Optimal Decision Trees

Zichao (David) Wang

June 2020

Before we start

Without loss of generality, throughout this presentation, we will simplify the decision trees (DT for short) as follows.

- We will always talk about the classification problem rather than the regression problem, since there is no fundamental difference between them.
- We will not distinguish the binary DT from n-ary DT and will use both interchangeably, since any n-ary DT is equivalent to a binary DT of larger heights.

Overview I

Overview II

Current section

1. A quick review on DT
    - Tree construction algorithm: use ID3 as an example.
    - Prevent overfitting: make our DT “less suboptimal”.
    - Pros & cons of DT

A quick review on DT

Tree construction algorithm: use ID3 as an example.

```
1: $A \leftarrow$ the best attribute for splitting {examples}
2: Decision attribute for this node $\leftarrow A$
3: for value in $A$ do
4:    Create a new child node.
5: end for
6: Split {examples} into child nodes.
7: for all [child_node, {subset}] do
8:    if {subset} is pure then
9:       Stop
10:   else
11:      Split(child_node, {Subset})
12:   end if
13: end for
```

Algorithm 1: Split(node, {examples})

A quick review on DT

Tree construction algorithm: use ID3 as an example.

- We can use *Information Gain (IG)* to measure the purity of a node: suppose we have a set of examples S and we want to evaluate the effect of splitting the dataset using attribute A.

$$IG(S, A) := H(S) - \sum_{v \in \{-1,1\}} \frac{|S_v|}{|S|} H(S_v)$$

  where S_v is the subset of S with $A = v$, and the *entropy* is defined as

$$H(S) = - \sum_{v \in \{-1,1\}} p_v \log p_v$$

- The entropy, just like in physics, measures the degree of chaos in our probability distribution.

A quick review on DT

Prevent overfitting: make our DT "less suboptimal".

- Direction 1: optimize a single tree.
  - Stop splitting when it is not statistically significant.
  - Grow a full DT, and then post-prune it on the CV set.
  - If we use IG as our metric, the DT will by nature favoring the attributes which can divide the training set into a lot of small subsets (think of the index of each observation as an extreme example). In this case, we can use *Gain Ratio (GR)* instead of IG during the splits.

$$SplitEntropy(S, A) = -\sum_{v\in\{-1,1\}} \frac{|S_v|}{|S|}\log\frac{|S_v|}{|S|}$$

$$GR(S, A) := \frac{IG(S, A)}{SplitEntropy(S, A)}$$

  - Modify the DT to make the decision boundaries oblique.
  - Run an overall optimization instead of greedy algorithm.
- Direction 2: use law of large numbers.
  - Random forest

A quick review on DT

Pros & cons of DT

- Pros
  - Interpretable: humans can understand the decision process.
  - Can easily handle the noise attributes (since their $IG \approx 0$).
  - Can handle missing data.
  - Compact: After pruning, number of nodes $\ll$ size of the dataset.
  - Very fast in prediction: $O(h)$ time complexity.
- Cons
  - Only axis-aligned splits of data.
  - It is a greedy algorithm, and therefore cannot guarantee the optimality.

Current section

OC1: the big picture

- Just like ID3, we shall still use the greedy algorithm to recursively construct the tree.
- Therefore, the general framework of ID3 needs no change:
    1. Find the best "multivariate split".
    2. Divide our data into two subsets using the split function we found.
    3. For each child node, if it is not pure, repeat step 1 & 2.
    4. Post-prune the DT if needed.
- The only step which needs further discussion is step 1.

Hyperplane search strategy of OC1: perturbation

- The strategy of searching through the space of possible hyperplanes is that we consistently perturb the current hyperplane into a new location.
- Denote our sample space as P, which contains n examples, each with d attributes. Each example belongs to a particular category.
- The equation of the current hyperplane H can be written as $\sum_{i=1}^{d}(a_i x_i) + a_{d+1} = 0$.
- Let $P_j := (x_{j1}, x_{j2}, ..., x_{jd})$ be the jth observation from P. If we substitute P_j into the equation for H, we get $V_j = \sum_{i=1}^{d}(a_i x_{ij}) + a_{d+1}$.
- $\mathrm{sgn}(V_j)$ tells us whether P_j is above or below the hyperplane H.

Hyperplane search strategy of OC1: perturbation

- OC1 perturbs the coefficients of H one at a time.
- Denote the current attribute index we are perturbing is m $(1 \leq m \leq d)$.
- Since we perturb only one attribute at a time, all other coefficients can be seen as constants, and V_j can be viewed as a function of a_m.
- Define
$$U_j = \frac{a_m x_{jm} - V_j}{x_{jm}} \tag{1}$$
Then the point P_j is above H if $a_m > U_j$ and vice versa.
- Thus, by fixing the value of $a_1, a_2, ..., a_{d+1}$ except a_m, we can obtain n constraints on a_m, using n data points in P.
- The problem then is to find a value for a_m that satisfies as many of these constraints as possible.

Hyperplane search strategy of OC1: perturbation

```
1: for $j = 1$ to $n$ do
2:     Use (1) to compute $U_j$.
3: end for
4: Sort $U_1, U_2, ..., U_m$ in the increasing order. Then possible splits are chosen as
   $\frac{U_1+U_2}{2}, \frac{U_2+U_3}{2}, ..., \frac{U_{m-1}+U_m}{2}$.
5: $a_{m1}$ = best univariate split among all the candidates above
6: Let $H_1$ be the result of substituting $a_{m1}$ for $a_m$ in H.
7: if impurity$(H) >$ impurity$(H_1)$ then
8:     $a_m = a_{m1}$
9:     stagnant $= 1$
10: else if impurity$(H) =$ impurity$(H_1)$ then
11:     $a_m = a_{m1}$ with probability $= 1 - \exp(-stagnant)$
12:     stagnant $=$ stagnant $+ 1$
13: else
14:     Do not update
15: end if
```

Algorithm 2: Perturb(H, m)

Hyperplane search strategy of OC1: perturbation

In terms of which coefficient among $a_1, ..., a_{d+1}$ we shall choose, we use the following method called *R-50*.

```
1: for $i = 1$ to 50 do
2:     Generate a random integer $m$ ranging in $[1, d+1]$.
3:     Perturb($H$, $m$)
4: end for
```

Algorithm 3: R-50

Hyperplane search strategy of OC1: jump out of local minima.

- Since algorithm 2 is similar to SGD, one of its big drawbacks is that we may be frequently stuck in the local minima.
- We can deal with the problem in two ways:
  1. perturbing the hyperplane in a random direction;
  2. re-running the perturbation algorithm with additional intial hyperplaces.
- The second method is rather trivial, and we will only focus on the first one.

Hyperplane search strategy of OC1: jump out of local minima.

When a hyperplane $H = \sum_{i=1}^{d} a_i x_i + a_{d+1}$ cannot be improved by the deterministic perturbation, we do the following.

- Let $R = (r_1, r_2, ..., r_{d+1})$ be a random vector. Let α be the amount by which we want to perturb H in the direction R.
- That is to say, after the random perturbation, the new hyperplane is $H_1 = \sum_{i=1}^{d}(a_i + \alpha r_i)x_i + (a_{d+1} + \alpha r_{d+1})$.
- Notice that the only variable here is α. Therefore, just like before, the n examples in P impose n constraints on α, depending on their categories.
- We use the perturbation algorithm 2 to compute the best value of α.
- If H_1 improves (i.e., lowers) the impurity, we adopt H_1 and continue with the deterministic coefficient perturbation procedure; otherwise stop and output H as the best possible split.

OC1: impurity measure, stopping criteria and pruning strategy

- Impurity metric: Gini impurity
- Stopping criteria: stop splitting when all the leaves are pure.
- Pruning strategy: the only pruning done by the paper consists of cutting off subtrees at nodes whose impurity measures are less than a certain threshold.

OC1: my implementation

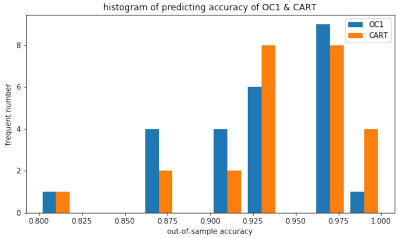
- I have implemented OC1 in Python in *oblique_decision_tree.py* and *gini.py*
- Compared with OC1 introduced above, my implementation is a bit simplified in the sense that I do not implement the random perturbation (in order to escape from local minima) and the post-pruning.
- For the pruning process, it is ok not to implement it because we can use “OC1 forest” in practice.

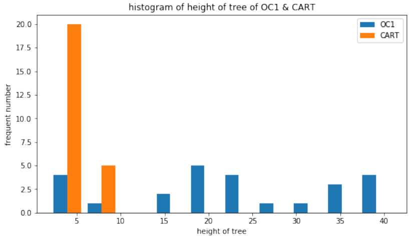
OC1: results and discussion

- We compare my simplified OC1 with *sklearn.tree.DecisionTreeClassifier*, which I believe is a modification of CART.
- We compare the performance in both single-tree and forest level.
- The evaluation metric is out-of-sample accuracy.
- The data set used here is iris dataset with 150 data points and 4 features.

OC1: results and discussion

- We use both OC1 and CART to fit a single tree 25 times. Train : test = 4 : 1.

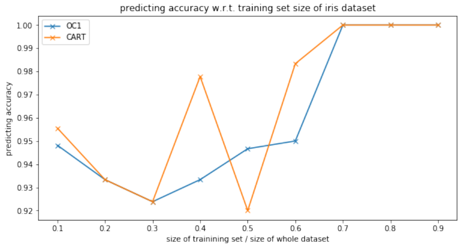




- In terms of accuracy, simplified OC1 achieves a similar level with modified CART.
- OC1 has much larger heights than CART, which in turn proves that *sklearn.tree.DecisionTreeClassifier* has been modified in some aspects in order to control the height and prevent overfitting.

OC1: results and discussion

- In terms of random forest, we set training set : whole dataset = 0.1, 0.2, ..., 0.9. In each case, we fit a forest of 50 trees using OC1 and CART respectively.



- Both forests do pretty good jobs, probably because iris is just a toy dataset and not noisy or complicated at all.
- But we do notice that as the size of training set grows, the accuracy of OC1 forest increases much more stably than CART forests.
- This probably means that in the forest level, OC1 forest better captures the data and thus has less overfitting.

Current section

Finding the optimal DT: the big picture

Before getting into any detail, let's first try to answer a fundamental question:

Fundamental question

Regardless of which methodology we adopt, what has to be done in order to find the optimal DT?

Finding the optimal DT: the big picture

The answer to the question above consists of the following steps.

- Model the current state of a DT, including both internal and leaf nodes.
- Define a loss function, since the big picture is to minimize the loss so that we can get the "optimal tree". The loss function should be a function of the prediction results and the true labels.
- Tackle a fundamental problem: the status of internal nodes has to be decided (i.e., fixed) in order to get our prediction results. So there is a time order between the internal nodes and leaves. How do we take gradient of both of them simultaneously?

Finding the optimal DT: the big picture

- If all these problems can be solved, there is nothing to keep us from find the so-called “optimal DT”.
- We will solve them one by one.

Finding the optimal DT: some notations

- m: number of leaf nodes. Since we will only talk about full binary trees, the number of leaves is thus $(m+1)$.
- x: $(p \times 1)$ vector representing a data point. Here p is the number of attributes.
- k: number of categories.
- i: index of internal nodes, $i = 1 : m$.
- j: index of leaves, $j = 1 : m + 1$.
- l: index of categories, $l = 1 : k$.
- $\mathcal{D} := \{x_z, y_z\}_{z=1:n}$: training set, where n is the size of training set, and $y_z \in \{1, 2, ..., k\}$ is the categorical label we want to predict.
- $v[i]$: the ith element of a column vector v. All indices start from 1 in this section.
- $\mathbb{I}_n$: the set of all the column vectors of length n, s.t. only one of the elements is 1 and all the others are 0.
- $\mathbb{H}^m := \{-1, 1\}^m$: the linear space of length-m vectors whose elements are either -1 or $+1$.

Finding the optimal DT: model internal nodes.

- We use the splitting function $s_i(x) : \mathbb{R}^p \mapsto \{-1, 1\}$ to represent an internal node.
- If $s_i(x) = -1$, it means that if we put observation x on node i, it will be classified into the left subtree, and vice versa.
- Just like OC1, we shall assume that $s_i(x) = \operatorname{sgn}(w_i^\top x)$, where $w_i : (p \times 1)$ is a column vector describing the decision process inside node i.
- Note that by introducing w_i, the decision boundaries can be oblique.
- There is no need to introduce an "intercept term" such as $s_i(x) = \operatorname{sgn}(w_i^\top x + b_i)$, since this can be achieved by adding a column of 1 to our X matrix.
- Therefore, based on the discussion above, we can use

  $$W_{(m \times p)} := (w_1^\top, w_2^\top, ..., w_m^\top)^\top$$

  to represent the current status of all internal nodes.

Finding the optimal DT: model leaf nodes.

- For a leaf node j, we use the predictive log-probability $\theta_j[l] := \log \Pr(y = l|j)$ to describe its prediction on class l.
- Apparently θ_j is a $(k \times 1)$ column vector.
- Similar with internal nodes, we can use

$$\Theta_{(m+1)\times k} := (\theta_1^\top, \theta_2^\top, ..., \theta_{m+1}^\top)^\top$$

to describe our current status of leaves in the DT.

Finding the optimal DT: link internal nodes to leaves.

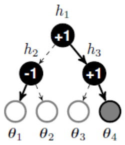
- The key to linking internal nodes to leaves is to define a latent vector

$$h := \operatorname{sgn}(\mathrm{Wx}) \in \{-1, 1\}^m := \mathbb{H}^m$$

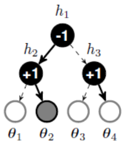
- Intuitively speaking, for a single input x, if we put it on each of the m internal nodes (no matter whether x will be directed to that node in reality), and combine all the results to a column vector, we will get h.
- Such a vector fully determines the leaf to which a data point is directed.

Finding the optimal DT: link internal nodes to leaves

- After having the latent vector h, let's further define a *tree navigation function* $f : \mathbb{H}^m \mapsto \mathbb{I}_{m+1}$ that maps an m-bit sequence of decisions (i.e., $h = \text{sgn}(Wx)$) to a one-hot vector specifying the selected leaf.
- Some examples are as below.



$$f([+1, -1, +1]^\top) = [0, 0, 0, 1]^\top = \mathbb{1}_4$$
$$\theta = \Theta^\top f(\mathrm{h}) = \theta_4$$



$$f([-1, +1, +1]^\top) = [0, 1, 0, 0]^\top = \mathbb{1}_2$$
$$\theta = \Theta^\top f(\mathrm{h}) = \theta_2$$

- Note that f is fully determined by Wx.
- Therefore, using the notation above, $\theta := \Theta^\top f(\text{sgn}(Wx))$ is the prediction of our DT when the input is x.

Finding the optimal DT: define the loss function.

- Clearly a generic loss function should be of the form $l = l(\theta, y)$ where θ is our predicted probability mass function and y is the true ground label.
- For a regression problem, y and θ are just scalars, and we can simply define $l(\theta, y) := (\theta - y)^2$
- For a multi-classification problem, we can use the *log loss*

$$l(\theta, y) := -\theta[y] + \log\big(\sum_{\beta=1}^{k} \exp\theta[\beta]\big)$$

- The goal of learning is just to find $\{W, \Theta\}$ that minimize the empirical loss given a training set $\mathcal{D}$

$$\mathcal{L}(W, \Theta; \mathcal{D}) := \sum_{(x,y)\in\mathcal{D}} l(\theta, y) = \sum_{(x,y)\in\mathcal{D}} l(\Theta^\top f(\operatorname{sgn}(Wx)), y) \tag{2}$$

Finding the optimal DT: is there something magical going on?

- Obviously by introducing W, Θ and f, we completed the task of modelling a DT.
- Moreover, it seems that we de-coupled the before-and-after order between internal and child nodes too!
- But how did we achieve that?
- We de-coupled W and Θ by incorporating some "useless information": for an observation x, from the root to the leaf it belongs to, it will only go through $\sim \log(m)$ nodes, but we incorporated the decision of every internal node in W.
- But that "redundant information", in turn, gives us the freedom to de-couple the dependency in the time axis!
- This is the real magical part of this paper.

Finding the optimal DT: make the loss function nice and smooth.

- Note that there is no way for us to compute $\frac{\partial \mathcal{L}(W,\Theta;\mathcal{D})}{\partial W}$ during the optimization process.
- The problem can be solved by the following observation

$$\operatorname{sgn}(Wx) = \arg\max_{h \in \mathbb{H}^m}(h^\top Wx)$$

- Using the equation and (2), we can re-express our loss function as

$$\mathcal{L}(W,\Theta;\mathcal{D}) = \sum_{(x,y)\in\mathcal{D}} l(\Theta^\top f(\hat{h}(x)), y) \\ \text{where } \hat{h}(x) = \arg\max_{h\in\mathbb{H}^m}(h^\top Wx) \tag{3}$$

- This is the form of loss function we shall use in the next step.

Finding the optimal DT: make the loss function nice and smooth.

- But the loss function (3) is still not easy to take gradient and thus needs further modification.
- Here we develop an upper bound for $l(\Theta^\top f(\mathrm{sgn}(Wx)), y)$

$$\begin{aligned} l(\Theta^\top f(\mathrm{sgn}(Wx)), y) &\leq \max_{g \in \mathbb{H}^m} (g^\top Wx + l(\Theta^\top f(g), y)) - \max_{h \in \mathbb{H}^m} (h^\top Wx) \\ &:= \mathrm{I} - \mathrm{II} \end{aligned} \tag{4}$$

- The proof of (4) can be found in reference 3.
- One important observation is that LHS can be achieved when $g = h = \mathrm{sgn}(Wx)$ in RHS.
- Term II (and also the first part in term I) is called the *inference problem*, as it infers the latent variable h.
- Term I is called the *loss-augmented inference*, as it augments the inference problem by an additional loss term.

Finding the optimal DT: make the loss function nice and smooth.

- Notice that in (4), the LHS does not depend on the scale of W, whereas the RHS does.
- To dig a little deeper, let's divert a bit and discuss the effect of $||W||$ on the upper bound. For $a \geq b \geq 0$,

$$\max_{g \in \mathbb{H}^m} (ag^\top Wx + l(\Theta^\top f(g), y)) - \max_{h \in \mathbb{H}^m} (ah^\top Wx) \leq$$
$$\max_{g \in \mathbb{H}^m} (bg^\top Wx + l(\Theta^\top f(g), y)) - \max_{h \in \mathbb{H}^m} (bh^\top Wx)$$

- That is to say, the larger the scale, the tighter the upper bound.

Finding the optimal DT: make the loss function nice and smooth.

- Therefore, we want the scale of W to be large.
- But on the other hand, when $||W||$ approaches $+\infty$, the loss term $l(\Theta^\top f(g), y)$ in term I of (4) becomes negligible compared with $g^\top Wx$. Therefore, the solutions to our loss-augmented inference and the original inference problem will be almost identical.
- This is not good, because even though a larger $||W||$ yields a tighter bound, it actually makes the bound approach the loss itself, and therefore becomes nearly piecewise constant, which is hard to optimize.
- In conclusion, we do not want $||W||$ to be either too large or too small. A reasonable approach would be to set a hard constraint on $||W||$.
- Let's come back to our optimization problem.

Finding the optimal DT: the optimization problem

- Finally, we formulate our optimization problem as

$$\text{minimize } \mathcal{L}'(W, \Theta; \mathcal{D}) = \sum_{(x,y)\in\mathcal{D}} \{\max_{g\in\mathbb{H}^m}(g^\top Wx + l(\Theta^\top f(g), y)) - \max_{h\in\mathbb{H}^m}(h^\top Wx)\}$$
$$s.t. \;\; ||w_i|| \leq \nu, \; \forall i \in \{1, ..., m\} \tag{5}$$

- Here w_i is the ith row of matrix W.
- $\mathcal{L}'(W, \Theta; \mathcal{D})$ is called the *surrogate objective.* This is the optimization problem we want to solve.
- We can apply constraints on each w_i separately because each node (represented by w_i) acts independently in the DT in terms of the splitting decision.

Finding the optimal DT: algorithm to solve the optimization problem

- We will definitely use the gradient descent algorithm family to solve the optimization problem.
- But during that process, we also have to evaluate our target function given a pair of (W, Θ) from time to time.
- That is to say, we need to solve the maximization sub-problem of term I and II.
- As discussed before, the solution to term II is just $\hat{h} = \operatorname{sgn}(Wx)$.
- If we can solve term I, then we can build the optimal DT following the algorithm on the next page.

Finding the optimal DT: algorithm to solve the optimization problem

```
1: Initialize $W^{(0)}, \Theta^{(0)}$ using OC1.
2: **for** $t = 0$ to $\tau$ **do**
3:     Sample a pair $(x, y)$ randomly from $\mathcal{D}$.
4:     $\hat{h} \leftarrow \mathrm{sgn}(W^{(t)}x)$ (solution to term II)
5:     $g \leftarrow \arg\max_{g \in \mathbb{H}^m} \{g^\top W^{(t)} x + l(\Theta^\top f(g), y)\}$ (solution to term I)
6:     $W^{(tmp)} \leftarrow W^{(t)} - \eta \hat{g} x^\top + \eta \hat{h} x^\top$ (SGD for $W$, $\eta$ is learning rate)
7:     **for** $i = 1$ to $m$ **do**
8:         $W_{i,:}^{(t+1)} \leftarrow \min\{1, \frac{\sqrt{\nu}}{||W_{i,:}^{(tmp)}||_2}\} \cdot W_{i,:}^{(tmp)}$ (projection operation)
9:     **end for**
10:    $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \frac{\partial}{\partial \Theta} l(\Theta^\top f(\hat{g}), y)\Big|_{\Theta = \Theta^{(t)}}$ (SGD for $\Theta$)
11: **end for**
```

Algorithm 4: Algorithm to construct the optimal DT

Finding the optimal DT: algorithm to solve the optimization problem

- So far, there is only one problem left: how to solve the maximization problem of term I

$$\hat{g}(x) = \arg\max_{g \in \mathbb{H}^m} \{ g^\top W x + l(\Theta^\top f(g), y) \}$$

- This is actually not as scary as it looks like.
- An observation that makes it tractable is that $f(g)$ can only take on $m+1$ distinct values, which correspond to terminating at one of the $m+1$ leaves and selecting a distribution from $\{\theta_j\}_{j=1}^{m+1}$.

Finding the optimal DT: algorithm to solve the optimization problem

- That is to say, we actually just need to solve the following problem

$$\hat{g}(x) = \arg\max_{g \in \mathbb{H}^m} \{g^\top W^x + l(\theta_j, y)\}, \;\; s.t. \;\; f(g) = \mathbb{I}_j = (0, .., 0, 1 \; (j\text{th index}), 0, ..., 0)^\top$$

- Now it becomes clear about how to solve the problem:
    1. For any given j, set all the binary bits in g corresponding to the path from the root to leaf j to be consistent with the path direction towards leaf j in our DT, according to W.
    2. Bits of g that do not appear on this path have no effect on the output of $f(g)$ and should be set based on $g[i] = \text{sgn}(w_i^\top x)$ to obtain the maximum $g^\top Wx$.
    3. Repeat step 1 & 2 for each j, and find the $j^\star$ that maximizes the target function. Then $\hat{g}(x) = f^{-1}(\mathbb{I}_{j^\star})$.

Finding the optimal DT: time complexity analysis

- Let's analysis the time complexity of the optimization problem above.
- Denote the height (i.e., depth) of our DT as $d \sim \log_2(m) \sim \log(m)$.
- For each data point, we search for each $j = 1 : m + 1$, and the within each internal node i, the time complexity of computing inner product $w_i^\top x$ is $\mathcal{O}(p)$.
- Thus the time complexity of each round of SGD is $\mathcal{O}(2^d p) \sim \mathcal{O}(mp)$.
- The complexity is too high to let us train a deep tree.

Finding the optimal DT: improve the time complexity.

- We can improve the time complexity by using a slightly different upper bound on the loss

$$l(\Theta^\top f(\text{sgn}(Wx)), y) \leq \max_{g \in \mathbb{B}_1(\text{sgn}(Wx))} (g^\top Wx + l(\Theta^\top f(g), y)) - \max_{h \in \mathbb{H}^m} (h^\top Wx) \quad (6)$$

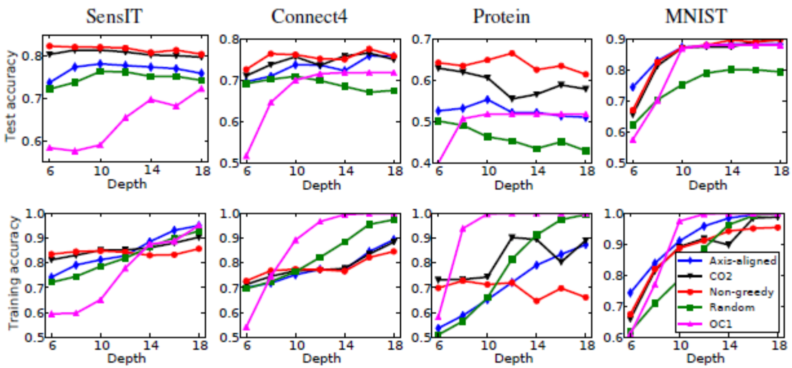
  where $\mathbb{B}_1(\text{sgn}(Wx))$ is the set of all vectors which differ from $\text{sgn}(Wx)$ by at most one bit.
- This set is called the *Hamming ball* of radius 1 around $\text{sgn}(Wx)$.
- The proof can be found in reference 3.
-
  - Since $\hat{g}$ and $\text{sgn}(Wx)$ can only differ in at most one bit;
  - and among all the bits in $\hat{g}$, we care about only d of them,
- $\hat{g}$ can only take $\sim d$ different values, and $f(\hat{g})$ can also take $\sim d$ different values, instead of $m = 2^d$ in the prior case.
- Therefore, after the improvement, the time complexity of each round of gradient descent is $\mathcal{O}(dp) = \mathcal{O}(p \log m)$.[1]

[1] It confuses me that time complexity here is $\mathcal{O}(d^2 p)$, according to the paper.

Finding the optimal DT

- We have now finished all the technical parts of finding the optimal DT.
- Let's see the results provided by the paper.

Finding the optimal DT: results and discussion



- In terms of test accuracy, non-greedy tree is the best among almost all the four datasets.
- Moreover, in terms of the shape similarity between in-sample and out-of-sample accuracy curves, the non-greedy tree is the best. This indicates that non-greedy tree does have less overfitting.

Current section

Take-aways

- OC1
    - OC1 is a promising extension on the regular DT.
    - OC1 is almost guaranteed to be better than the regular DT, since the latter is a subset of OC1.
- Non-greedy optimal DT
    - Finding the “optimal DT” is a complicated problem, which is still under active research now.
    - The difficulties mostly come from the before-and-after order between the internal nodes and leaves. This order makes it difficult to apply optimization techniques.
    - The problem above can be solved by incorporating some information which may be redundant for now but useful later in the evolution of our DT.

References

Sreerama K. Murthy, Simon Kasif, Steven Salzberg and Richard Beigel
OC1: A randomized algorithm for building oblique decision trees
Proceedings of AAAI (1993)

Mohammad Norouzi, Maxwell Collins, Matthew A. Johnson, David J. Fleet and Pushmeet Kohli
Efficient Non-greedy Optimization of Decision Trees
Advances in Neural Information Processing Systems 28 (NIPS 2015)

Mohammad Norouzi, Maxwell Collins, Matthew A. Johnson, David J. Fleet and Pushmeet Kohli
Efficient Non-greedy Optimization of Decision Trees: Supplementary Material
Advances in Neural Information Processing Systems 28 (NIPS 2015)