Ensemble Models

Multiple models used together.

$\{M_1, M_2, M_3, M_4 \dots M_k\}$ —combine→ More powerful Model.

4 types of ensembles

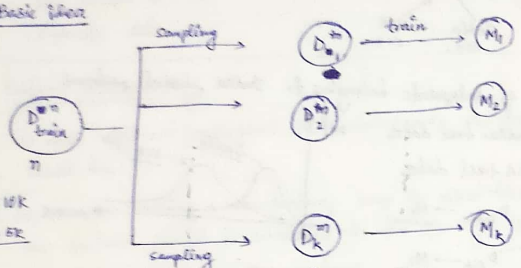
1. Bagging (Bootstrapped Aggregation).
2. Boosting
3. stacking
4. cascading

Mostly used in kaggle as well as Real world scenarios.

Key Aspect → More different the models are, the better & powerful it becomes.

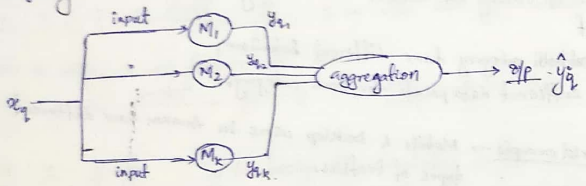
Bagging (Bootstrapped Aggregation).

Basic idea



Training Time

* sampling done with replacement.



Runtime

Aggregation → classification → majority vote
Aggregation → regression → mean / median

variance → degree of change in model with change in data.

* Say we changed 100 pts. of 10K.
* Only a subset of the datasets will get affected.
* That also to very little extent.
* So, overall model doesn't change much.

Bagging → can reduce variance in a model without impacting the bias.

Model error = $Bias^2$ + Variance.

If the base models have low bias + high variance, then bagging is just the perfect ensemble.

B

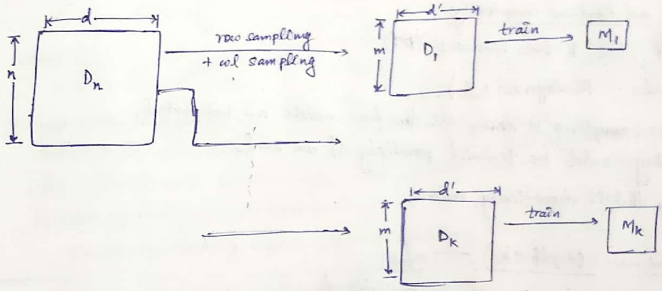
Bagging (Mi's) → low-bias; reduced variance.

Ex → DT of good depth → low bias & high variance.
bagging → Random forests

Random forests

Random bootstrap sampling. — collection of decision trees

Random Forest → DT (base model) + Bagging + column sampling.
↓
feature bagging.



* Sampled datasets now have different sets of features each.

Mi's → Decision Trees of reasonable depth → (low bias + high variance).

Model M_i uses D_i for training.

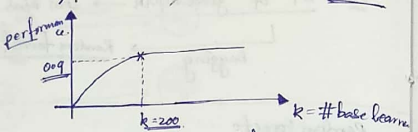
So, $(D_n - D_i)$ used as cross-validation set.
↳ Out of Bag (OOB) samples.

RF → very powerful.

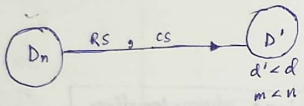
Bias-variance trade-off

M = aggregat^n($m_1, m_2 \cdots m_k$).

found via cross-valid^n.



* $k \uparrow$, variance $\downarrow$.
* bias(M) ≈ bias(m_i)



$\frac{d'}{d}$ = col. sampling Rate.

$\frac{m}{n}$ = row. sampling rate.

low col. SR ↓, row SR ↓ → variance ↓.

* Usually people fix CSR & RSR and keep increasing no of models to reduce variance.

Train and Runtime Complexity

Ex → RF with k-base learners (DT)

Train – O(n log n × d × k).

* Once sampling is done, as the base models are independent, they could be trained parallely if we want.
So, that'll dramatically reduce.

Runtime → O(depth * k) — v.fast

This can also be parallelized.

Space complexity → O(DT * k) ⟵ — convinient as DT ⇒ if else cond^n.

Extremely Randomized tree

Only change fm RF → while handling real value features, instead of trying each & every value after sorting, ~~try~~ for threshold, try from a random sample of numbers. to make a bit faster & introduce some randomization.

Boosting

reduce var.

Bagging – high variance, low bias (base-models) + randomization (CS, RS) + aggregation.

Boosting – low-variance, high bias + additively combine.

reduce bias while keeping var low

core idea – reduces bias.

Step 0 :– $D_{train} \{x_i, y_i\}_{i=1}^n \longrightarrow (M_0) \longrightarrow$ (high bias) i.e. high train error.

$y_i = h_0(x_i)$

Init error$_i$ = $y_i - F_0(x_i)$.

$F_0(x) = h_0(x)$.

↳ Model at end of step 0.

for any step k :→

$D \{x_i, e_i\}_{i=1}^n \longrightarrow (M_k)$

$e_i = h_k(x_i)$

where $e_i = y_i - F_{k-1}(x_i)$

$F_k(x) = \sum_{j=0}^{k} \alpha_j h_j(x)$ → Model at end of stage k.

additive weighted models

* since at each stage we are training the base model on the residual error of prev stage, it ends up having low residual error.

each base model $h_i(x)$ trained to fit the residual error of the previous stage.

examples of boosting algo → Gradient Boosting, AdaBoost, etc.

Residuals, loss functions and Gradients

Residual at end of stage k :→ $e_i = y_i - F_k(x_i)$.

Train model M_{k+1} by fitting its function h_{k+1} to $\{x_i, e_i\}_{i=1}^n$

* Assuming regression, we define loss at end of stage k as

$L(y_i, F_k(x_i)) = (y_i - F_k(x_i))^2$

$\frac{\partial L}{\partial F_k(x_i)} = 2(y_i - F_k(x_i)) \times (-1)$.

$\Rightarrow \boxed{\frac{-\partial L}{\partial F_k(x_i)} = 2(y_i - F_k(x_i))}$

* negative-gradient ≈ pseudo-residual

↓

Gradient Boosting utilizes this.

So, what gradient boosting does is

$$D \longrightarrow M_K$$
$$\{x_i, e_i\}_{i=1}^{n} \qquad e_i = h_K(x_i).$$

where e_i = pseudo-residual i.e. $\frac{-\partial L}{\partial F_{k-1}(x)}$.

* This allows us to use any differentiable loss function.

So, we can minimize hinge loss using ~~regression~~ decision trees.

* So, Gradient Boosted Decision Trees (GBDT) tend to perform better than random forests.

Refer → Gradient Boosting (wikipedia article), for exact algo.

Input: Training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M (no of base models).

Algorithm:-

1. Initialize model with constant value:-

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

If L is sq-loss $\delta = \text{mean}(y_i)$.

2. For $m = 1$ to M:

1. compute so called pseudo-residuals

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

2. Fit a base learner (e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$

3. Compute multiplier δ_m by solving the following 1-dimension optimization problem

$$\delta_m = \underset{\delta}{\operatorname{argmin}} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \delta h_m(x_i))$$

4. update the model:-

$$F_m(x) = F_{m-1}(x) + \delta_m h_m(x).$$

3. Output $F_M(x)$

Regularization and Shrinkage

At the end of the algorithm, we'll get

$$F_M(x) = h_0(x) + \sum_{m=1}^{M} \delta_m h_m(x)$$

M :- # base models. (hyperparameter).

M↑ => overfit↑. => variance↑.

Optimal 'M' could be found by cross validation and GridSearch or RandomSearch.

Alternatively Shrinkage

$$F_m(x) = F_{m-1}(x) + \nu \delta_m h_m(x) \qquad 0 < \nu \leq 1$$

ν → learning rate.

ν↑ => overfit↑ => variance↑.

ν↓ => variance↓.

So now we have 2 hyper-parameters :- M & ν.

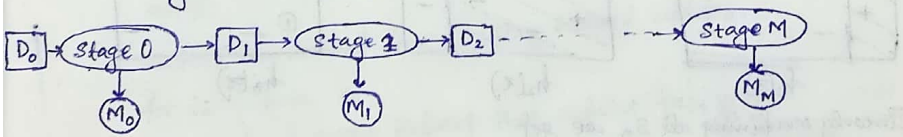
CV with GridSearch / RandomSearch (M, ν).

Train & Runtime Complexity (GBDT)

Train :- O(n log n * d * M). M :- # base-learners.

unlike RF, GBDT not parallelizable.

Boosting is a serial algorithm.



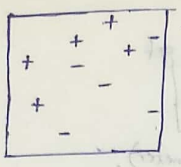
Runtime :→ O(depth * M) Space :- O(each tree * M + M) ← δm

~~Imp~~ AdaBoost

└ Special case of Gradient Boosting. See - university of pennsylvania example.

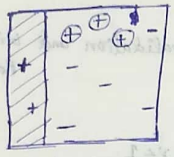
Application :- face detection.

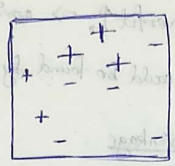
Core idea :→ Increase wt of misclassified points.



— original training data.
lets say we train DT of depth = 1 (high bias, low variance).

1st stage

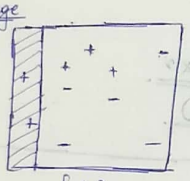


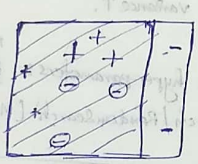


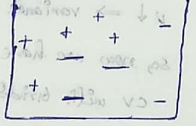
exponentially increased wt of misclassified pts.

$h_1(x)$

2nd stage

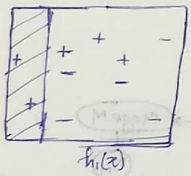


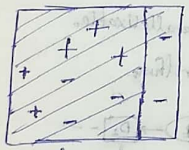


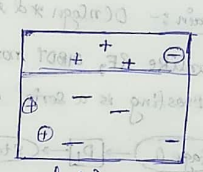


$h_1(x)$ $h_2(x)$

3rd Stage

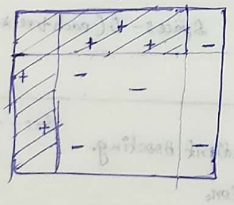






$h_1(x)$ $h_2(x)$ $h_3(x)$

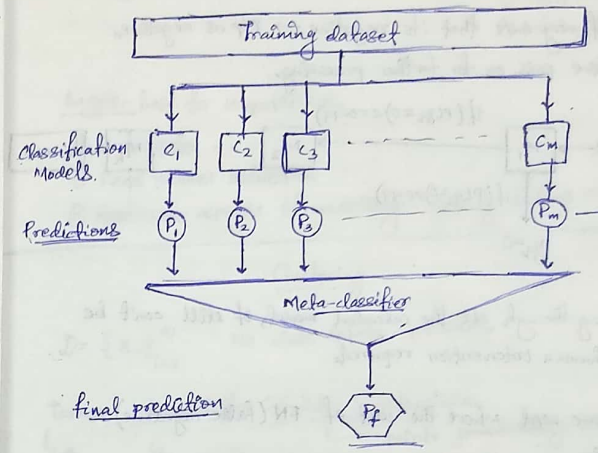
linearly combining all 3, we get



GBDT more often used in internet companies than AdaBoost.

Stacking Models

library - mlxtend.classifier - see official docs for details.



Algorithm: Stacking

Input :- training data $D = \{x_i, y_i\}_{i=1}^{n}$ $(x_i \in \mathbb{R}^m, y_i \in Y)$

Output :- An ensemble classifier H.

1. learn first-level classifiers (base models).
   for $i \leftarrow 1$ to T do
   Learn a base classifier h_i based on D.
   end for.
2. Construct new dataset from D using trained models.
   for $i \leftarrow 1$ to m do
   construct a new dataset that contains $\{x_i', y_i\}$
   where $x_i' = \langle h_1(x_i), h_2(x_i), h_3(x_i), \ldots h_T(x_i) \rangle$
   end for.
3. learn a second-level classifier. (meta classifier).
   learn a new classifier h' based on the newly constructed dataset.
   return $H(x) = h'(h_1(x), h_2(x), h_3(x), \ldots h_T(x))$

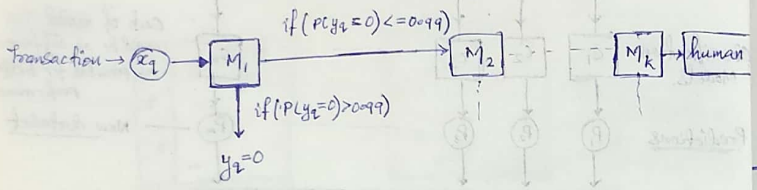
* $h_i(x)$ - can be — o/p label / probability value.

Kaggle uses stacking extensively.

Cascading Models

Ex → predicts credit card transaction is fraud or not.

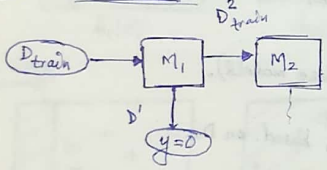
core idea:- If very sure that its negative → o/p as negative.
else pass on for further processing.



After processing through all the cascaded models if still can't be decided, human intervention required.

These models are used where the cost of FN (false negative) is very high.

Ex → Cancer detection



$D^2_{train} = D_{train} - D'$ and so on.

There could be more complex architecture also for cascade models.

Kaggle Vs Real World

① Kaggle cares about a single metric:- log-loss, F1-score, AUC, etc.
Real world - multiple metrics with different priorities.

This is because in real-world we try to solve a business problem i.e. increase sales, etc.
So, we map this biz. metric → multiple ML metrics.

② Kaggle → very complex ensembles. → goal to improve 1 metric.
This might be impractical in real world due to ① low latency
② interpretability
③ training time.

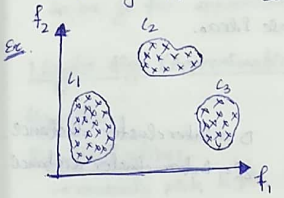
Kaggle - best for competitions.
① Participate
② Read winner solutions.
③ Learn aspects of modelling - cleaning, feature engg, EDA, etc.

Clustering

$D = \{x_i\}_{i=1}^{n}$ No class labels provided.

Task:- group/cluster "similar" datapoints.
→ points geometrically close to each other.

Ex.



(a) points in a cluster are close together.
(b) points in different clusters are far away.

How to measure how well the algo performed?
→ All prev metrics are based on y_i's.

Most used clustering →
① K-means
② Hierarchical.
③ DB-scan.

Unsupervised Learning

Classification & Regression → Supervised learning.
$\{x_i, y_i\}$ the labels supervised/guided the learning process.

clustering → unsupervised learning, No y_i available.

Semi-supervised learning → $D = D_1 \cup D_2$
$|D_1| \ll |D_2|$
$\{x_i, y_i\}$ small
$\{x_i\}$ large

Application of clustering — data mining

① e-commerce → Group similar customers based on their purchasing behavi
so as to offer deals to a group of similar customers
who are more likely to take the deal.

② Image segmentation → grouping/clustering similar pixels.
forms base of object detection.

③ Data labelling → cluster the data into 2 groups.
check some instance of 1 group to know whether
its positive/negative class.
saves a lot of time.

Metrics for clustering

All the metrics have the same common base idea.

① Maximize inter-cluster distance.

② Minimize intra-cluster distance.

D = intercluster distance
d = intra-cluster distance

One example

Dunn Index →

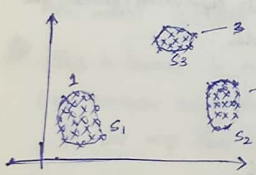
$$= \frac{\min D}{\max d}$$

D is high => good clustering.

K-means clustering

k:- # clusters. — hyperparameter, found similar to CV.

* Each cluster has a centroids.



k=3

$S_1 \cup S_2 \cup S_3 = D$

$S_1 \cap S_2 = \phi$

$S_2 \cap S_3 = \phi$

$S_1 \cap S_3 = \phi$

$$c_i = \frac{1}{n} \sum_{x_j \in S_i} x_j$$

centroid of S_i

K-means — centroid based clustering.

Others are ① density based clustering.
② hierarchical clustering.

* Once we get the k-centroids, each point is assigned to the nearest centroid.

Mathematical formulation

Objective function →

$$\underset{c_1, c_2, \cdots c_k}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x \in S_i} \|x - c_i\|^2$$

It basically says that find centroids such that intracluster distance are minimized.

such that $S_i \cap S_j = \phi$.

* This problem is NP-Hard. Exponential time complexity.
So we go for approximation algorithms.

Lloyd's Algo (Approximation)

Algorithm →

① Initialization:-
→ randomly pick k pts from D and call them $c_1, c_2 \cdots c_k$.

② Assignment —

```
for each point x_i in D
    select the nearest c_j.
    Add x_i to set S_j.
```

③ Recompute centroid / update

recalculate/update c_j's as follows.

$$c_j = \frac{1}{|S_j|} \sum_{x_i \in S_j} x_i$$

④ ~~Recompute centroid update.~~
→ ~~recalculate/update~~

⑤ Repeat ② & ③ untill convergence: — no significant difference in c_i's

K-means — initialization sensitive.

└ final clusters depend on initialization. — Refer slides.

One solution

→ Repeat k-means multiple times with different initializations.

Pick the best clustering based on

* smaller intracluster distance.
* larger intercluster distance.

Alternate solution (K-Means++)

① pick the first centroid randomly from D (c_1).

② for each non-centroid point in D, calculate distance of x_i to its closest centroid (d_i).

③ Pick a point out of all these points with a probability of d_i. & nominate it as next centroid.

④ Repeat ② & ③ to get all the centroids.

* We don't directly select the pt with maximum distance so as to avoid impact of outliers.

Limitations of K-Means — refer slides.

* K-means tries to create clusters of same sizes. (density may vary).

① clusters of different sizes.

② clusters of different density.

③ non-convex shapes.

Overcoming limitations

① clusters of different sizes → increase K↑ and then again re-group clusters.

K-medoids

* centroids shouldn't compulsorily be one of the training set points.

* Also, they may mend up having fractional components which may make them less interpretable.

Ex → BoW

* Idea of k-medoids is to select one of the training set points which is close to the actual centroid.

Partitioning Around Medoids (PAM)

① Initialisation → similar to K-Means++ probabilistic method.

② Assignment → same ✓ → assign to closest medoid.

③ update →

swap each medoid with each non-medoid point.

If loss decreases, do nothing, else undo the swap.

Loss is given by

$$\sum_{i=1}^{k} \sum_{x \in S_i} \|x - m_i\|^2$$

m_i ← medoid

Advantage of medoids

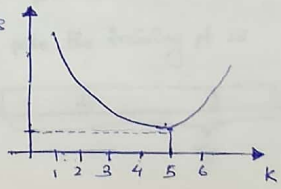
① Interpretable

② can use similarity matrix also, hence kernelization.

Determining the right 'K'

① domain-knowledge — If we know only two classes possible (+)ve & (−)ve. Then, no of clusters = 2.

② elbow method



Time complexity

k-means - $O(n * k * d * i)$

- n: #pts
- d: dim
- k: #clusters
- i: #iterations

$\approx O(nd)$

$k \leq 10$ typically.

$i \leq 300$

Space complexity - $O(nd + kd)$

$\approx O(nd)$