Automation of fine-tuning Hyper-parameters using pseudo-dimension

Priyanka Aravindan

February 2, 2023

Contents

1 Introduction and Organisation of the report

Although Machine Learning has become the common buzzword and used in every domain, there are still a lot of scope for research, especially mathematical and algorithmic research. I am mainly interested in generic algorithms that works on such wide and diverse data with high mathematical basis to be converge to the expected result. This report is a summary of what I have learnt during the internship with Dr Kim Thang Nguyen.

The ultimate goal is to understand pseudo-dimension and how it can used to tune hyper-parameters in algorithms, with an small implementation of example of how to model a Machine learning problem into a constraint satisfaction problem and use the knowledge of pseudo-dimension. For which I start the report with introducing machine learning aspects such as PAC Learning, VC - dimension and Pseudo dimension. Further I proceed to explore the bounds of pseudo-dimension with examples of Knapsack and Maximum-Weight Independent Set (MWIS). Then finally how we use these information to model an algorithm that uses Clustering and a toy example I implemented.

2 Aspects of Machine Learning

In this section I explain the basics that are needed to understand the Algorithm and proofs that is explained in the later sections.

2.1 PAC Learning

The PAC model belongs to that class of learning models which is characterized by learning from examples. In these models, say if f is the target function to be learnt, the learner is provided with some random examples (actually these examples may be from some probability distribution over the input space, as we will see in a short while) in the form of $(X, f(X))$ where X is a binary input instance, say of length n, and $f(X)$ is the value (boolean TRUE or FALSE) of the target function at that instance.

Based on these examples, the learner must succeed in deducing the target function f which we can now express as $f : \{0, 1\}^n \rightarrow 0, 1$. Before moving on, lets consider an important question. What actually should we mean by 'succeeded' ? Please agree that the notion of success better be reasonable enough so as to enable us to find interesting algorithms meeting its criteria. Hence,

- We may allow a small probability that the learning algorithm fails. This would be in some kind of worst cases where the examples that are picked up from the distribution and provided to the learner don't contain much information to help deduce the target function. In fact, as you would be able to see later in the illustrations, PAC Learning actually tries to reduce the probability of getting such unlucky examples by considering a sufficient number of examples for learning a particular concept. In other words, the PAC learning strategy places an upper bound on the probability of committing an error by placing a minimum bound on the number of examples/samples it takes to learn a target concept.
- We may spare the learner from learning the exact concept. If the learner can find a function h which is close to f i.e. if h is an approximates f , we would still be interested in it.

The above two factors that we used to explain the meaning of success are quantified as ϵ and δ in the formal PAC literature. ϵ gives an upper bound on the error in accuracy with which h approximated f and δ gives the probability of failure in achieving this accuracy. Using both these quantities, we can express the definition of a PAC Algorithm with more mathematical clarity. Consequently, we can say that, to qualify for PAC Learnability, the learner must find with probability of at least $1 - \delta$, a concept h such that the error between h and f is at most ϵ.

To make more mathematical sense of the above ideas, we define a probability distribution D over the input domain $X : \{0, 1\}^n$. So when we say that we pick an example at random, we mean that we are picking an example from this distribution. Also, its imperative to mention that the all the examples that learner is provided with are independent of each other. So, if we are going to consider the total probability of encountering a sequence of examples over D , we will use the product rule i.e. we would be multiplying the probabilities. A major role played by D, in our setting is that it conveniently and mathematically quantifies the error with which h approximates f. i.e. we can write $error(h, f) = Pr_{x \in D}(h(x) \neq f(x))$

This section casts the problem of choosing the best algorithm for a poorly understood application domain as one of learning the optimal algorithm with respect to an unknown instance distribution.

Our basic model consists of the following ingredients.

- A fixed computational or optimization problem Π.For example, Π could be the problem of computing a maximum-weight independent set of a graph or a clustering problem.
- An unknown distribution D over instances $x \in \Pi$.
- A set $\mathcal{A}$ of algorithms for Π. For example, A could be a finite set of SAT solvers or an infinite family of greedy heuristics.
- A performance measure cost: $\mathcal{A} \times \Pi \to [0, H]$, indicating the performance of a given algorithm on a given instance. Two common choices for cost are the running time of an algorithm and, for optimization problems, the objective function value of the solution produced by an algorithm.

The "application-specific" information is encoded by the unknown input distribution D, and the corresponding "application-specific optimal algorithm" $\mathcal{A}_D$ is the algorithm that minimizes or maximizes (as appropriate) over the algorithms $A \in \mathcal{A}$. The error of an algorithm $A \in \mathcal{A}$ for a distribution D is In our basic model, the goal is: Learn the application-specific optimal algorithm from data (i.e., samples from D).

More precisely, the learning algorithm is given m i.i.d. samples $x_1, \cdots, x_m \in \Pi$ from D and (perhaps implicitly) the corresponding performance $cost(A, x_i)$ of each algorithm $A \in \mathcal{A}$ on each input x_i. The learning algorithm uses this information to suggest an algorithm to use on future inputs drawn from D. We seek learning algorithms that almost always output an algorithm of A that performs almost as well as the optimal algorithm in A for D-learning algorithms that are probably approximately correct (PAC). A learning algorithm $L(\epsilon, \delta)$-learns the optimal algorithm in A from m samples if, for every distribution D over Π, with probability at least $1 - \delta$ over m samples $x_1, \cdots, x_m$ D, L outputs an algorithm with error at most ϵ.

2.2 VC Dimension

A set S of size of m is shattered by H if $|\Pi_H(S)| = 2^m$, i.e. all possible labelings of the set S are realized by functions in H. $VC - dim(H)$ = cardinality of the largest set shattered by H

Example. (Intervals) For the case when $H = intervals$, it is illustrated in Figure 1 that H can shatter S of 2 points but cannot shatter S of 3 points. Therefore, $VC - dim(intervals) = 2$. Note: we can see that, we need to show $VC - dim$ is at least some number d and then show that $VC - dim$ is at most d to draw the conclusion that $VC - dim = d$. To show $VC - dim$ is at least d, we need to find just one set of d points that are shattered (not for every set of d points). To show $VC - dim$ is at most d, we need to show every set of $d + 1$ points is not shattered.

Example. (Axis-aligned Rectangles) For the case when H = axis-aligned rectangles, $VC - dim = 4$. (Illustrated in Figure 2)

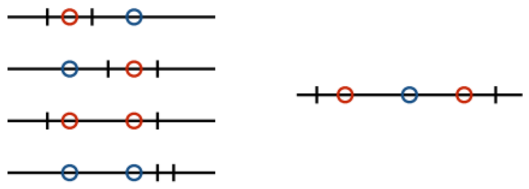


Figure 1: : Left: Case for 2 points that all labelings are realized. Right: For any three points, when the middle point has “-” label and the other two have “+” labels, this means the interval must contain all three points, which means it can not be shattered.

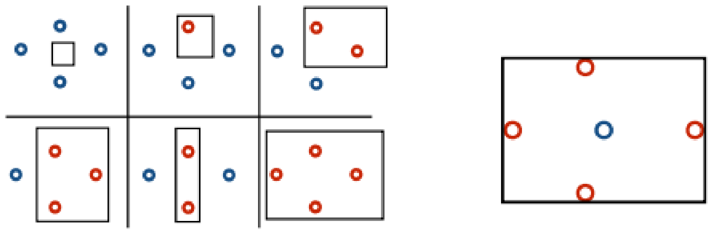


Figure 2: Left: A set of 4 points that can be shattered by axis-aligned rectangles. Right: For any 5-point set, we can choose the topmost, bottom-most, leftmost and rightmost points and assign “+” to them, and the remaining point is assigned to “-”. Any rectangle that contains the “+” points must also contain “-”, which means this case cannot be shattered.

2.3 Pseudo Dimension

Let H denote a set of real-valued functions defined on the set X. A finite subset $S = \{x_1, ..., x_m\}$ of X is (pseudo-)shattered by H if there exist real-valued witnesses $r_1, ..., r_m$ such that, for each of the 2^m subsets T of S, there exists a function $h \in H$ such that $h(x_i) > r_i$ if and only if $i \in T($ for $i = 1, 2, ..., m)$. The pseudo-dimension of H is the cardinality of the largest subset shattered by H (or $+\infty$, if arbitrarily large finite subsets are shattered by H). pseudo-dimension is a natural extension of the VC dimension from binary-valued to real-valued functions.

2.4 Uniform Convergence

Let H be a class of functions with domain X and range in $[0, H]$, and suppose H has pseudo-dimension d_H. For every distribution D over X, every ϵ ¿ 0, and every $\delta \in (0, 1]$, if

$$m \geq c\left(\frac{H}{\epsilon}\right)^2\left(d_H + ln\left(\frac{1}{\delta}\right)\right) \tag{1}$$

for a suitable constant c (independent of all other parameters), then with probability at least $1 - \delta$ over m samples $x_1, ..., x_m$ D

$$\left|\left(\frac{1}{m}\sum_{i=1}^{m} h(x_i)\right) - \mathbb{E}_{x\ D}[h(x)]\right| < \epsilon \tag{2}$$

We can identify each algorithm $A \in \mathcal{A}$ with the real-valued function $x \rightarrow cost(A, x)$. Regarding the class $\mathcal{A}$ of algorithms as a set of real-valued functions defined on Π, we can discuss its pseudo-dimension, as defined above. We need one more definition before we can apply our machinery to learn algorithms from $\mathcal{A}$. (This part is a theorem - the proof will not be discussed)

2.5 Empirical Risk Minimization (ERM)

Fix an optimization problem Π, a performance measure cost, and a set of algorithms $\mathcal{A}$. An algorithm L is an ERM algorithm if, given any finite subset S of Π, L returns an algorithm from $\mathcal{A}$ with the best average performance on S. For example, for any Π, cost, and finite $\mathcal{A}$, there is the trivial ERM algorithm that simply computes the average performance of each algorithm on S by brute force and returns the best one. Fix parameters $\epsilon > 0$, $\delta \in (0, 1]$, a set of problem instances Π, and a performance measure cost. Let $\mathcal{A}$ be a set of algorithms that has pseudo-dimension d with respect to Π and $cost$. Then, any ERM algorithm $(2\epsilon, \delta)$ learns the optimal algorithm in $\mathcal{A}$ from m samples, where m is defined as in equation 1

The same reasoning applies to learning algorithms beyond ERM. For example, with the same assumptions as in Corollary 3.4, an algorithm from $\mathcal{A}$ with approximately optimal (over A) average performance on a set of samples is also approximately optimal with respect to the true input distribution. This observation is particularly useful when the ERM problem is computationally difficult but can be approximated efficiently. trivially at most $log_2|\mathcal{A}|$. The following subsections demonstrate the much less obvious fact that natural infinite classes of algorithms also have small pseudo-dimension.

3 Learning greedy heuristics

The goal of this section is to bound the pseudo-dimension of many classes of greedy heuristics. Throughout this section, the performance measure cost is the objective function value of the solution produced by a heuristic on an instance, where we assume without loss of generality a maximization objective.

3.1 Definitions and examples

Our general definitions are motivated by greedy heuristics for NP-hard problems. For example:

- Knapsack. The input is n items with values $v_1, ..., v_n$, sizes $s_1, ..., s_n$, and a knapsack capacity C. The goal is to compute a subset $S \subseteq \{1, 2, ...n\}$ with maximum total value $\sum_{i \in S} v_i$, subject to having total size $\sum_{i \in S} s_i$ at most C. Two natural greedy heuristics are to greedily pack items (subject to feasibility) in order of nonincreasing value vi or in order of nonincreasing density v_i/s_i (or to take the better of the two).

- Maximum-Weight Independent Set (MWIS). The input is an undirected graph $G = (V, E)$ and a non-negative weight w_v for each vertex $v \in V$. The goal is to compute an independent set-a subset of mutually nonadjacent vertices-with maximum total weight. Two natural greedy heuristics are to greedily choose vertices (subject to feasibility) in order of nonincreasing weight w_v or nonincreasing density $w_v/(1 + deg(v))$. (The intuition for the denominator is that choosing v "uses up" $1 + deg(v)$ vertices-v and all of its now-blocked neighbors.) The latter heuristic also has an adaptive variant, where the degree $deg(v)$ is computed in the subgraph induced by the vertices not yet blocked from consideration, rather than in the original graph.

In general, we consider object assignment problems, where the input is a set of n objects with various attributes, and the feasible solutions consist of assignments of the objects to a finite set R, subject to feasibility constraints. The attributes of an object are represented as an element ξ of an abstract set. For example, in the Knapsack problem, ξ encodes the value and size of an object; in the MWIS problem, ξ encodes the weight and (original or residual) degree of a vertex. In the Knapsack and MWIS problems, $R = \{0, 1\}$, indicating whether or not a given object is selected. By a greedy heuristic, we mean algorithms of the following form While there remain unassigned objects:

1. Use a scoring rule σ (described below) to compute a score $\sigma(\xi_i)$ for each unassigned object i, as a function of its current attributes ξ_i.

2. For the unassigned object i with the highest score,

   use an assignment rule to assign i a value from R and, if necessary, update the attributes of the other unassigned objects. For concreteness, assume that ties are always resolved lexicographically. A scoring rule assigns a real number to an object as a function of its attributes. Assignment rules that do not modify objects' attributes yield nonadaptive greedy heuristics, which use only the original attributes of each object (such as v_i or v_i/s_i in the Knapsack problem, for instance). In this case, objects' scores can be computed in advance of the main loop of the greedy heuristic. Assignment rules that modify object attributes yield adaptive greedy heuristics, such as the adaptive MWIS heuristic described above.

In a single-parameter family of scoring rules, there is a scoring rule of the form $\sigma(\rho, \xi)$ for each parameter value ρ in some interval $I \subseteq \mathbb{R}$. Moreover, σ is assumed to be continuous in ρ for each fixed value of ξ. Natural examples include Knapsack scoring rules of the form and MWIS scoring rules of the form $w_v/(1 + deg(v))\rho$ for $\rho \in [0, 1]$ or $\rho \in [0, \infty)$. A single-parameter family of scoring rules is κ-crossing if, for each distinct pair of attributes ξ, ξ', there are at most κ values of ρ for which $\sigma(\rho, \xi) = \sigma(\rho, \xi')$. For example, all of the scoring rules mentioned above are 1-crossing rules. For an example assignment rule, in the Knapsack and MWIS problems, the rule simply assigns i to "1" if it is feasible to do so, and to "0" otherwise. In the adaptive greedy heuristic for the MWIS problem, whenever the assignment rule assigns "1" to a vertex v, it updates the residual degrees of other unassigned vertices (two hops away) accordingly. We call an assignment rule β-bounded if every object i is guaranteed to take on at most β distinct attribute values. For example, an assignment rule that never modifies an object's attribute is 1-bounded. The assignment rule in the adaptive MWIS algorithm is n-bounded, where n is the number of vertices, as it only modifies the degree of a vertex (which lies in $\{0, 1, 2, ..., n - 1\}$). Coupling a single-parameter family of κ-crossing scoring rules with a fixed β-bounded assignment rule yields a (κ, β)-single-parameter family of greedy heuristics. All of our running examples of greedy heuristics are (1,1)-singleparameter families, except for the adaptive MWIS heuristic, which is a (1,n)-single-parameter family.

3.2 Upperbound on Pseudodimension

We next show that every (κ, β)-single-parameter family of greedy heuristics has small pseudo-dimension. This result applies to all of the concrete examples mentioned above.

If $\mathcal{A}$ is a (κ, β)-single-parameter family of greedy heuristics for an object assignment problem with n objects, then the pseudodimension of $\mathcal{A}$ is $O(log(\kappa\beta n))$. In particular, all of our running examples are classes of heuristics with pseudo-dimension $O(logn)$.

Proof. Recall from the definitions that we need to upper bound the size of every set that is shatterable using the greedy heuristics in $\mathcal{A}$. For us, a set is a fixed set of s inputs (each with n objects) $S = \{x_1, ..., x_s\}$. For a potential witness $r_1, ..., r_s \in \mathbb{R}$, every algorithm $A \in \mathcal{A}$ induces a binary labeling of each sample x_i, according to whether $cost(A, x_i)$ is strictly more than or at most r_i. We proceed to bound from above the number of distinct binary labelings of S induced by the algorithms of A, for any potential witness. Consider ranging over algorithms $A \in \mathcal{A}$-equivalently, over parameter values $\rho \in I$. The trajectory of a greedy heuristic $A \in \mathcal{A}$ is uniquely determined by the outcome of the comparisons between the current scores of the unassigned objects in each iteration of the algorithm. Because the family uses a κ-crossing scoring rule, for every pair i, j of distinct objects and possible attributes ξ_i, ξ_j with $\xi_i \neq \xi_j$, there are at most κ values of ρ for which there is a tie between the score of i (with attributes ξ_i) and that of j (with attributes ξ_j). Because σ is continuous in ρ for every fixed ξ, the relative order of the score of i (with ξ_i) and j (with ξ_j) remains the same in the open interval between two successive values of ρ at which their scores are tied. The upshot is that we can partition I into at most $\kappa + 1$ intervals, such that the outcome of the comparison between i (with attributes ξ_i) and j (with attributes ξ_j) is constant on each interval. In the case that $\xi_i = \xi_j$, because we break ties between equal scores lexicographically, the outcome of the comparison between $\sigma(\rho, \xi_i)$ and $\sigma(\rho, \xi_j)$ is the same for all $\rho \in I$. Next, the s instances of S contain a total of sn objects. Each of these objects has some initial attributes. Because the assignment rule is β-bounded, there are at most $sn\beta$ object-attribute pairs (i, ξ_i) that could possibly arise in the execution of any algorithm from A on any instance of S. This implies that, ranging across all algorithms of A on all inputs in S, comparisons are only ever made between at most $(sn\beta)^2$ pairs of object-attribute pairs (i.e., between an object i with current attributes ξ_i and an object j with current attributes ξj). We call these the relevant comparisons. For each relevant comparison, we can partition I into at most $\kappa+1$ subintervals such that the comparison outcome is constant (in ρ) in each subinterval. Intersecting the partitions of all of the at most $(sn\beta)^2$ relevant comparisons splits I into at most $(sn\beta)^2\kappa + 1$ subintervals such that every relevant comparison is constant in each subinterval. That is, all of the algorithms of A that correspond to the parameter values ρ in such a subinterval execute identically on every input in S. The number of binary labelings of S induced by algorithms of A is trivially at most the number of such subintervals. Our upper bound $(sn\beta)^2\kappa + 1$ on the number of subintervals exceeds 2^s, the requisite number of labelings to shatter S, only if $s = O(log(\kappa\beta n))$.

4 Mixed Integer Linear Programming

I start with a small introduction and definitions for linear program formulations without discussing the methods to solve and further proceeding to explain about the formulation of Mixed integer linear program.

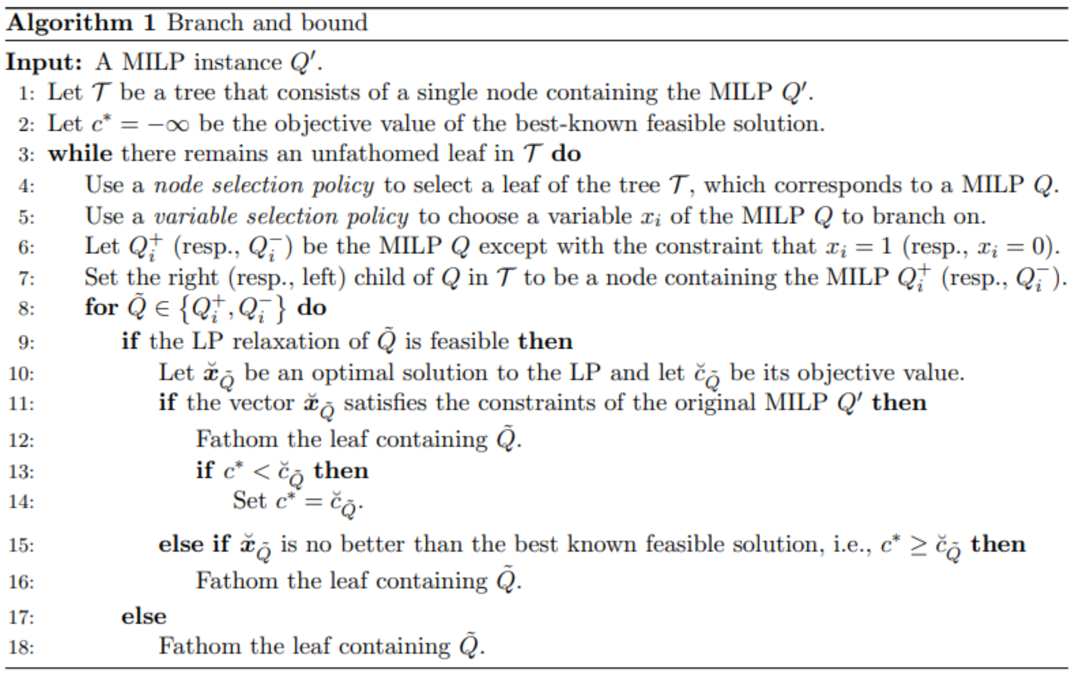
4.1 Definition

The linear-programming models are continuous, in the sense that decision variables are allowed to be fractional. Often this is a realistic assumption. It is of the form

$$\begin{aligned}\text{max/min} \sum_{j=1}^{n} c_j x_j \\ \text{sub to} \sum_{j=1}^{n} a_{ij} x_j = b_i \forall i \in \{1, 2, \cdots, m\} \\ x_j \geq 0 \forall i \in \{1, 2, \cdots, n\}\end{aligned}$$

Note that the constrains can also be $\sum_{j=1}^{n} a_{ij} x_j \geq b_i \forall i \in \{1, 2, \cdots, m\}$ or $\sum_{j=1}^{n} a_{ij} x_j \leq b_i \forall i \in \{1, 2, \cdots, m\}$. In case of integer programming there is an extra constrain that x_j can only be integer. A generalization of this is Mixed Integer Linear Program(MILP) which is some x_j are integers and some are continuous variables. MILP is NP-Hard.

4.2 MILP Tree search Algorithm



Algorithm 1 Branch and bound

```
Input: A MILP instance $Q'$.
 1: Let $\mathcal{T}$ be a tree that consists of a single node containing the MILP $Q'$.
 2: Let $c^* = -\infty$ be the objective value of the best-known feasible solution.
 3: while there remains an unfathomed leaf in $\mathcal{T}$ do
 4:     Use a node selection policy to select a leaf of the tree $\mathcal{T}$, which corresponds to a MILP $Q$.
 5:     Use a variable selection policy to choose a variable $x_i$ of the MILP $Q$ to branch on.
 6:     Let $Q_i^+$ (resp., $Q_i^-$) be the MILP $Q$ except with the constraint that $x_i = 1$ (resp., $x_i = 0$).
 7:     Set the right (resp., left) child of $Q$ in $\mathcal{T}$ to be a node containing the MILP $Q_i^+$ (resp., $Q_i^-$).
 8:     for $\tilde{Q} \in \{Q_i^+, Q_i^-\}$ do
 9:         if the LP relaxation of $\tilde{Q}$ is feasible then
10:             Let $\breve{\boldsymbol{x}}_{\tilde{Q}}$ be an optimal solution to the LP and let $\breve{c}_{\tilde{Q}}$ be its objective value.
11:             if the vector $\breve{\boldsymbol{x}}_{\tilde{Q}}$ satisfies the constraints of the original MILP $Q'$ then
12:                 Fathom the leaf containing $\tilde{Q}$.
13:                 if $c^* < \breve{c}_{\tilde{Q}}$ then
14:                     Set $c^* = \breve{c}_{\tilde{Q}}$.
15:             else if $\breve{\boldsymbol{x}}_{\tilde{Q}}$ is no better than the best known feasible solution, i.e., $c^* \geq \breve{c}_{\tilde{Q}}$ then
16:                 Fathom the leaf containing $\tilde{Q}$.
17:         else
18:             Fathom the leaf containing $\tilde{Q}$.
```

Figure 3: MILP Tree Search Algorithm

MILPs are typically solved using a tree search algorithm called branch-and-bound. Given a MILP problem instance, Bran and Bound relies on two subroutines that efficiently compute upper and lower bounds on the optimal value within a given region of the search space. The lower bound can be found by choosing any feasible point in the region. An upper bound can be found via a linear programming relaxation. The basic idea of BB is to partition the search space into convex sets and find upper and lower bounds on the optimal solution within each. The algorithm uses these bounds to form global upper and lower bounds, and if these are equal, the algorithm terminates, since the feasible solution corresponding to the global lower bound must be optimal. If the global upper and lower bounds are not equal, the algorithm refines the partition and repeats. In more detail, suppose we want to use BB to solve a MILP Q'. BB iteratively builds a search tree T with the original MILP Q' at the root. In the first iteration, T consists of a single ode containing the MILP Q'. At

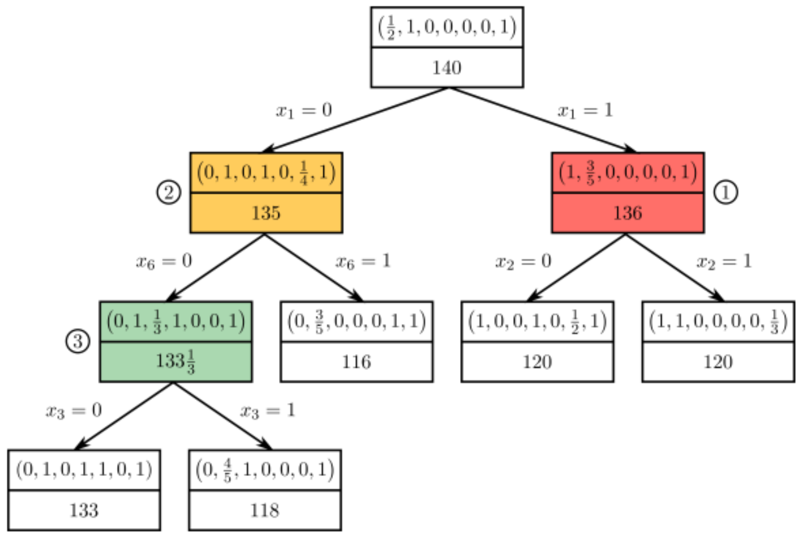

$(\frac{1}{2},1,0,0,0,0,1)$
140
$x_1=0$
$x_1=1$
2
$(0,1,0,1,0,\frac{1}{4},1)$
135
$(1,\frac{3}{5},0,0,0,0,1)$
136
1
$x_6=0$
$x_6=1$
$x_2=0$
$x_2=1$
3
$(0,1,\frac{1}{3},1,0,0,1)$
$133\frac{1}{3}$
$(0,\frac{3}{5},0,0,0,1,1)$
116
$(1,0,0,1,0,\frac{1}{2},1)$
120
$(1,1,0,0,0,0,\frac{1}{3})$
120
$x_3=0$
$x_3=1$
$(0,1,0,1,1,0,1)$
133
$(0,\frac{4}{5},1,0,0,0,1)$
118

Figure 4: Illustration of MILP tree search Algorithm with an example

each iteration, BB uses a node selection policy (which we expand on later) to select a leaf node of the tree T , which corresponds to a MILP Q. BB then uses a variable selection policy to choose a variable x_i of the MILP Q to branch on. Specifically, let Q_i^+(resp., Q_i^-) be the MILP Q except with the additional constraint that $x_i = 1$ (resp., $x_i = 0$). BB sets the right (resp., left) child of Q in T to be a node containing the MILP Q_i^+(resp., Q_i^-). BB then tries to "fathom" these leafs: the leaf containing Q_i^+(resp., Q_i^-) is fathomed if:

1. The optimal solution to the LP relaxation of Q_i^+(resp., Q_i^-) satisfies the constraints of the original MILP Q'
2. The relaxation of is infeasible, so Q_i^+(resp., Q_i^-) must be infeasible as well.
3. The objective value of the LP relaxation of Q_i^+(resp., Q_i^-) is smaller than the objective value of the best known feasible solution, so the optimal solution to Q_i^+(resp., Q_i^-) is no better than the best known feasible solution.

BB terminates when every leaf has been fathomed. It returns the best known feasible solution, which is optimal. See Algorithm 1 for the pseudocode. The most common node selection policy is the best bound policy. Given a BB tree, it selects the unfathomed leaf containing the MILP Q with the maximum LP relaxation objective value. Another common policy is the depth-first policy, which selects the next unfathomed leaf in the tree in depth-first order.

4.3 Example

$$\begin{aligned} \max \quad & 40x_1 + 60x_2 + 10x_3 + 10x_4 + 3x_5 + 20x_6 + 60x_7 \\ \text{sub to} \quad & 40x_1 + 50x_2 + 30x_3 + 10x_4 + 10x_5 + 40x_6 + 30x_7 \leq 100 \\ & x_1, \cdots, x_7 \in \{0,1\} \end{aligned} \tag{1}$$

Each rectangle denotes a node in the BB tree. Given a node Q, the top portion of its rectangle displays the optimal solution $\tilde{x}_Q$ to the LP relaxation of Q, which is the MILP (1) with the additional constraints labeling the edges from the root to Q. The bottom portion of the rectangle corresponding to Q displays the objective value $\tilde{c}_Q$ of the optimal solution to this LP relaxation, i.e., $\tilde{c}_Q$ = (40, 60, 10, 10, 3, 20, 60)$\cdot$ $\tilde{x}_Q$. In this example, the node selection policy is the best bound policy and the variable selection policy selects the "most fractional" variable: the variable x_i such that $\tilde{x}_Q[i]$ is closest to 1/2, i.e., $i = argmax\{min\{1 - \tilde{x}_Q[i], \tilde{x}_Q[i]\}\}$. In Figure 1, the algorithm first explores the root. At this point, it has the option of exploring either the left or the right child. Since the optimal objective value of the right child (136) is greater than the optimal objective value of the left child (135), BB will next explore the pink node (marked 1). Next, BB can either explore either of the pink node's children or the orange node (marked 2). Since the optimal objective value of the orange node (135) is greater than the optimal objective values of the pink node's children (120), BB will next explore the orange node. After that BB can explore either of the orange node's children or either of the pink node's children. The optimal objective value of the green node (marked 3) is higher than the optimal objective values of the orange node's right child (116) and the pink node's children (120), so BB will next explore the green node. At this point, it finds an integral solution, which satisfies all of the constraints of the original MILP (1). This integral solution has an objective value of 133. Since all of the other leafs have smaller objective values, the algorithm cannot find a better solution by exploring those leafs. Therefore, the algorithm fathoms all of the leafs and terminates.

4.4 Variable selection in MILP tree search

Variable selection policies typically depend on a real-valued score per variable x_i. (Score-based variable selection policy). Let score be a deterministic function that takes as input a partial search tree T , a leaf Q of that tree, and an index i and returns a real value ($score(T, Q, i) \in \mathbb{R}$). For a leaf Q of a tree T , let $N_{T,Q}$ be the set of variables that have not yet been branched on along the path from the root of T to Q. A score-based variable selection policy selects the variable $argmax_{x_j \in N_{T,Q}} score(T, Q, j)$ to branch on at the node Q.Most fractional. In this case, ($score(T, Q, i) = \{min\{1 - \tilde{x}_Q[i], \tilde{x}_Q[i]\}\}\}$. The variable that maximizes ($score(T, Q, i)$ is the "most fractional" variable, since it is the variable such that $\tilde{x}_Q[i]$ is closest to 1/2.

5 Clustering problem

A solution to the cluster problem is usually to determine a partitioning that satisfies some optimality criterion. This optimality criterion may be given in terms of a function f which reflects the levels of desirability of the various partitions or groupings. This target is the objective function. Assuming there are n accounts (objects) and m features for each account we seek to partition these n accounts in m dimensional spaces into meaningful clusters, K in number. The clustering is achieved by minimizing intracluster similarity and maximizing intercluster dissimilarity. Mathematically, the problem can be formulated as an optimization problem as follows: For a given or to be suitably chosen $K \in N$

$$\begin{aligned} &\min f(C) \\ &\text{subject to } C = (C_1, \cdots, C_k), \qquad C_1 \cup \dot{} \cdots \cup \dot{C}_k = \Pi \end{aligned}$$

whereby $\Pi = \{x_1, \cdots, x_n\}$ is the set of objects to be grouped in K disjoint clusters C_k. Finally, f is a nonnegative objective function. Its minimization aims at optimizing the quality of clustering.

5.1 Formulation Clustering into MILP

We can formulate k-means clustering as a MILP by assigning a binary variable x_i to each point p_i where x_i = 1 if and only if p_i is a center, as well as a binary variable y_{ij} for each pair of points p_i and p_j , where y_{ij} = 1 if and only if p_j is a center and p_j is the closest center to p_i. We want to solve the following problem:

$$\begin{array}{lll} \min & \sum_{i,j\in[n]} d(p_i,p_j)\, y_{ij} & \\ \text{s.t.} & \sum_{i=1}^{n} x_i = k & \\ & \sum_{j=1}^{n} y_{ij} = 1 & \forall i \in [n] \\ & y_{ij} \le x_j & \forall i,j \in [n] \\ & x_i \in \{0,1\} & \forall i \in [n] \\ & y_{ij} \in \{0,1\} & \forall i,j \in [n]. \end{array}$$

Figure 5: MILP formulation of Clustering (k-means) algorithm

5.2 Learning Algorithm

For the case where we wish to learn the optimal tradeoff between two scoring rules, we provide an algorithm that finds the empirically optimal parameter $\cap\mu$ given a sample of m problem instances. Our algorithm stems from the observation that for any tree-constant cost function cost and any problem instance Q, $cost(Q, score1 + (1-\mu)score2)$ is simple: it is a piecewise-constant function of μ with a finite number of pieces. Given a sample of problem instances $S = Q_1, ..., Q_m$, our ERM algorithm constructs all m piecewise-constant functions, takes their average, and finds the minimizer of that function. In practice, we find that the number of pieces making up this piecewise-constant function is small, so our algorithm can learn over a training set of many problem instances.

5.3 Implementation

I have constructed a combination of c++ and python code to understand the workings of above mentioned MILP tree search algorithm using Google OrTools for modelling the Linear Programming part and I have used IRIS data set without the labeling to train, with k = 3, to train the data. When fed to the normal MILP and the result of it was inconclusive and I am cuttenly working on writing the full machine learning code part of the problem.

6 Conclusion and Future Expectations

During the course of this internship, I learnt more about the mathematical basics of Machine Learning from PAC Learning and Psedo dimention combined with to implementation of Linear Programming using ORTools in C++ and Python. These methods can also be used when considering Online Algorithms and these techniques can molded and modeled to fit any application that could be expressed in terms of a Linear Program.

References

[1] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik, “Learning to Branch”, *arXiv:1803.10150v2 [cs.AI]* , 16 May 2018.

[2] Rishi Gupta and Tim Roughgarden, “Data-Driven Algorithm Design”, *COMMUNICATIONS OF THE ACM*, VOL. 63, June 2020.

[3] Maria-Florina Balcan, “Data-driven Algorithm Design”, *arXiv:2011.07177v1 [cs.DS]*, 14 Nov 2020.