

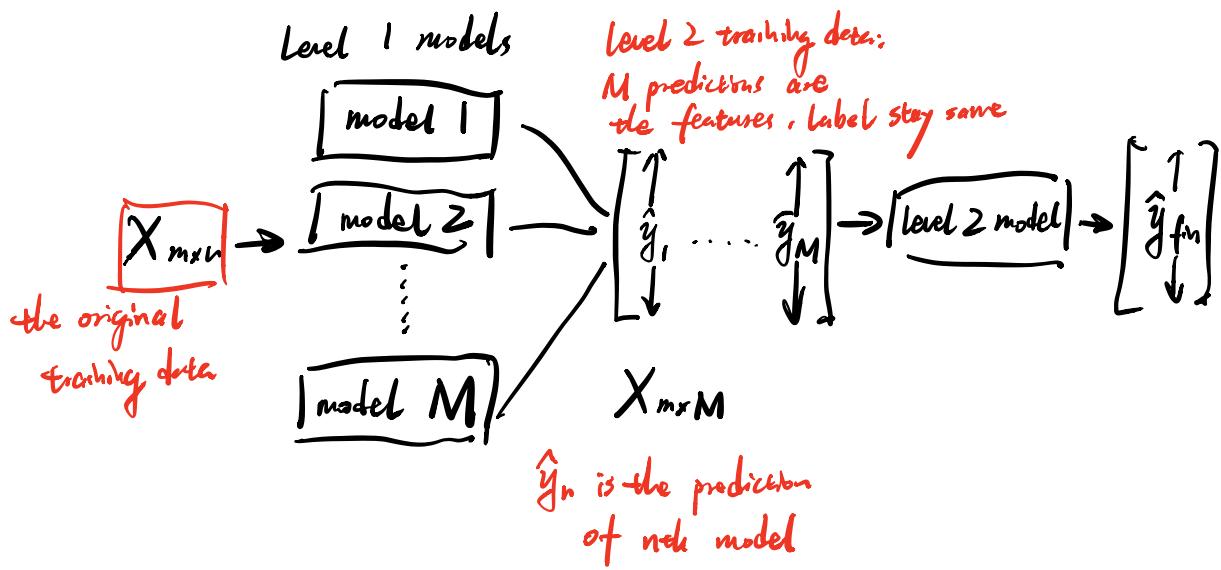
Ensemble Method:

1. Stacking

Taking a weighted combination of the predictions of a total of S different classifiers.

The weight can be a simple average, or learned via a secondary classification/regression process.

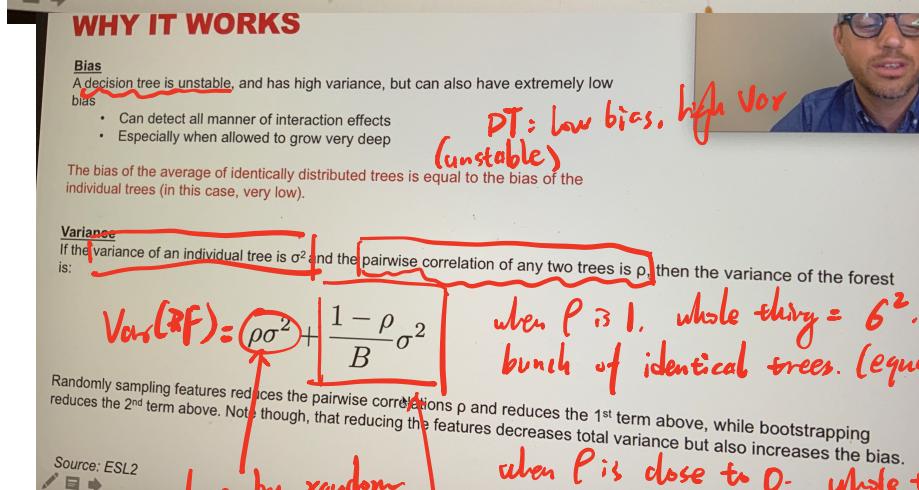
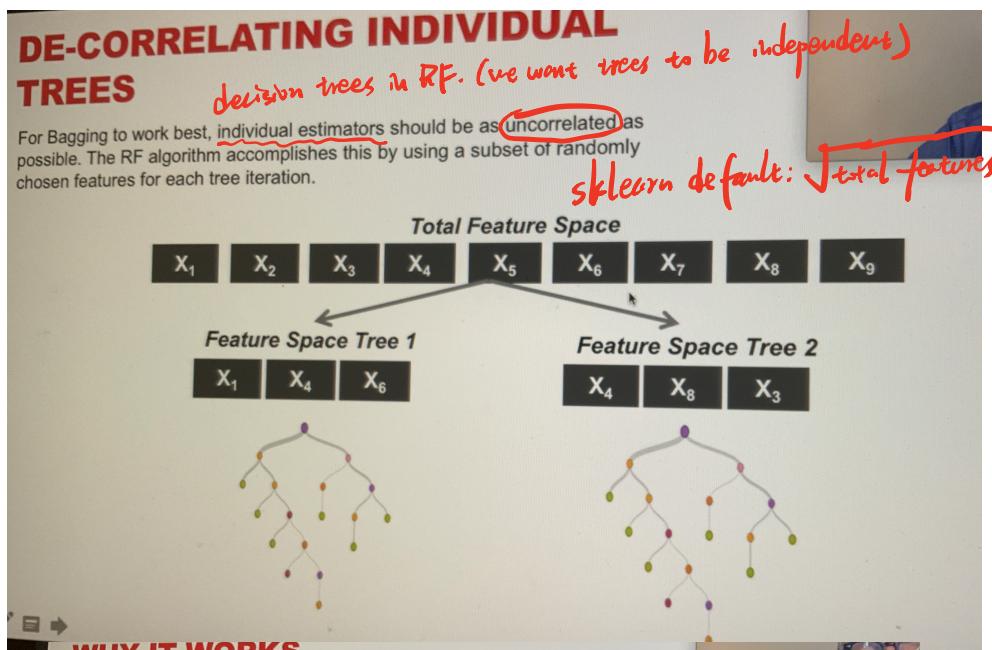
$$f(X_i) = \sum_{s=1}^S w^s \times f^s(X_i)$$



In order for learned stacking to generalize, the stage 2 model needs to be built from data that is separate from stage 1 models. Otherwise, the system risks overfitting.

Bagging: (Bootstrap + Aggregation)

1. create a bootstrap sample of the data
2. fit the model on the bootstrap sample
3. make the prediction on train/test data using the model
4. repeat step 1.2.3 for n times, storing each prediction
5. Make the final prediction by averaging the bootstrapped predictions.



reduce by
 selected features | $\frac{1}{B} \sigma^2$, decrease as $B \uparrow$ (more trees, less var)
 reduce by
 bootstrapping

TUNING

RF's are quick to set up and might do fairly well straight out of the box.
 But nonetheless, tuning is always recommended.



Forest Level Parameters

- # trees (`n_estimators`) – increasing this decreases variance, but increases training time.
- # of features to sample (`max_features`) – the number of features sampled in each tree.
 Reducing this # increases the bias but decreases the RF variance. sklearn default: \sqrt{d}

(don't need to tune a tree)

Tree Level Parameters

(we want to overfit individual tree)

- # intermediate nodes (`max_depth`) – the size of the tree. Usually you don't want to limit this (i.e., set `max_depth=None`) as deep as possible
- Size of intermediate notes (`min_sample_split`) – the number of instances in an intermediate node, before splitting (usually good to set to 1).

We usually want to over-fit the individual trees, and as always, use some hold-out method to optimize Forest level parameters.

Out-of-bag error (built-in RF validation property)

for each bootstrap sample. $\frac{1}{3}$ of sample were not selected to build the tree, so they serve as the validation set.

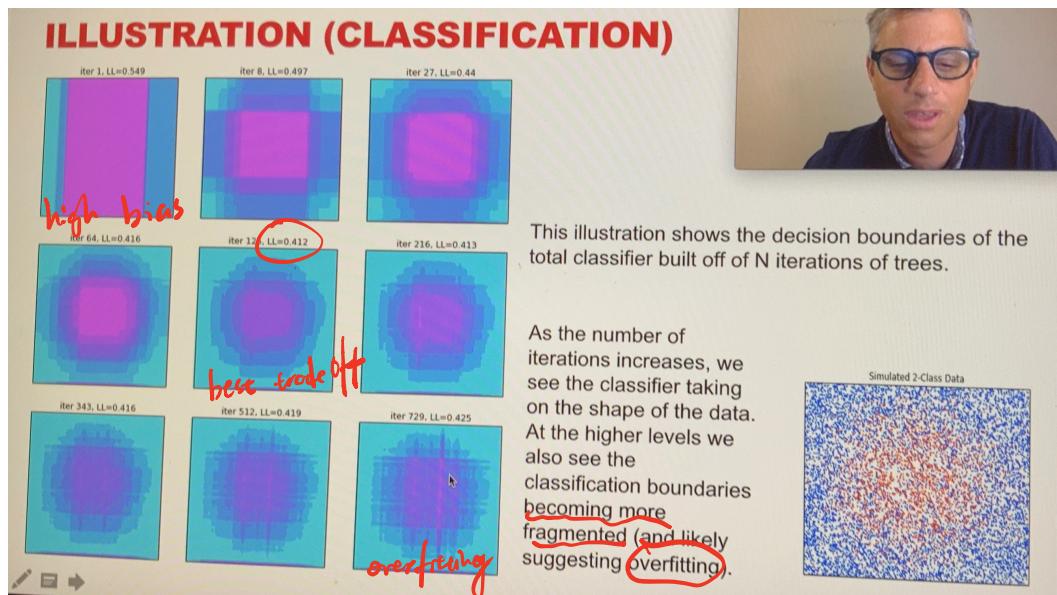
so RF on each OOB sample and average the error can give us the out-of-sample error estimation.

(RF has the built-in validation property called OOB error)

Gradient Boosting

We start with our best guess, then compute the error at each observation, and fit an DT to the residuals. The fitted DT becomes the next $f(x)$ that gets added to the ensemble.

Different from the RF, GBT is an iterative process, where each tree is fit on the residuals of the cumulative predictions from all prior trees.



Modern varieties : XGBoost, LightGBM, CatBoost

which one to use is an empirical question

kNN:

$$\hat{y}(x) = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i$$

or we can take the weighted average. weight can be the distance. (closer neighbor can influence more)

Property:

1. Non-parametric (the decision tree)
2. Instance-based learning (don't need to train)

Challenges:

1. Expensive Search (`sklearn.fur()` allows more efficient search)
2. Curse of dimensionality (when $d \uparrow$, all points are equally apart)
3. Scale matters (features with higher magnitude tend to dominate, need to normalize first)

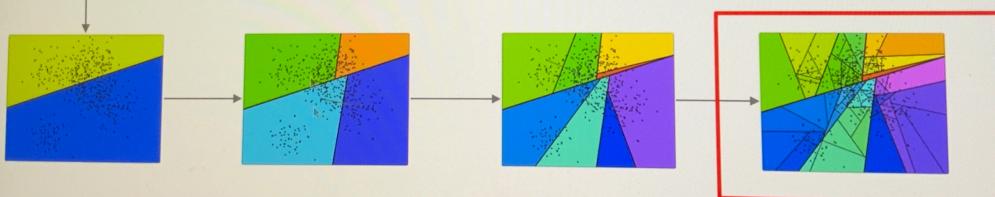
Approximate Nearest Neighbors $O(\log N)$ use binary tree

CREATING THE NEIGHBORHOODS

There are multiple approaches, so we'll use the ANNOY* package as an example



The core idea is we iteratively take random partitions of the X space. In step 1 we select two points at random, then identify the hyper-plane that is equidistant from the points. In step 2 we do the same, but perform this operation solely within each separate half of the previous step's hyper-plane. We continue this procedure until we've reached our designate stopping criteria.



SEARCHING FOR NEIGHBORS

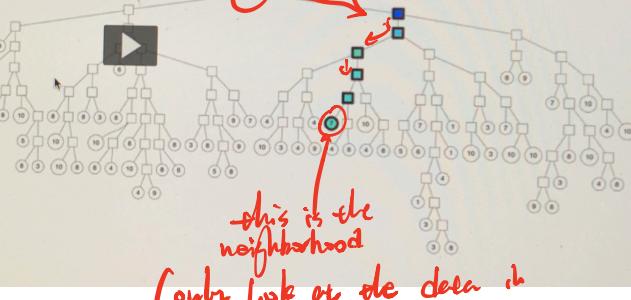


After K recursive splits, our partitioned data can be represented with a binary tree data structure. Finding a set of neighbors amounts to taking an input x' , traversing the tree to see which node it lands in, and taking the neighbors from that node.



$x_i \times \text{hyperplane } (+\text{ or } -)$.

this is the neighborhood
(only look at the data in this neighborhood)



Problem 1: you are limited to just the neighbors in the terminal node of the tree.

Solution 1: you don't have to use a single terminal node. Nearyby splits can also be considered

2. You can build a forest using multiple trees, with different starting points. Run the algorithm D times to get D separate neighborhoods for a point. When scoring, the candidate set becomes the union of all D neighborhoods.

THE TRADEOFFS



Problem: You may miss true nearest neighbors that are on the boundaries of the partitions

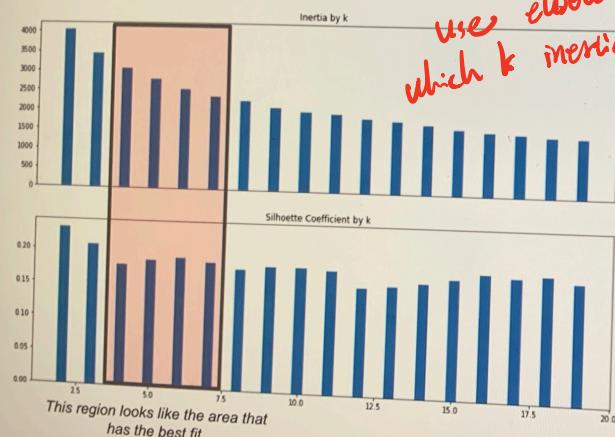


Solution: You can build a forest using multiple trees. Run the full algorithm D times to get D separate neighborhoods for a point. When scoring, the candidate set becomes the union of all D neighborhoods.

Clustering (do dimension reduction first \Rightarrow stable clusters)

Quantify Goodness of fit to find K (# of clusters)

WE CAN QUANTIFY GOODNESS OF FIT TO IDENTIFY THE RIGHT K



use elbow method
which k inertia decreases the most

The inertia measures the within cluster sum of squares distance from the mean.

$$\text{inertia} = \sum_{j=1}^k \sum_{x_i \in C_j} \|x_i - \mu_j\|^2$$

within cluster distance

a = the mean distance between a sample and all other points in the same cluster

b = the mean distance between a sample and all other points in the nearest cluster

$$s = \frac{b - a}{\max(a, b)}$$