NMBU

Norwegian University of Life Sciences

Ensemble learning

Chapter 7 in Python Machine Learning TE

Ensemble learning

- Simplicity of (statistical classifier) models
- DL requires huge amount of data

https://cdn-0.scatteredquotes.com/wp-content/uploads/2019/04/Avengers-Endgame-Quotes-7.jpg

What we will cover

- What is ensemble models → combining classifications

- How to ensemble models
  - → Majority voting
  - → Bagging – bootstrapping samples, single classifier
  - → Boosting – weak learners focusing on hard to classify samples
    - AdaBoost/XGBoost

- Evaluation and tuning

Resources

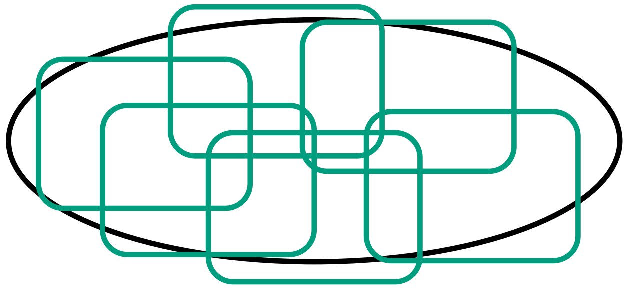
- Python Machine Learning TE, Chapter 7, pages 219-254.
- Jupyter notebook: Chapter 7, part 1

- Ensemble methods in scikit-learn:
  http://scikit-learn.org/stable/modules/ensemble.html

Ensemble Learning

- **Main idea:** Many models in combination can perform better than one -> combine individual classifiers into a meta-classifier.

- **Bagging (Bootstrap aggregating):**
  - Reduce overfit by combining different models built from different subsets of the available training data.



- **Boosting:**
  - Hierarchy of weak learners with different tasks.
  - Combine weak learners into a common, flexible, high performance model.

Analogy - dietary advice

- **Many experts & many opinions**

- **Single learner analogy:**
  - Listen to one expert
  - Large bet on single product?
  - May be disappointed

- **Ensemble learning:**
  - Listen to many experts
  - Construct consensus
  - More robust advise
  - Reduce chance of disappointment

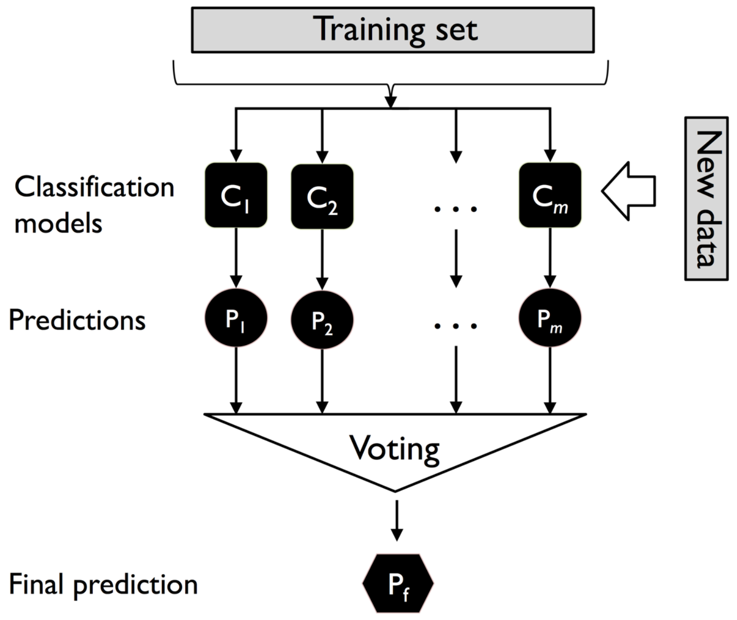
Majority Voting

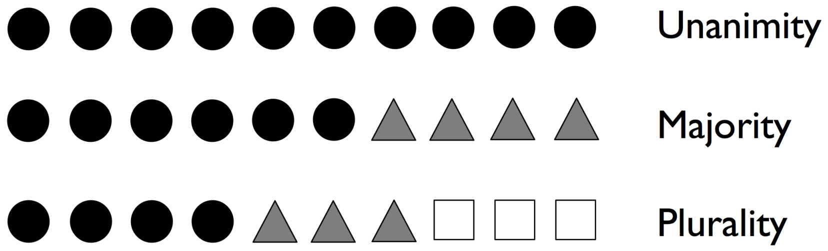
→ Majority voting (binary) simply means that we select the class label that has been predicted by the majority of classifiers. (Generalized to multiclass settings, the principle is called plurality voting.)

[p. 233-236]

Majority voting

- Train several classifiers and/or subsets and predict a sample

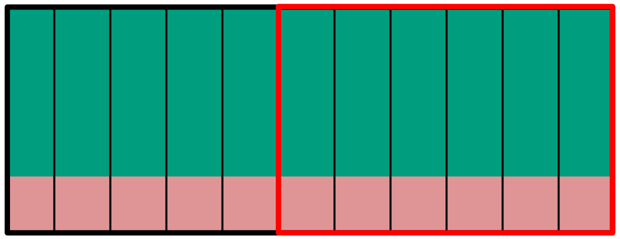




- Choose the class that received the most votes among the classifiers.

$$\hat{y} = mode\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \dots, C_m(\boldsymbol{x})\}$$

Combinatorial argument



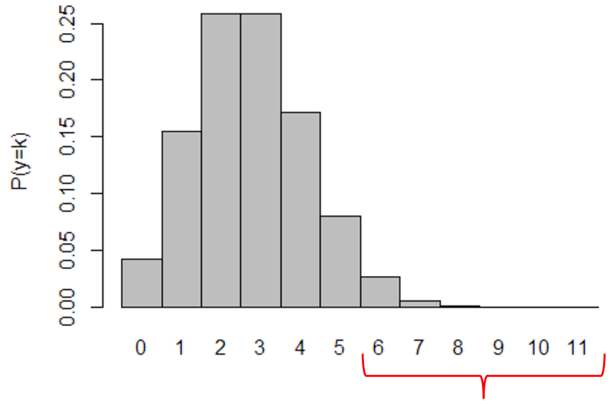
- Assume *n* independent two-class classifiers:
  - equal error rate *ε*
  - uncorrelated errors
  - *Consider the situation where more than k ≈ n/2* are wrong

- Probability of more than *k* errors the in ensemble (majority of error):

$$P(y \geq k) = \sum_k^n \binom{n}{k} \varepsilon^k (1-\varepsilon)^{n-k}$$

For *n* = 11 and *ε* = 0.25:

$$P(y \geq 6) = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1-0.25)^{n-k} = 0.034$$



- Note: In practice errors are seldom completely uncorrelated

Jupyter: Chapter 7, part 1

General majority vote

- **Including weights in the majority voting:**

$$\hat{y}(\boldsymbol{x}) = arg\max_g\{\textstyle\sum_{j=1}^{m} w_j\, \chi_A\big(C_j(\boldsymbol{x}) = g\big): g = 1, \dots, G\}$$

  - G classes (g = 1, …, G), m classifiers (j = 1, …, m), $C_j(\boldsymbol{x})$ is classification of **x**
  - Weighted (by the w_js) counting of classifier outcomes (χ_A is 1 for class hit)
  - w_js can reflect the general confidence in each classifier, similar to a prior

- Example with m = 3 classifiers, G = 2 classes:
  - w = [0.2, 0.2, 0.6], (weights of the 3 classifiers)
  - C($\boldsymbol{x}$) = [0, 0, 1] (two classifiers vote for “0”, one for “1”)

  => $\hat{y}(\boldsymbol{x}) = arg\max_g\{g = 0\!: 0.4, g = 1\!: 0.6) = 1\}$

Posterior probabilities

- Change from class membership to probability (p_{gj}) of class membership (predict => `predict_proba`)

$$\hat{y}(\boldsymbol{x}) = arg\max_{g} \sum_{j=1}^{m} w_j\, p_{gj}$$

  - (Product of general confidence and class membership probability)

- Example (same classifier weights: w = [0.2, 0.2, 0.6]):

$$C_1(\boldsymbol{x}) \rightarrow [0.9, 0.1],\ C_2(\boldsymbol{x}) \rightarrow [0.8, 0.2],\ C_3(\boldsymbol{x}) \rightarrow [0.4, 0.6]$$

$$p(g = 0 | \boldsymbol{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(g = 1 | \boldsymbol{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42$$

$$\hat{y}(\boldsymbol{x}) = arg\max_{g} [\, p(g = 0 | \boldsymbol{x}), p(g = 1 | \boldsymbol{x})] = 0$$

Implementing a majority vote classifier

- We will make an estimator in the scikit-learn style: http://scikit-learn.org/stable/developers/develop.html

- To reduce the amount of code needed, we use inheritance from BaseEstimator and ClassifierMixin which include some mandatory functions:. *get_params, set_params, score*.

- To make *estimators* compatible with pipelines we need *fit/fit_transform* and *transform* functions. As this is a *classifier*, the *transform* parts are not needed.

- Including *predict* enables prediction, *predict_proba* enables ROC AUC and similar.

Jupyter: Chapter 7, part 1

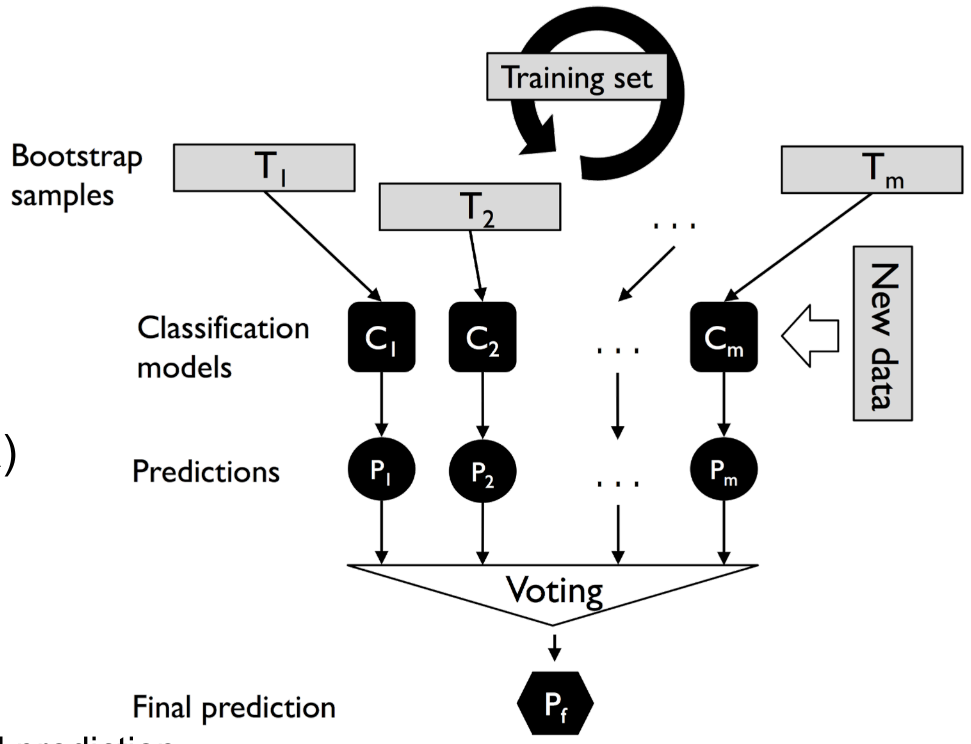
Bagging

→ Draw bootstrap samples (random samples with replacement) from the initial training dataset to fit the individual classifiers in the ‘ensemble’.

[p. 243-248]

Bagging – bootstrap aggregation

- Building an ensemble of classifiers from bootstrap samples.

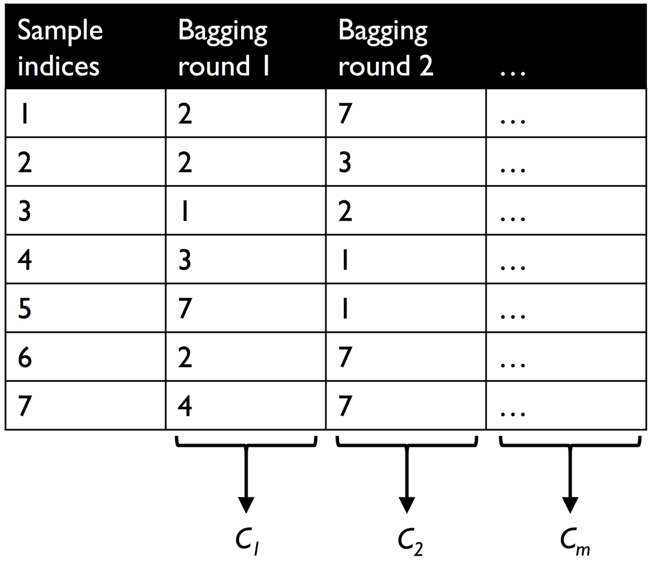


- Repeatedly sample with replacement from the training data.
  - (Replacement causes duplicate observations in the training data)
- Train the classifier on each sample and predict new data.
- Vote for final prediction.
- Note: In the models of regression, the final prediction is made by averaging the predictions of the all-base model, and that is known as bagging regression.

Bagging “in a nutshell”

- Typically we use unpruned **decision trees** as our individual classifiers
  - **Note:** Training accuracy can be forced to 100% with single sample end-nodes.

- **Strategy:**
  1. Fit a classifier for each of the *m* bootstrap samples (bootstrap aggregation).
  2. Apply all the *m* classifiers for prediction.
  3. Make the final (ensemble) prediction by Majority voting based on the *m* individual classifiers.

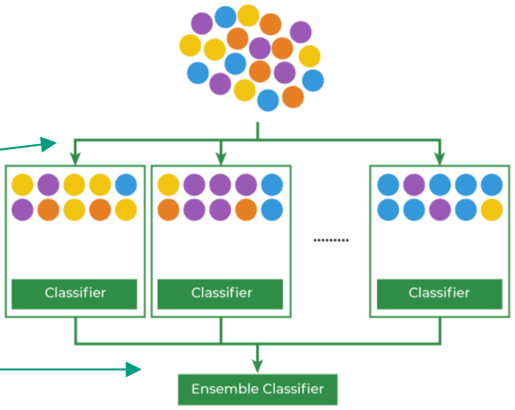


Sample indices	Bagging round 1	Bagging round 2	…
1	2	7	…
2	2	3	…
3	1	2	…
4	3	1	…
5	7	1	…
6	2	7	…
7	4	7	…

Jupyter: Chapter 7, part 2

Bagging Classifier – code example*

- Basic steps:
  1. Bootstrap sampling
  2. Base model training (individually)
  3. Aggregation/Majority voting



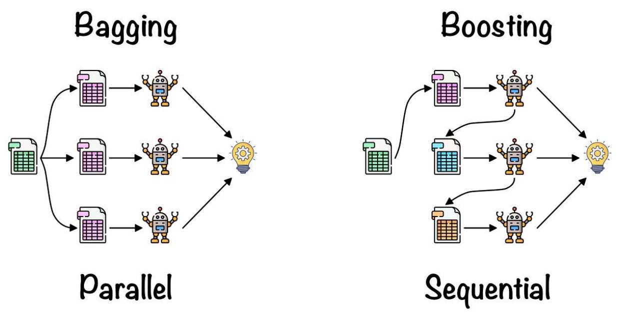
- `Classifier instance from scratch`
- `from sklearn.ensemble import BaggingClassifier`

*https://www.geeksforgeeks.org/ml-bagging-classifier/

Boosting

→ Weak learners subsequently learn from misclassified training examples to improve the performance of the ‘ensemble’.

[p. 249-257]



Boosting “in a nutshell”

- Very simple individual classifiers (weak learners), usually a decision tree stump (a tree with only one split).

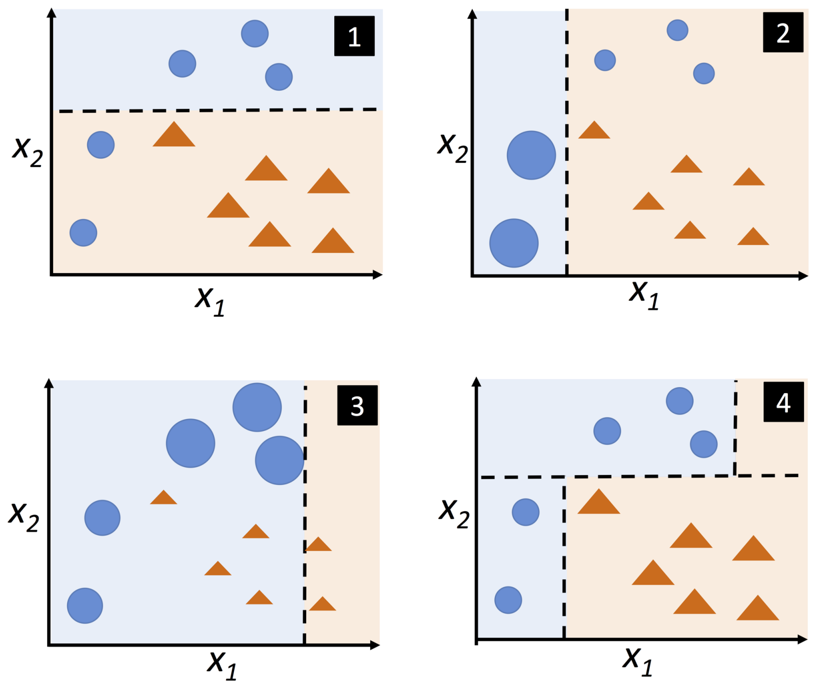
- Focus on the samples that are hard to classify:
  - Re-learning based on up-weighting of previous misclassifications.

- Many different boosting algorithms exist, e.g., AdaBoost, LogitBoost, BrownBoost, LPBoost, XGBoost, …
  - AdaBoost directly available in scikit-learn, together with (Histogram) Gradient Boosting (Histogram is experimental in sklearn 0.24.1)
  - xgboost available in scikit format (pip install XGBoost (Mac/Linux), compilation on Windows)
  - BrownBoost available through Intel daal libraries (pip install daal (different interface))

The AdaBoost

- Complete training set every time
  - Repeated reweighting
  - Learn from previous mistakes

1. First classification
2. Increase weights on misclassified samples, decrease on the correct ones.
3. Repeat, usually several times
4. Combine the classifiers obtained in all the previous steps by weighted majority voting.



AdaBoost, detailed description

1. Set the weight vector **w** to uniform weights, where $\sum_i w_i = 1$.
2. For j in m boosting rounds, do the following:
    a. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.

First time all the sample weights in $\boldsymbol{w}$ are equal.

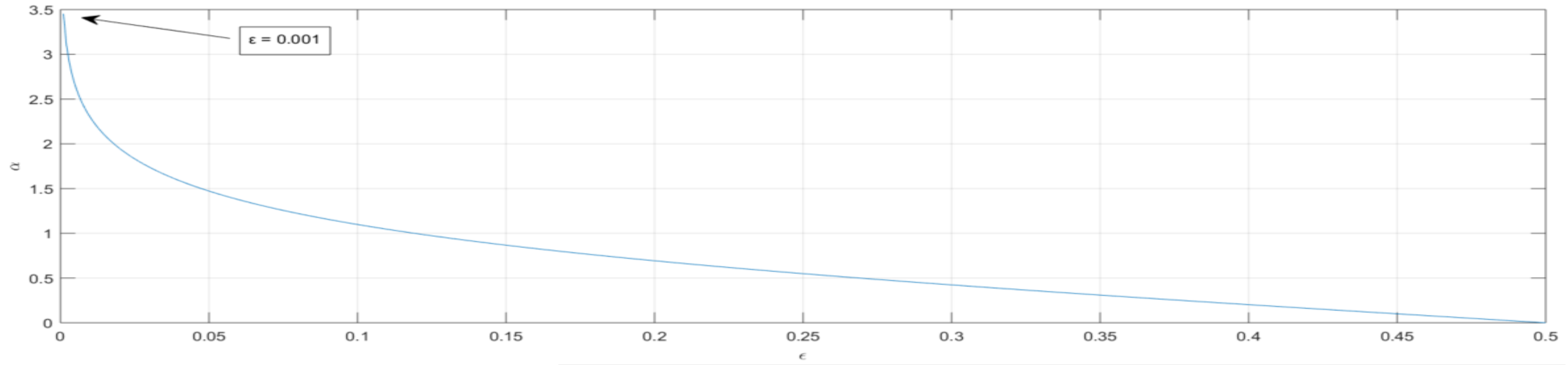
In later iterations weights are adjusted to emphasize samples that seems hard to classify correctly.

AdaBoost, detailed description

1. Set the weight vector **w** to uniform weights, where $\sum_i w_i = 1$.
2. For j in m boosting rounds, do the following:
    a. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
    b. Predict class labels: $\hat{y} = \text{predict}(C_j, \boldsymbol{X})$.
    c. Compute weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} \neq \boldsymbol{y})$.

Use the sample weights **w** to emphasize errors from hard to classify samples

AdaBoost, detailed description



c. Compute weighted error rate: $\varepsilon = \boldsymbol{w} \cdot \left(\hat{\boldsymbol{y}} \neq \boldsymbol{y} \right)$.

d. Compute coefficient: $\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$.

The coefficient α_j quantifies our confidence in the current round of classifications.

It will also be used in the subsequent updating of the sample weights ($\boldsymbol{w}$) (emphasizing samples that are hard to classify).

AdaBoost, detailed description

1. Set the weight vector **w** to uniform weights, where $\sum_i w_i = 1$.
2. For j in m boosting rounds, do the following:
   a. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
   b. Predict class labels: $\hat{y} = \text{predict}(C_j, \boldsymbol{X})$.
   c. Compute weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} \neq \boldsymbol{y})$.
   d. Compute coefficient: $\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$.
   e. Update weights: $\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$.

Next round's weights are an adjustment of this round's weights according to the confidence in this round and correct/incorrect classification **(y_i = +/- 1)**, x = element-wise product resulting in a vector.

Note: In the notation below, element-wise multiplication is represented by the cross symbol (×) and the dot-product between two vectors by a dot symbol (·), see more details on pages 250-54:

AdaBoost, detailed description

1. Set the weight vector **w** to uniform weights, where $\sum_i w_i = 1$.
2. For j in m boosting rounds, do the following:
   a. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
   b. Predict class labels: $\hat{y} = \text{predict}(C_j, \boldsymbol{X})$.
   c. Compute weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} \neq \boldsymbol{y})$.
   d. Compute coefficient: $\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$.
   e. Update weights: $\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$.
   f. Normalize weights to sum to 1: $\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$.
3. Compute the final prediction: $\hat{\boldsymbol{y}} = \left(\sum_{j=1}^{m} \left(\boldsymbol{\alpha}_j \times \text{predict}(C_j, \boldsymbol{X}) \right) > 0 \right)$.

Jupyter: Chapter 7, part 2

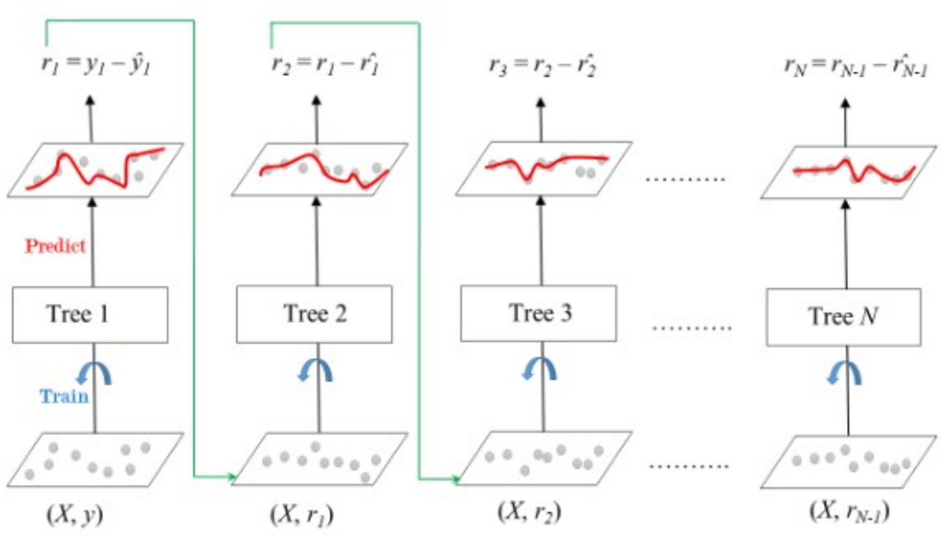
Gradient Boosting

- AdaBoost: Adjusts the weights of training instances to focus on the ones that **previous classifiers misclassified**.
- Gradient Boosting: Builds each tree on the **residuals of the previous tree's predictions**, effectively focusing on reducing the errors.

AdaBoost	Gradient Boosting
During each iteration in AdaBoost, the weights of incorrectly classified samples are increased, so that the next weak learner focuses more on these samples.	Gradient Boosting updates the weights by computing the negative gradient of the loss function with respect to the predicted output.
Calculates the weights for the weak classifiers in the final decision based on their accuracy.	Uses a learning rate to shrink the contribution of each tree to prevent overfitting and improve model robustness.
AdaBoost uses simple decision trees with one split known as the decision stumps of weak learners.	Gradient Boosting can use a wide range of base learners, such as decision trees, and linear models.
AdaBoost is more susceptible to noise and outliers in the data, as it assigns high weights to misclassified samples	Gradient Boosting is generally more robust, as it updates the weights based on the gradients, which are less sensitive to outliers.

Gradient Boosting

1. Initialize a base model
    1. Compute residual
    2. Build a weak learner to the residuals
    3. Compute the gradients of the loss
    4. Update the model
2. Make the final prediction



Step-by-step walk through* (regression)

- Dataset

Age	Category	Weight (kg)	Price (USD)	Pseudo residuals
25	Electronics	2.5	123	-33
34	Clothing	1.3	56	-100
42	Electronics	5.0	345	189
19	Homeware	3.2	98	-58

- Loss function (MSE)

$$\frac{1}{2}(Observed\ - Predicted)^2$$

- Initial prediction

$$\frac{123+56+345+98}{4} = 156$$

$$\begin{tikzcd}\text{Average} \arrow[d] \\ 156\end{tikzcd}$$

*https://www.datacamp.com/tutorial/guide-to-the-gradient-boosting-algorithm

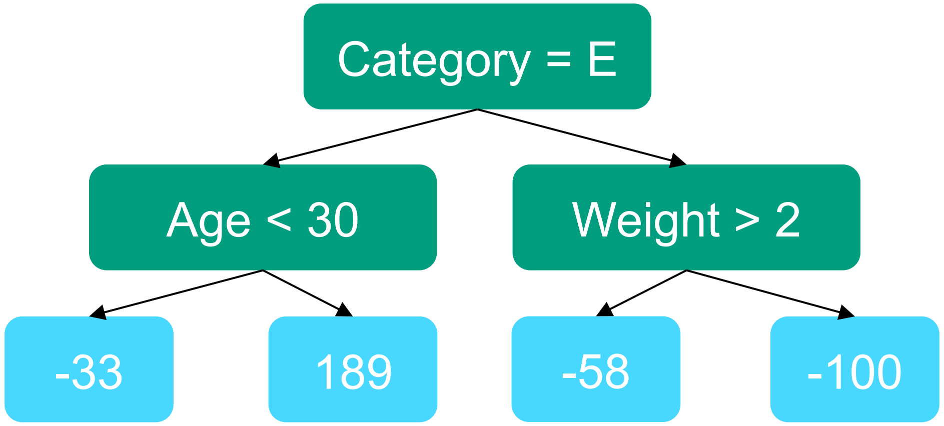
Step-by-step walk through*

- Dataset

Age	Category	Weight (kg)	Price (USD)
25	Electronics	2.5	123
34	Clothing	1.3	56
42	Electronics	5.0	345
19	Homeware	3.2	98

Pseudo residuals
-33
-100
189
-58

- Build a weak learner



*https://www.datacamp.com/tutorial/guide-to-the-gradient-boosting-algorithm

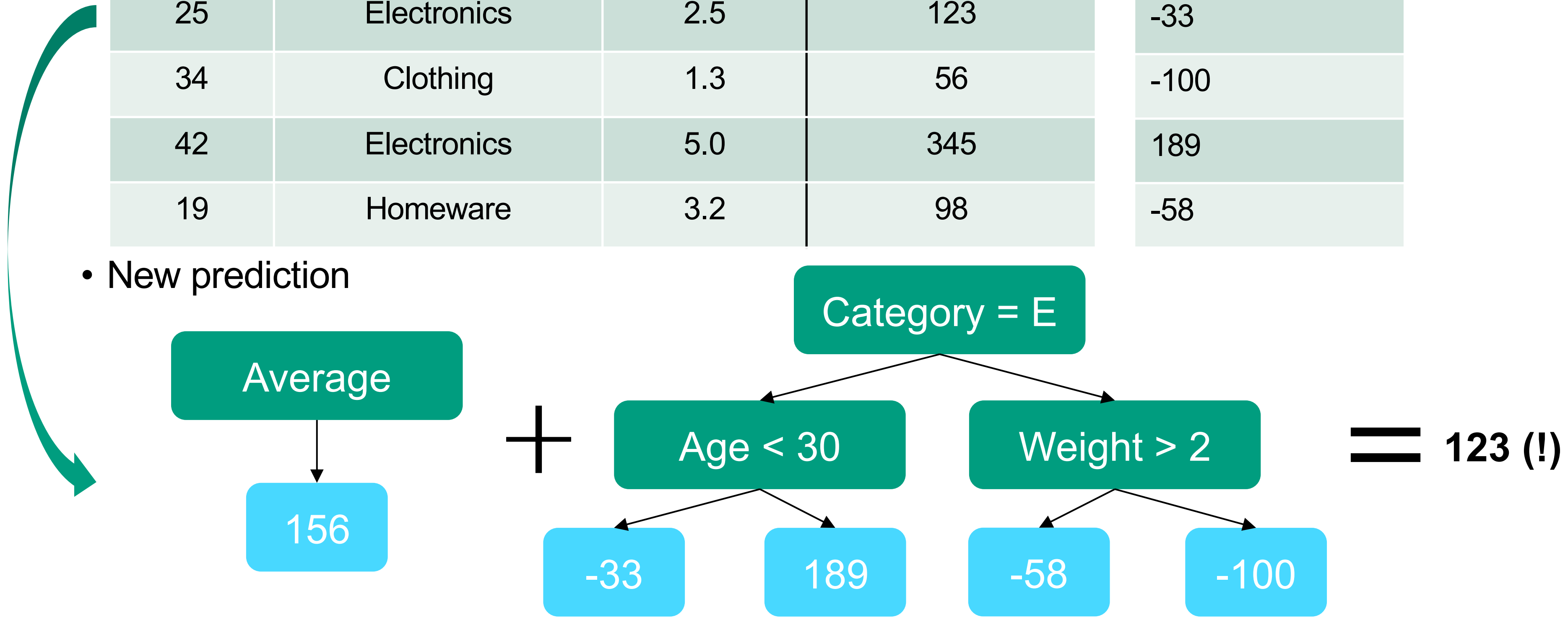
Step-by-step walk through*

- Dataset

Age	Category	Weight (kg)	Price (USD)
25	Electronics	2.5	123
34	Clothing	1.3	56
42	Electronics	5.0	345
19	Homeware	3.2	98

Pseudo residuals
-33
-100
189
-58

- New prediction



*https://www.datacamp.com/tutorial/guide-to-the-gradient-boosting-algorithm

Step-by-step walk through*

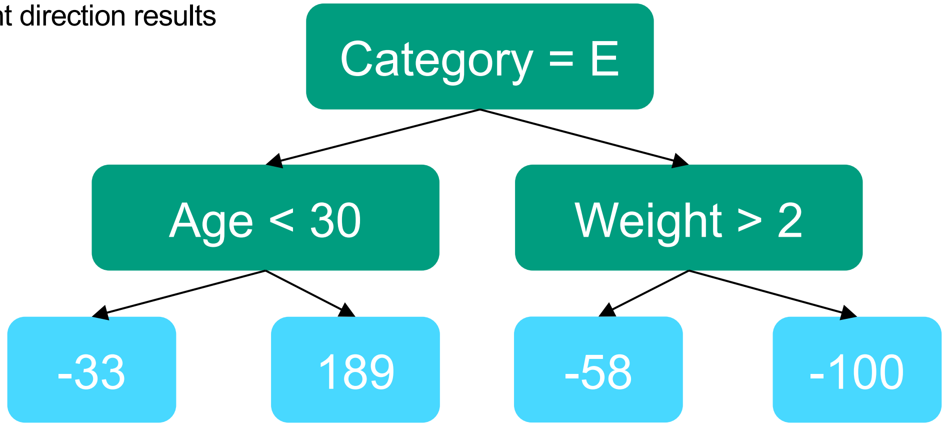
- Dataset

Age	Category	Weight (kg)	Price (USD)
25	Electronics	2.5	123
34	Clothing	1.3	56
42	Electronics	5.0	345
19	Homeware	3.2	98

(New) Pseudo residuals
-29.7
-90
170.1
-52.2

- Learning rate $\eta \to 0.1$ (lots of small steps in the right direction results in better prediction)

- New predictions
  - Sample 1: $156 + \eta * (-33) = 152.7$
  - Sample 4: $156 + \eta * (-58) = 150.2$



*https://www.datacamp.com/tutorial/guide-to-the-gradient-boosting-algorithm

Step-by-step walk through*

- Dataset

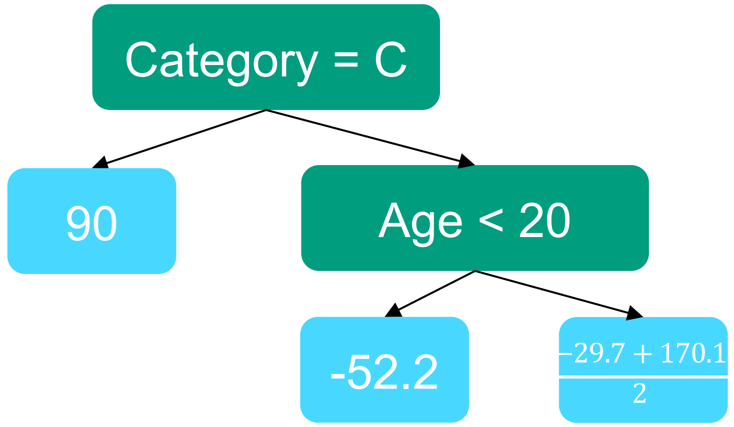
Age	Category	Weight (kg)	Price (USD)
25	Electronics	2.5	123
34	Clothing	1.3	56
42	Electronics	5.0	345
19	Homeware	3.2	98

(New) Pseudo residuals
-36.7
-81
153.1
-46.9

- New weak learner of residuals
- New predictions

$$1.\ 156 + (\eta * -33) + (\eta * 70.2) = 159.7$$

$$4.\ 156 + (\eta * -58) + (\eta * 52.2) = 144.9$$



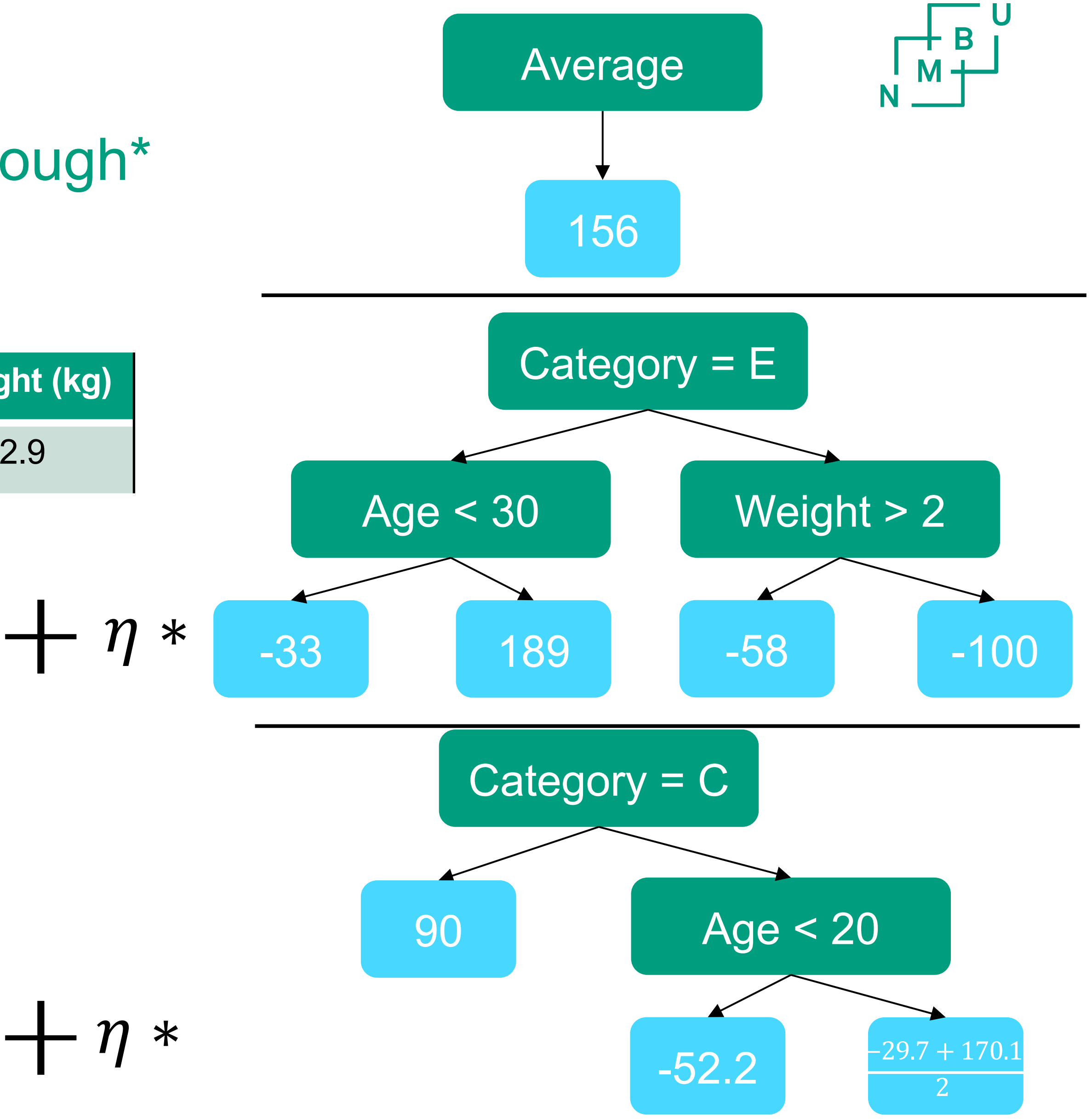
*https://www.datacamp.com/tutorial/guide-to-the-gradient-boosting-algorithm

Step-by-step walk through*

- New prediction

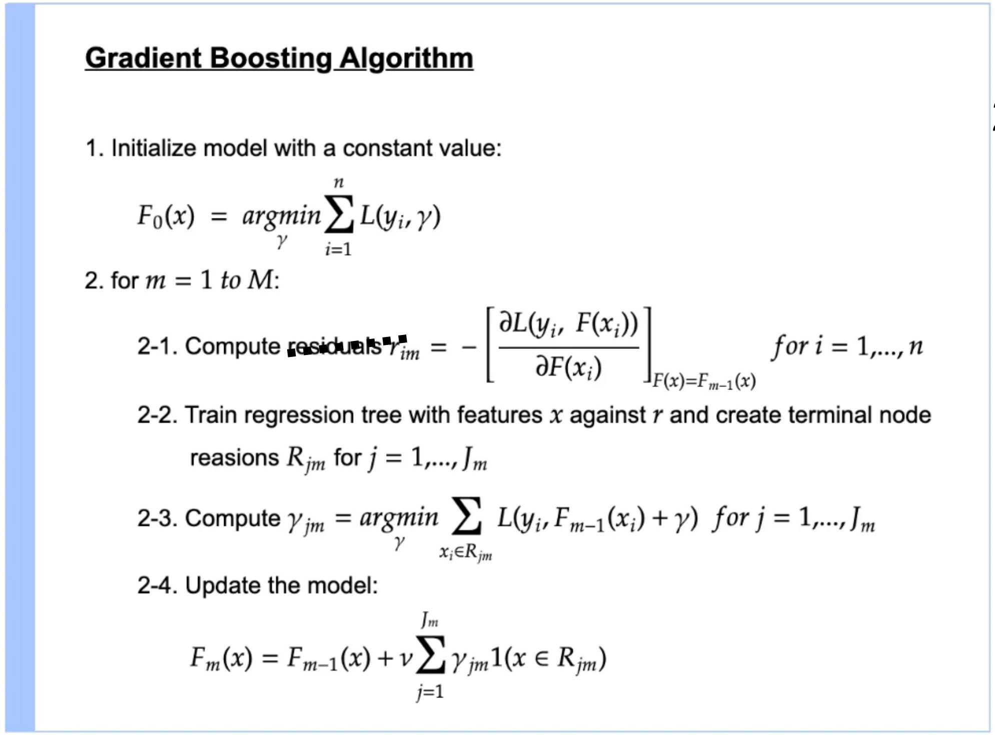
Age	Category	Weight (kg)
30	Electronics	2.9

- $156 + (0.1 * 189)$
  $+(0.1 * 70.2) = 144.12$



*https://www.datacamp.com/tutorial/guide-to-the-gradient-boosting-algorithm

Gradient Boosting – algorithmic overview*



Gradient Boosting Algorithm

1. Initialize model with a constant value:

$$F_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$$

2. for $m = 1$ to M:

2-1. Compute residuals $r_{im} = -\left[\frac{\partial L(y_i,\ F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$ $for\ i = 1,\dots,n$

2-2. Train regression tree with features x against r and create terminal node reasions R_{jm} for $j = 1,\dots,J_m$

2-3. Compute $\gamma_{jm} = \underset{\gamma}{argmin} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$ $for\ j = 1,\dots,J_m$

2-4. Update the model:

$$F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{jm} 1(x \in R_{jm})$$

Source: adapted from Wikipedia and Friedman's paper

1. Initial prediction -> searching for a value that minimize the loss (mean)
2. Loop through M trees
    1. Derivative of the loss function with respect to previous predictions
    2. Build a tree
    3. Minimising the average loss of the residuals on all the samples that belong in the terminal node R_{jm}
    4. New prediction of the combined models

*https://towardsdatascience.com/all-you-need-to-know-about-gradient-boosting-algorithm-part-1-regression-2520a34a502

Gradient Boosting – code example

- Notebook *Gradient_Boosting.ipynb*
  - Regression
  - Classification

The XG-Boost

- Aka eXtreme Gradient Boosting

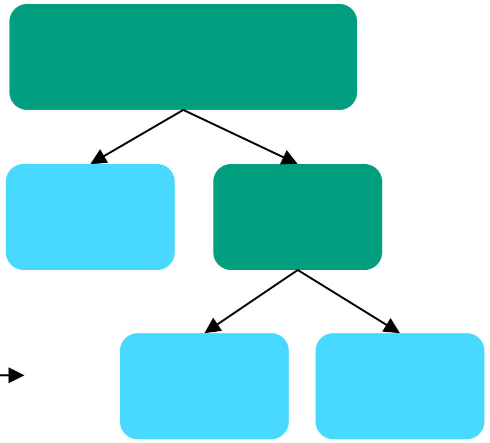
1. Initial prediction
2. Compute the residuals
3. Build a weak learner
   1. (XG-unique*) Decision/Regression tree on the residuals (greedy algorithm logic)
   2. Similarity score is the evaluation metric for nodes
   3. Gain score is an evaluation criterion for trees.
   4. Pruning if Gain Score < γ, the branch is pruned
4. New Prediction = Previous Prediction + (Learning rate) x (New Prediction)

* Regularized decision tree
Lambda: regularization parameters which reduces the predictions **sensitivity to individual observations.**
Gamma: specifies the minimum loss reduction required to make a further partition of the tree and controls the **actual structure of the trees**.

*XG-unique Trees

$$Gain = Similarity_{left} + Similariy_{right} - Simialarity_{root} \longrightarrow$$

$$Similarity\ score = \frac{(sum\ of\ residuals)^2}{Number\ of\ residuals + \lambda} \longrightarrow$$



The XG-Boost

Advantage	Disadvantage
Performance (track record on Kaggle competitions)	**Computational Complexity** (more computationally intensive steps)
Scalability (suitable for large datasets)	**Overfitting** (especially when trained on small datasets)
Customizability (range of hyperparameters that can be adjusted, making it highly customizable)	**Hyperparameter tuning** (importance of properly tune the parameters)
Interpretability (naturally provide feature importance through gain and similarity score/cover)	**Memory requirements**

XG Boost – code example

- Notebook *XGboost.ipynb*
  - Classification

Summary

Bagging versus Boosting

- **<u>Bagging</u>**
  - A parallel ensemble of classifiers, aiming to improve classification performance by decreasing model variance.
  - Especially useful for improving accuracy of unstable models and prevent against overfitting. <u>Works as averaging across "overfitted" models</u>.
  - Can degrade the performance of stable classifiers like k-NN.
  - Example: Random forests

- **<u>Boosting</u>**
  - A sequential ensemble of classifiers (next classifier emphasizes errors made by the previous one), aiming at decreasing model bias.
  - <u>Combines underfitted models</u>.
    - The individual classifiers must be at least better than random guessing (but not necessarily "too good").
  - Examples: Gradient boosting, AdaBoost

Summary

- **Majority voting:**
  - (Weighted) consensus from many classifiers.
  - Also used “under the hood” in multiclass classification problems.

- **Ensemble learning is:**
  - computationally more time consuming,
  - prone to overfitting,
  - but capable of improving on other classifiers (reduce bias and variance).