ENSEMBLES OF MODELS

BAGGING & BOOSTING & STACKING

Ensembles of models

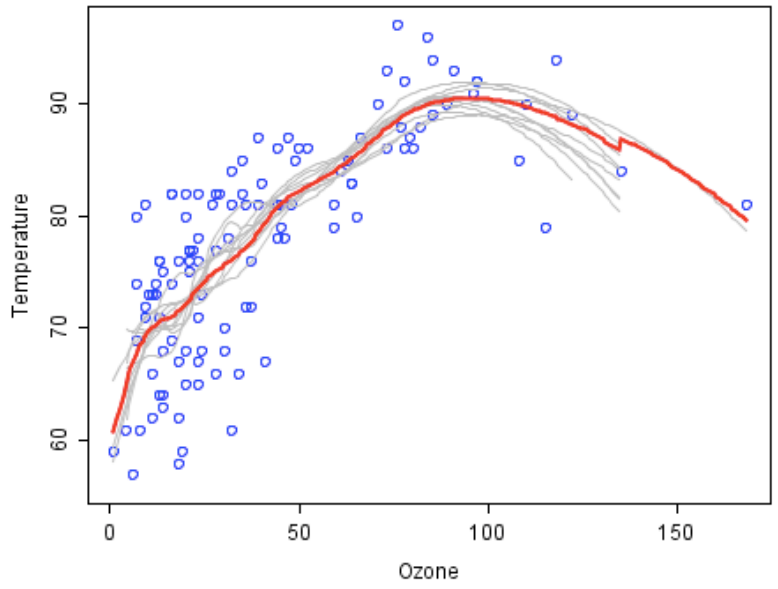
- **Ensemble** = a model which is a collection of models (collection of **base models**). e = $\{b_1, b_2, .., b_T\}$
- Main types:
  - Bagging: base models are independently trained
    - Random Forests: base model = tree Bagging + Randomization
  - Boosting: base models are trained sequentially
    - Gradient boosting: base model = regression tree
  - Stacking: base models are heterogeneous and are organized hierarchically

BAGGING

Bagging (Bootstrap aggregating)

- Motivation: some training methods tend to overfit the training set. They are also called unstable / high variance methods. That is, if the training sample changes slightly, the model obtained is "quite" different (relatively speaking). In other words, they tend to overfit.
  - Unstable: neural networks, trees (with large max depth), ...
  - Stable: linear, KNN, Support Vector Machines, ...

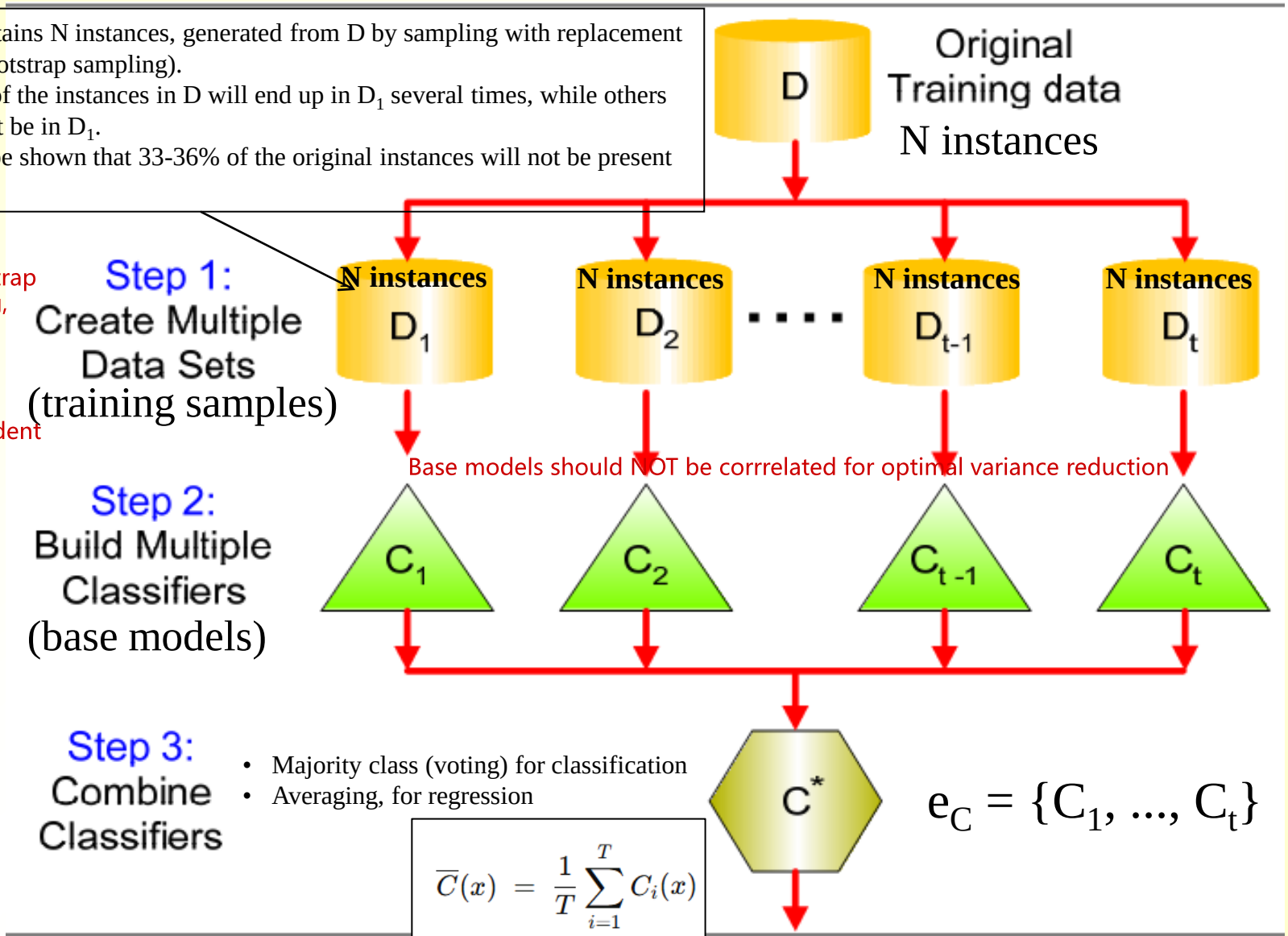
- Bagging strategy:
  - Simulate having several training samples
  - Given that biases in the training set are random, train several models with different training samples and average them



Bagging (Bootstrap aggregating)

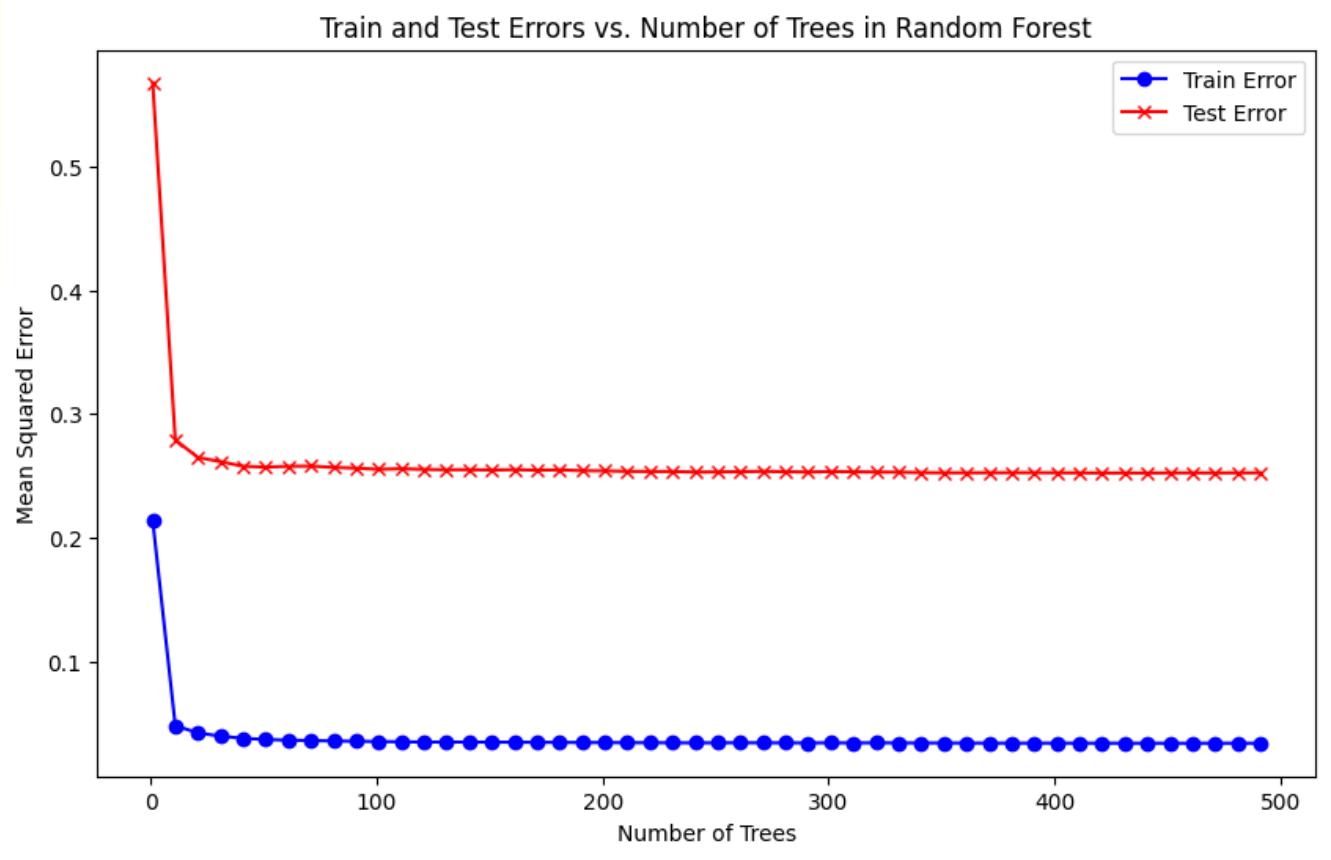
- D_1 contains N instances, generated from D by sampling with replacement (i.e. bootstrap sampling).
- Some of the instances in D will end up in D_1 several times, while others will not be in D_1.
- It can be shown that 33-36% of the original instances will not be present in D_1

In bootstrap sampling, the different data sets arent independent



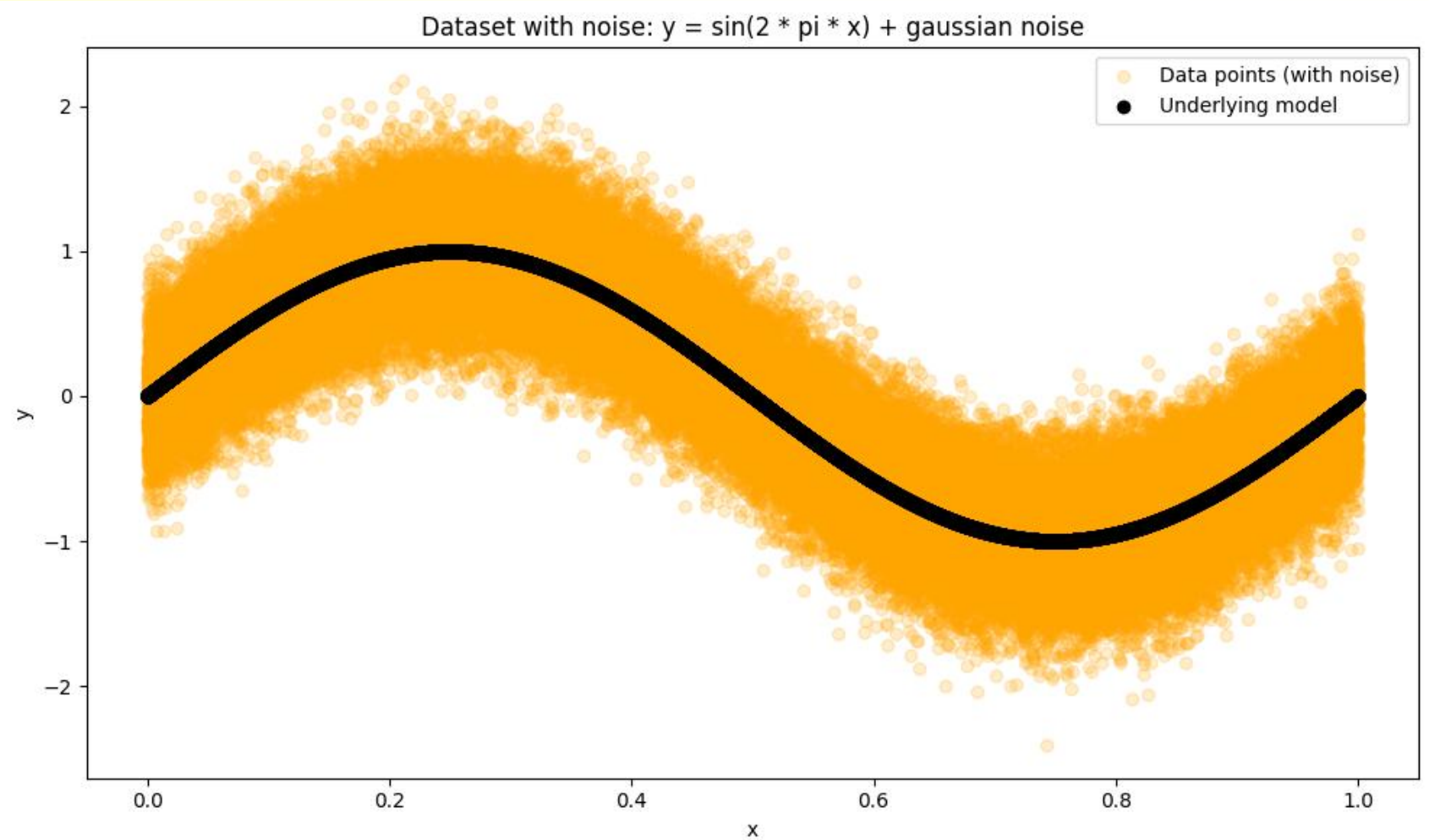
Bagging and error

Typically, the more models, the better (lower) the error. Bagging does not overfit by adding more base models.



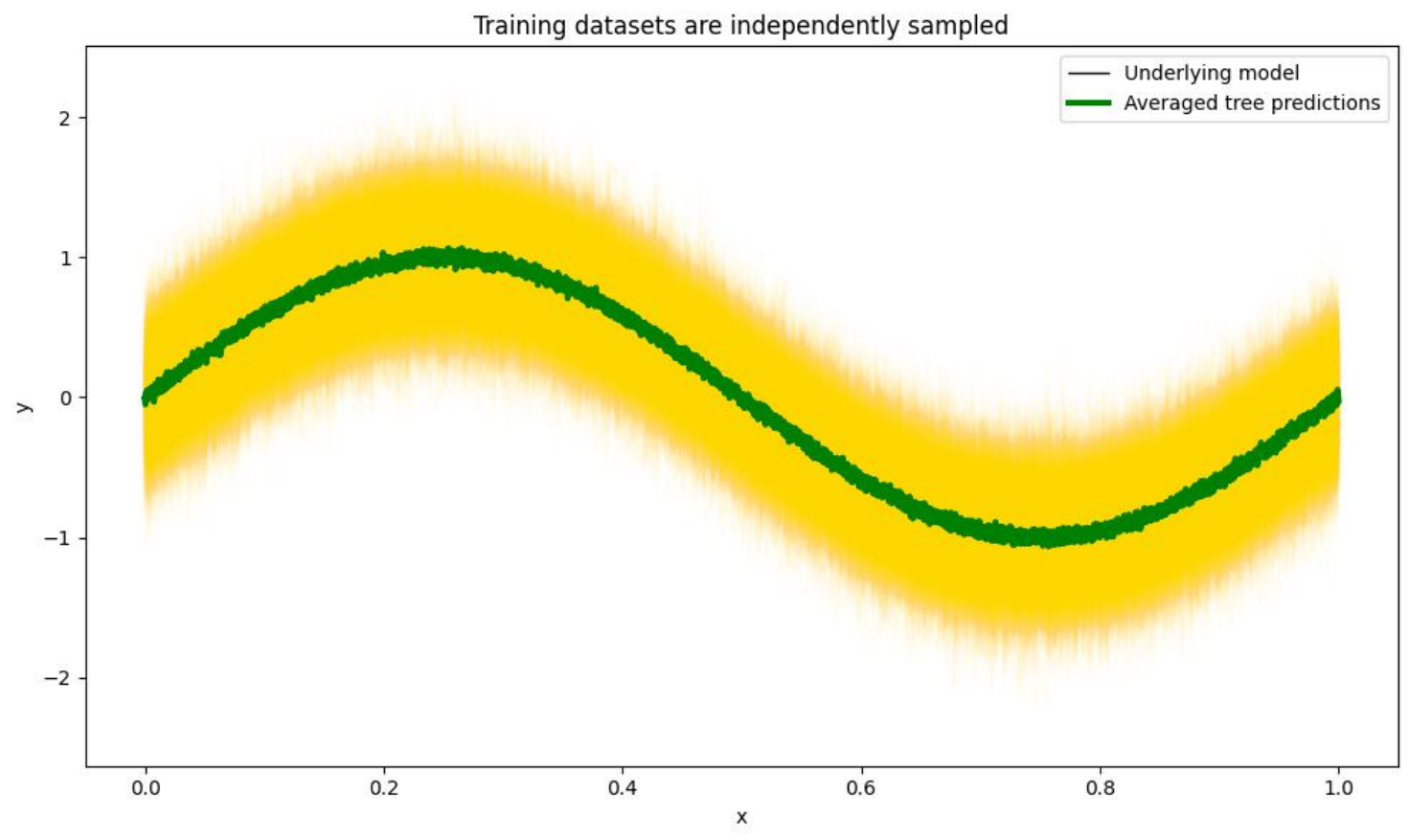
Number of base models $e_C = \{C_1, ..., C_t\}$

Bagging: how much variance is reduced?



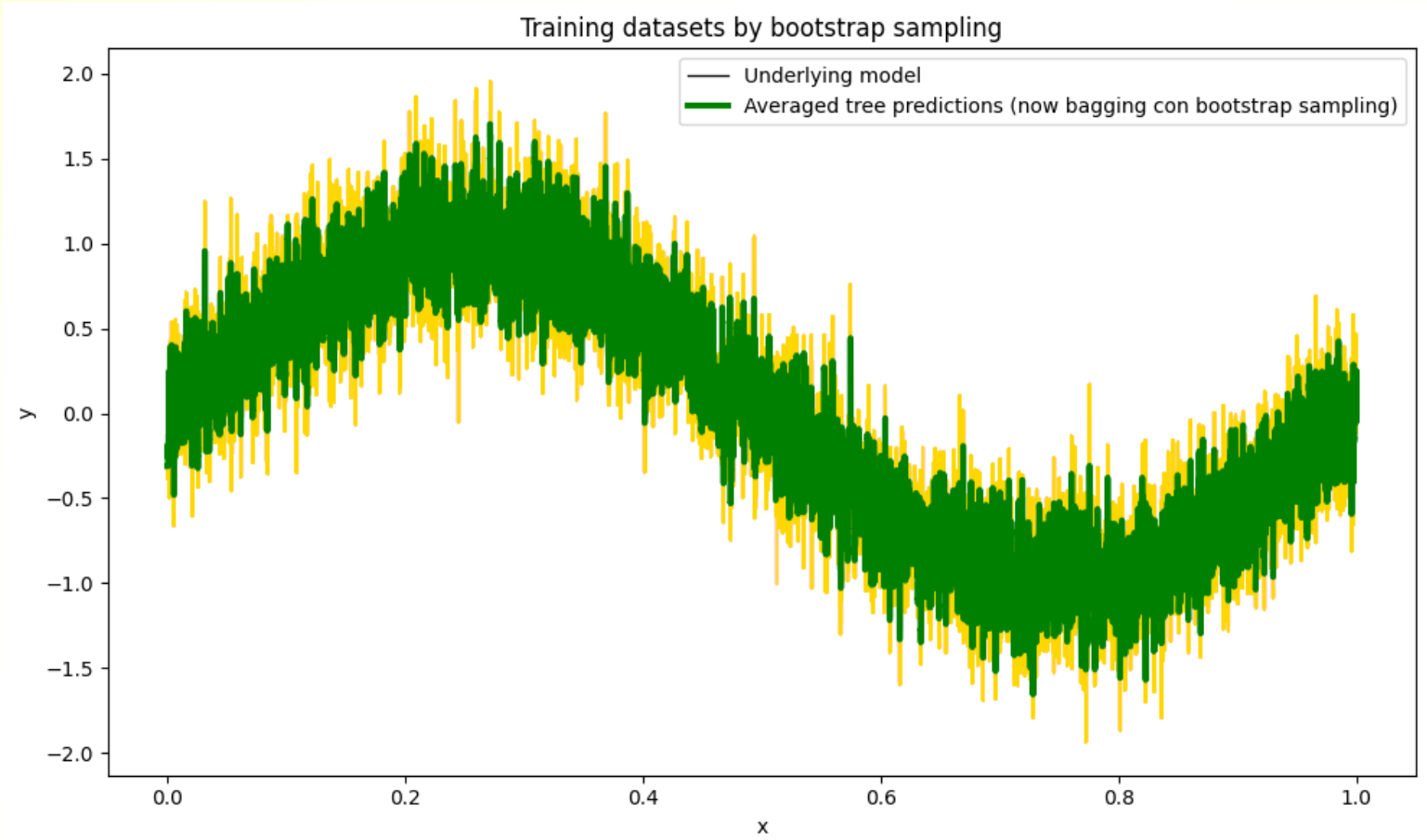
Bagging: how much variance is reduced?

- if the multiple training datasets (300) were independently sampled: → NOT BAGGING



Bagging: how much variance is reduced?

- But the multiple training datasets (300) are not independent (bootstrap sampling from a single training dataset), resulting in high correlation between the trees:



Bagging: how much variance is reduced?

$$\bar{C}(x) = \frac{1}{T}\sum_{i=1}^{T} C_i(x)$$

Variance from the bagging tree

$$\mathrm{Var}(\bar{C}) = \rho\,\sigma_C^2 + \frac{1-\rho}{T}\,\sigma_C^2$$

Correlation between base models

Variance of base models

variance of individual trees

Number of base models

Base models will be highly correlated when data samples are

- If base models (trees) are highly correlated, then Bagging variance reduction is very small.

$$\mathrm{Var}[\bar{C}(x)] \approx \sigma_C^2 \qquad (\rho \approx 1)$$

- If models are much less correlated, then Bagging variance reduction is large

$$\mathrm{Var}(\bar{C}(x)) = \frac{\sigma_C^2}{T}$$

Randomization

- There should be a diversity of base models: in an extreme case, if all base models were the same (highly correlated), computing their average would give no reduction in variance.
- Bootstrapped sampling achieves this to some degree.
- Randomization is a way of making the base models less correlated.

IT IS NOT WHAT BAGGING DOES, IN BAGGING THE DIFFERENT BASE MODELS ARE CORRELATED

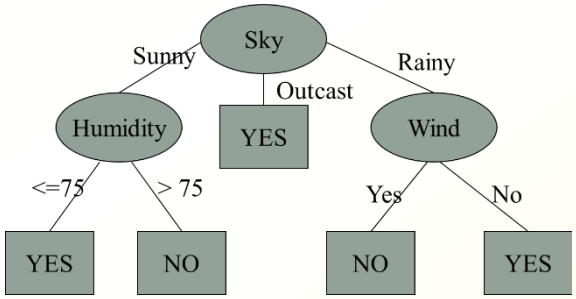
Randomization

- Randomization is an additional way of creating a set of diverse (less correlated) base models.
- Some training methods are stochastic: everytime they are run, they generate a different model (even if the training set is the same)
  - E.g. : neural networks, because initial weights / parameters are random.
- The basic tree training algorithm is deterministic. But it can be modified so that it becomes stochastic.
- Random Forests modify the tree training algorithm so that it becomes stochastic. This modification is called "randomization"
- In order to get a variety of base models, RF use both:
  - bootstrap sampling
  - and **randomization.**

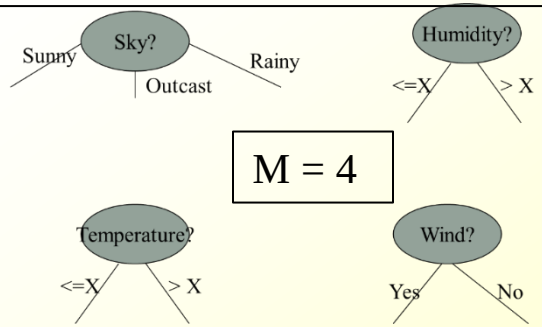
} 2 sources of variability (overcomes the problem of correlation between base models)

Random Forests (RF)

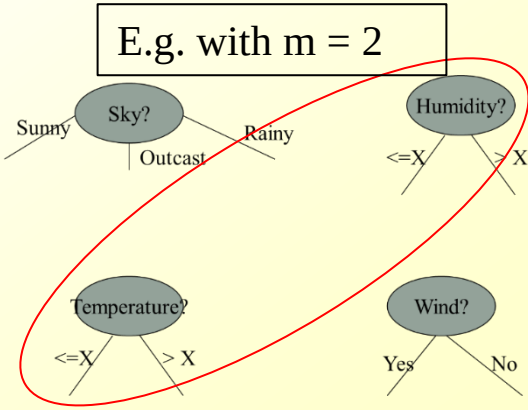
- RF = **Bagging** with decision trees. It uses:
  1. Random sampling with replacement (that is, bootstrap sampling)
  2. Randomization: when choosing an attribute for a node, the best is selected from a random subset of $m < M$ attributes (instead of the entire set of M attributes).



– The standard algorithm for training trees:
  1. Start with the empty tree. Current node = root node
  2. Compute the entropies of all available features (*M*) and select the best one for the current node
  3. Go to 2, and continue growing the tree recursively



– The randomized version
  1. Start with the empty tree
  2. S = Select a random feature subset of size *m* out of all features
  3. Compute the entropies of features in S and select the best one for the current node
  4. Go to 2, and continue growing the tree recursively



Random Forests (RF)

- RF = **Bagging** with decision trees. It uses:
  - Random sampling with replacement (that is, bootstrap sampling)
  - Randomization: when choosing an attribute for a node, the best is selected from a random subset of size $m < M$ attributes (instead of the entire set of M attributes).
    - For instance, if there are M=16 attributes and m=3, then 3 attributes are randomly chosen for every nod, and then the best (smallest entropy) of the three is selected
    - m is usually called *mtry*, or *max_features* in sklearn).
    - Typically $m=sqrt(M)$ for classification and $m = M/3$ for regression.
    - If $m == M$, then RF ~ Bagging

Random Forests

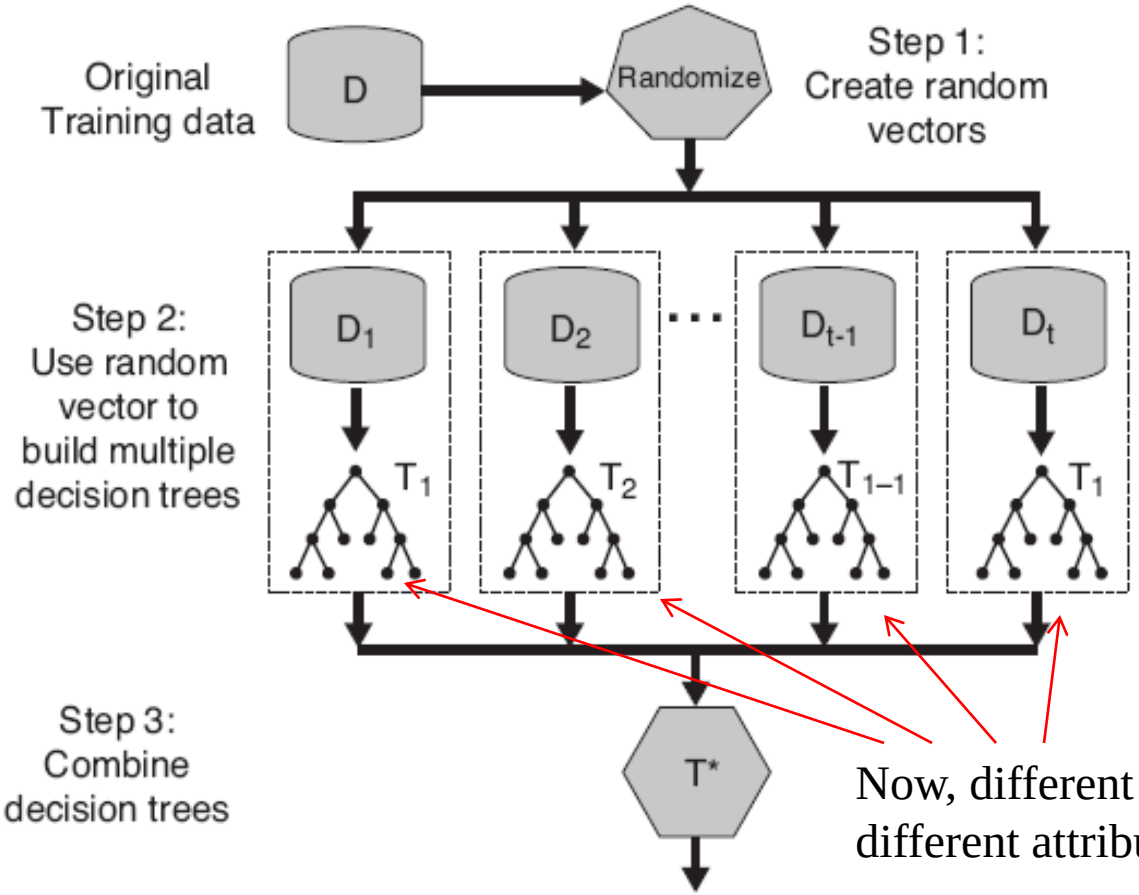


Figure 5.40. Random forests.

Results of Random Forests

- Usually RF outperform single decision trees

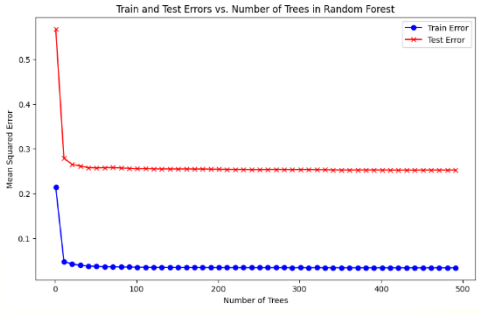
Test set misclassification error (%)

Data set	**Forest**	**Single tree**
Breast cancer	2.9	5.9
Ionosphere	5.5	11.2
Diabetes	24.2	25.3
Glass	22.0	30.4
Soybean	5.7	8.6
Letters	3.4	12.4
Satellite	8.6	14.8
Shuttle $\times 10^3$	7.0	62.0
DNA	3.9	6.2
Digit	6.2	17.1

Hyper-parameter tuning of Random Forests

```
class sklearn.ensemble.RandomForestClassifier(n_estimators=100, *, criterion='gini', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='sqrt', max_leaf_nodes=None,
min_impurity_decrease=0.0, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False,
class_weight=None, ccp_alpha=0.0, max_samples=None)                                                    [sour
```

- Main hyper-parameters:
  - n_estimators: number of trees (base models) in the ensemble: there should be enough trees in the ensemble. The default (100) usually works well, but it can also be tuned.
  - **max_features**: mtry or m. Size of the random subset of features to be compared at every node. Possible values: sqrt, log2, or a number between 1 and the maximum number of features in the dataset.
    - It can also be a real number between 0 and 1 (fraction of features)
  - max_depth / min_samples_split: hyper-parameters of the trees (base models)
  - njobs is not a hyper-parameter, but allows the RF to be trained in parallel.



Random Forests. Out of bag estimation (OOB)

- We already know that in order to estimate future performance (model evaluation), it is necessary to use a different test set than the one used for training
- This is typically done by means of train/test (a.k.a. hold-out) and also by means of crossvalidation
- But RF are able to provide model evaluation without using a test set (called **out-of-bag OOB evaluation**)
- It takes advantage that some instances are not present in some of the training sets D_i. Given that they have not been used for training, they can be used for testing the associated tree T_i
- The error of instance *x* will be computed by using a **reduced** ensemble: only the trees where *x* was not used for training
- In any case, if different models (RF and others) are going to be compared, it is better to use the same model evaluation technique for all of them.

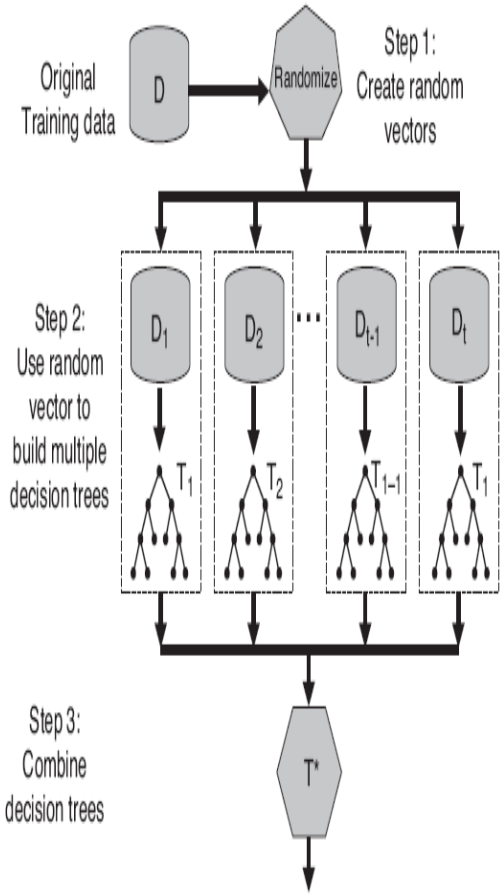
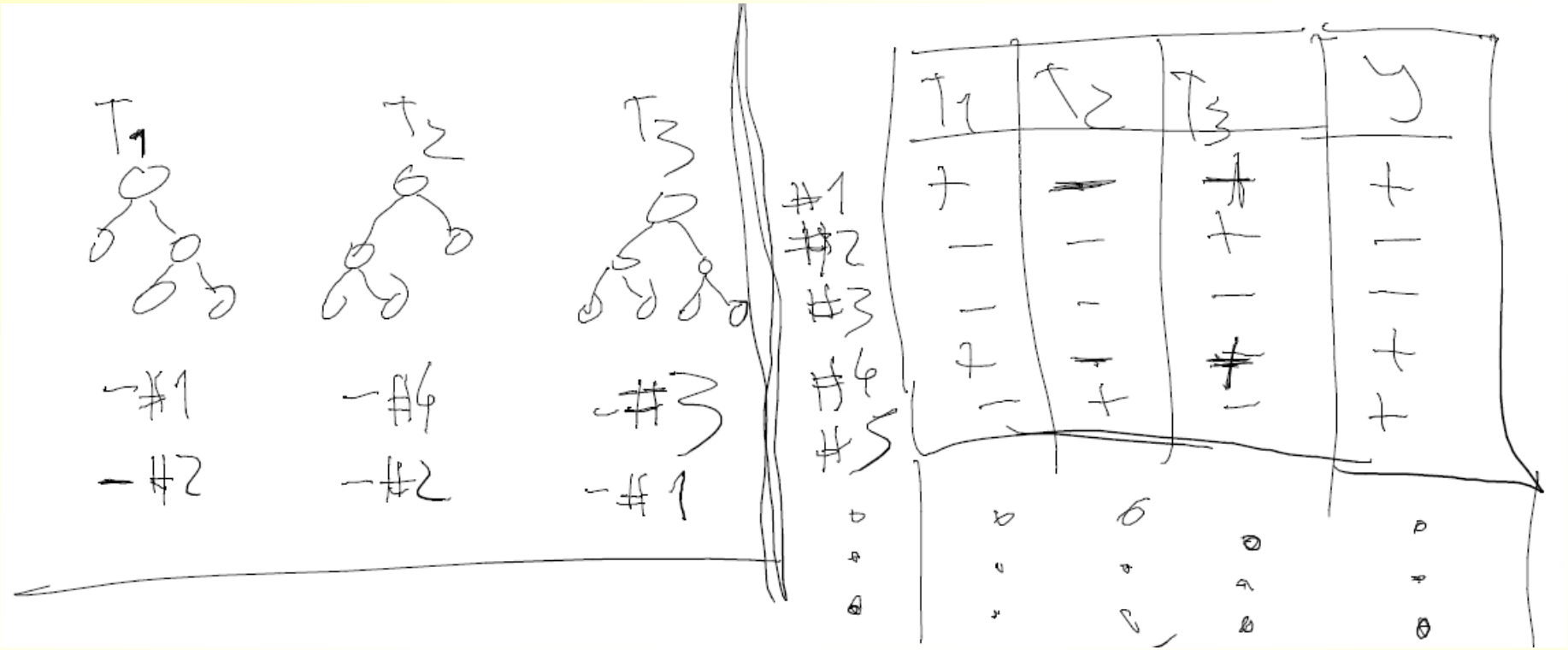


Figure 5.40. Random forests.



REPASAR COMO FUNCIONA OOB

What would OOB be, using the first five instances?

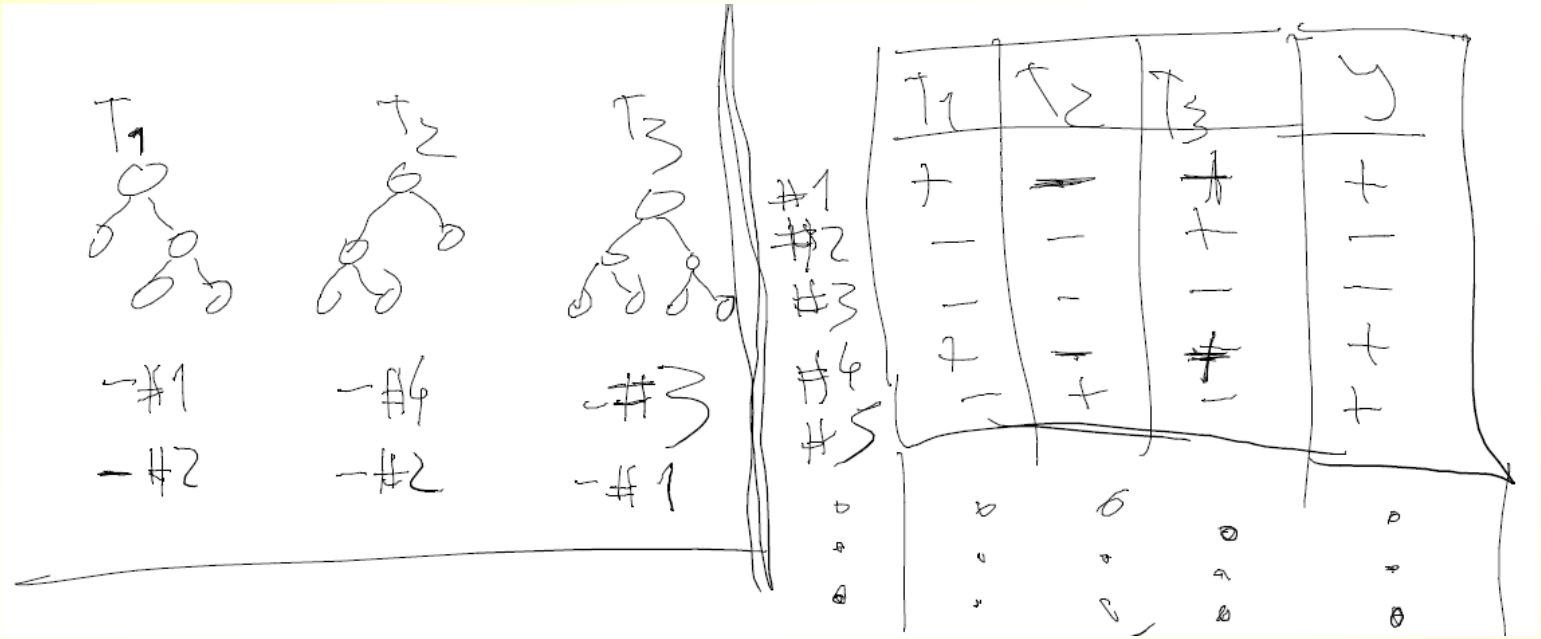
Possible answers:

- a: 3/5
- b: 3/4
- c: 2/5

- #1, T1 (+) and T3 (+), output of the reduced ensemble: +; y=+. 1 success
- #2, T1 (-) and T2 (-), output of the reduced ensemble: -; y=-. 1 success
- #3, T3 (-), output of the red. ens.: -; y=-. 1 success.
- #4, T2 (-) , output of the red. ens.: -; y=+, 1 faillure
- #5, none
- Anwer: 3 successes out of 4: OOB accuracy = 3/4

Possible answers:

- a: 3/5
- **b: 3/4**
- c: 2/5



```python
# Load the data
california = fetch_california_housing()
X, y = california.data, california.target

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train the model
rf = RandomForestRegressor(n_estimators=100, oob_score=True, random_state=42)
rf.fit(X_train, y_train)

# Obtain the out-of-bag predictions
oob_predictions = rf.oob_prediction_
# Calculate the out-of-bag MAE
oob_mae = mean_absolute_error(y_train, oob_predictions)

# Predict on the test data
y_pred = rf.predict(X_test)
# Calculate the test MAE
test_mae = mean_absolute_error(y_test, y_pred)

# Output the results
print(f'Out-of-Bag MAE: {oob_mae}')
print(f'Test MAE: {test_mae}')
```

```
Out-of-Bag MAE: 0.33205583527473714
Test MAE: 0.32754256845930246
```

Random Forests. Summary

- Only two new hyper-parameters: number of trees in the ensemble (*T*, *n_estimators*) and size of the attribute subset (*m*, *mtry*, *max_features*).
  - But there are additional hyper-parameters for the trees: criterion, max_depth, min_samples_split, ...)
- Usually it outperforms single decision trees
- **Faster** than standard Bagging (because only *mtry* << M attributes are considered for each node)
- We get Out-of-bag (OOB) estimation (estimation of future performance) for free
- It is able to return **probabilistic predictions**: if 90 trees say "+" and 10 say "-", probability of "+" is 0.9

Extremely randomized trees (Extra-trees)

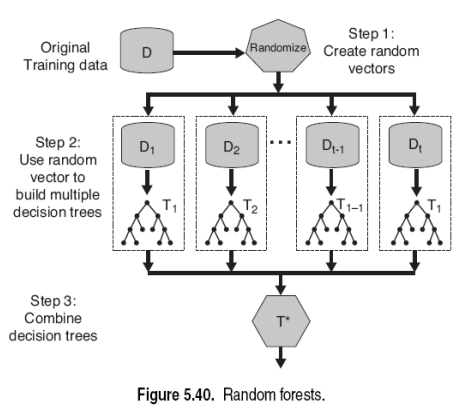


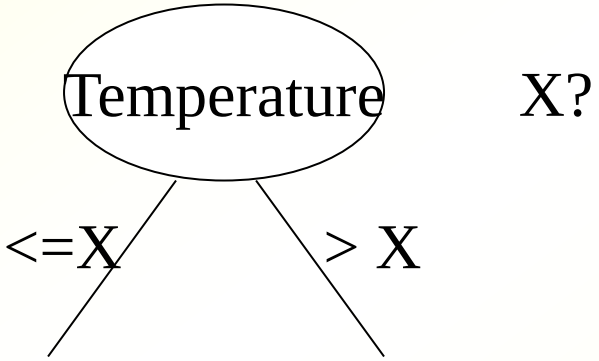
Figure 5.40. Random forests.

- Motivation:
  - Increase randomization of base models
  - Reduce training time: most of the training time in trees is spent when selecting the threshold for numeric attributes
- Extra-trees work similarly to RF, but:
  - the whole training set is used for all base models (random sampling with replacement is NOT used in Extra-trees)
  - For every decision tree node, *mtry* attributes are chosen randomly (out of M), like in standard RF, but:
    - If the attribute is numeric, the threshold is chosen randomly.
      - E.g. Let's suppose there are 6 features $\{a_1, a_2, a_3, a_4, a_5, a_6\}$
      - If, for instance, max_features=3, then 3 features will be selected randomly: $\{a_2, a_4, a_5\}$
      - For each selected feature a_i, a random threshold t_i will be generated: $\{a_2<t_2, a_4<t_4, a_5<t_5\}$
      - The average entropy of each of the three will be computed; and the best $a_i<t_i$ will be selected.

Reminder: what to do for continuous attributes?

A binary (two-valued) attribute is created by computing a threshold X

Nota: only some thresholds are shown. The best one is X=84 with average entropy = 0.83



64 – Yes, 65-No, 68 – Yes, 69 – Yes, 70 – Yes, 71 – No, 72 – YesNo, 75 – YesYes, 80 – No, 81 – Yes, 83 – Yes, 85 - No

HP = 0.83

X=84

4 No, 9 Yes

1 No, 0 Yes

64 – Yes, 65-No, 68 – Yes, 69 – Yes, 70 – Yes, 71 – No, 72 – YesNo, 75 – YesYes, 80 – No, 81 – Yes, 83 – Yes, 85 - No

HP = 0.93

2 No, 4 Yes

X=71.5

3 No, 5 Yes

64 – Yes, 65-No, 68 – Yes, 69 – Yes, 70 – Yes, 71 – No, 72 – YesNo, 75 – YesYes, 80 – No, 81 – Yes, 83 – Yes, 85 - No

HP = 0.89

1 No, 4 Yes

X=70.5

4 No, 5 Yes

Extremely randomized trees (Extra-trees)

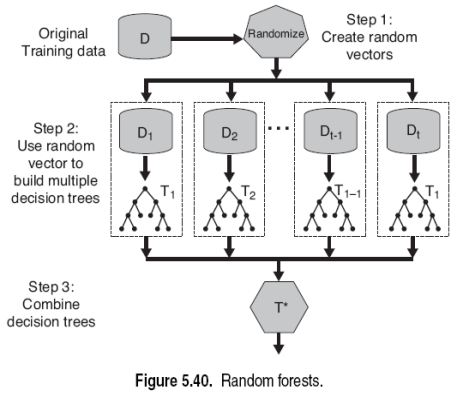


Figure 5.40. Random forests.

- Motivation:
  - Increase randomization of base models
  - Reduce training time: most of the training time in trees is spent when selecting the threshold for numeric attributes
- Extra-trees work similarly to RF, but:
  - the whole training set is used for all base models (sampling with replacement is NOT used in Extra-trees)
  - For every decision tree node, *mtry* attributes are chosen randomly (out of M), like in standard RF, but:
    - If the attribute is numeric, the threshold is chosen randomly.
    - **If the attribute is categorical, a binary decision is selected randomly:**
      - **E.g. : if attribute A has values {a,b,c,d,e}, a binary decision could be:**
        - **if(A ∈ {c,e}) left else right.**
    - **Notice that all nodes (categorical and numerical) are binary.**

Extra-trees results

- In 12 classification and 12 regression problems

Table 2 Win/Draw/Loss records (corrected *t*-tests) comparing the algorithm in the column versus the algorithm in the row

	Classification problems					Regression problems				
	PST	TB	RS*	RF*	ET^d	PST	TB	RS*	RF*	ET^d
PST	–	8/4/0	11/1/0	11/1/0	10/2/0	–	10/2/0	8/4/0	10/2/0	10/2/0
TB	0/4/8	–	7/5/0	7/5/0	7/5/0	0/2/10	–	0/9/3	1/11/0	4/8/0
RS*	0/1/11	0/5/7	–	0/8/4	2/10/0	0/4/8	3/9/0	–	4/8/0	4/7/1
RF*	0/1/11	0/5/7	4/8/0	–	5/7/0	0/2/10	0/11/1	0/8/4	–	3/7/2
ET^d	0/2/10	0/5/7	0/10/2	0/7/5	–	0/2/10	0/8/4	1/7/4	2/7/3	–

- PST = single trees (trained with CART)
- TB = Tree Bagging
- RF = Random Forests

Extra-trees results

- ET takes much less time

Table 4 Computing times (msec) of training (ensembles of $M = 100$ trees)

Classification problems					Regression problems			
Dataset	ST	TB	RF^d	ET^d	Dataset	ST	TB/RF^d	ET^d
Waveform	68	4022	1106	277	Friedman1	7	372	284
Two-Norm	66	3680	830	196	Housing	12	685	601
Ring-Norm	101	4977	1219	251	Hwang-f5	16	917	742
Vehicle	80	5126	1500	685	Hwang-f5n	15	948	748
Vowel	236	14445	4904	694	Pumadyn-32fh	251	13734	9046
Segment	291	18793	5099	1053	Pumadyn-32nm	221	11850	9318
Spambase	822	55604	9887	8484	Abalone	73	4237	3961
Satellite	687	45096	11035	5021	Ailerons	495	29677	26572
Pendigits	516	34449	12080	5183	Elevators	289	15958	13289
Dig44	4111	259776	67286	9494	Poletelecomm	497	28342	26576
Letter	665	44222	17041	14923	Bank-32nh	613	34402	20178
Isolet	37706	2201246	126480	11469	Census-16H	597	35207	27900

ENSEMBLES OF MODELS

BOOSTING

Boosting

- Motivation: to iteratively improve (to boost) weak models (weak learners)
  - Weak learner: method that obtain models that are better than chance, but not much better
  - Trees with low depth are weak learners (e.g. decision stumps are trees with max depth = 1)
- Boosting adds **sequentially** a new model h_t to the ensemble

  $\{h_0, h_1\} + h_2 \Rightarrow \{h_0, h_1, h_2\} \Rightarrow \ldots \Rightarrow$
- The general idea is that the new base model h_t improves the ensemble trained so far (e.g. h_t succeeds for the instances for which h_{t-1}failed)
- There are several boosting techniques (e.g. Adaboosting), but nowadays, Gradient Boosting has become the standard (and other Boosting techniques are considered variants)

GRADIENT BOOSTING

- Adaptation of boosting for regression based on the idea of gradient descent on function space
- Basic idea: new base model h_t is computed to approximate the difference (pseudo-residual) between the current ensemble output and the target output.

GRADIENT BOOSTING

- Let's suppose that **weak** model $f_0(x)$ is trained to aproximate response variable y
- Definition: pseudo-residual r for f_0 = error made by $f_0(x)$ on x_i
  - $r_0(x_i) = y_i - f_0(x_i)$
- Let $h_0(x)$ be a model that approximates pseudo-residuals:
  - i.e its training set is $\{(x_1, y_1 - f_0(x_1)), \ldots, (x_N, y_N - f_0(x_N))\}$
- In principle, model $f_1(x) = f_0(x) + h_0(x)$ will approximate better the response variable y
  - In the best case, $f_0(x_i) + h_0(x_i) = f_0(x_i) + y_i - f_0(x_i) = y_i$
- Repeat T times: $h_0, h_1, \ldots, h_{T-1}$

GRADIENT BOOSTING

- Repeat T times:
  - $f_0(x) = ?$
  - $f_1(x) = f_0(x) + h_0(x)$
  - $f_2(x) = f_1(x) + h_1(x) = f_0(x) + h_0(x) + h_1(x)$
  - $f_3(x) = f_2(x) + h_2(x) = f_0(x) + h_0(x) + h_1(x) + h_2(x)$
  - …
  - $f_T(x) = f_{T-1}(x) + h_{T-1}(x) = f_0(x) + h_0(x) + \ldots + h_{T-1}(x)$

GRADIENT BOOSTING

1. Initialize $f_0(x)$ = mean(y_i)
2. FOR t = 0 TO (T-1)
   - A new model h_t is going to be added to the ensemble
   - Compute a new training set, whose outputs are the "pseudo-residuals", where for each (x_i, y_i), the new training instance is $(x_i, y_i - f_t(x_i))$
     - $\{(x_1, y_1 - f_t(x_1)), (x_2, y_2 - f_t(x_2)), \ldots, (x_N, y_N - f_t(x_N))\}$
     - Where partial ensemble so far: $f_t(x) = f_0(x) + c_0 h_0(x) + \ldots + c_{t-1} h_{t-1}(x)$
   - Train a new model $h_t(x)$ to fit the pseudo-residuals
   - Compute c_t to optimize the error of:

     $f_t(x) + c_t h_t(x)$
   - Update the partial ensemble: $f_{t+1}(x) = f_t(x) + c_t h_t(x)$

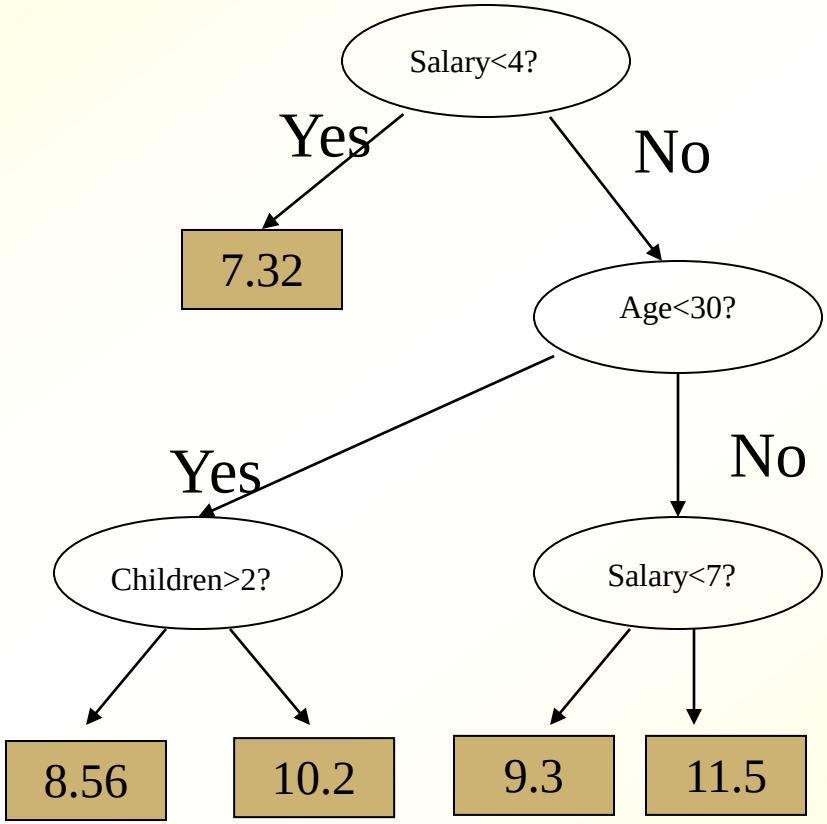
GRADIENT BOOSTING

- A GB ensemble is a collection of base models, for regression:
  - $f_T(x) = \mathbf{f_0}(x) + c_0\, \mathbf{h_0}(x) + c_1\, \mathbf{h_1}(x) + \ldots + c_{T-1}\, \mathbf{h_{T-1}}(x)$
  - base models trained sequentially, improving the ensemble so far

- Training set = $\{(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), \ldots, (\mathbf{x_N}, y_N)\}$
- $f_0(x)$ = mean($\{y_i\}$)
- $f_1(x) = f_0(x) + c_0\, h_0(x)$
  - where $h_0(x)$ is a model trained with the pseudo-residual dataset:
    - $\{(x_1, y_1 - f_0(x_1)), (x_2, y_2 - f_0(x_2)), \ldots, (x_N, y_N - f_0(x_N))\}$
- $f_2(x) = f_1(x) + c_1\, h_1(x)$
  - where $h_1(x)$ is a model trained with the pseudo-residual dataset:
    - $\{(x_1, y_1 - f_1(x_1)), (x_2, y_2 - f_1(x_2)), \ldots, (x_N, y_N - f_1(x_N))\}$
- ...
- $f_T(x) = f_{T-1}(x) + c_{T-1}\, h_{T-1}(x) = \text{mean}(\{y_i\}) + c_0\, h_0(x) + \ldots + c_{T-1}\, h_{T-1}(x)$

GRADIENT BOOSTED TREES

- It is gradient boosting with regression trees.



Original California Housing dataset

X_train

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

y_train
1.030
3.821
1.726
0.934
0.965

Initial weak model f0 = mean(y_train) (dummy regression model)
Current partial ensemble = {f0}

X_train

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

f0_preds_train
2.071947
2.071947
2.071947
2.071947
2.071947

```
y_train.mean()
```

New training dataset for training base model $\mathbf{h_0}$
Current partial ensemble = {f0}

X_train

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

y_train_0

y_train		f0_preds_train
1.030		2.071947
3.821		2.071947
1.726	-	2.071947
0.934		2.071947
0.965		2.071947

Pseudo-residuals
y-f0(x)

New training dataset for training base model $\mathbf{h_0}$

c.p.e. {f0}

X_train

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

y_train_0

y_train		f0_preds_train
1.030		2.071947
3.821		2.071947
1.726	-	2.071947
0.934		2.071947
0.965		2.071947

Pseudo-residuals
y-f0(x)

MedInc	HouseAge	AveRooms	AveBedrms	y_train_0
3.2596	33.0	5.017657	1.006421	-1.041947
3.8125	49.0	4.473545	1.041005	1.749053
4.1563	4.0	5.645833	0.985119	-0.345947
1.9425	36.0	4.002817	1.033803	-1.137947
3.5542	43.0	6.268421	1.134211	-1.106947

New training dataset for training base model h_0

c.p.e. {f0}

X_train

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

y_train_0

y_train		f0_preds_train
1.030		2.071947
3.821		2.071947
1.726	-	2.071947
0.934		2.071947
0.965		2.071947

Pseudo-residuals
y-f0(x)

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

y_train_0
-1.041947
1.749053
-0.345947
-1.137947
-1.106947

```
h0 = DecisionTreeRegressor(max_depth=3, random_state=42)
h0.fit(X_train, y_train_0)
```

New training dataset for training base model $\mathbf{h_0}$

c.p.e. f1 = {f0, h0}

X_train

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

y_train_0
-1.041947
1.749053
-0.345947
-1.137947
-1.106947

Pseudo-residuals
y-f0(x)

```
h0 = DecisionTreeRegressor(max_depth=3, random_state=42)
h0.fit(X_train, y_train_0)
```

Current partial ensemble f1: {f0, h0}
Making predictions: f1(x) = f0(x) + h0(x)
Typically, an optimized weight c0 is also used: f1(x) = f0(x) + c0 * h0(x)

New training dataset for training base model $\mathbf{h_1}$

c.p.e. f1 = {f0, h0}

X_train

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

y_train_1 = pseudo-residuals y-f1(x)

y_train		f1_preds_train
1.030		1.866884
3.821		2.735653
1.726	-	1.866884
0.934		1.625823
0.965		2.735653

```
f1_preds_train = f0_preds_train + h0.predict(X_train)
```

New training dataset for training base model $\mathbf{h_1}$

c.p.e. f1 = {f0, h0}

X_train

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

y_train_1 = pseudo-residuals y-f1(x)

y_train		f1_preds_train
1.030		1.866884
3.821		2.735653
1.726	-	1.866884
0.934		1.625823
0.965		2.735653

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

y_train_1
-0.836884
1.085347
-0.140884
-0.691823
-1.770653

New training dataset for training base model $\mathbf{h_1}$

c.p.e. f1 = {f0, h0}

X_train

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

y_train_1 = pseudo-residuals y-f1(x)

y_train		f1_preds_train
1.030		1.866884
3.821		2.735653
1.726	-	1.866884
0.934		1.625823
0.965		2.735653

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

y_train_1
-0.836884
1.085347
-0.140884
-0.691823
-1.770653

```
h1 = DecisionTreeRegressor(max_depth=3, random_state=42)
h1.fit(X_train, y_train_1)
```

New training dataset for training base model $\mathbf{h_1}$

c.p.e. f2 = {f0, h0,h1}

X_train

MedInc	HouseAge	AveRooms	AveBedrms
3.2596	33.0	5.017657	1.006421
3.8125	49.0	4.473545	1.041005
4.1563	4.0	5.645833	0.985119
1.9425	36.0	4.002817	1.033803
3.5542	43.0	6.268421	1.134211

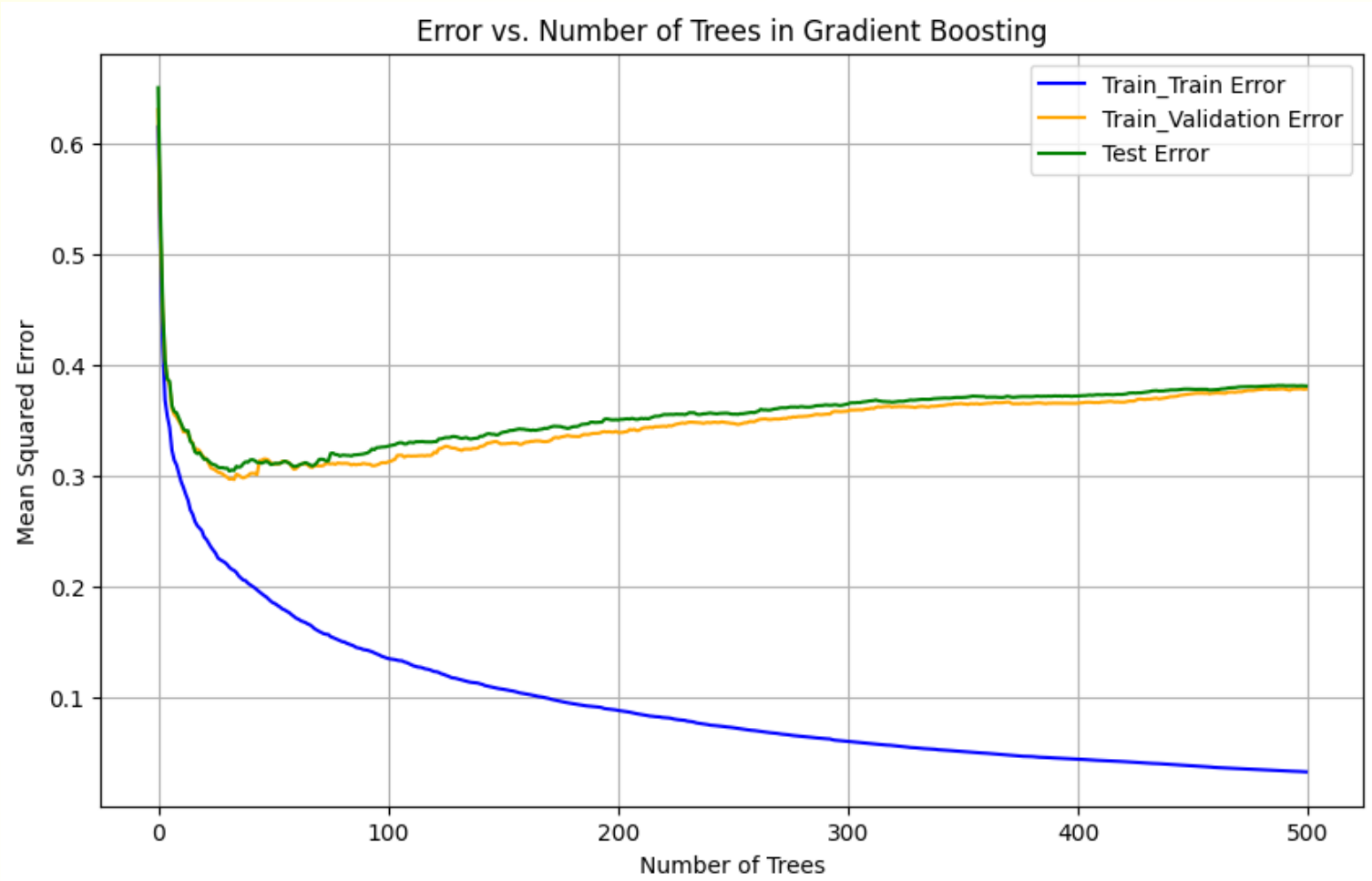
y_train_1 = pseudo-residuals y-f1(x)

y_train_1
-0.836884
1.085347
-0.140884
-0.691823
-1.770653

Current partial ensemble f2: {f0, h0, h1}
Making predictions: f1(x) = f0(x) + h0(x) + h1(x)
Typically, an optimized weight c1 is also used: f1(x) = f0(x) + c0 * h0(x) + c1 * h1(x)

Gradient Boosting

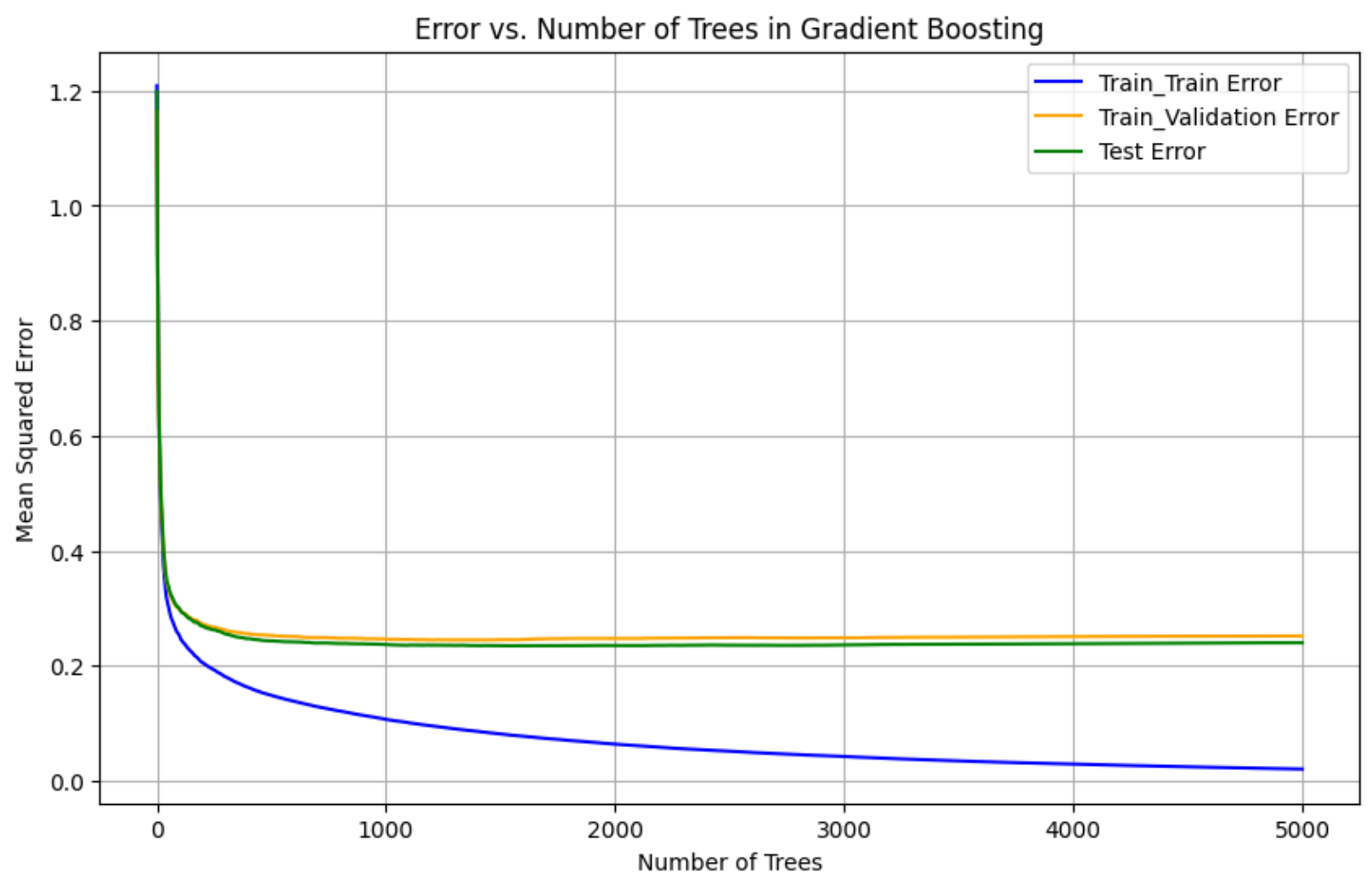
- Warning: possible overfitting if the number of base models is large.
- GB may use some of the models for memorizing noisy instances.
- E.g. California dataset



Boosting and overfitting

- Hyper-parameter T (number of trees): the larger, the more complex the model. Too complex models may result in overfitting.
  - **n_estimators** (in sklearn)
- **learning_rate** ($0< \alpha <1$): $f_t(x) = f_{t-1}(x)+c_{t-1}\ \alpha\ h_{t-1}(x)$
  - Less overfitting because each model *h* has less influence on the final result
  - small α require more trees in the ensemble (larger T)
  - Sometimes the learning rate α is also called *v*

- California dataset with learning rate = 0.1 (instead of 1.0)



Gradient Boosting and overfitting

- Hyper-parameter maximum depth of trees
- Stochastic gradient boosting:
  - At each iteration t, h_t is trained with a random sample (e.g. 80% of the training set) without replacement
  - Overfitting is avoided to some extent, because the training sample changes for every base model.
  - hyper-parameter **subsample** in sklearn

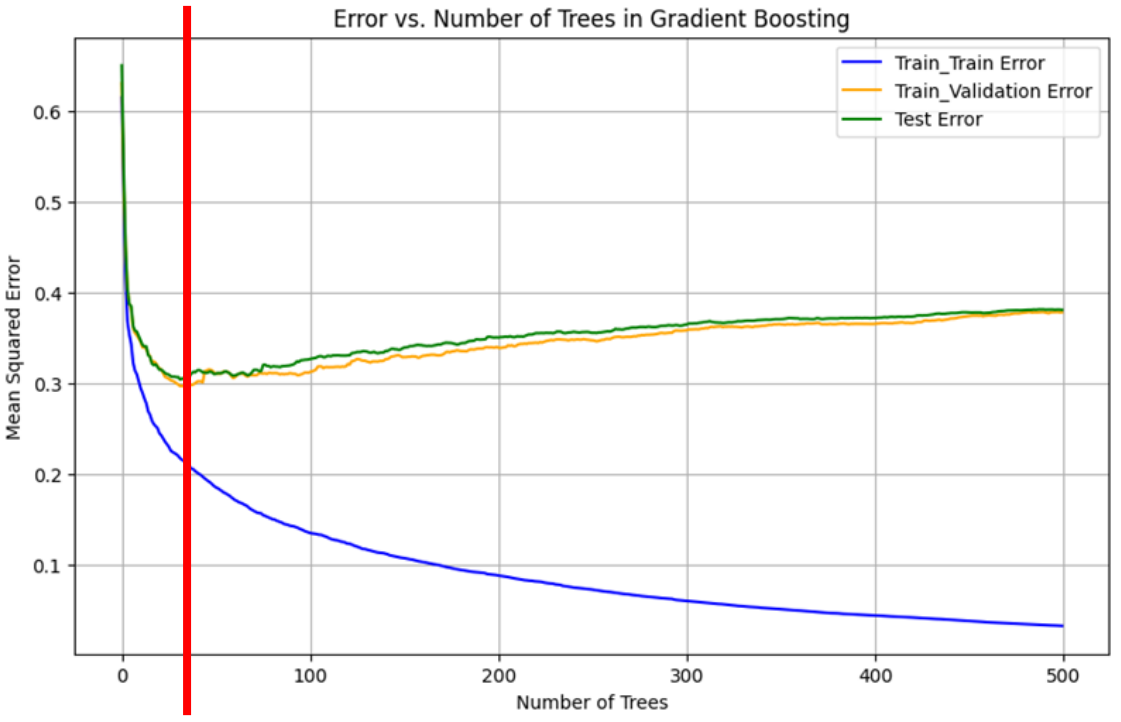
Gradient Boosting HPO

```
# Define the parameter grid
param_grid = {
    'max_iter': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'min_samples_leaf': [20, 40, 60]
}

# Initialize the HistGradientBoostingRegressor
hist_gb_reg = HistGradientBoostingRegressor(random_state=42)

# Initialize the GridSearchCV object with 3-fold cross-validation
grid_search = GridSearchCV(hist_gb_reg, param_grid, cv=KFold(n_splits=3),
                           scoring='neg_mean_absolute_error', verbose=1)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)
```

Gradient Boosting HPO with early-stopping

class sklearn.ensemble.**HistGradientBoostingRegressor**(*loss='squared_error', *, quantile=None, learning_rate=0.1, max_iter=100, max_leaf_nodes=31, max_depth=None, min_samples_leaf=20, l2_regularization=0.0, max_bins=255, categorical_features=None, monotonic_cst=None, interaction_cst=None, warm_start=False, early_stopping='auto', scoring='loss', validation_fraction=0.1, n_iter_no_change=10, tol=1e-07, verbose=0, random_state=None*) [source]

- early_stopping = True
- validation_fraction is the proportion of the training data used for determining the number of trees in the ensemble.
- n_iter_no_change: if no improvement for 10 iterations, stop growing the ensemble.
- A random subsample of the training data is used for validation. No other validation strategies can be used (e.g. no crossvalidation, no validation with no-shuffling, …)



GB HPO WITHOUT EARLY STOPPING

```
# Define the parameter grid
param_grid = {
    'max_iter': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'min_samples_leaf': [20, 40, 60]
}

# Initialize the HistGradientBoostingRegressor
hist_gb_reg = HistGradientBoostingRegressor(random_state=42)
```

GB HPO WITH EARLY STOPPING

```
# Define the parameter grid (excluding max_iter)
param_grid = {
    'learning_rate': [0.01, 0.1, 0.2],
    'min_samples_leaf': [20, 40, 60]
}

# Initialize the HistGradientBoostingRegressor with early stopping parameters
hist_gb_reg = HistGradientBoostingRegressor(max_iter=300,
                                            early_stopping=True,
                                            validation_fraction=0.1,
                                            n_iter_no_change=50,
                                            tol=0.01, random_state=42)
```

ENSEMBLE IMPLEMENTATIONS.

- In sklearn:
  - Bagging:
    - `ensemble.BaggingClassifier([base_estimator, ...])`
    - `ensemble.BaggingRegressor([base_estimator, ...])`
  - Faster:
    - `ensemble.RandomForestClassifier([...])`
    - `ensemble.RandomForestRegressor([...])`
    - `ensemble.ExtraTreesClassifier([...])`
    - `ensemble.ExtraTreesRegressor([n_estimators, ...])`
  - Boosting:
    - `ensemble.AdaBoostClassifier([...])`
    - `ensemble.AdaBoostRegressor([base_estimator, ...])`
    - `ensemble.GradientBoostingClassifier(*[, ...])`
    - `ensemble.GradientBoostingRegressor(*[, ...])`
  - Faster:
    - `ensemble.HistGradientBoostingRegressor([...])`
    - `ensemble.HistGradientBoostingClassifier([...])`

ENSEMBLE IMPLEMENTATIONS.

- External to sklearn (but they can be used from within sklearn)
  - Boosting:
    - xgboost:
      - https://www.kaggle.com/stuarthallows/using-xgboost-with-scikit-learn
      - https://xgboost.readthedocs.io/en/latest/
    - lightgbm:
      - https://lightgbm.readthedocs.io/en/latest/
    - catboost: (useful when some categorical attributes have many different values)
      - https://catboost.ai/

Gradient Boosting for classification

- Gradient Boosting is for regression
- But a classification problem can be transformed into a regression one if the response variable is 0 or 1 (for two-class problems). Similarly to logistic regression, the prediction of the model is going to be the probability of the class (rather than the class itself).
- Gradient Boosting minimizes the logistic loss (or log loss, or cross-entropy loss), for binary classification:

$$L(y,p) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1-y_i)\log(1-p_i)]$$

- And this can be achieved by computing pseudo-residuals as **$(y-f_i(x))$**, with $f_i(x)$ predicting the probability of class 1.

Gradient Boosting for classification

- If there are several classes, for each class, an ensemble can be trained to separate that class from the rest (one vs. all). The final class would be the class with maximum probability.
- E.g. for three classes C1, C2, C3:
  - e1(x) trained to separate C1 from {C2, C3}
    - e1(x) = prob(class(x) == C1)
  - e2(x) trained to separate C2 from {C1, C3}
    - e2(x) = prob(class(x) == C2)
  - e3(x) trained to separate C3 from {C1, C2}
    - e3(x) = prob(class(x) == C3)
  - Then class(x) = maxarg_Ci (e1(x), e2(x), e3(x))