CS3244 : Machine Learning

Semester 1 2023/24

Lecture 3 : kNN, k-d Tree, Decision Tree, Random Forest, Gradient Boosting

Xavier Bresson

https://twitter.com/xbresson

Department of Computer Science

National University of Singapore (NUS)

Material used for preparation

- Prof Kilian Weinberger, CS4780 Cornell, Machine Learning, 2018
  - https://www.cs.cornell.edu/courses/cs4780/2018fa
- Prof Min-Yen Kan, CS3244 NUS, Machine Learning, 2022
  - https://knmnyn.github.io/cs3244-2210

Outline

- kNN
- k-d tree
- Decision tree
- Bagging
- Boosting
- Conclusion

Outline

- kNN
- k-d tree
- Decision tree
- Bagging
- Boosting
- Conclusion

kNN

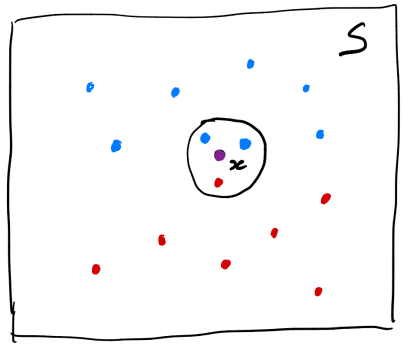
- kNN algorithm
  - Assumption : Close data points have similar labels, i.e. class or regression value.
  - Algorithm : For a test data point x, assign the most common labels in its k nearest neighbors in the training set S.
  - Formalization
    - kNN classification :

$$f_k(x) = \text{mode}\big(\{y : (x,y) \in S_x^k\}\big)$$

$$\text{with } S_x^k = k - \text{nearest neighbor of } x \text{ defined as}$$

$$S_x^k = \{x' : \max_{x' \in S_x^k} \text{d}(x,x') \leq \text{d}(x,x''),\ \forall x'' \in S \setminus S_x^k\}$$

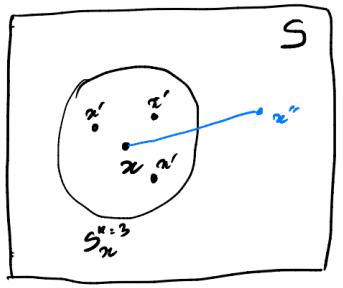
$$\text{and } \text{d}(x,z) = \sum_i \big(|x_i - z_i|^p\big)^{1/p},\ p = 1 \text{ (Manhattan)},\ p = 2 \text{ (Euclidean)}$$



kNN binary classification with k=3
x belongs to blue class

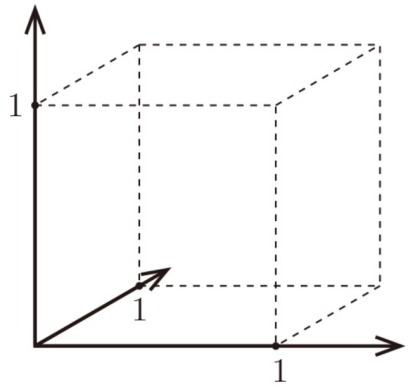
    - kNN regression :

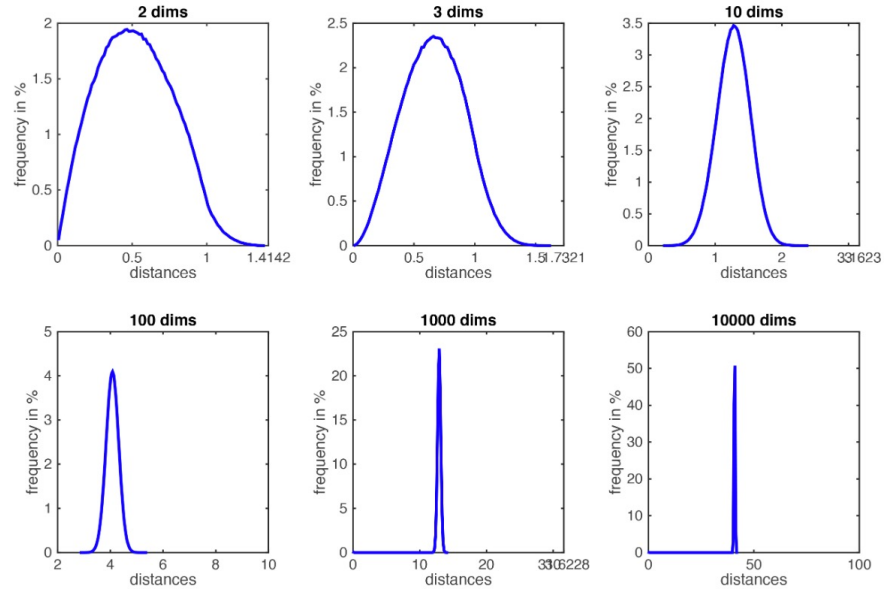
$$f_k(x) = \text{mean}\big(\{y : (x,y) \in S_x^k\}\big)$$



Curse of dimensionality

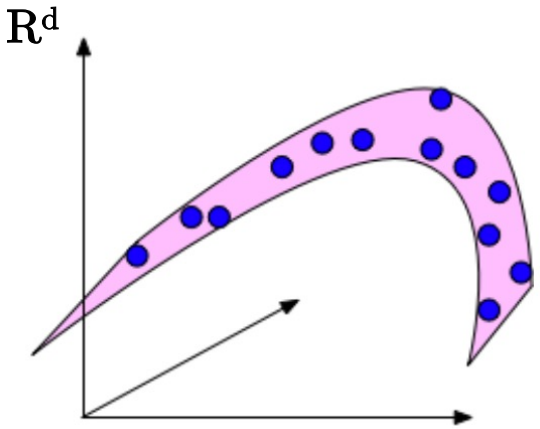
- Euclidean/L^p distances between points :
  - k-NN assumption "close data points have similar labels" works if we can define a meaningful distance between two data points.
  - Unfortunately, in high-dim spaces, data points sampled from a random probability distribution, are far from each other with (almost) the same distance value.
    - Let us sample points uniformly at random within the unit cube and let us the compute the distance between all pair of points when the dimensionality increases.





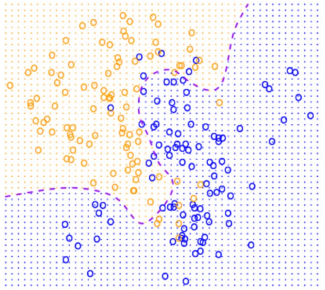
Blessing of structure

- Real-world data does not follow a random probability distribution!
  - Data has structure, s.a. edges, textures for natural images.
  - This means that data lie in a much lower dimensional sub-space than R^d.
    - For example, images of human face can be accurately described with e.g. 50 features s.a. male/female, blond/dark hair, etc, although the original image lie in a space of 1M dimensions (1,000 x 1,000 pixels).



kNN summary

- kNN algorithm is the simplest machine learning technique for classification (binary and multiple classes) and regression.
  - It is expressive as it can produce non-linear boundary decision.
- As n→∞, kNN becomes provably very accurate, but also intractable.
  - Real-world applications have a limited n number of training data.
- As d→∞, curse of dimensionality kicks in and kNN breaks for Euclidean distances.
- kNN works if distances are semantically meaningful.
  - Combining kNN and deep learning representation is today a strong baseline.



Outline

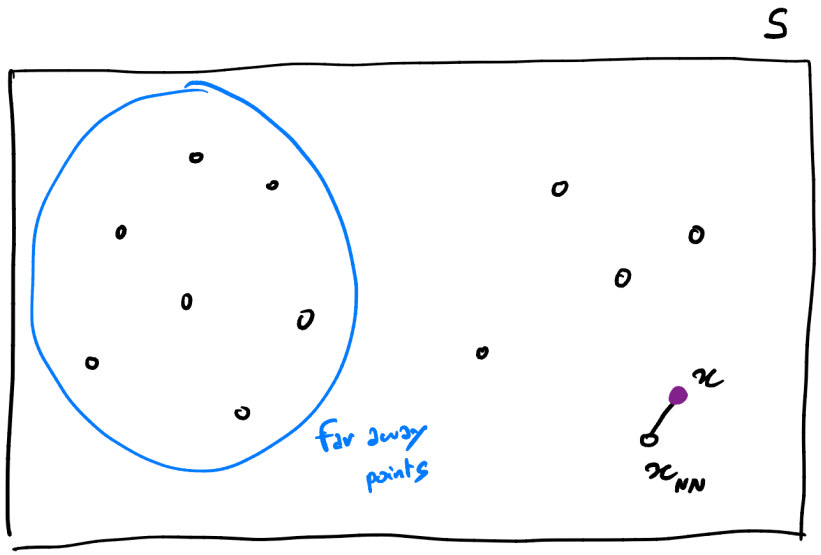
- kNN
- **k-d tree**
- Decision tree
- Bagging
- Boosting
- Conclusion

kNN complexity

- kNN time complexity is O(n.d.k), where n the number of training data, d the number of data dimensions and k is the number of k nearest neighbors.
  - This complexity means that kNN becomes very slow and memory consuming when n is large but we want to have n as large as possible to get the best possible accuracy.
  - Can we improve the speed? Yes, by leveraging data structure.

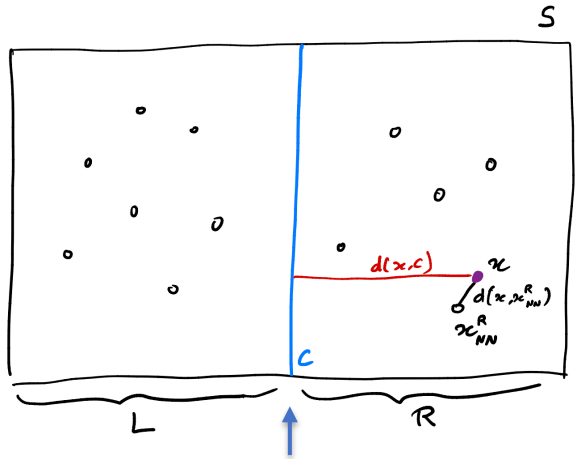
k-d tree

- General idea : When we search for the closest point(s), most data points are actually far away and hence there is no need to compute the distances for these far away points.
- How to achieve this goal?
  - Solution is to partition the d-dim feature space with a binary tree structure.

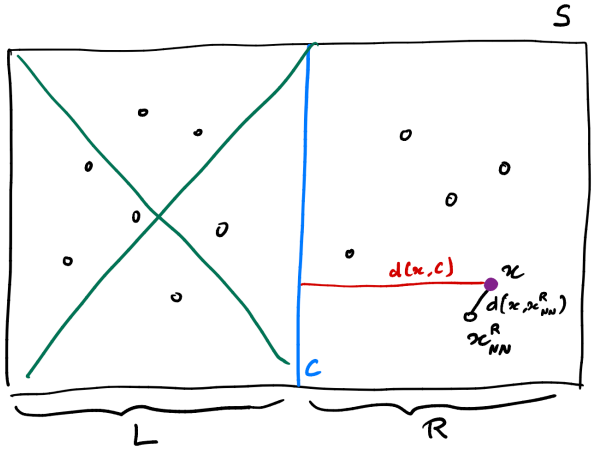


k-d tree speeds up kNN

- For example, let us consider the full dataset and one partition as follows:
  - Make a cut along one feature dimension that divides the data into two sets, i.e. Left and Right, with approximatively the same number of data in each half.
  - Consider a new data point x, or which we want to find the closest neighbor.
  - Identify which set the data x lies, here the right set R.
  - Find the nearest neighbor x_{NN}^{R} in R, it requires O(n/2).
  - Compute the distance d(x,C) between x and the cut C.
  - If $d(x,C) > d(x, x_{NN}^{R})$ then all x in L can be discarded (by triangular inequality) and kNN gets a 2x speed-up!



Cut/split the space S into 2 sets R and L with ≈ the same number of points



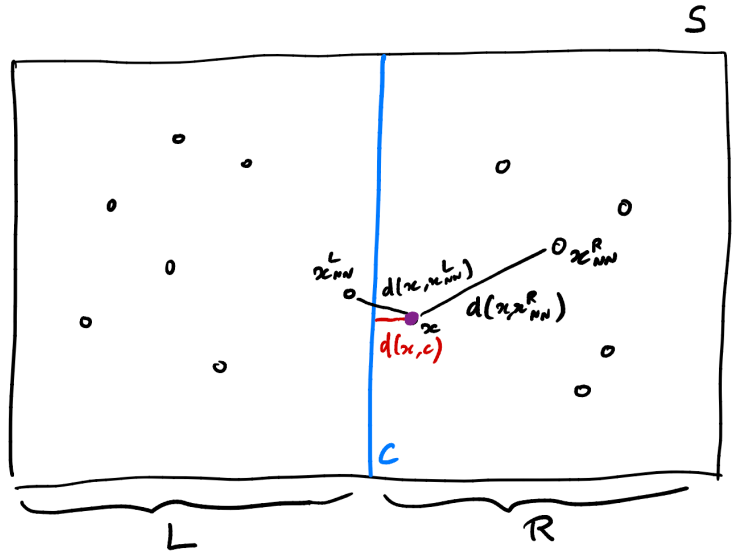
$$d(x, x_{NN}^{R}) < d(x, C)$$

$$\Downarrow$$

All points in L cannot be NN, we can discard/prune the space by a factor 2.

k-d tree speeds up kNN (on average)

- Q: What happens if $d(x,C) < d(x, x_{NN}^{R})$?
  - It is possible that the NN lies in L – so we need to compute all distances $d(x,x^{L})$.
  - Speed complexity is then O(n), same as kNN.
  - Worst case complexity of k-d tree is kNN complexity, but it is actually much better in practice (average complexity).



$$d(x, x_{NN}^{R}) > d(x, C)$$

$$\Downarrow$$

We need to compute the distance to all points in L.

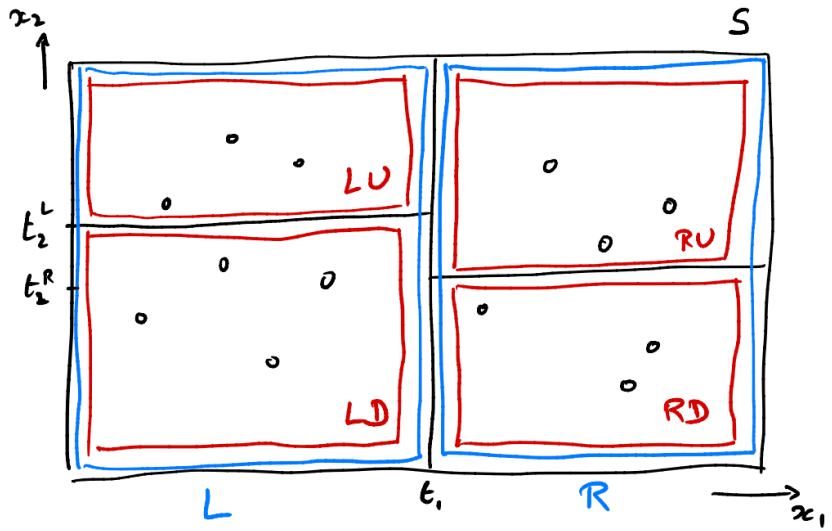
$$\text{if } d(x, x_{NN}^{R}) < d(x, x_{NN}^{L}) \text{ then } x_{NN} = x_{NN}^{R},$$

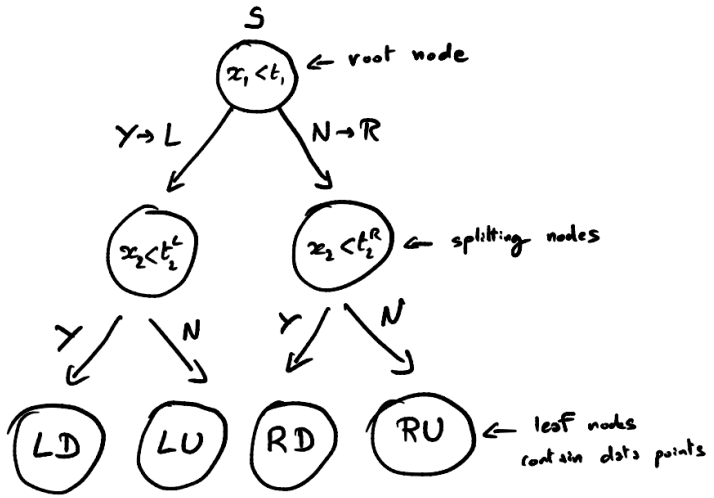
$$\text{otherwise } x_{NN} = x_{NN}^{L}.$$

Complexity is O(n).

k-d tree structure

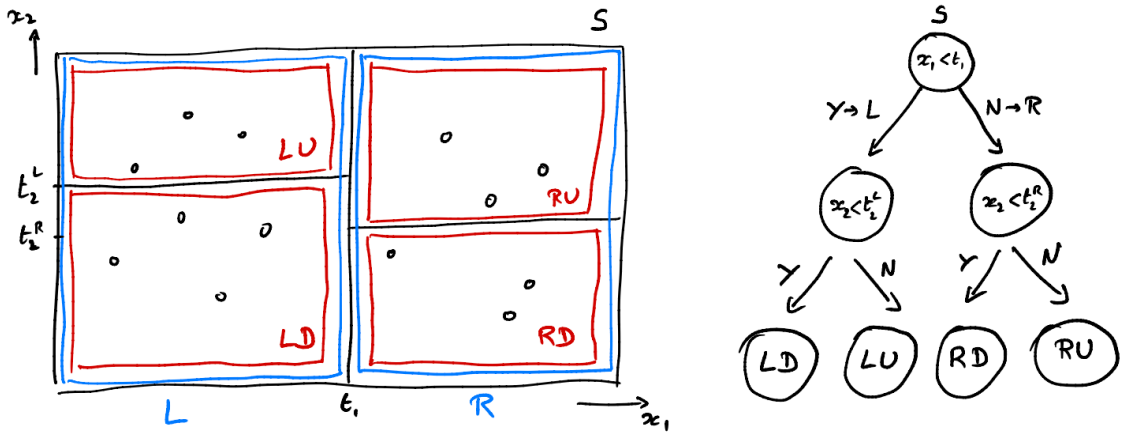
- Tree construction / training stage
  - Split recursively in half along each feature dimension.
  - Iterate over all feature dimensions.
  - Tree depth is quite small depth = $O(\log_2 n)$, e.g. $\log_2 10^3 \approx 10$, $\log_2 10^6 \approx 20$, $\log_2 10^9 \approx 30$
- How to select which next feature dimension?
  - A good heuristic is to select the feature dimension that captures the largest variation of data (similar to PCA).





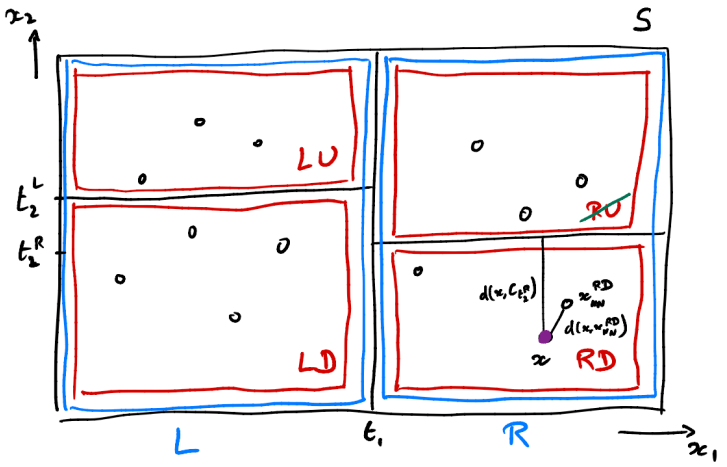
k-d tree search

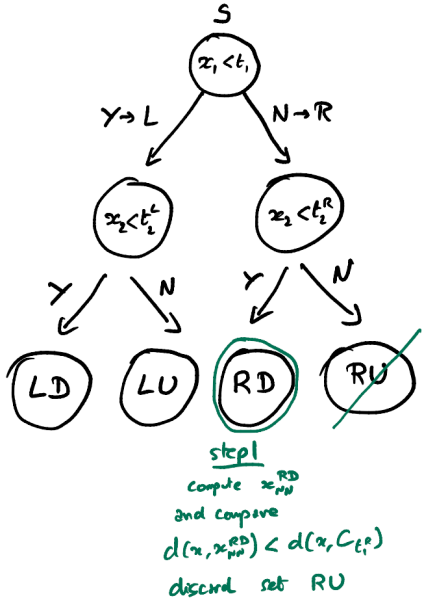
- Example of NN search process in 2D
  - Which order of search?
  - Which partitions to prune?

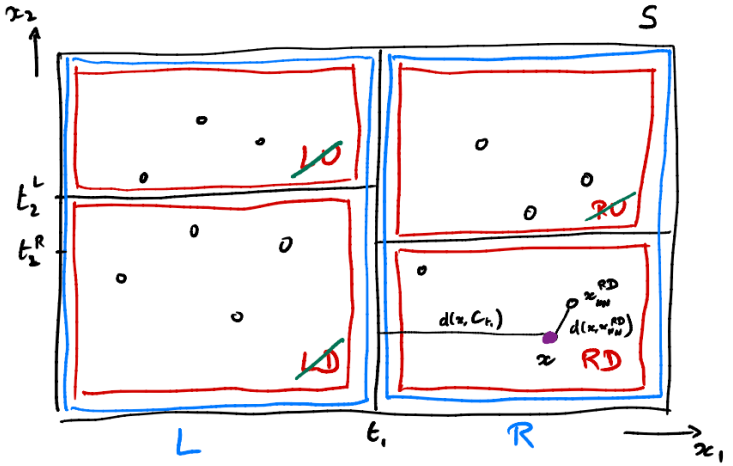


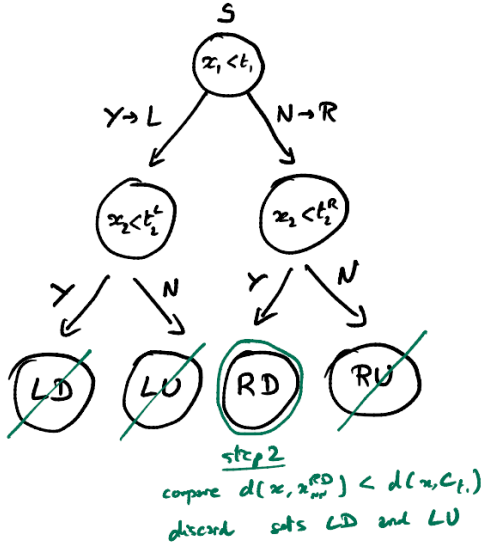
k-d tree search #1

- Example of NN search process in 2D
  - Which order of search?
  - Which partitions to prune?
  - Best case scenario O(n/4)



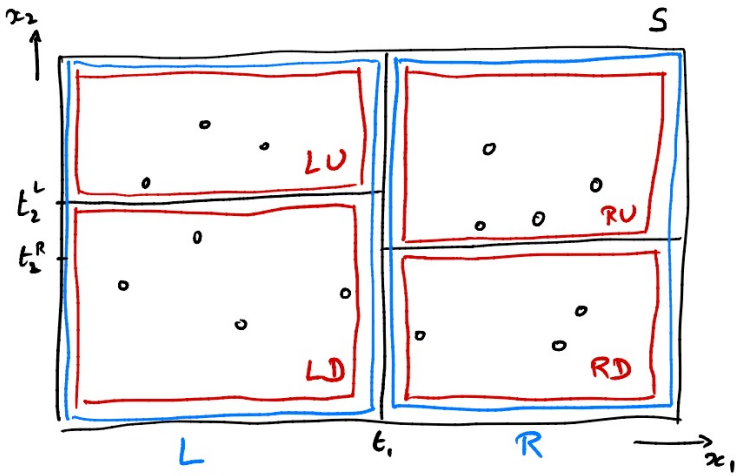


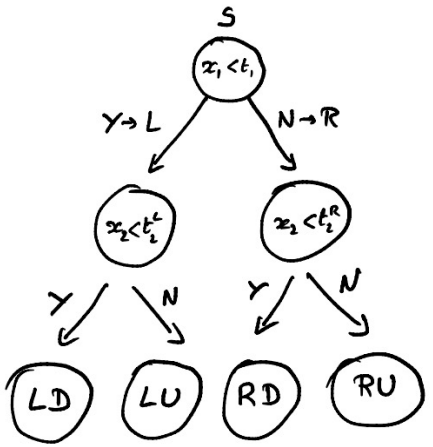


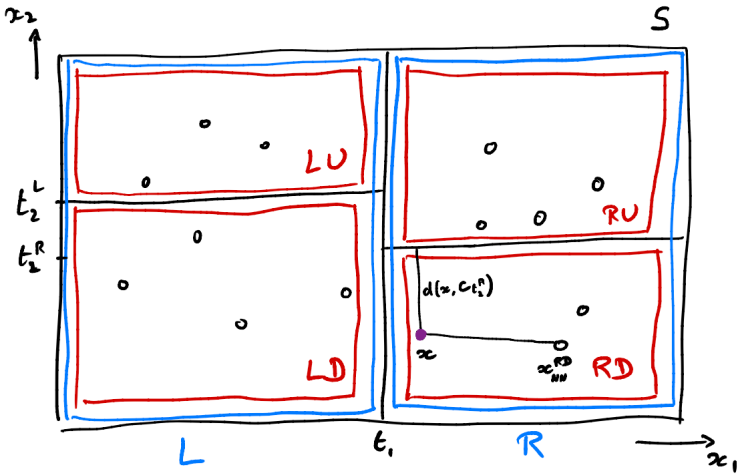


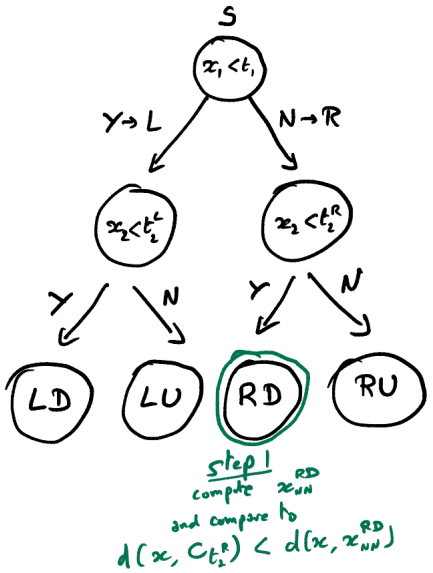
k-d tree search #2

- Example of NN search process in 2D
  - Which order of search?
  - Which partitions to prune?
  - Worst case scenario O(n)



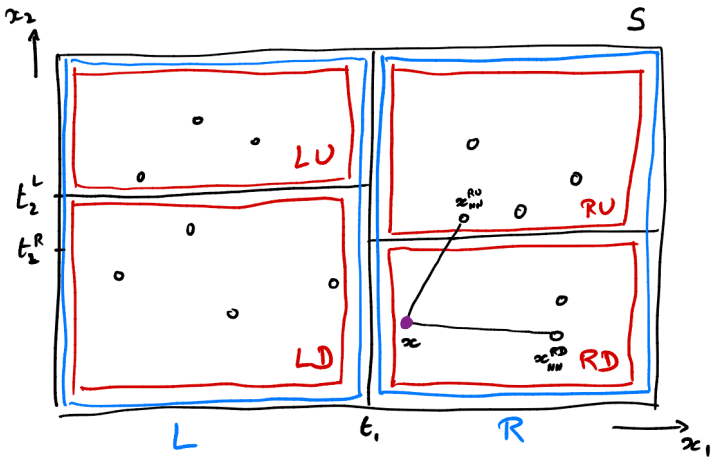


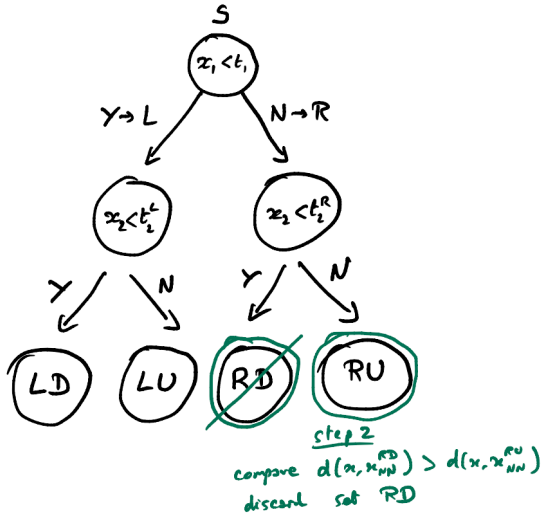


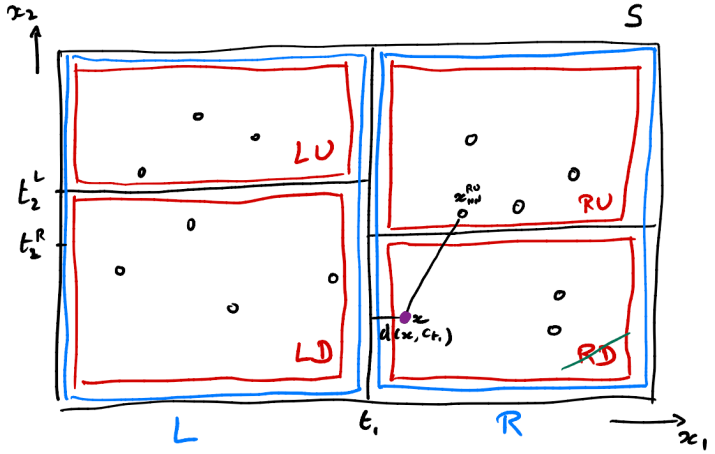


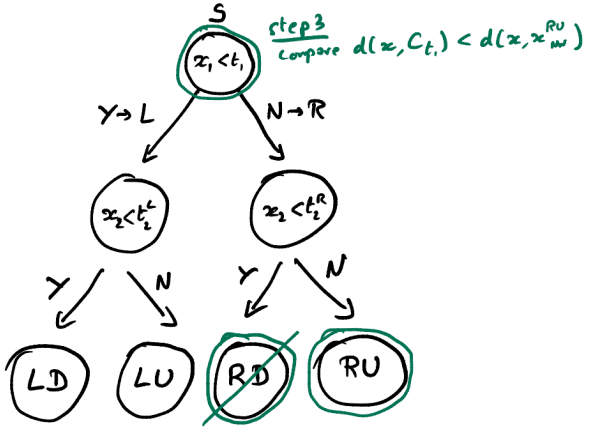
k-d tree search

- Example of NN search process in 2D
  - Which order of search?
  - Which partitions to prune?
  - Worst case scenario O(n)



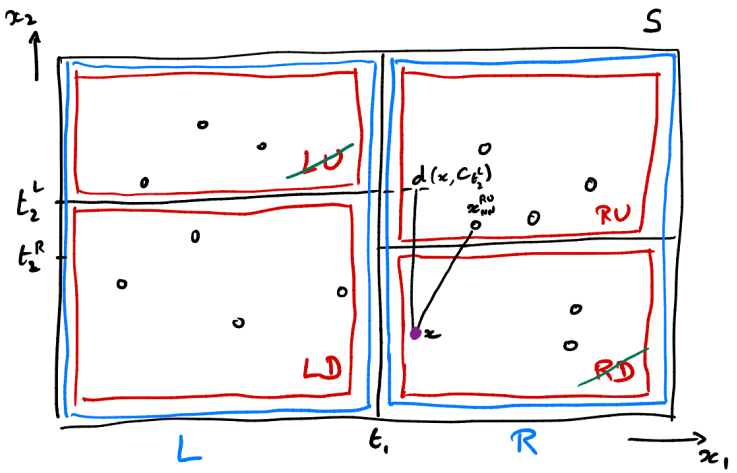


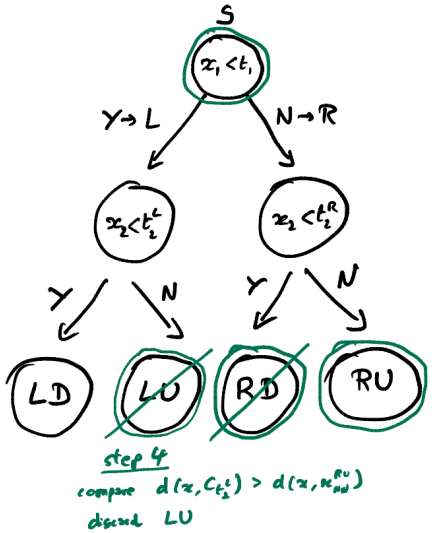


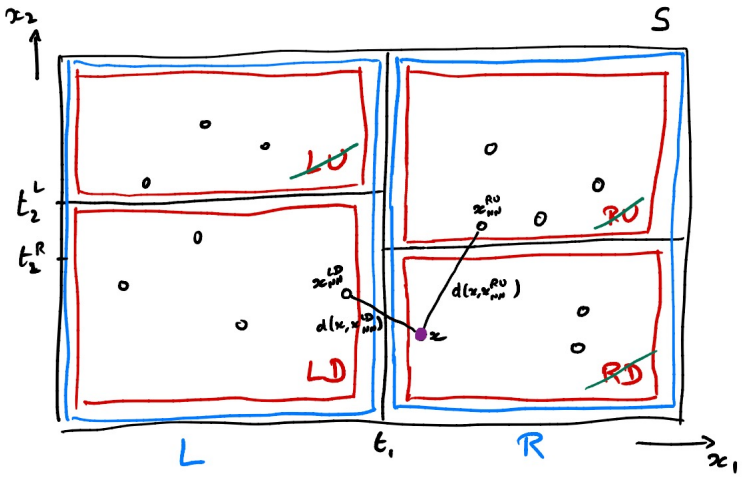


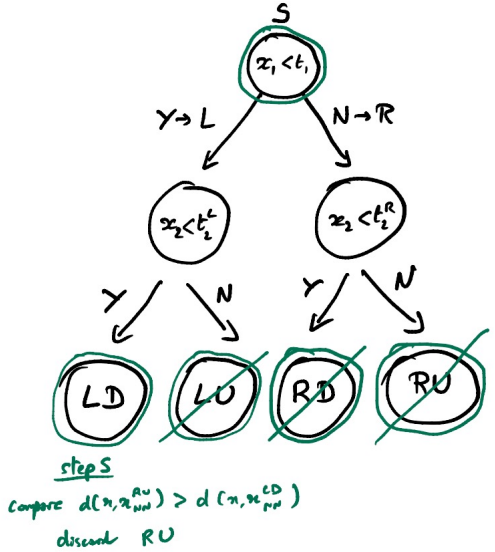
k-d tree search

- Example of NN search process in 2D
  - Which order of search?
  - Which partitions to prune?
  - Worst case scenario O(n)









k-d tree complexity

- Suppose k=1 (i.e. nearest neighbor)
- Training / building k-d tree
  - Space/memory complexity (worst case) :

$$O(2^p) \rightarrow O(n) \text{ with } p = O(\log_2 n)$$

#nodes in the tree

(Space complexity for data : $O(nd)$)

  - Time/speed complexity (worst case) :

$$\sum_p O\left(\frac{n}{2^p} \times 2^p\right) = O(pn) \rightarrow O(n \log_2 n)$$

#hierarchy levels

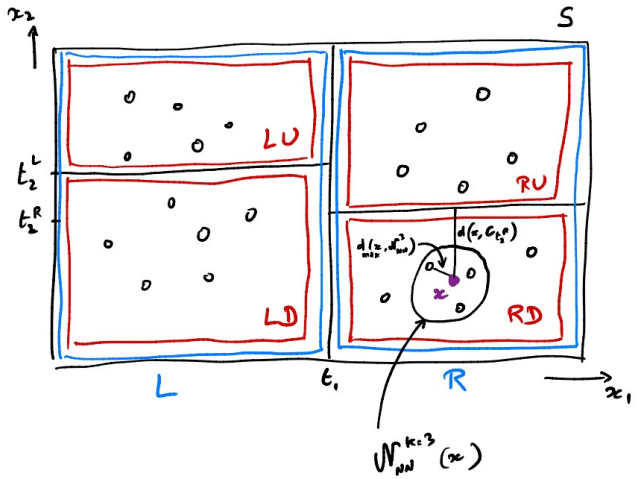
#regions

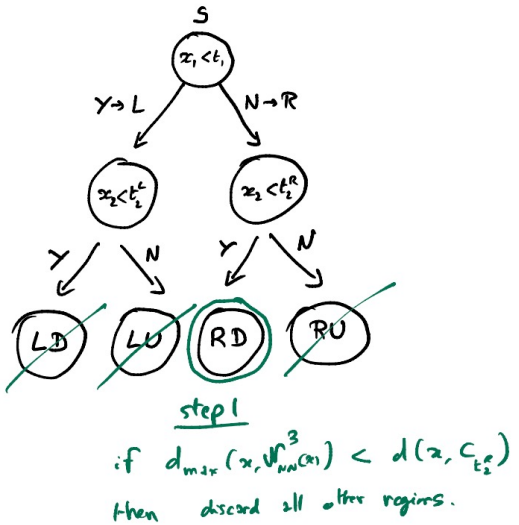
splitting time per feature dimension (features are ordered)

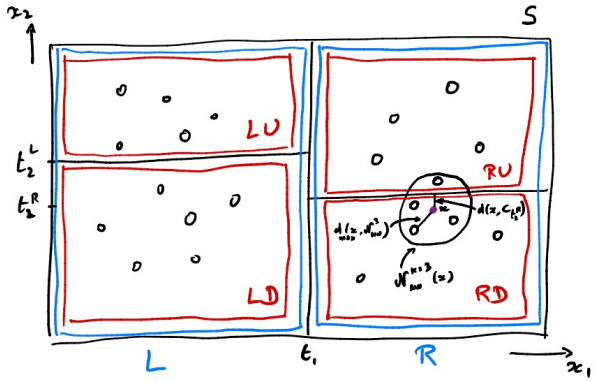
- Inference / NN search
  - Time/speed complexity (best case) : $O(p + d\frac{n}{2^p}) \rightarrow O(\log_2 n + d)$ with $p = O(\log_2 n)$
  - Time/speed complexity (worst case/NN) : $O(dn)$
  - Time/speed complexity (average case/tricky) : $O(d \log_2 n)$

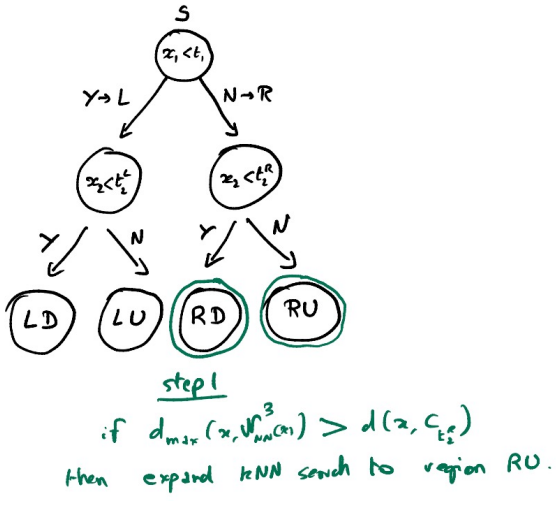
k-d tree with k nearest neighbors

- Note that the "k" in the name "k"-d tree means the number of data features.
- Note that the "d" in the name k-"d" tree means "dimension".
- In our lecture, we call the number of data features d and the number of nearest neighbors k.



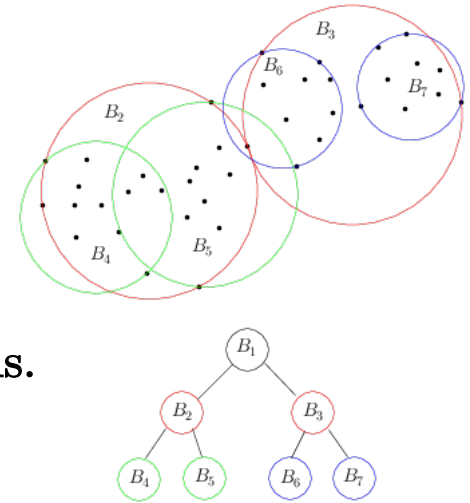






Summary

- kNN is slow because it does a lot of unnecessary pairwise distance computations.
- k-d tree partitions the feature space so we can discard space partitions that are further away than our closest k neighbors.
- Pros :
  - Exact kNN, but approximation can be used e.g. no backtracking in parent nodes.
  - Easy to implement.
  - Average inference complexity is $O(d.\log_2 n)$, compared to $O(d.n)$ with kNN.
- Cons :
  - Cuts are axis-aligned which does not generalize well to higher dimensions.
  - [Not included] Ball tree partitions the manifold of data points (assumption), as opposed to the whole space. This performs much better in higher dimensions.



Outline

- kNN
- k-d tree
- Decision tree
- Bagging
- Boosting
- Conclusion

Motivation

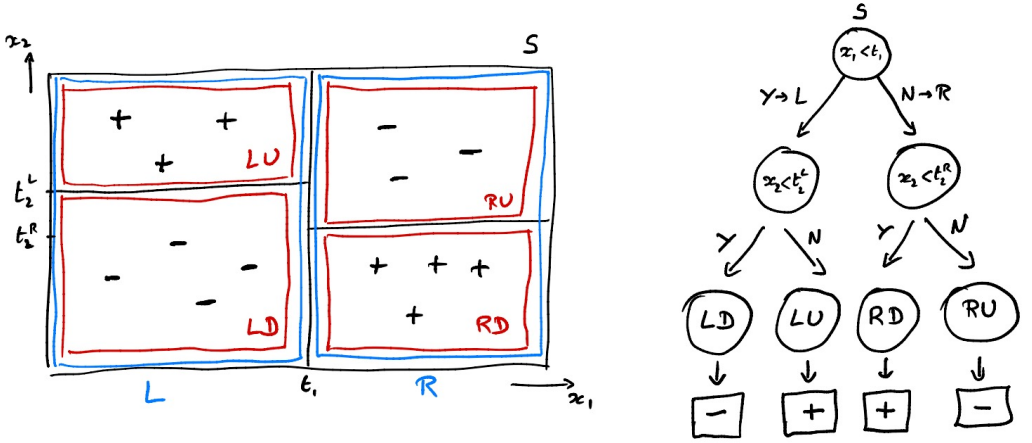
- kNN requires to store the full training set to make a prediction.
  - When n becomes large, it becomes intractable.
- Real-world assumption : Most data are not random and usually concentrate in regions with the same predicted target e.g. class or regression value.
  - This enables faster nearest neighbor search with k-d tree data structure.
- However, the ultimate goal is not to find the closest data points, but to solve a classification or a regression problem.
  - The data identity, i.e. data features, is irrelevant for the classification/regression task.
  - What is critical is to identify areas where all points have the same class label.
  - For example, if a test point falls into a cluster of 1,000 points with all positive class label, then we know that its kNN will all be positive before computing the distances to the 1,000 points.

Decision tree

- Decision tree leverages the idea that a data point has the same class label or same regression value when it falls into a cluster of same label or same regression value.
- Major advantage : There is no need to load the full training set for inference.
- Instead, we can build and load a tree structure that recursively splits the feature space into regions with similar label/value.

Construction

- Decision tree construction / training stage
  - Start from the root node of the tree that represents the entire dataset.
  - Split this set into two halves with approximatively the same size by cutting along a feature dimension e.g. x_1 with a threshold value t_1. This produces two sets of data points R and L.
  - Threshold t_1 and dimension x_1 are chosen such that the resulting children nodes R and L are purer than their parent node S w.r.t. class label or regression value.
  - If all points in R have the same e.g. positive class label and all points in L have also the same negative class label, then the decision tree is done.
  - If not, the current leaf nodes are split again until all leaves are pure, i.e. all data points in the node have the same label.

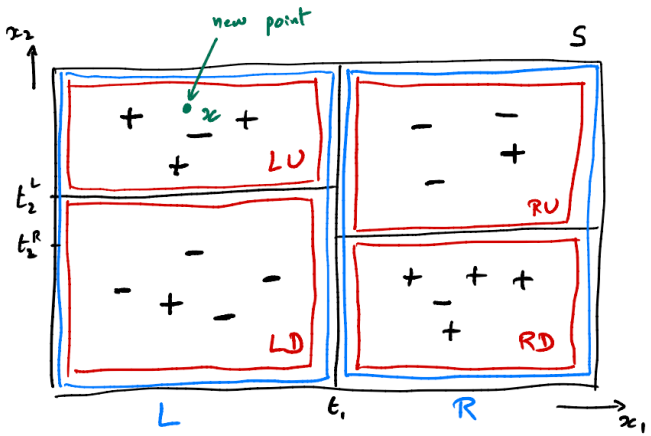


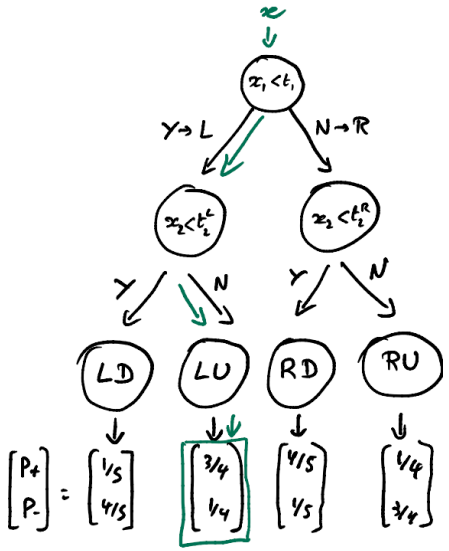
Inference

- Inference with decision tree
  - Once the tree is constructed, there is no need to keep the training set in memory.
  - What we need to store
    - The tree structure which has a depth of $\log_2 n \leq 30$.
    - Class probability/regression value in the final leaf nodes.
      - If pure classes, only class label is stored.
  - Decision tree does not require any distance computation.
    - The cut is based on feature value.
    - Hence, inference is very fast with $O(\log_2 n)$, independent of feature dimension d.

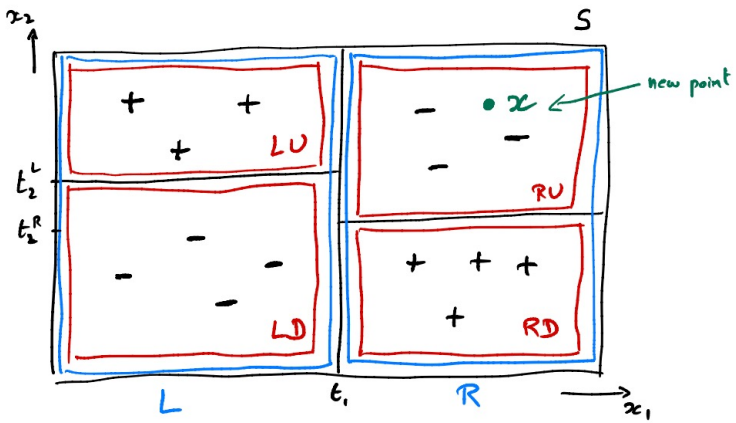
Inference

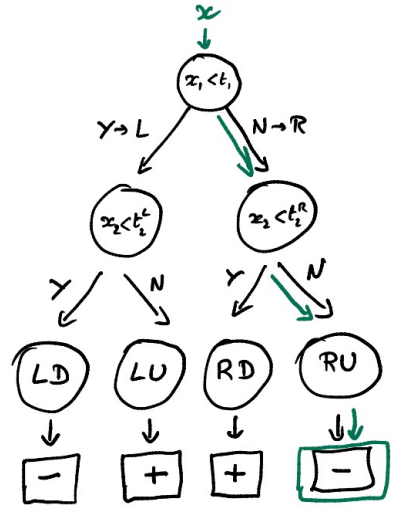
- Inference with decision tree





class probability





class label

Optimal decision tree

- Q: Can we build a decision tree that is
  - Maximally compact, i.e. small depth.
  - Only has pure leaves.
- Yes in theory, if no two data points have same features but different labels.
- No in practice, as finding a minimum size tree is NP-hard.
- But there exists a greedy algorithm that can approximate effectively small decision trees.
  - We split the data recursively by minimizing a function that measures label purity in the children's nodes.

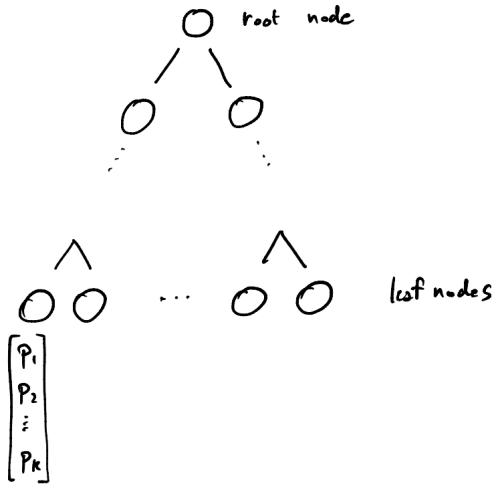
Purity function

Define

$S : \{(x_i, y_i)_{i=1}^n\},\ x_i \in \mathbb{R}^d,\ y_i = \{1, ..., c\},\ c$ is the number of classes

$S_k = \{(x, y) : y = k\},\ s_k \subset S,\ S = S_1 \cup ... \cup S_c$

$p_k = \frac{|S_k|}{|S|}$ fraction of data with label k



We want pure leaf nodes, i.e. $p_k = 1$ for a specific k and $p_{k'} = 0, \forall k' \neq k$

The worst case is when all leaves are random prediction, i.e. $p_k = \frac{1}{c}, \forall k$

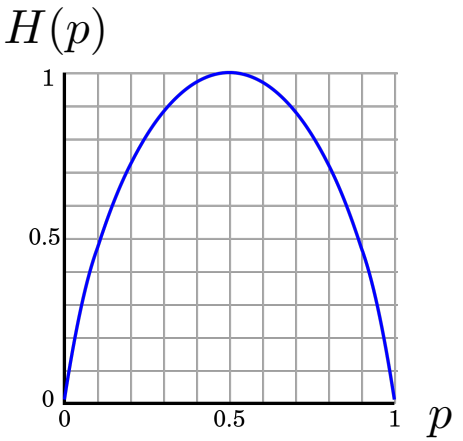
To avoid the worst case, we will maximize the KL distance between the random prediction and the best candidate p obtained by splitting

$$\max_p \ \mathrm{KL}(p, q) = \sum_{k=1}^{c} p_k \log \frac{p_k}{q_k}, \ \text{with } q_k = \frac{1}{c}, \forall k$$

Entropy

- **Maximizing the KL distance reduces to minimizing the entropy :**

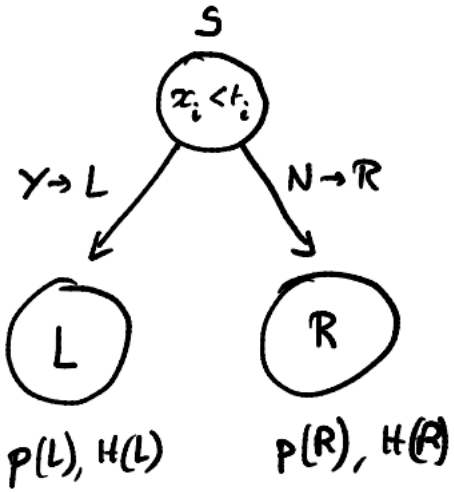
$$
\begin{aligned}
\mathrm{KL}(p,q) &= \sum_{k=1}^{c} p_k \log \frac{p_k}{q_k}, \text{ with } q_k = \frac{1}{c}, \forall k \\
&= \sum p_k \log p_k - p_k \log q_k, \text{ with } q_k = \frac{1}{c} \\
&= \sum p_k \log p_k + p_k \log c \\
&= \sum p_k \log p_k + \log c \sum p_k, \text{ with } \sum p_k = 1 \\
&= \sum p_k \log p_k + \log c \\
\max_p \ \mathrm{KL}(p,q) &= \max_p \ \sum p_k \log p_k + \cancel{\log c} \\
&= \min_p \ -\sum p_k \log p_k \\
&= \min_p \ H(p) \text{ Entropy}
\end{aligned}
$$



Entropy of binary tree

- **Entropy of binary tree :**

$$H(S)$$



$$H(L,R) = p(L)H(L) + p(R)H(R)$$
$$= \frac{|L|}{|S|}H(L) + \frac{|R|}{|S|}H(R)$$

Information gain

- Definition : The information that is gained by splitting a set of data points.
- In the case of decision tree, the splitting is controlled by a specific feature value, i.e. $x_i < t_i$.
- The entropy of the subsets $S_1,...,S_c$ is defined as

$$H(S_1, ..., S_c) = \sum_{k=1}^{c} p(S_k) H(S_k)$$
$$= \sum_{k=1}^{c} \frac{|S_k|}{|S|} H(S_k)$$

- Finally, the information gain (IG) is the difference between the entropy of the original set S and the weighted sum of the entropy of the subset S_k.

$$\mathrm{IG}(S, S_1, ..., S_c) = H(S) - H(S_1, ..., S_c)$$
$$= H(S) - \sum_{k=1}^{c} p(S_k) H(S_k)$$

Feature and threshold selection

- Goal is to find subsets that maximizes the information gain, achieving the purest possible subsets.
  - Identifying the purest subsets is to find a feature x_i and a threshold value t_i.
- Decision tree construction (pseudo-code)
  - While leaf nodes are not pure (or ≥ threshold)
    - Loop over (remaining) feature dimensions, i.e. $x_1, x_2, \ldots, x_d$
      - Loop over n thresholds (e.g. middle points between two consecutive points, such as $t_i = (x_{i+1} - x_i)/2$)
        - Compute information gain for R and L
        - Save (dimension, threshold value) with maximum information gain.
    - Split space with best (dimension, threshold value) and remove dimension x_i from loop.
- Exact complexity is O(n.d). Approximations are used in practice for speed-up.

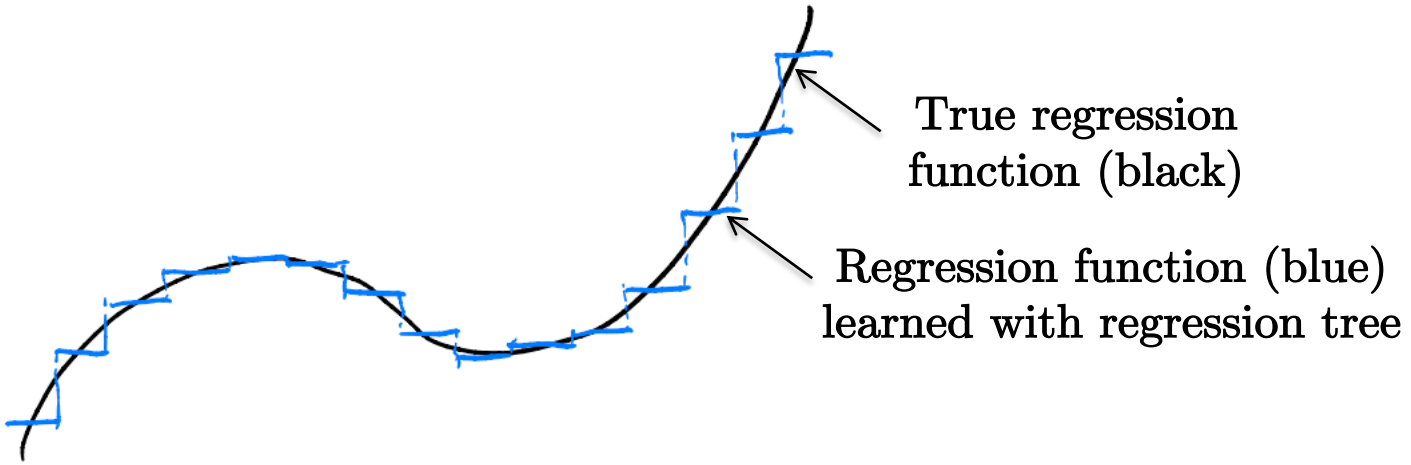
Regression tree

- It is straightforward to extend decision tree to other task s.a. regression as long as a purity function can be defined for the new task :

$$\mathrm{H}(S) = -\sum p_k \log p_k \quad \rightarrow \quad \mathrm{L}(S) = \frac{1}{|S|} \sum_{(x,y)\in S} (y - \bar{y}_S)^2 \text{ (variance)}$$

$$\text{with } \bar{y}_S = \frac{1}{|S|} \sum_{(x,y)\in S} y \text{ (mean)}$$

Classification task | Regression task



Complexity

- Training / building decision tree
  - Space/memory complexity (worst case) :

$$\underbrace{O(2^p)}_{\#\text{nodes in the tree}} \rightarrow O(n) \text{ with } p = O(\log_2 n)$$

  - Time/speed complexity (worst case) :

$$\sum_p O\Big(\frac{n}{2^p} \times 2^p\Big) = O(pn) \rightarrow O(n \log_2 n)$$

$\frac{n}{2^p}$: purity time

- Inference / NN search
  - Time/speed complexity : $p \rightarrow O(\log_2 n)$

Outline

- kNN
- k-d tree
- Decision tree
- **Bagging**
- Boosting
- Conclusion

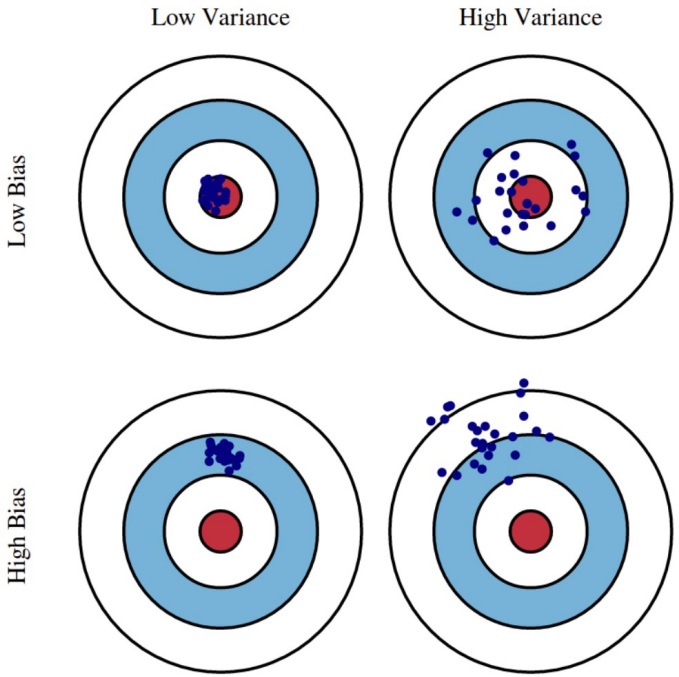
Bagging

- Decision trees have great advantages at inference
  - Speed complexity is very fast, i.e. $O(\log_2 n) \leq 30$
  - Memory complexity is low, i.e. $O(n)$ independent of d
- However, these techniques have high variance performance.
  - This means that the quality of the classification/regression solutions vary significantly (see next slide).
    - They are known as weak learners (classifiers or regressors).

Bias and variance (week 5)

- Quality of predictive models are evaluated by their bias-variance properties.
- For example, let assume that the task of the model is to predict the red center of the target below

Low variance and low bias
The perfect model!



High variance and low bias
The model is able to find the correct solution on average.

Low variance and high bias
The model favors some solutions, far from the true ones.

High variance and high bias
The worst model!
The model has not only bad bias but also large variance.

Bias and variance

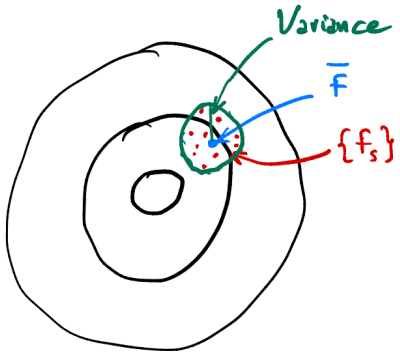
- Long history of analysis of the bias-variance trade-off (but recently questioned by deep learning).
  - It is highly challenging to design the perfect model (i.e. low bias and low variance).
- Formalization

$$\mathbb{E}_{(x,y)}\left[(f_S(x) - y)^2\right]$$

Mean over data points; data; Predictive model; target; Assumption: no noise

$$= \mathbb{E}_{(x,y)}\left[(f_S(x) - \bar{f}(x) + \bar{f}(x) - y)^2\right], \text{ with } \bar{f}(x) = \int_{S' \subset S^*} f_{S'}(x) p(S') dS'$$

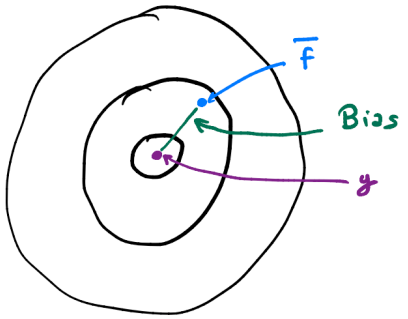
Average predictor; True data distribution



$$= \mathbb{E}_{(x,y)}\left[\underbrace{(f_S(x) - \bar{f}(x))^2}\right] + \mathbb{E}_{(x,y)}\left[\underbrace{(\bar{f}(x) - y)^2}\right]$$

Error between prediction model and average over all predictors
Variance

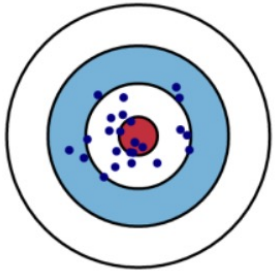
Error between average predictor and target
Bias²



Reducing variance

- Decision trees have low bias but high variance, i.e. solutions vary around the true solution.
- Goal : Design a technique that reduces variance, i.e.

$$\min \ \mathbb{E}_{(x,y)}\left[(f_S(x) - \bar{f}(x))^2\right]$$



- Most common idea is to take the average of multiple solutions, a.k.a. ensemble technique :

$$f_S(x) \approx \hat{f}(x) = \frac{1}{m}\sum_{j=1}^{m} f_{S_j}(x) \rightarrow \bar{f}(x) \ \text{ as } \ m \rightarrow \infty$$

$$\text{where } \{S_1, ..., S_m\} \subset S^* \text{ (true data distribution)}$$

Reducing variance

- Why averaging classifiers reduces variance ?
  - Because of the law of large numbers :

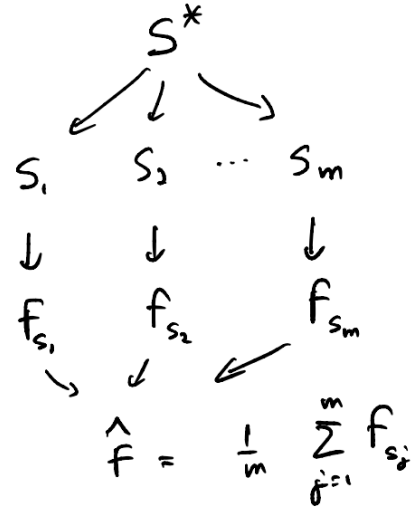
    The law of large numbers states for i.i.d. (independent and identically distributed) random variable x_i with mean $\bar{x}$:

$$\frac{1}{m}\sum_{i=1}^{m} x_i \rightarrow \bar{x} \text{ as } m \rightarrow \infty$$

  - Apply to learners : Assume we have m training datasets $S_1,\ldots,S_m$ sampled from S^*, the true data distribution.

    Train a learner on each training set and average the result:

$$\hat{f} = \frac{1}{m}\sum_{j=1}^{m} f_{S_j} \rightarrow \bar{f} \text{ as } m \rightarrow \infty$$

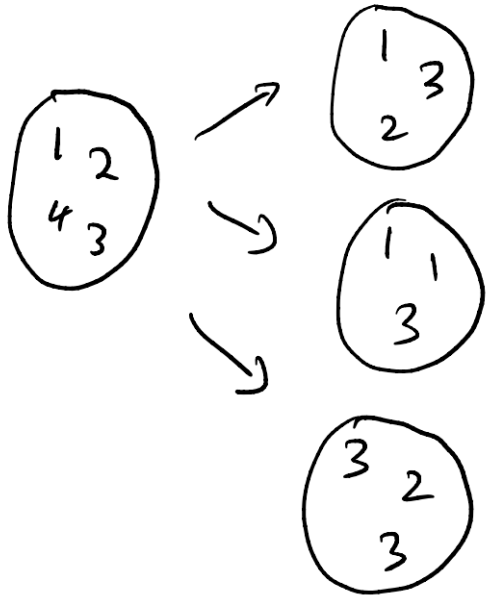


Bagging

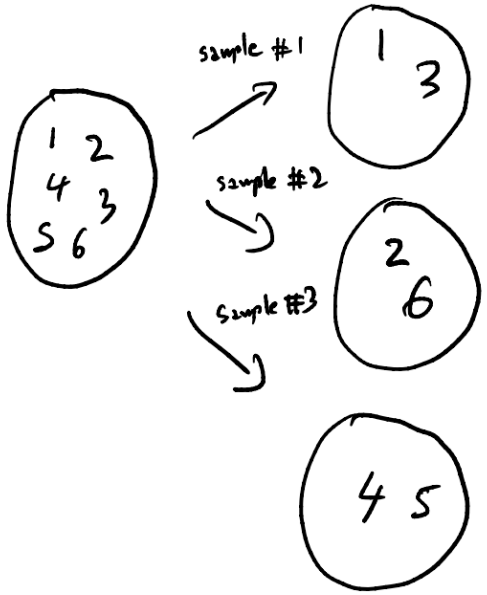
- Averaging several classifiers/regressors will decrease the variance and make the ensemble classifier/regressor more accurate.
- But we do not have access to more training sets $\{S_1,..,S_m\}$ than the original set S.
  - Because we do not know the true distribution S*.
  - How do we create new training sets?
    - Solution is bagging.
- Bagging algorithm
  - Sample m datasets $S_1,..,S_m$ from original S with replacement.
  - For each training set S_j, train a classifier f_{Sj}.
  - Final/ensemble classifier is $\hat{f}(x) = \frac{1}{m}\sum_{j=1}^{m} f_{S_j}(x)$

Sampling with replacement

- What is sampling with replacement?
  - When a data is selected, it continues to be part of the set and can be sampled again (unlike sampling without replacement, once a data is selected then it is removed from the set and cannot be sampled again).



Sampling with replacement



Sampling without replacement

Sampling with replacement

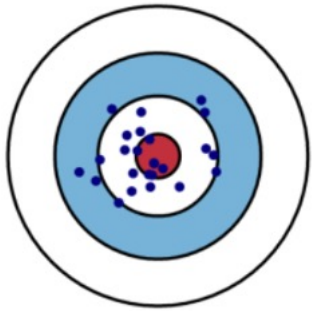
- When we sample without replacement, two samples are not independent, i.e. not i.i.d.
  - For example, let us consider a set of n balls numbered from 1 to n. If we first sample the ball number 3, then the next sampled ball will depend on the first ball as the remaining set of balls is {1,2,4,5,6}.
- However, when we sample with replacement, two samples are independent, i.e. a sample does not affect another sample, i.e. they are i.i.d.
  - Based on the example above, if we first sample the ball number 3 and we replaced this ball in the set, then the next sampled ball will not depend on the first ball as the set of balls is the unchanged original set {1,2,3,4,5,6}.

Bagging

- As sampling with replacement produces i.i.d. data, then training sets S_j are i.i.d.
- Applying the law of large numbers guarantees the following asymptotic result :

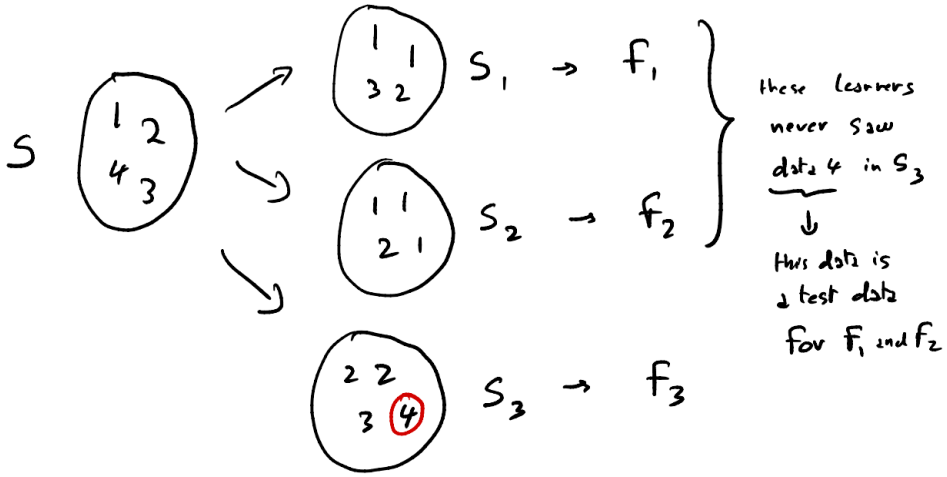
$$\hat{f} = \frac{1}{m}\sum_{j=1}^{m} f_{S_j} \rightarrow \bar{f}$$

- As a result, in practice, bagging reduces the variance quite effectively.
- We say that bagging reduces the variance without increasing the error of an unbiased classifier.
  - Unbiased classifier produces the correct solution is average.



Bagging

- Advantages
  - Easy to implement
  - Easy to reduces variance for high variance classifiers/regressors.
  - Bagging also provides an error estimate of the test error (for free).
    - During sampling S_j, some training data x_k will not be selected and hence can act as a test data for the ensemble of classifiers.

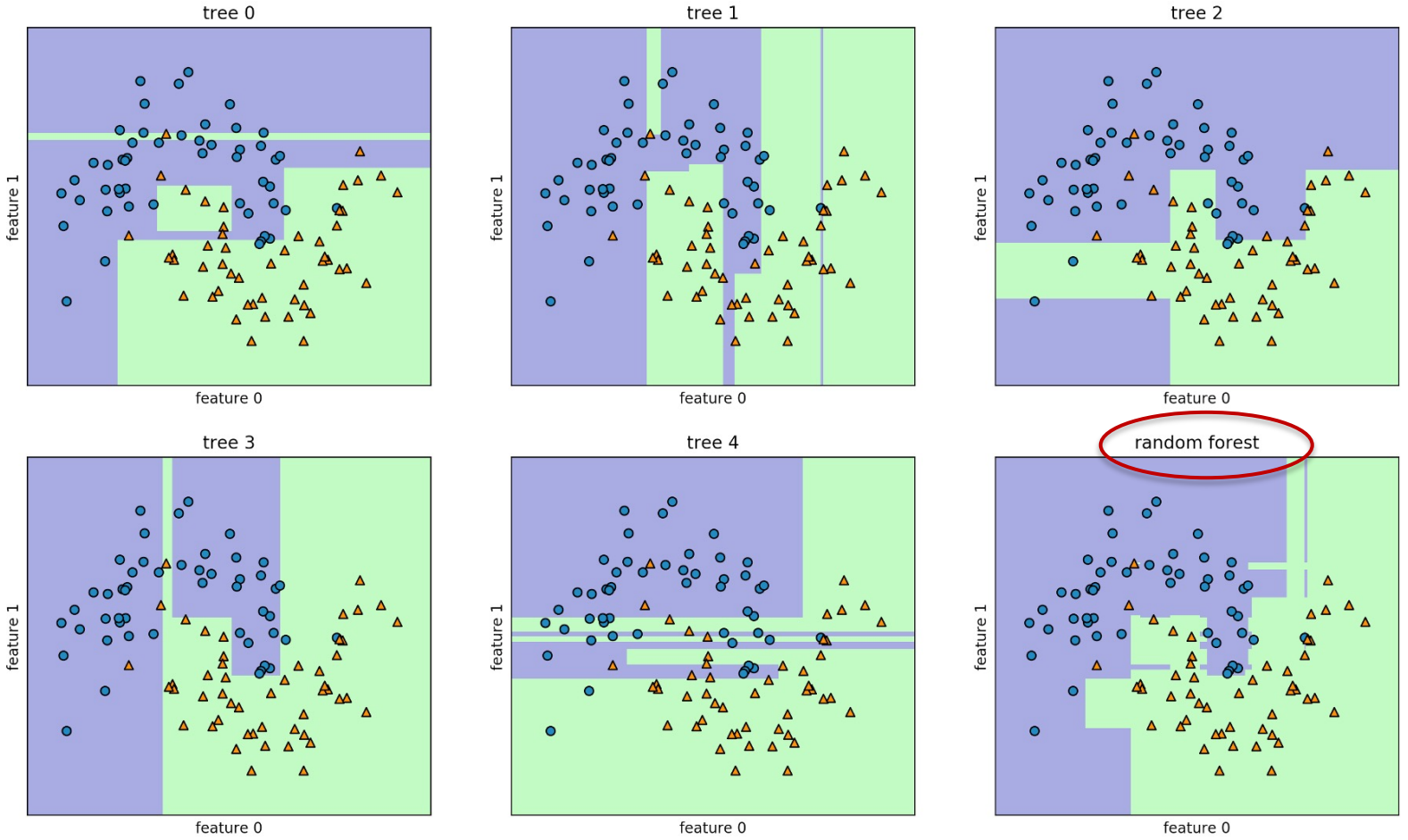


Random forest

- One of the most popular and useful bagging algorithms is random forest.
- Random forest is an ensemble of decision trees.
- Algorithm
  - Sample m training datasets $S_1,\ldots,S_m$ from S with replacement.
  - For each S_j, train a decision tree f_{Sj} with one important modification :
    - Only consider a randomly small number of k of splits with $k \leq d$ features.
    - Goal is to make sure that all classifiers f_{Sj} are all very different. As such, they will make different errors at test time, but averaging will correct most of the errors.
  - Final classifier is $\hat{f}(x) = \frac{1}{m}\sum_{j=1}^{m} f_{S_j}(x)$
- Hyper-parameters
  - $k = \sqrt{d}$ is a good heuristic
  - m is as large as computational resource permits

Random forest

- Example with two-moon binary classification

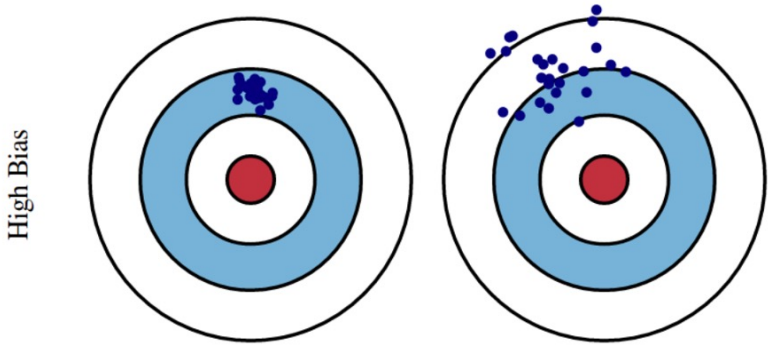


Outline

- kNN
- k-d tree
- Decision tree
- Bagging
- Boosting
- Conclusion

Boosting

- Let us consider the case where classifiers/regressors have high bias, i.e. these prediction models have large errors on the training set.
  - An example of such high-bias models are decision trees with limited depth, e.g. value 4.
- Q: Can we design an ensemble method that combines a large number of weak learners to generate a strong learner with low bias?
  - Yes, this class of algorithms is called boosting.
  - Boosting reduces bias.



Boosting

- Vanilla boosting algorithm
  - Assume we have an ensemble classifier at step t = T, i.e. $F_T(x) = \sum_{t=1}^{T} \alpha_t f_t(x), \ \alpha_t > 0$
  - Prediction error is defined with a loss function as L(f) as $L(f) = \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)$
  - We would like to add a new (weak) learner f to F_T to decrease the prediction error as much as possible. The weak learner f is selected by minimizing the following loss :

$$f_{t+1} = \text{argmin}_{f \in H} \ L(F_T + \alpha f), \ \alpha > 0 \text{ and small}$$

H: hypothesis space

  - After we found the new learner f_{t+1}, we simply add it to F_T :

$$F_{T+1} = F_T + \alpha f_{t+1}$$

Gradient boosting

- How do we solve the optimization problem?

$$f_{t+1} = \text{argmin}_{f \in H} \; L(F_T + \alpha f)$$

- We use a first-order Taylor approximation of L :

$$L(F + \alpha f) \approx L(F) + \alpha \underbrace{\langle \nabla L(F), f \rangle}_{\text{gradient}}$$

$$\approx L(F) + \alpha \sum_{i=1}^{n} \frac{\partial L}{\partial F(x_i)} . f(x_i)$$

$$\text{Example with MSE} : L(F(x_i)) = \frac{1}{2}\big(F(x_i) - y_i\big)^2$$

$$\frac{\partial L}{\partial F(x_i)} = \big(F(x_i) - y_i\big) . \frac{\partial (F(x_i) - y_i)}{\partial F(x_i)} = F(x_i) - y_i$$

Gradient boosting

- Optimization problem :

$$\begin{aligned}\text{argmin}_{f \in H}\ L(F_T + \alpha f) &\approx \text{argmin}_{f \in H}\ \cancel{L(F)} + \alpha \langle \nabla L(F), f \rangle \\ &\approx \text{argmin}_{f \in H}\ \sum_{i=1}^{n} \frac{\partial L}{\partial F(x_i)} . f(x_i)\end{aligned}$$

- At last, we need an algorithm that computes f ∈ H, H is known as the hypothesis space, i.e. a space of solutions for the task at hand.
- As the goal of boosting is to reduce the bias of predictors, the space H is supposed to contain high-bias models s.a. decision trees with limited depth.
- Another major advantage of gradient boosting is that solution f does not have to be exact, i.e. any approximation f can used as long as the dot product is negative (see next slides for justification) :

$$\sum_{i=1}^{n} \frac{\partial L}{\partial F(x_i)} . f(x_i) < 0$$

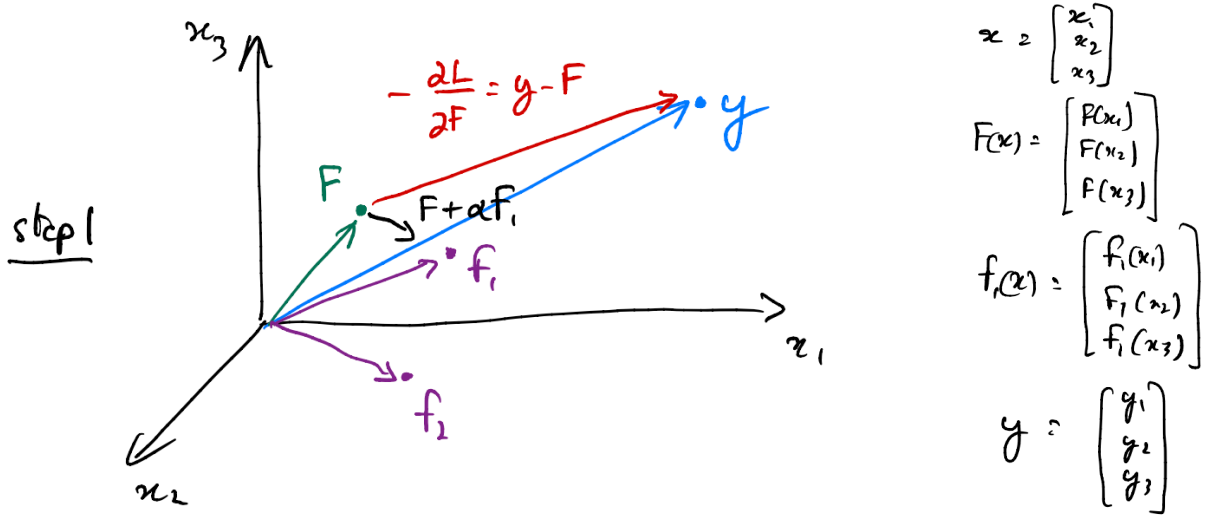
Gradient boosting

- **Step-by-step optimization :**

$$\text{Suppose } L(F) = \frac{1}{2}(F-y)^2, \text{ then } \frac{\partial L}{\partial F} = F - y$$

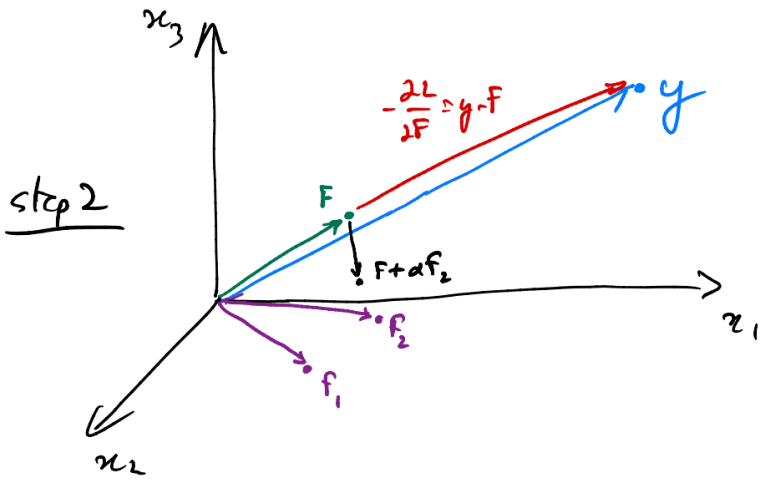
$$\text{Then } \operatorname{argmin}_{f \in H} \langle \frac{\partial L}{\partial F}, f \rangle \Leftrightarrow \operatorname{argmax}_{f \in H} \langle -\frac{\partial L}{\partial F}, f \rangle$$

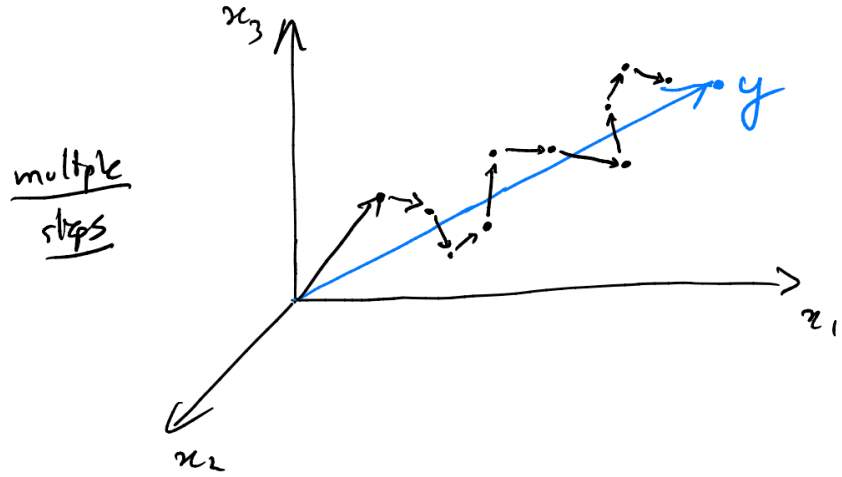
Find candidate f that best aligns to –gradient, which always points to the solution y.



Gradient boosting

- Step-by-step optimization :





Gradient boosting (AnyBoost)

- **Pseudo-code**

Input: $L, \alpha, \{(x_i, y_i)\}, \mathcal{A}$ (predictor)

$F_0 = 0$

For $t = 0 : T$

$g_i = \dfrac{\partial L(F_t(x_i), y_i)}{\partial F(x_i)}$ (gradient)

$f_{t+1} = \text{argmin}_{f \in H} \sum_{i=1}^{n} g_i . f(x_i),$ with $f = \mathcal{A}(\{(x_i, y_i)\})$

if $\sum_{i=1}^{n} g_i . f_{t+1}(x_i) < 0$ then

$F_{t+1} = F_t + \alpha f_{t+1}$

else

return F_t

Summary

- Boosting is a powerful technique to turn weak learners into a strong learner.
- Class of boosting algorithms
  - Gradient boosting
    - Classification and regression tasks
    - Weak learners are regression trees of depth 4
  - Adaptative boosting (AdaBoost)
    - Specific case of binary classification (cannot be applied directly to multi-class and regression)
    - Specific case of exponential loss, i.e. $L(F) = \sum_{i=1}^{n} e^{-y_i . F(x_i)}$
    - Step size α can be computed optimally (closed-form solution)
    - Training error decreases exponentially, O(logn) convergence (can be proved)
- A hybrid algorithm that combines advantages of bagging and boosting can be designed :
  - Stochastic gradient boosting : sub-sample with replacement + low-depth trees

Outline

- kNN
- k-d tree
- Decision tree
- Bagging
- Boosting
- Conclusion

Conclusion

- kNN is a simple and expressive learning technique but time and memory consuming.
- k-d tree speeds up kNN by discarding far away data points.
- Decision tree improves the memory complexity (no loading of data points required) and speeds up inference with tree structure.
- Bagging is an ensemble method that combines a large number of weak learners with high variance to generate a strong learner with low variance.
- Boosting is an ensemble method that combines a large number of weak learners with high bias to generate a strong learner with low bias.
- Bagging/boosting are universal, i.e. agnostic of the algorithm used.
  - Use these ensemble techniques to boost your algorithm accuracy by a few percentage, e.g. to win Kaggle competitions ☺.

Questions?