$$
$$R_2(j,s) = \{x \mid x_j > s\}.$$

29

(2)Splitting Criterion

We pick (j^*, s^*) that minimizes the **impurity** after the split.

For regression trees:

Minimize **sum of squared errors** (SSE):

$$(j^*, s^*) = \arg\min_{j,s} \left[\sum_{x_i \in R_1(j,s)} (y_i - \bar{y}_{R_1})^2 + \sum_{x_i \in R_2(j,s)} (y_i - \bar{y}_{R_2})^2 \right],$$

where $\bar{y}_{R_k}$ is the mean of y_i in region R_k.

For classification trees:

Minimize **impurity measure** such as:

- **Gini impurity**: $G = \sum_k p_k(1 - p_k) = 1 - \sum_k p_k^2$
- **Entropy**: $H = -\sum_k p_k \log p_k$

and choose (j^*, s^*) that minimizes the **weighted average impurity** after the split.

30

(3)Recursive Partitioning

The process repeats recursively on each region, forming a hierarchical tree structure until:

- A stopping criterion is met (e.g. max depth, min samples per leaf), or
- The region is pure.

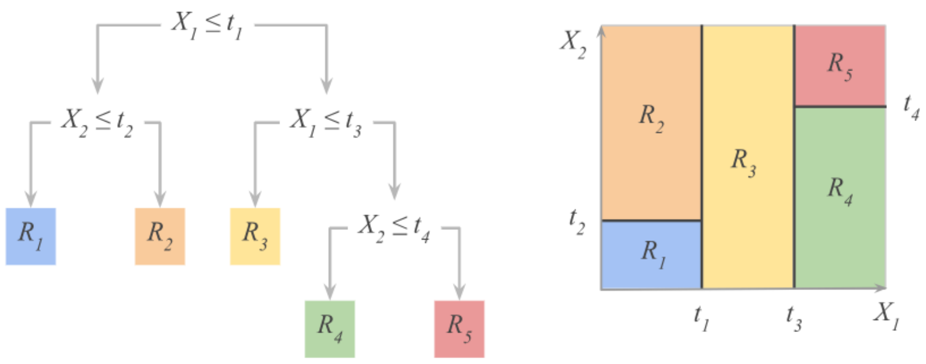
Each **leaf node** stores a prediction:

$$\hat{y}_R = \begin{cases} \text{majority class in } R, & \text{classification} \\ \text{mean of } y_i \text{ in } R, & \text{regression.} \end{cases}$$

31

31

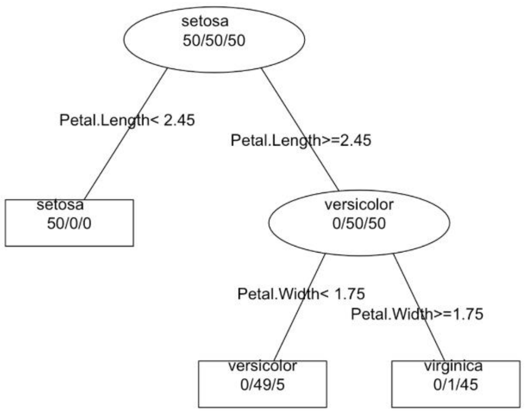
(4)Decision Tree Illustration

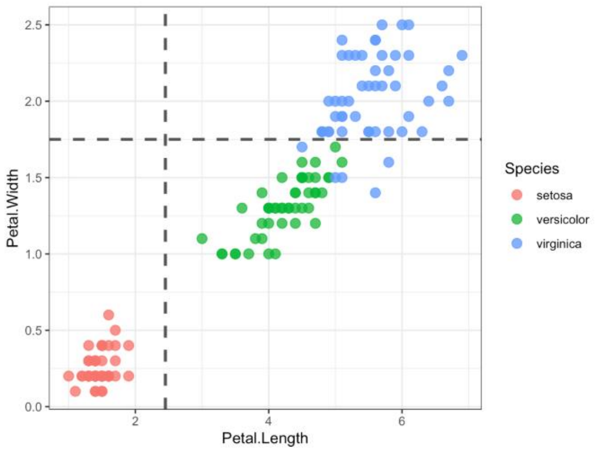


32

32

(4)Decision Tree Illustration





33

33

(5)Limitations of a Single Tree

- High variance: small changes in data → very different trees
- Prone to overfitting, especially when deep
- Piecewise-constant predictions (discontinuous boundaries)

(6)Random Forests: Ensemble of Decision Trees

A **Random Forest** overcomes overfitting by **averaging many de-correlated trees.**

Let $T_1, T_2, \ldots, T_B$ be B trees trained on bootstrap samples.

Training Procedure

1. **Bootstrap sampling**:
   For each tree b, draw a dataset D_b by sampling n points **with replacement** from training data.
2. **Random feature selection**:
   At each split, select a random subset of features (of size $m < d$) to consider for splitting.
3. **Grow tree fully** (no pruning).

34

Prediction

- **Regression**:

$$\hat{f}_{\mathrm{RF}}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$$

- **Classification**:

$$\hat{y}_{\mathrm{RF}}(x) = \arg\max_k \sum_{b=1}^{B} \mathbf{1}\left[T_b(x) = k\right]$$

Each tree has high variance, but their average reduces variance dramatically.

Variance Reduction by Averaging

If the individual trees have variance σ^2 and pairwise correlation ρ, then the ensemble variance is:

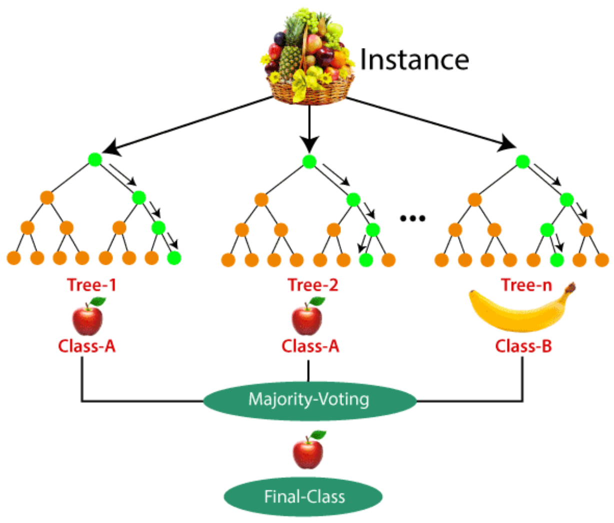
$$\mathrm{Var}_{\mathrm{ensemble}} = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2.$$

Hence, low correlation (via randomization) + many trees $B \Rightarrow$ **stable, low-variance predictor**.

35

35

Intuition Diagram



36

36

Appendix B: Boosting as introduced by Robert Schapire (1989)

(1)Setup

We begin with:

- A **hypothesis space** $\mathcal{H} = \{h_1, h_2, \ldots, h_M\}$
- A weak learning algorithm that can produce classifiers $h_t(x) \in \{-1, +1\}$ that perform only slightly better than random on weighted data.
- Training set:

$$D = \{(x_i, y_i)\}_{i=1}^{n}, \quad y_i \in \{-1, +1\}.$$

The goal: **combine several weak hypotheses** into a strong final classifier with low training error.

37

37

(2)Schapire's 3-hypothesis construction (original idea)

Schapire (1989) first proved that a weak learner (accuracy > 0.5) can be boosted.

Let the weak learner produce three hypotheses $h_1, h_2, h_3 \in \mathcal{H}$, trained on reweighted distributions:

1. h_1 trained on uniform weights.
2. h_2 trained emphasizing samples misclassified by h_1.
3. h_3 trained emphasizing points where h_1 and h_2 disagree.

Then the **final strong classifier** is:

$$H(x) = \text{sign}\big(h_1(x) + h_2(x) + h_3(x)\big).$$

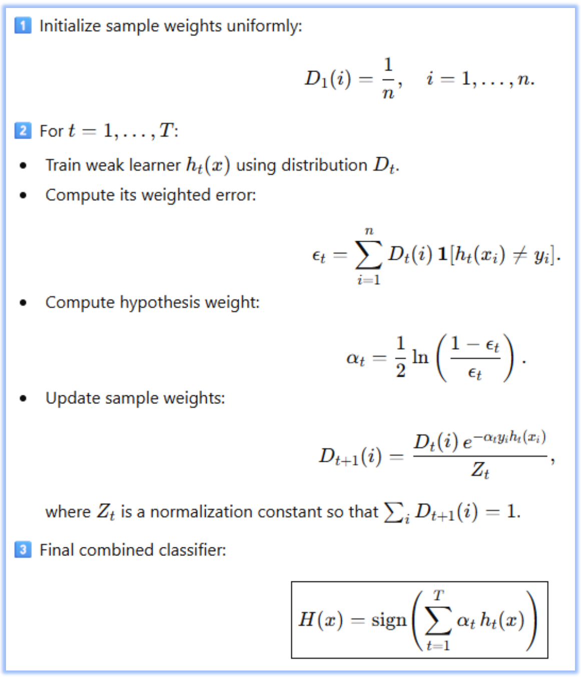
This theoretical result established the **existence** of a boosting procedure.

38

38

(3)Generalized iterative boosting (AdaBoost view)

Later (Freund & Schapire, 1995), this idea was extended into the **AdaBoost algorithm**, which iteratively reweights data and combines weak hypotheses:



1 Initialize sample weights uniformly:

$$D_1(i) = \frac{1}{n}, \quad i = 1, \dots, n.$$

2 For $t = 1, \dots, T$:

- Train weak learner $h_t(x)$ using distribution D_t.
- Compute its weighted error:

$$\epsilon_t = \sum_{i=1}^{n} D_t(i)\, \mathbf{1}[h_t(x_i) \neq y_i].$$

- Compute hypothesis weight:

$$\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right).$$

- Update sample weights:

$$D_{t+1}(i) = \frac{D_t(i)\, e^{-\alpha_t y_i h_t(x_i)}}{Z_t},$$

where Z_t is a normalization constant so that $\sum_i D_{t+1}(i) = 1$.

3 Final combined classifier:

$$\boxed{H(x) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t\, h_t(x)\right)}$$

39

(4)Objective interpretation

AdaBoost can be shown to minimize the **exponential loss**:

$$\min_{f(x)=\sum_t \alpha_t h_t(x)} \sum_{i=1}^{n} e^{-y_i f(x_i)}.$$

Each iteration greedily reduces this loss, hence *boosting* the ensemble performance.

(5)Key intuition

- Boosting **focuses** subsequent weak learners on examples the current ensemble misclassifies.
- The final strong classifier is a **weighted majority vote** over weak hypotheses:

$$H(x) = \operatorname{sign}\left(\sum_t \alpha_t h_t(x)\right).$$

- **Schapire's (1989) theorem** showed that weak learnability implies strong learnability. AdaBoost (1995) made that constructive.

40

40