Learning to Vote

Chunheng Jiang

October 14, 2018

Our ultimate goal is to design or employing existing machine learning methods to train voting rules with certain axiomatic properties. The learnability for voting rules satisfying some desirable fairness axioms is very useful. Once a new voting rule is proposed, it may have some special fairness properties. It's pretty hard to handcraft another voting rule with the same axiom. However, we could learn a similar voting rule from instances, or even a simpler rule with the exactly same property. It's promising to learn all those existing voting rules efficiently and have their good genes to design a single meta-voting rule, that satisfies more fairness criteria, or at least with a higher satisfiability to all fairness criteria than any existing voting rule.

The very first step to learn a voting rule or mechanism from instances is to construct a training set. The input profiles and the corresponding winner(s) are provided by an expert or oracle voting rule, which we are attempting to learn. We query experts or oracles with preference profiles, they reveal us their favorite candidate(s). We extract important feature from these query-answer pairs and create the training set. Here we make a brief note on the features we extracted from preference profile (a comprehensive survey on existing voting rules again is required).

To make the training schema scalable for varied number of voters and candidates, we extract the following signals from voting preference profiles: configuration parameters (the size of the profile or the number of voters n and the number of candidates m), the positional features of candidates (number of voters who have a candidate on the 1st, 2nd or 3rd place, all candidates' positional features are includes), pairwise comparisons features (winning or losing points in all head-to-head competitions, number of games which a candidate beats others or be beaten), non-linear extreme features (whether a Condorcet winner, whether a best candidates for beating (being beaten by) the largest (smallest) number of candidates, whether a worst candidate for beating (being beaten by) the smallest (largest) number of candidates) and so on. Xia and Conitzer [36] proposed the concept of general scoring rule, we expect the training algorithm is valid for the broader general scoring rule rather than the specific positional scoring rules or Condorcet rules.

Besides training set on with voting predictions, we have to construct a training set reflecting the fairness axioms that we are interested in. We adopted the idea proposed by Xia [35] to augment the voting rule training set with some specific fairness criteria. Take the Condorcet consistency for instance. To select the Condorcet winner, firstly the learned voting rule should be able to detect the sufficient condition for Condorcet winner's existence. Based on the conditions, the evidence could be very important feature for the learning algorithm to collect. Besides, we could borrow the idea in learning to rank, try to learn a meta-voting rule based on the existing voting rules. Most direct approach is to construct the training set based on popular existing voting rules, especially those scoring rules. Their scoring schema could be vital features for learning algorithm to learn. Actually, the finer the features we extract, the better the prediction performance of the learned voting rule.

Also, we will prove that whether a class of voting rules is learnability and show sound reason to use as fewest instances to train in good performance as possible (*VC-dimension and out-of-sample error*). To train the voting rule, we can incorporate the point-wise, the pairwise and the triple-wise axiomatic constraints into the error function by borrowing the idea from the famous RankNet in learning to rank.

To make the learned model easy to explain, we tend to learn based on linear functions or decision trees, especially the decision tree with linear combinations of features as its internal decision nodes. For example, Procaccia et al [23,24] proposed a voting tree to learn voting rules. There are some other works that relies on neural network and tries to learn Condorcet and Borda rules [3].

1 Notation

- Set of alternatives, candidates, or agents $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$
- Set of voters $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$
- Set of preference rankings $\Pi(\mathcal{A}) = \{\pi_1, \pi_2, \ldots, \pi_{m!}\}$
- Preference profile

$$\Pi(\mathcal{A}, \mathcal{V}) = \left\{ K_1 * \sigma_1 + \ldots + K_{m!} * \sigma_n \middle| \sum_{i=1}^{m!} K_i = n, K_i \geq 0, \sigma_i \in \Pi(\mathcal{A}), 1 \leq i \leq m! \right\}$$

2 Learning Voting Rules

Learning voting rules with some axiomatic properties are formulated as the following objective function

$$\min_{r \in \mathcal{H}} e_{r,\text{train}} = e_{r,\text{oracle}} + \sum_k \lambda_k e_{r,\text{criteria}_k}. \tag{1}$$

with a regularization term $\sum_k \lambda_k e_{r,\text{criteria}_k}$. Here, $\lambda_k \geq 0$ is a penalty coefficient. The larger the value, the higher the penalty the learned rule will receive for violating one axiomatic constraint. Generally speaking, no voting rule could satisfy all fair criteria human expect. To create a voting rule satisfying as many axioms as possible, machine learning is introduced. The algorithm learns from labelled examples and approximately fit to the exact voting principle in the mind of some an expert, sometimes called oracle.

The training set have two parts: the query results. The results is collected according to a series of queries which are submit to the expert in black-box. Another part presents in the form of summing term. Each voting axiomatic criterion could be introduced to impose the constraints over the structure of the learned voting rule. If the later part in the error function is eliminated ($\lambda_k = 0, \forall k$, then the learning algorithm just learns how exactly the black-box works internally. Certainly, we also could use different values for different criteria. The rule in the mind of the expert could be very simple. If it keeps in private,

people are hard to guess. To learn the rule, we could just use all the collected data to approximate its mathematical structure. Certainly, we expert that the rule could be much perfect, then it should satisfy many fairness properties which human value high. To impose limitations over the learned model, the most straightforward approach is to design an objective function, which takes the axiomatic properties into consideration based on the assumption that these properties could be expressed in terms of mathematical expression. However, it's extremely hard to make that happen. Since it's relatively easy to evaluate whether a voting mechanism satisfies certain axiom. We therefore refer to discrete examples by introduce extra data to augment the data provided by the expert. We expect that the learned mechanism could have minimum prediction error on the data from different sources. Especially, it could keep balance between its approximation to the ground-truth expert and have higher satisifiabilities over all axioms. For pointwise, such as Condorcet winner criterion, it should have high probability to detect the Condorcet winner and choose it as the winner if he/she is available. For the pointwise axioms, such as neutrality. It should always tell consistent result even permutations take place on all voters' preferences. We notice that to evaluate the satisfiability of pairwise axioms. We should evaluate the rule over all pairs' of corresponding permutation profiles.

To make the mechanism learned from instances make correct predictions, it must be a fitness to the selected voting rule to certain extend, at the same time, it should correctly predict the selection of Condorcet winners from the Condorcet profiles. The second part plays the same role in LASSO's regularization term, imposed constraints on the voting mechanism we are trying to recover. When the voting rule be able to make correct prediction on all profiles, and could also correctly select the Condorcet winner. The more accurate the voting rule could make predictions on the second subsets, the higher the voting rule's satisifiability to the CWC.

Suppose that the learned voting rule have a overall error on all preference profiles $\mathbf{e}$, we should also compute its error over the selected voting rules $\mathbf{e_1}$ and its prediction error on those profiles $\mathbf{e}_2$. Ideally speaking, if we could perfectly train a voting rule, it should

make highly consistent predictions of profiles with the selected voting rule, and the prediction error over profiles which represent certain voting fairnesses should also be very low. The lower the second error, the better the learned voting rule satisfies the corresponding axiom. There may be a error bound, in mathematical analysis. To get a tight error bound, we refer to the experimental analysis and voting rule-learning approach to verify it.

To demonstrate the generalization ability of the learned voting rule, the most intuitive way is make a scatter plot on the prediction errors (e, e_1 and e_2) and the numbers of voters.

To learn a voting mechanism with the Condorcet consistency and the neutrality criterion, we have to collect the right instances from the queries submitted to the expert and the Condorcet profiles. Both are the basic requirements for a Condorcet consistent expert rule to satisfy. Therefore, we sample the profiles from them and imposed permutations on them to augment the training set.

Generating samples according to satisfaction of Condorcet criterion and neutrality, then use multiple standard learning algorithms (e.g. random forest) to learn a voting rule. Then, evaluate the learned rule by three criteria: error rate with Borda, satisfaction of Condorcet, and satisfaction of neutrality. For each criterion, also include the train errors.

Learning a multivariate decision tree to approximate the positional scoring rules, then general scoring rule and keeping trade-offs for its satisfiabilities of different axiomatic properties.

All positional scoring rules do not satisfy Condorcet consistency and consistency at the same time. Therefore, it's very important to train a voting rule which seeks a balance point between the satisfaction with the two axiomatic properties in dilemma. Besides, it's not easy to evaluate the satisfiability of consistency criterion, we have to refer to sampling techniques, such as rejection-accept sampling approach. Keep the problem in mind, and figure out ways to deal with it.

2.1 Fairness Criteria

Fairness, also known as axiomatic property, sometimes axiom in brief, is used for evaluate and compare voting rules. It's nearly certain that no single voting rule could satisfies all existing axiomatic properties. These desirable axioms represent people's wishes. Wish that the voting rule could satisfies many of them, and be a fair rule. To understand, we make an incomplete summary of axiomatic properties. They are anonymity, neutrality, consistency/separability, finite local consistency, Condorcet consistency, Condorcet Loser, Pareto criterion (unanimity), non-dictatorship (stronger: anonymity), independence of irrelevant alternatives (IIA), homogeneity, resolvability, monotonicity (see a perfect example), prudence, continuity, Smith criterion, reversal symmetry, public-enemy criterion (If a candidate is ranked last by a majority of voters, then s/he should not win the election.)

We note that a comprehensive survey on the relations of these voting criteria are required. For example, the Condorcet consistency implies a majority criterion.

2.2 Fixed-majority Criterion

Fixed-majority criterion is a multi-winner analogue of the simple majority criterion introduced by Debord. It requires that if there is a simple majority of the voters, each of whom ranks the same k candidates in their top k positions (perhaps in a different order), then these k candidates should form a unique winning committee.

Given k candidates $\mathcal{A}_k$ in $\mathcal{A}$. If these k candidates as whole receives majority of top-k votes, it's guaranteed that the winner(s) must be in $\mathcal{A}_k$. Suppose that we can efficiently search the **smallest** $\mathcal{A}_k$ such that they are ranked by majority of the voters as their top-k choices. It will help us the reduce the number of nodes to explore in STV DFS approaches, and speed up the computation.

1. Multi-winner Analogues of Plurality Rule: Axiomatic and Algorithmic

Table 1: Candidates 1,2 and 3 receive majority votes

1	1	1	1	1
1	2	3	4	4
2	3	1	1	3
3	1	2	2	1

2. Efficient Voting via The Top-k Elicitation Scheme

As shown in Table 1, candidates 1, 2 and 3 receive the majority votes because they are ranked by majority voter as their top-3 choices. The winners must come from $\{1,2,3\}$.

2.3 Structure of Profiles with a Condorcet Winner

Suppose that we have m candidates and n voters in a voting, then there will be total number of $(m!)^n$ possible voting preferences presented by those voters, where $m!$ is the total number of all feasible strict linear ordering which could be imposed on the candidates. Here we are trying to illustrate how to anatomy the voting preference profile space, which as a result, could greatly help to compute the exact number of profiles with a Condorcet winner. To be convenient, we denote the preference profile space as $\Pi[m,n]$. Each voter has his or her own preferential ranking of these candidates, and has its counterpart in the permutation space. Hereafter, we denote a voter's preference ranking as $\pi \in \Pi[m,n]$.

A good representation approach could assist us to analysis many axiomatic properties of voting rules. We represent each preference relation on the candidates as a directed graph or its adjacent matrix. Then, the preference profile space is a spanned by all $m!$ permutations of those candidates, no matter how many voters there are. Each profile in the space could be represented as a linear combination of those basis permutations.

For each permutation, we extract all possible head-to-head contests. If one alternative win over the other one, he gain a point from the alternative; otherwise no point to gain. For all candidates and a permutation, we could construct a matrix to record the result.

For example, there are three candidates $\{A, B, C\}$ and a preference ranking of them is $A > B > C$. Its representation matrix could be written as

$$P_{A>B>C} = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

We have all pairwise comparison matrices of those permutations

$$\{P_{A>B>C}, P_{A>C>B}, P_{B>A>C}, P_{B>C>A}, P_{C>A>B}, P_{C>B>A}\}.$$

Also, we notice that all of them are singular matrix, and have a same rank $m-1$. They are used as the basis matrices to represent the preference profile space in terms of linear combinations.

Given a (m, n) voting, for any preference ranking $\pi \in \Pi[m, n]$, we have a non-negative integer vector $K \in \mathbb{N}^{m!}$ and the following representation form

$$P_\pi = \sum_{i=1}^{m!} k_i P_i,$$

with a constraint imposed on the coefficients, that is $\sum_i k_i = n$, where k_i means that the number of voters who share a equivalent preference ranking P_1 in the permutation space, and their sum should be equal to the total number of voters or votes. Besides that, if the preference rankings are sorted in order, the last one should be the transpose matrix of the first one, i.e. $P_{m!} = P_1^T$. Actually, there is a permutation matrix P_{12}, such that $P_2 = P_{12}^T P_1 P_{12}$. The relationship is straightforward, we can swap the position of two candidates in P_1 and transform to P_2. Similarly, the principle can be applied to any two basis matrix P_i and P_j. There must exist one permutation matrix P_{ij} such that $P_j = P_{ij}^T P_i P_{ij}$. According to the relation shown by P_1 and $P_{m!}$, we have $Q = P_{12}P_{23}\ldots P_{m!-1,m!}$ and $Q^T P_1 Q = P_{m!} = P_1^T$.

What we are interested in is how to decompose the votes of those profiles who have a Condorcet winner. Although many of them may do not have a Condorcet winner, but the population of profiles with a Condorcet winner is also large. To compute the exact

number of profiles with a Condorcet winner, and figure out their votes distribution, we formulate a constraint integer program

$$P = \sum_{1 \le i \le m!} k_i P_i, \tag{2}$$

$$s.t. \quad \sum_{i=1}^{m!} k_i = n, \tag{3}$$

$$k_i \in \mathbb{N}, 1 \le i \le m!. \tag{4}$$

To guarantee a Condorcet winner, which must beat all opponents in head-to-head competitions. It implies that all elements of one row of $\Delta = P - P^T$ should be positive except the one at the diagonal position. There must exist the only $1 \le i \le m!$, such that

$$\delta_i = e_i^T (P - P^T) = e_i^T \sum_{1 \le i \le m!} k_i (P_i - P_i^T), \tag{5}$$

where $\delta_{ii} = 0$ and $\delta_{ij} = 0, \forall 1 \le j \ne i \le m!$, $e_i = (0, \ldots, 0, 1, 0, \ldots, 0)^T$. If a feasible matrix P satisfies the above constraints, we could safely concluded that the i-th candidate in the profile must be the Condorcet winner. The size of the feasible region is closely related to the number of voters, the larger the number of voters, the harder to search all feasible solutions. However, we could take advantage some symmetrical relationships of parameters on the constraints to reduce the searching time.

If we could find all feasible solutions, we could construct all possible profiles with a Condorcet winner. It will be extremely important for evaluate those profiles generated from large-size voters.

2.3.1 Sufficient Conditions of the Existence of a Condorcet Winner

To learn a general voting rule that satisfies a specific point-wise axiomatic property (e.g. Condorcet Winner Criterion, CWC) , the learning algorithm must be able to explore the structure of the preference profiles with a Condorcet winner. Also, we research other domain restrictions, like single-peaked preference, to better understand the sufficient conditions for the existence of a Condorcet winner in a voting preference profile.

2.3.2 Similarity of the Condorcet Winner Consistent Voting Rules

We learn voting rules from instances based on the ground truth voting rules, and cluster them with the help of their fitness models or approximation agents. Based their the clustering properties, we could figure out the important reasons which make them Condorcet consistent.

2.4 Borda rule and monotonicity

Suppose c is a Borda winner of the preference profile $P = \{R_1, R_2, \ldots, R_n\}$ over m candidates C by n voters V, where R_i is the preference ranking over C by the voter $v_i \in V$, $i = \{1, 2, \ldots, n\}$. Therefore, c's Borda score is not less than the Borda score each of other candidates has received, namely

$$f(c, P) = \max_{k=1}^{m} f(c_k, P).$$

If one of the voter, w.l.o.g, saying v_1, changes his/her mind, and raises candidate c's ranking in his/her preference ranking list from R_i to R_i' and $P' = \{R_1', R_2, \ldots, R_n\}$. The up-ranking could increase c's Borda score at least one point, while others' do not increase at all, even one candidate's points decreases by one, $f(c, P') > f(c, P)$, which won't harm c's place as a Borda winner. As a result, Borda rule satisfies the monotonicity criterion.

2.5 Neutralize learned rule

Suppose we have learned a voting rule f from instances, and the learned rule may fail the neutrality criterion. To make it as neutral as possible, we would like to perform some explicit permutations on the learned rule to create an ensemble $r = \{f_{M_1}, f_{M_2}, \ldots, f_{M_{m!}}\}$ voting rule, based on all possible permutations over m candidates. Each member f_{M_i} of r is a permuted version of the learned rule f with the permutation $M_i \in \mathcal{M}$, $1 \leq i \leq m!$, where $\mathcal{M} = \{M_1, M_2, \ldots, M_{m!}\}$ is the permutation set over m candidates.

How to construct the rule member f_{M_i} is the core of this section. Now, we define a general form of ensemble rule r from f with the help of the permutations $\mathcal{M}$ as following

$$r = \{f_{M_1}, f_{M_2}, \ldots, f_{M_{m!}}\}, \quad f_{M_i}(P) = g_i(M) f(h_i(M)(P)), \quad 1 \leq i \leq m!, \ \forall P \in \Pi[m,n]. \tag{6}$$

The general voting rule r makes predictions based on the majority rule

$$r(P) = \underset{1 \leq k \leq m}{\arg\max} \sum_{1 \leq i \leq m!} I(f_{M_i}(P) = k). \tag{7}$$

If the general voting rule satisfies the neutrality criterion, we have the following result

$$r(M(P)) = M(r(P)), \quad \forall P \in \Pi[m,n], M \in \mathcal{M}. \tag{8}$$

Suppose we perform permutations on the learned voting rule f and the corresponding predictions, therefore $g_i(M) = M_i$, $h_i(M) = M_i^{-1}$, and $f_{M_i}(P) = M_i f(M_i^{-1}(P))$, where $1 \leq i \leq m!$. It's guaranteed that the formulation makes a neutral voting rule.

Proof. Without loss of generality, we denote $M_1 = I$ the identity permutation, and $f_{M_1} = f$. Suppose that $f_{M_i}(P) = M_i f(M_i^{-1}(P))$, then we have $r(P) = \bigcup_{1 \leq i \leq m!} M_i f(M_i^{-1}(P))$. According to the definition of neutrality, we feed the general voting rule a permuted profile $M(P)$, $\forall M \in \mathcal{M}$. The prediction of the permuted profile is

$$r(M_k(P)) = \bigcup_{1 \leq i \leq m!} M_i f(M_i^{-1}(M_k(P))).$$

We know that when $i = k, M_i f(M_i^{-1}(M_k(P))) = M_k f(P) = M_k f_{M_1}(P)$, and therefore $M_k f_{M_1}(P) \in M_k r(P)$. Since $M_k f_{M_1}(P) \in M_k r(P)$, it implies that $M_k r(P) \bigcap r(M_k(P)) \neq \varnothing$ for each $k \in [1, m!]$, and r created with the first structure satisfies the neutrality criterion. □

Learn a general scoring rule that integrates both positional scoring rules (plurality and Borda count), and Condorcet methods (e.g. Copeland, RankedPairs). We will connect learning to rank with social choice rules, and propose some new idea to learn a general

rule consistent with as many voting rules as possible, and to make it fair satisfying as many axioms as possible.

To learn the social choice mechanism, we can transform it to a classification problem. Each candidates (alternative or agent) is a class, and each profile is an input example. The winner of the profile based on certain voting rule is the target. There are untold number of classification algorithms could be directly applied to solve the problem.

We would like to investigate decision tree and neural network. The decision tree has a big advantage for its ease of explanation, and could be able to learn the Condorcet method, which includes many pairwise comparisons. The neural network, although hard to interpret the complex structure of model, has better performance in learning a positional voting rule. There are many ways to integrate these two characteristics of neural network and decision tree.

We have explored the newly developed deep neural forests, however the experiments on data do not very promising. The current implement focuses on the decision tree side. A further development could be extended to the neural network learning.

Besides, we should also reuse the existing decision tree algorithms, especially those having linear combination of input features in node splitting.

3 Preference Profile Sampling Model

To generate random preference profile, researchers proposed several probability models on voting[7], among which the most widely known ones are the Impartial Culture (IC), Anonymous Impartial Culture (AIC), and the Real Society (RS) assumptions.

- IC: Voters are completely independent, every profile from $\Pi(\mathcal{A})^n$ is equiprobable, each voter selects her preferences according to an uniform probability distribution on the set of linear orderings. Their decisions are completely random and indepen-

dent. The assumption of IC is a worse case analysis, and produce cyclic preference profiles with a high probability.

- AIC: Voters are slightly independent [22].
- RS: Voters are highly dependent, all of them are belonging to certain parties – Metropolis Algorithm.

Based on the anonymous impartial culture assumption, EĞECIOĞLU and Giritligil [7] proposed to generate the anonymous equivalent classes that the voters are anonymous. Let's see an example with $m = 2$ and $n = 3$. There are $(m!)^n = 8$ possible preference profiles, as indicated in Table 2.

Table 1: Preference Profiles of $n = 3$ Voters over $m = 2$ Candidates $\{a, b\}$ and the AECs

	a a a b b b	
b a a a b b	a b a b a b	a a b b b a
b b a a a b	a b b b a a	b a b a b a
	b b b a a a	

According to Table 2, there are four rows and each of them represents one anonymous equivalent class. For example, the first AEC includes all preference profiles that all voters sharing the same preference of $a \succ b$, i.e. $3 * a \succ b$. The last AEC represents all voters sharing the same preference that $b \succ a$, i.e. $3 * b \succ a$. The second and third one is $2 * a \succ b + b \succ a$ and $a \succ b + 2 * b \succ a$, respectively. Given m candidates and n voters, there are $\binom{n+m!-1}{m!-1}$ AECs in total and $\binom{3+2!-1}{2!-1} = 4$.

3.1 Voting Features

Considering each profile as a query, and for each alternative we extract the individual ranking information, including the number of voters who cast it as first choice, second choice, and so on. Another vital component could be explore is the pairwise competition profile. Given the relative place in a ballot, we could cumulate relevant data of each pair of alternatives, especially the pair made from a winner and its opponents.

We create an integrated (pointwise and pairwise) LETOR method with the pointwise voting features of each individual alteranative and the head-to-head comparisons. From the pointwise features, we learn a linear function to produce pointwise scores. From the pairwise features, we learn another linear model. Therefore, both kinds of features combined to model the preference relations among alternatives.

Given $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ and a profile $P \in \Pi(\mathcal{A})^n$. We extract from P the following pointwise features:

$$\begin{array}{cccccc} a_1 \rightarrow & x_{11} & x_{12} & \cdots & x_{1m} & \leftarrow x_1, \\ a_2 \rightarrow & x_{21} & x_{22} & \cdots & x_{2m} & \leftarrow x_2, \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_m \rightarrow & x_{m1} & x_{m2} & \cdots & x_{mm} & \leftarrow x_m, \end{array}$$

where x_{ij} is the number of voters ranking a_i as his or her jth choice, $i, j = 1, 2, \ldots, m$.

Based on the relative places in a preference ranking, we have the vote distribution over the positional differences between $a_i, a_j \in \mathcal{A}$. Let $z_{ij}(k;\pi)$ be the number of voters who place a_i and a_j on his or her preference ranking with a positional difference k, which has $2(m-1)$ possible values $k \in \{\pm 1, \pm 2, \ldots, \pm(m-1)\}$. Accumulating these features over all possible rankings in P, we have the pairwise positional features $Z = (z_{ij})_{m \times m \times 2(m-1)}$

$$z_{ij} = \Big[z_{ij}(1-m), \ldots, z_{ij}(-2), z_{ij}(-1), z_{ij}(1), z_{ij}(2), \ldots, z_{ij}(m-1)\Big]^T \in \mathbb{R}^{2(m-1)},$$

and

$$z_{ij}(k) = \sum_{\pi \in P} z_{ij}(k;\pi) = \sum_{\pi \in P} [[\pi(a_i) - \pi(a_j) = k]], \quad i, j = 1, 2, \ldots, m.$$

Suppose an oracle voting rule r chooses $a_i \in \mathcal{A}$ as the winner, i.e. $r(P) = a_i$. The pairwise features between a_i and $a_j \in \mathcal{A}$ can be utilized to construct the preference model

$$\begin{array}{cccccccc} (a_i, a_1) \rightarrow & z_{i1}(1-m) & \cdots & z_{i1}(-1) & z_{i1}(1) & \cdots & z_{i1}(m-1) & \leftarrow z_{i1}, \\ (a_i, a_2) \rightarrow & z_{i2}(1-m) & \cdots & z_{i2}(-1) & z_{i2}(1) & \cdots & z_{i2}(m-1) & \leftarrow z_{i2}, \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ (a_i, a_m) \rightarrow & z_{im}(1-m) & \cdots & z_{im}(-1) & z_{i1}(1) & \cdots & z_{im}(m-1) & \leftarrow z_{im}. \end{array}$$

According to the above definitions of the two kinds of features, they present some obvious inter-dependent relations

$$\sum_{j=1}^{m} x_{ij} = \sum_{j=1}^{m} x_{ji} = \sum_{k \in \{\pm 1, \ldots, \pm(m-1)\}} z_{ij}(k) = n, \forall i, j = 1, \ldots, m.$$

Given profile P and a true voting rule r, with all pairs of alternatives $r(P)$ and $a_j \in P$, we propose the following integrated LETOR model

$$\begin{aligned} \min_{w,\theta} \quad & L\Big[f(\mathcal{A} \times \mathcal{A}; w, \theta), r\Big] \\ s.t. \quad & Pr\Big(r(P) \succ_f a_j\Big) = \Big[1 + e^{-s_j(P)}\Big]^{-1} \\ & s_j(P) = f(r(P), a_j; w, \theta), \forall 1 \leq j \leq m \end{aligned}$$

to learn r the true voting rule with all the pointwise positional features of the individual alternatives and all associated pairwise features. Here $L(f, r)$ is an in-sample loss function of the learned pairwise voting rule

$$f(a_i = r(P), a_j; w, \theta) = g(x_i; w) - g(x_j; w) + h(z_{ij}; \theta), w \in \mathbb{R}^m, \theta \in \mathbb{R}^{2(m-1)}$$

to the true voting rule r. It could be defined as a cross entropy loss function over all input profiles $P \in \mathbb{P}$, i.e.

$$L\Big[f(\mathcal{A} \times \mathcal{A}; w, \theta), r\Big] = -\frac{1}{|\mathbb{P}|} \sum_{P \in \mathbb{P}} \sum_{j=1}^{m} Pr\Big(r(P) \succ_r a_j\Big) \log Pr\Big(r(P) \succ_f a_j\Big).$$

Therefore, we have the final voting learning method formulated as following

$$\begin{aligned} \min_{w,\theta} \quad & -\tfrac{1}{|\mathbb{P}|} \sum_{P \in \mathbb{P}} \sum_{j=1}^{m} Pr\Big(r(P) \succ_r a_j\Big) \log Pr\Big(r(P) \succ_f a_j\Big) \\ s.t. \quad & Pr\Big(r(P) \succ_f a_j\Big) = \Big[1 + e^{-s_j(P)}\Big]^{-1} \\ & s_j(P) = f(r(P), a_j; w, \theta), \forall 1 \leq j \leq m. \end{aligned}$$

There are many methods to formulate the pairwise preference probability $Pr\left(a_i \succ a_j\right)$ from individual scores or pairwise scores, e.g. Mallows' model, Plackett-Luce model, Bradley-Terry model and logistic transformation used in RankNet[2].

3.2 General Scoring Rule

Xia and Conitzer [36] proposed the generalized scoring rules, which united Condorcet method and other positional scoring rules. We extract from each profile all the absolute and the relative positional information, and from which the source preference profile can be immediately reconstructed.

Let $x_{a,b}^{ij}$ be the number of voters who place $a \in \mathcal{A}$ in position i and $b \in \mathcal{A}$ in position j. When tie is not allowed, $x_{a,b}^{ij} = 0$ for $i = j$ or $a = b$. The features are called *generic features*, from which all the related positional features and pairwise features could be constructed.

- Positional Features: the number of voters who rank a as his or her i-th choice

$$x_a^i = \sum_{b \in \mathcal{A}} \sum_{j=1}^{m} x_{a,b}^{ij}, \forall 1 \leq i \leq m.$$

- Pairwise Features: the number of voters who prefer a to b

$$x_{a \succ b} = \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} x_{a,b}^{ij}, \forall a, b \in \mathcal{A}.$$

We aim to learning a general scoring rule $f(x; w)$ from the extracted profile preference features, where w is the parameter vector of the scoring function. There are many methods to learn a linear model. For example, neural network, SVM, decision tree et al. The weight is the atomic components. However, these extracted features require further refinement, because it may not be able to reveal some crucial information, e.g. the strength of the relative preference. A vote $a \succ c \succ d \succ b$ presents a stronger relative preference of $a \succ b$ than $a \succ b \succ c \succ d$, and the later conveys much stronger preference of $a \succ b$ than that of $c \succ d \succ a \succ b$. Both positions of a and b in comparison are important features that deserve additional attention.

3.2.1 Decision Tree Voting

Considering $\mathcal{A} = \{a,b,c\}$ and its $3! = 6$ possible linear orderings

$$\Pi(\mathcal{A}) = \{a \succ b \succ c, a \succ c \succ b, b \succ a \succ c, b \succ c \succ a, c \succ a \succ b, c \succ b \succ a\}.$$

Given a preference profile $P \in \Pi(\mathcal{A})^n$, we can have the following features: the number (or percentage) of each possible linear ordering; the number (or percentage) of pairwise comparisons, including $\{a \succ b, b \succ a, a \succ c, c \succ a, b \succ c, c \succ b\}$. We notice the following relations for the linear ordering and the pairwise comparisons

$$\begin{aligned}
\#(a \succ b) &= \#(a \succ b \succ c) + \#(a \succ c \succ b) + \#(c \succ a \succ b),\\
\#(a \succ c) &= \#(a \succ b \succ c) + \#(a \succ c \succ b) + \#(b \succ a \succ c),\\
\#(b \succ a) &= \#(b \succ a \succ c) + \#(b \succ c \succ a) + \#(c \succ b \succ a),\\
\#(b \succ c) &= \#(a \succ b \succ c) + \#(b \succ a \succ c) + \#(b \succ c \succ a),\\
\#(c \succ a) &= \#(b \succ c \succ a) + \#(c \succ a \succ b) + \#(c \succ b \succ a),\\
\#(c \succ b) &= \#(a \succ c \succ b) + \#(c \succ a \succ b) + \#(c \succ b \succ a).
\end{aligned}$$

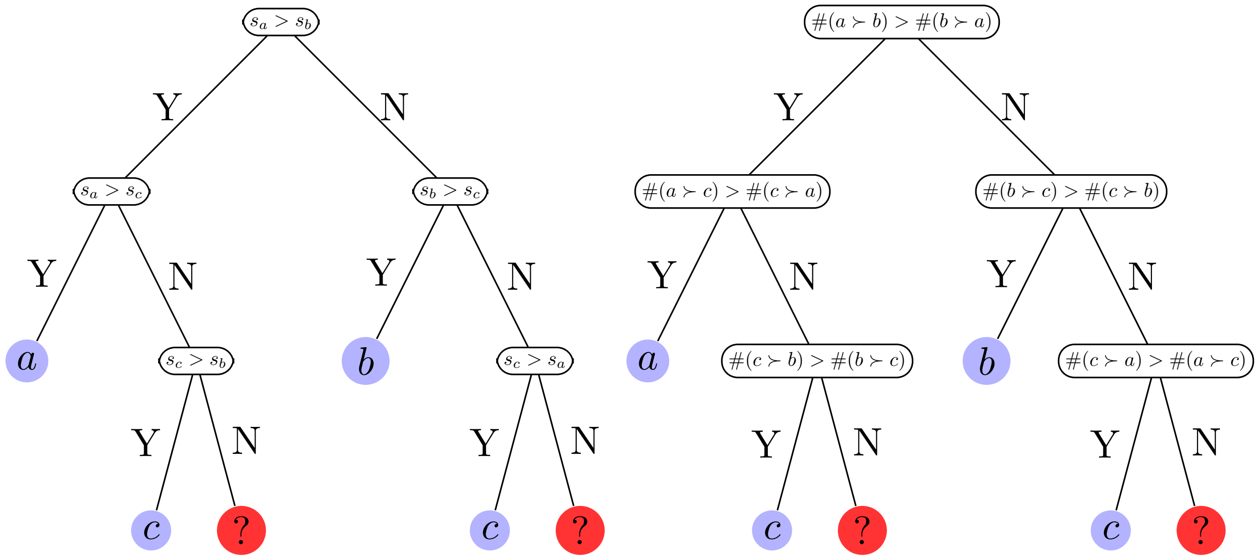

Fig. 1: Decision Tree Voting Procedure of Borda (Left) and Condorcet (Right) Method

The Borda rule is to compute each candidate's Borda scoring with a positional weight vector $w = (w_1, w_2, w_3)^T$. Therefore, we have the Borda scores

$$
\begin{aligned}
s_a &= w_1 * \Big[\#(a > b > c) + \#(a > c > b)\Big] + w_2 * \Big[\#(b > a > c) + \#(c > a > b)\Big] + w_3 * \Big[\#(b > c > a) + \#(c > b > a)\Big], \\
s_b &= w_1 * \Big[\#(b > a > c) + \#(b > c > a)\Big] + w_2 * \Big[\#(a > b > c) + \#(c > b > a)\Big] + w_3 * \Big[\#(a > c > b) + \#(c > a > b)\Big], \\
s_c &= w_1 * \Big[\#(c > a > b) + \#(c > b > a)\Big] + w_2 * \Big[\#(a > c > b) + \#(b > c > a)\Big] + w_3 * \Big[\#(a > b > c) + \#(b > a > c)\Big],
\end{aligned}
$$

and rank candidates based on their Borda scores. The one with largest Borda score wins. As indicated in Fig. 1, Borda rule and Condorcet method for 3 candidates can be formulated with a decision tree of at least depth 3.

3.2.2 Weight Refinement

To make the explanation for model explicit, we decompose the original weight w to two components: the weight α of preferred position, and the weight β of relative position margin, i.e. $w = \alpha * \beta$. Moreover, the symmetry property should also be brought into consideration. With the decomposition, the total number of parameters is reduced. There are m possible positions for $m = |\mathcal{A}|$, so

$$
\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_m)^T \in \mathbb{R}^m,
$$

where α_i indicates the weight of a preferred candidates locating at position i. Even the last element α_m can be eliminated or directly assigned a zero weight $\alpha_m = 0$ because the last place is unfavorable. The relative position margin can be fixed, the possible values are $\{\pm 1, \pm 2, \ldots, \pm(m-1)\}$, corresponding to a relative positional weights

$$
\beta = (\beta_{1-m}, \ldots, \beta_{-2}, \beta_{-1}, \beta_1, \beta_2, \ldots, \beta_{m-1})^T \in \mathbb{R}^{2(m-1)}.
$$

The relative importance of each element in α and β should be different, but there exists some well-defined constraints on them. In general, we expect that

$$
\alpha_1 \geq \alpha_2 \geq \ldots \geq \alpha_m, \quad \beta_{1-m} \leq \beta_{-2} \leq \beta_{-1} \leq \beta_1 \leq \beta_2 \ldots \leq \beta_{m-1}.
$$

4 Voting and Preference Data Set

- PrefLib.org house over 3,000 preference datasets used by the community.
- Olympic Election of Hosting Places: Mathematics in Sport
- Ariel Procaccia introduced an awesome example of election with four candidates. Given a fixed preference profile, different voting rules however produces distinctively different outcomes.

5 Data Set

We generate three profile data sets:

- SOC-1: All possible permutation rankings over m candidates, each permutation receives one vote, therefore each profile has $m!$ votes. Here, $3 \leq m \leq 10$.
- SOC-2: All possible combinations of permutation rankings over m candidates for given number of votes n. Besides, voters' names are not considered. Here, $m = \{3,4,5\}$ and $2 \leq n \leq 5$. There are more than 9 million distinct profiles for $m = 5, n = 4$, it's extremely large. To limit the size of our data set, we just generate the profiles of $n = \{2,3\}$ for $m = 5$.
- SOC-3: Randomly generated profiles with specific m and n. Here $m = \{10,20,\ldots,100\}$ and $n = \{10,20,\ldots,100\}$. For each (m,n), we sample $s = 1,000$ profiles.
- SOC-4: Randomly generated single-peaked preference profiles[34] with specific m and n, where $m = \{10,20,\ldots,100\}$ and $n = \{10,20,\ldots,100\}$, we sample $s = 1,000$ profiles for each pair (m,n). To generate the single-peaked votes, we are required to provide the x-axis. In our implementation, the x-axis preference ranking is $0 < 1 < \ldots < m-1$.

There are 8 profiles (each file contains a profile) in SOC-1, $421,685$ profiles in SOC-2 (see Table 5) and $10 \times 10 \times 1,000 = 10^5$ in SOC-3.

n	$m = 3$	$m = 4$	$m = 5$
2	21	300	7,260
3	56	2,600	295,240
4	126	17,550	9,078,630
5	252	98,280	225,150,024

Table 1: Anonymous Equivalent Classes

To make our heuristic method convincing, we concentrate on a subset of these profiles. These profiles are called hardcases. Here, we give the definition of the hardness of a profile when searching the winner using a voting rule (it is specific for STV, similarly we can define the hardness for Baldwin, Coombs, Nanson and RankPairs).

1. ***Extremely Easy Cases*** are profiles which do not contain any tie in terms of plurality score from the very beginning.
2. ***Easy Cases*** are profiles with a single winner, who can be selected by **ONLY** repeatedly applying the Heuristic algorithm (directly eliminate one or multiple candidates at one time without considering the ordering to eliminate them).
3. ***Hard Cases*** are profiles that can not be processed by the Heuristic at some point.

When $m > n$, some alternatives must have zero plurality score and must be eliminated through the heuristic.

6 Experimental Settings

There are setting that can be easily turn on/off in our experiments include **C**aching, **P**runing, **H**euristic, **S**ampling and Priority Function in Priority Queue. Except for Q which indicates the priority function used, all other parameters are now boolean or equivalent $\{0,1\}$.

The Caching is used in checking whether a state has been visited. There are various implementations for this goal. For example, keeping a list of each states, keeping a hash table for all visited states. Experimental results indicated that the later approach is more efficient.

The Pruning is a step to check the state if all remaining alternatives have been selected in previous steps. If it's true, the program absolutely does not need to continue exploring the current branches (*prematurely pruning a branch will certainly affect the number of times of an alternative being elected as a winner*).

The Sampling step is applied before the actual election begins. It comes from the brute force method with a specific priority ranking in eliminating alternatives when there are ties. The priority ranking (aka permutation over alternatives) are randomly generated. The number of priority ranking (or size of sampling) we generated can affect the number of winners produced from this procedure. It plays as a rough estimation even it can find all the winners sometimes. Sampling causes randomness for the later procedures, therefore it's turn off when we concern more about the number of nodes to extend. Therefore, we have another important indicator, the effective number of alternatives in the pruned profile. What really matters are these left.

Different from the standard DFS method to search winners, the priority function is used for Best First Search. How to define the priority of each state (the set of candidates remain at the current node) is the crucial factor affecting the search efficiency.

We expect the heuristic and the priority function are helpful in reducing the running

time, reducing the number of nodes to extend. It's obvious, the beneficial in running time saving for running the heuristic and updating the priorities is counteracted. To make the comparison fair, we also count out the times of computing the plurality scores (scoring time) and expect that both strategies can reduce the total number of times in scoring compared with the standard DFS method.

Another aspect is the preprocessing. It's trivial, but very important for fair comparison. It's used before the actual election happens, and eliminates those alternatives who have never appeared as any voter's top choice. Their plurality scores are zero and have no chance to be elected at all even based on standard DFS method.

7 Searching Efficiency

Based on the above description of our experimental settings, here we would like to introduce an important indicator to measure the performance of the voting rule with various strategies.

The proposed heuristic searching algorithm (detailed information could be found in next subsection) and the priority function are expected to reduce the number of times to scoring and the number of nodes produced. The current outcomes in our experiments are not promising. It's pretty hard to dramatically reduce both values. We introduce another metric the **Searching Efficiency**. It's a curve with the percentage of number of nodes produced as the x-axis and the percentage of number of winners found as the y-axis. The domain is a 1×1 box, i.e. $[0,1] \times [0,1]$. Without Sampling, at the very beginning stage of the election, no node and no winner corresponding $(0,0)$, and when finished, it corresponds $(1,1)$. It's obvious, the Sampling will change what it looks like. Extreme cases can be a flat line, or a curve with large y-intercept which is above the curve produced from standard DFS method.

We expect the proposed heuristic or priority function helps to lift the curve as much as possible, which can demonstrate that the proposed method can find all the winners much soon than other approaches. That's what we called searching efficiency.

8 Heuristic Search Algorithm

The DFS approach is utilized to search all possible STV winners based on various tie-breaking rule, and each search path from top-to-bottom of the tree corresponds to a different tie-breaking rule.

The heuristic search algorithm (HSA) provides two strategies (a) eliminate a set of candidates directly without considering the tie-breaking rule in use, and (b) early terminate search procedure with some heuristic conditions to speed up DFS.

A direct elimination of weak candidates from profile requires the following steps:

- sort candidates based on their plurality scores
- place candidates to different hierarchical tiers and with candidates more favored at a higher tier
- search from top to low tier and eliminate all those lower tiers if their total plurality score is strictly smaller than the one they immediately followed.

The early stop strategy includes considering whether the remaining candidates has been evaluated at a visited node and whether all of them have been elected as the winners. The later condition may be rare, once happens, any attempt to extend the tree must be a waste and an early stop can prevent it.

The effect of this heuristic is in doubt. The situation that the heuristic can eliminate multiple alternatives one time is really rare. Then, it actually does nearly the same work as the standard DFS method. Each time, at most one alternative can be eliminated from the profile.

8.1 Priority Functions

To make as few as possible exploration, the next state we expect to visit the one which has the highest chance to produce a winner who has never been chose before. Let's imagine some special cases:

1. All alternatives in the state are known winners. It's useless and the Pruning cuts it off.
2. The size of state is small, and none of them are known winners. It's absolutely the top priority to visit these states.
3. The plurality scores for one alternative who has never been chose before is largest. It certainly deserves our attention.

Even three special cases reveals many important features that we can investigate to find a better priority function. These features can be *depth*, *size of the state*, *number of alternatives who never win before*, *the plurality score of the alternative who never wins before*. Based on our observation, we can design some priority functions leading us to extend a better state.

Looking back on our recursion DFS, the alternative with highest chance of being elected before will be eliminated at top priority. It reduces the chance as much as possible, even prevents that known winner being reelected again. **The recursion DFS places higher a weight on *depth*, and then refers to additional feature (e.g. whether an alternative is a known winner, could (s)he win with a higher probability in the next round) to break ties, i.e. selecting from those who stays at the same depth the next to eliminate. The additional features should increase the chance of producing a new winner who has never been seen before.**

Let P be the profile, S be the current state, and W be the set of known winners. Furthermore, we denote U be the set of the remaining alternatives in S but donot present in W. Therefore, $U = S \setminus W$ or $U = S - W$. All alternatives in U are those who never win before. Thus, we have $U \cap W = \varnothing$. The priority function is written as F.

- $F(S,P) = m * \Big[m - \|S\|\Big] + \|U\|$, where $m - \|S\|$ shows the depth of the current state in the tree. It prefers to extend the nodes at deeper layers, and detect the states with more alternatives in U.
- $F(S,P) = \Big(\|U\| * \| \max_{u \in U} P(u)\Big) / \Big(\|W\| * \max_{w \in W} P(w)\Big)$
- $F(S,P) = \Big(\|U\| * \| \max_{u \in U} B(u)\Big) / \Big(\|W\| * \max_{w \in W} B(w)\Big)$
- $F(S,P) = \sum_{u \in U} P(u)/n$
- $F(S,P) = 2 \sum_{u \in U} B(u)/m(m-1)n$

We realize that if the state's priority changes frequently, sometimes may deteriorate the overall performance. Even worse, it may lead to a Breadth First Search, a very terrible way to extend the voting tree. How to avoid such kind of unstable updating? The previous estimate should not been completely discard. Then, we propose a weight decay approach to reuse the previous estimation of a state. It will be much stable than calculating the weight from scratch. We call the latest data shows us a local estimate and the previous computing can be broad and even global evaluation. The core idea is that the recent feedback from the election contains more important and accurate information of the overall situation. Therefore, at the same time, we should boost the estimation based upon the statistics (e.g. winning frequency) of the alternatives in the current state and the election outcomes (who is the new winner).

The ordering of nodes in fringe should be considered as an important signal and tuning it based on the added data. Let T be the fringe, and $T(S)$ be age of state S kept in T. The recently added state has its age $T(S) = 0$. DFS gives young state higher weights than old ones to make sure it is a LIFO (Last-In-First-Out) stack, and its implicit priority function is $F(S,P) = \|T\| - T(S)$. Age is an important feature, the frequency of one alternatives being elected in previous rounds can also provide some insights for us. Those states contain many alternatives with higher frequencies are given a low priority.

- $F(S,P) = (\|T\| - T(S))/[1 + \sum_{a \in S} f(a)]$, where $f(a)$ is the frequency of a being elected.
- $F(S,P) = (\|T\| - T(S))/[1 + \sum_{a \in S} (f(a) * B(a)/\text{MaxB}(m))]$, where $B(a)$ is a's Borda score, $\text{MaxB}(m) = m(m-1)n$ is the maximum Borda score that an alternative can get from n voters over m alternatives.
- $F(S,P) = (\|T\| - T(S))/[1 + \sum_{a \in S} (1 + f(a) * P(a)/n)]$ where $P(a)$ is the Plurality score of a.
- $F(S,P) = \alpha F(S,P,t-1) + F(S,P,t)$, where $\alpha \in [0,1]$ is the damping factor. $F(S,P,t)$ is the same as we did in previous versions which discard their previous estimation.

Another direction valuable to explore is machine learning, specifically the *reinforcement learning* (e.g. AlphaGo). Suppose we have learned a election function which can scoring each alternative in a state to show its probability of being elected. We can take advantage the probability prediction to choose a better next state. Here, we have to solve an important searching problem: **you have a perfect prediction model, from which you know who are the winners, how would you walk from the top-to-bottom of an implicit voting tree with a minimum path, and find all the winners.** All the possible paths are ruled by the specific voting rule. If the tree is fully extended, it will be a trivial question.

Let us denote g the learned model and define several priority function F based on g:

- $F(S,P) = \sum_{u \in U} g(u,P)/\|S\|$
- $F(S,P) = \max_{u \in U} g(u,P)/\max_{a \in S} g(a,P) \in (0,1]$
- $F(S,P) = \sum_{u \in U} g(u,P)/\sum_{a \in S} g(a,P) \in [0,1]$

According to our observation on the learned model, it fails to search the winner effectively. However, there is an interesting pattern in the number of nodes to visit for each new winner – winners are found at the end stage and the number of nodes to extend is very close. One possible reason is the model makes it a BFS-based voting approach. We

would like to explore those states with more tied-alternatives, and try our best to prevent BFS from happening.

8.2 Approximation

We can also design some approximation method to solve the problem. Sampling could be the easiest one approach to get close to the exact solution. We proposed another method with machine learning, and made predictions from states, therefore we built a rough image of the big picture. All predicted winners will be assigned the same weights for being predicted one time. The cumulated weights will be used to guide our heuristic search. More specifically, we first extend the tree based on BFS, with all tied states, we make prediction and compute the weights of all predicted winners. The predicted winner who gets the smallest weight will have top priority to be reevaluated and extended. To make the number of nodes extended as small as possible, we may set a predefined depth to explore.

Given a prediction model f with prediction accuracy (recall, precision) e. Let's use it to predict the winners in multi-round STV election method, what's the probability of making error based on the aggregated results of top k layers of the tree. The least number of layers we have to extend to make prediction with 100% accuracy. Assume the children can produce a higher accurate prediction, could we overthrow the inconsistent predictions that the parent node makes with all that of its children nodes. Take care of those unable to detected, they must have some special characteristics that prevent us to identify them. These special characteristics maybe helpful to design a good heuristic.

8.3 Bound the Number of Nodes

Given a perfectly symmetric profile of m alternatives, the number of voters is $n = m!$, we would like to derive the minimum number of nodes such that we can compute all possible

winners from the profile. We notice that, in $m!$ preference rankings of alternatives $A = \{1, 2, \ldots, m\}$, there are m preference rankings contains each of $(m-1)!$ preference rankings of $A-\{i\}$, $i = 1, 2, \ldots, m$. It indicates that each sub-preference-rankings of $A-\{i\}$ are equally distributed over $m!$ preference rankings of A. Let $N(m)$ denote the minimum number of nodes, the following equation must hold

$$N(m) = 1 + N(m-1) + (m-1),$$

where the first term is the root node that contains all alternatives, the second term is the minimum number of nodes we have to extend for the branch without $i \in A$. From one branch, we can have all alternatives except one alternative get one change of being elected. Then, we need at most extend $m-1$ nodes from top-to-bottom along one path to find the remaining alternative and select it as the winner. The number is represented by the third term in above equation.

Based on the equation, we can get $N(m) = N(m-1) + m = N(m-2) + (m-1) + m = N(2) + 3 + 4 + \ldots + m$, where $N(2)$ is the minimum number of nodes to extend, it's obvious $N(2) = 3$, therefore $N(m) = m(m+1)/2$. For a perfectly symmetric profile of $m = 10$ alternatives, we have to extend at least $N(10) = 10 * 11/2 = 55$ nodes.

For other kinds of profiles, is it possible to compute the optimal number of nodes to extend? If not, could we get a bound for the number? Suppose we have an sub-optimal search rule, and given a specific profile, how can be get the optimal path from the sub-optimal search paths. If we can answer at least one of the above questions, we will be able to compute the maximum number to reduce. Suppose a branch contains only one single winner, however it may contains some ties. If we can prune this kind of branch after we have found a winner and we are quite sure that it won't produce more than one winner, the number of nodes reduced must be very large. In order to discover the winners as soon as possible (early discovery), we expect that the number of nodes to extend for a new winner should be optimal. For example, the optimal number of nodes to get the first winner should be the size of the start state. Suppose a profile has a unique winner, the

early discover will has only one result, and will be always the optimal one for a rule that walks from top-to-bottom without disruption. How about the remaining winners? Could we find a way to compute the optimal values if there are multi-winners?

Based upon the idea, we can make improvement over the multi-round STV. The approach is to train a model that could detect a profile with single winner with (or close to) 100%. Then, we could use the model to avoid many unnecessary explorations.

8.4 Select the Best or Avoid the Worst

Most researchers are designing a fair voting rule or mechanism to choose a candidates. However, avoiding mistakes in decision making on the other side has not received an equivalent amount of attention. Electing a warmonger presidential candidate into office is generally worse than having a mediocre but harmless president. Sometimes, a mistake could lead to even a disaster for the whole society.

8.5 Randomization Analysis

Expected depth of any internal node in the voting tree, or the expected depth of a leaf node. To the multi-round winner determination procedure, it can be divided into two steps:

1. Estimate the probability of a candidate being a winner
2. Estimate the expected depth to place the candidate at a leaf node

9 Multivariate Decision Tree

Most decision trees are univariate, in which each decision node splits training examples based on one single input feature or variable. The multivariate decision trees (a.k.a *oblique tree*) is different, it splits data points based on a subset of the entire feature space at decision nodes. It's able to formulate a more complex structure presented in the input data set than the univariate decision tree.

An intuitive approach is to learn an (local or global) optimal hyperplane and separate data points to two half-spaces. Various methods have been proposed to search the hyperplanes. For example, the Classification and Regression Tree, a.k.a CART[1] is one of the first algorithms that allowed multivariate split. Based on a heuristic hill climbing method with backward feature elimination, CART searches an optimal linear separator in each internal decision nodes. Murthy et al.[20] made a thorough extension of CART's strategy and proposed the efficient Oblique Classifier 1 (OC1). It searches for the best univariate split as well as the best oblique split, and it only employs the oblique split when there is an improvement over the univariate split. OC1 uses both a deterministic heuristic search (as in CART) for finding local-optima and a non-deterministic search for escaping local-optima. Ittner and Schlosser[9] proposes using OC1 over an augmented feature space, generating non-linear decision trees. The key idea is to create new features by considering all possible pairwise products and squares of the original features.

You and Fu[37] learned a linear combination of features at each decision node based on the Fletcher-Powell descent method. Park and Sklansky[21] induced a piece-wise linear discriminant that cut a maximal number of *Tomek links* based on the principle of locally opposed clusters of objects. A Tomek link connects an opposed pair of data points for which the circle of influence between the pair doesn't contain any other points, as shown in Fig.2. Tenmoto et al.[32] proposed to construct a piecewise linear classifier which cuts multiple Tomek links based on[21]. We could implement the clustering procedure based on the density peak method[26].[27] proposed to combine neural network and decision tree

- neural tree network.

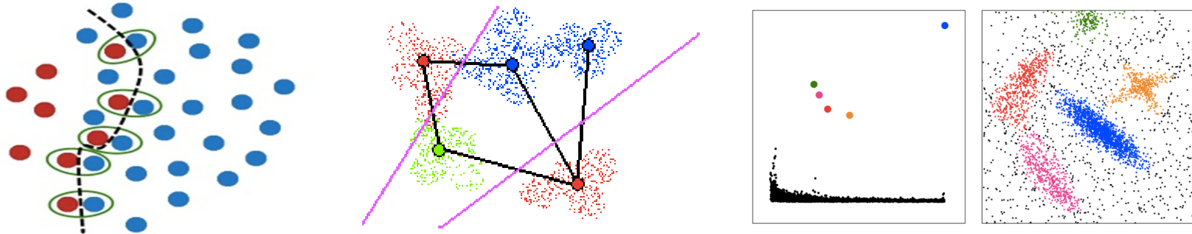

Fig. 2: Tomek Link, Maximum Cut Hyperplane and Density Peaks

9.1 Tomek Links and Condensed Nearest Neighborhood

Gabriel Graph, piecewise linear classification.

9.2 Maximum Cut Planes

To simulate the procedure of using a linear combination of input features to split the input space, and construct a decision tree-like model with multiple hyperplanes. The basic idea is to recursively search an optimal linear hyperplane to split the current input space into two regions. Then, in similar pattern split the half regions until the split regions satisfying certain criterions.

There are numerous ways to search those optimal linear functions. Each method relies on a specific loss function to iteratively approximate the voting procedure. In this section, we talk about one special process – maximum cut planes. Each hyperplane is a linear function that cut through majority of the neighbor-pairs.

Let X be the input space, and $x_i, x_j \in X$ are two input examples with different labels, i.e. $y_i \neq y_j$. In our case, each input example represents a preference profile. We have a well-defined distance measure $d : X \times X \mapsto \mathbb{R}$, and a predefined threshold $\epsilon > 0$. Those pairs of

input examples with distance less than ϵ form the ϵ - neighborhood $N(X;\epsilon)$, i.e.

$$N(X;\epsilon) = \Big\{(x_i, x_j) | d(x_i, x_j) \leq \epsilon; y_i \neq y_j; x_i, x_j \in X\Big\}.$$

Each element in $N(X;\epsilon)$ is called an ϵ - neighbor pair.

Suppose there is a linear function $f : X \mapsto \mathbb{R}$. The linear function could split the whole input space to two sub-regions: positive one X_+ and negative one X_-, it's certain that $X = X_+ \cup X_-$. Examples in the positive and negative regions respectively satisfy the following relations

$$\begin{cases} f(x;w) \geq 0, & x \in X_+, \\ f(x;w) \leq 0, & x \in X_-. \end{cases}$$

We say the linear function f cuts an ϵ - neighbor pair $(x_i, x_j) \in N(X;\epsilon)$ if $f(x_i;w)f(x_j;w) \leq 0$. The linear model is a cut plane separating x_i and x_j to different sides. To search an optimal linear function f that cuts through most number of ϵ - neighbor pairs, we have

$$w^* = \arg\max_{w} \sum_{(x_i,x_j) \in N(X;\epsilon)} [[f(x_i;w)f(x_j;w) < 0]]$$

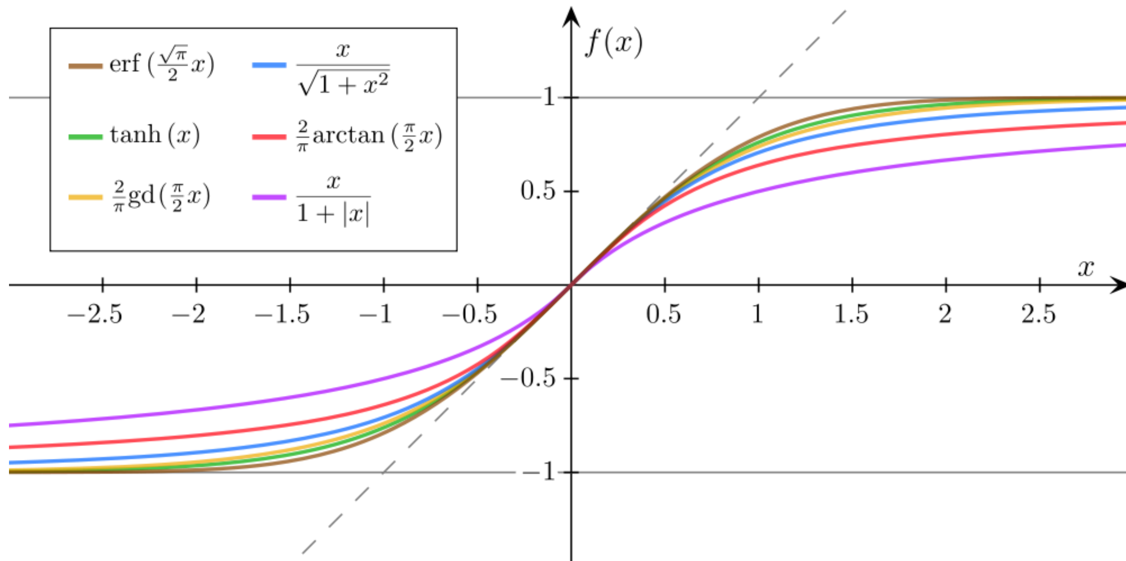

Fig. 3: Sigmoid Functions

It's very hard to optimize a non-smooth function, we choose to maximize one of its upper bounds. With the fact that $[[x < 0]] \leq \log(1 + e^{-x})$, then we choose to maximize a smooth function[14] [1]

$$w^* = \arg\max_w L(X; w) = \sum_{(x_i, x_j) \in N(X;\epsilon)} \log\left\{1 + e^{-f(x_i;w) f(x_j;w)}\right\}.$$

Therefore, we have the gradient of $L(X; w)$ w.r.t w

$$\begin{aligned}\nabla_w L &= - \sum_{(x_i, x_j) \in N(X;\epsilon)} \left[1 + e^{-f(x_i;w) f(x_j;w)}\right]^{-1} \left[x_i w^T x_j + x_j w^T x_i\right] \\ &= - \sum_{(x_i, x_j) \in N(X;\epsilon)} \left[(w^T x_j) x_i + (w^T x_i) x_j\right] / \left[1 + e^{-(w^T x_i)(w^T x_j)}\right].\end{aligned}$$

9.2.1 Speedup Iteration

The current updating strategy is not efficient, many alternative method can be adopted to possiblely speed up the optimization.

- Eliminate redundant features, i.e. zero padding features (e.g. the number or percentage of votes that put both a and b at the first place) or strongly related features are eliminated.
- Mutual neighbor pairs receive higher weights for their mutual-strength relations. Zero padding features are assigned zero weight, and the features in favorite places are given higher weights (ordered weight for the positional features).
- Approximated analytic solution to the optimization problem.
- Alternative smooth functions to approximate an indicator function.

[1] Alternatively, we can approximate it with a logistic function[25]

$$[[x > 0]] \approx [1 + e^{-\alpha x}]^{-1},$$

where $\alpha > 0$ is a scale constant. A larger α makes the approximation closer to the indicator function, and tends to be more non-smooth[14].

- Initial weighting strategy, e.g. the maximum pairwise condordance, the angular histogram distribution method (set a neighbor-pair as the basis vector, construct an angular histogram over all neighbor pairs relative to the basis vector, the the most dense norm vector is selected as the initial weight for the linear separator) , ordered and ℓ_1 or ℓ_2 normalized weighting constraints, inheriting separator from the parent node. Another method is similar to the approach in Boosting method, the closest neighbor-pairs are assigned higher weights so that be selected at high priority at the next iteration, because these pairs are the hardest ones to separate.

- Using different distance kernel function to measure the geometric nearness, including the Gaussian kernel based on the Euclidean distance

  $$K_{Gaussian}(x,y) = \exp\left\{-\frac{d^2_{Euclidean}(x,y)}{2\sigma^2}\right\},$$

  and the Entropic divergence kernel based on Kullback-Leibler divergence (or relative entropy)

  $$K_{Entropic}(x,y) = \exp\left\{-\frac{d^2_{KL}(x,y)}{2\sigma^2}\right\}.$$

9.2.2 Generalization

Borda rule can be presented with a multi-variate decision tree, and we handcraft linear functions for Borda rule, and directly generating multidimensional feature vectors. Then, labeling these data points with the handcrafted functions to construct the training set. Empirically, if we could learn Borda rule with high enough accuracy, it indicates that the number of representative data points is the key to learn the general scoring rule, including the Condorcet method with high generalization. The simulation with a smaller number of voters fails to provide the structure for the preference profiles.

9.2.3 Parameters & Hyper-parameters

To grow a decision tree, we have to optimize many parameters or hyper-parameters, including the maximum depth d_{max}, the maximum number of leaf nodes l_{max}, the minimum size of each leaf node l_{sz}, the measure used to split variable $s_{\mathcal{M}}$, the minimum split size s_{sz}, whether tree pruning is required. For the case of maximum cutting plane method, additional parameters are introduced: the number K of nearest neighbors or the threshold of distance $\bar{d}$ to define the neighborhood, the approximation smooth function f of the indicator function, the method to optimize the objective function (batch, mini-batch or stochastic gradient descent method), the learning rate η, the maximum iterations n_{max} in searching a local optimal cutting plane (or separator), the stop criterion in terms of misclassification error or cutting ratio or the surrogate objective value, the regularization coefficients $\lambda_1 > 0$ for ℓ_1 and $\lambda_2 > 0$ for ℓ_2 (if regularization is available).

9.2.4 VC-Bound

We are studying the misclassification probability bounds for decision trees, especially for multivariate decision trees. Based on the bound, we can estimate the generalization error (a.k.a out-of-sample error) and estimate the sufficient number of training set required for a specific thresholded error bar ϵ.[8] shown that with high probability any decision tree of depth no more than d that is consistent with n training examples has misclassification probability no more than $O(\frac{|\log n|}{n}\sqrt{M \cdot d_{VC}(\mathcal{C}) \cdot \log d})$, where $\mathcal{C}$ is the class of node decision functions, and M is the effective number of leaves, which is smaller than L the true number of leaf nodes.

9.2.5 Boolean Functions Learning

Each decision tree is a Disjunctive Normal Form (DNF) – a disjunction of conjunctive clauses. It is very useful in automated theorem proving.

10 Multi-Class Perceptron

Each class is related to a predictor and data points are classified based on the predictions from all predictors. The predictor which produce the largest prediction value will assign data points the corresponding label.

Suppose there are K classes in prediction, and the corresponding predictors are

$$f_k(x) = w_k^T x.$$

Given a data point x, it has a label $\hat{y}$ according to the predictions by all the K predictors

$$\hat{y} = \arg\max_k f_k(x).$$

The multi-class perceptron algorithm is very straightforward, just like the perceptron algorithm in binary classification. No need to compute gradients, it updates weights by adjusting the weight vectors when there is misclassified data points. Suppose there is a training data point (x, y), which is unable correctly classified by the predictors, i.e. $y \neq \hat{y}$. The algorithm update the weights of the predictor f_y and $f_{\hat{y}}$ as follows

$$\begin{cases} w_y &= w_y + \eta x, \\ w_{\hat{y}} &= w_{\hat{y}} - \eta x, \end{cases}$$

where $\eta > 0$ is a pseudo-learning rate to control the step-size in updating the existing weights.

11 Deep Neural Decision Forests[11]

- $x \in X$: an input example
- $y \in Y$: an valid label
- $\mathbb{I}$: the internal or decision nodes
- $\mathbb{L}$: the terminal or leaf nodes
- $\ell \in \mathbb{L}$: one leaf node
- $i \in \mathbb{I}$: one decision node
- k: the depth of the decision tree (root node is the zeroth layer)
- n: the number of decision nodes
- m: the dimension of the input x
- T: the deep neural decision tree
- $\theta_i \in \mathbb{R}^m$: the weight of the decision node i
- $d_i(x;\theta)$: the probability of x goes into the left branch of i
- $f_i(x;\theta)$: the decision function of node i for x
- b_{il}, b_{ir}: the nodes belonging to the right (left) branch of i
- $\mu_\ell(x;\theta)$: the probability of x reaching to the leaf node ℓ
- $\pi_\ell(y)$: the probability of having the label y at the leaf node ℓ
- $P_T(y|x;\theta,\pi)$: the probability of assigning x with the label y

There are some important properties or close relations between different variables. At each leaf node, there is a probability distribution over the labels Y, and we have

$$\sum_{y \in Y} \pi_\ell(y) = 1, \quad \forall \ell \in \mathbb{L}.$$

At each decision node, for any x, its probability of reaching different leaves could be that

$$\sum_{\ell \in \mathbb{L}} \mu_\ell(x;\theta) = 1.$$

With the decision function, the probability of turning left is

$$d_i(x;\theta) = \sigma(f_i(x;\theta)) = \frac{1}{1 + e^{-f_i(x;\theta)}},$$

and the probability of turning right is

$$\bar{d}_i(x;\theta) = 1 - d_i(x;\theta) = \frac{1}{1 + e^{f_i(x;\theta)}}.$$

Actually, $f_i(x;\theta)$ could have many complex structure, e.g. neural network. Here, we just adopt one simplified version, which is a linear function, i.e. $f_i(x;\theta) = \theta_i^T x$, θ_i becomes the only effective parameter.

To minimize the log-form loss over the training set, we adopt the stochastic gradient descending method. Here the function is defined as follows:

$$\begin{cases} \mathcal{L}(\theta,\pi;D) = & \sum_{(x,y) \in D} L(\theta,\pi;x,y)/|D| \\ L(\theta,\pi;x,y) = & -\log P_T(y|x;\theta,\pi) \end{cases} \tag{9}$$

Here the decision model of the decision tree could be viewed as an average of the predictions of each leaf node, i.e.

$$P_T(y|x;\theta,\pi) = \sum_{\ell \in \mathbb{L}} \mu_\ell(x;\theta)\pi_\ell(y). \tag{10}$$

where

$$\mu_\ell(x;\theta) = \prod_{i \in \mathbb{I}} d_i(x;\theta)^{I(\ell \in b_{il})} \bar{d}_i(x;\theta)^{I(\ell \in b_{ir})}. \tag{11}$$

11.1 Gradient w.r.t θ

The neural decision tree contains two types of parameters, one from decision nodes and the other from leaf nodes. We compute the gradient of the loss function with respect to θ and π:

$$\begin{aligned} \frac{\partial L(\theta,\pi;x,y)}{\partial\theta_i} &= -\frac{1}{P_T(y|x;\theta,\pi)}\frac{\partial P_T(y|x;\theta,\pi)}{\partial\theta_i} \\ &= -\frac{1}{P_T(y|x;\theta,\pi)}\sum_{\ell\in\mathbb{L}}\pi_\ell(y)\frac{\partial\mu_\ell(x;\theta)}{\partial\theta_i} \end{aligned} \tag{12}$$

Let's focus on $\mu_\ell(x;\theta)$ and have an equivalent transformation

$$\mu_\ell(x;\theta) = e^{\log\mu_\ell(x;\theta)} = e^{\sum_{i\in\mathbb{I}}\left[I(\ell\in b_{il})\log d_i(x;\theta)+I(\ell\in b_{ir})\log\bar{d}_i(x;\theta)\right]},$$

Then, we can obtain the derivation of $\mu_\ell(x;\theta)$ w.r.t θ_i

$$\begin{aligned} \frac{\partial\mu_\ell(x;\theta)}{\partial\theta_i} &= \mu_\ell(x;\theta)\left[\frac{I(\ell\in b_{il})}{d_i(x;\theta)} - \frac{I(\ell\in b_{ir})}{\bar{d}_i(x;\theta)}\right]\frac{\partial d_i(x;\theta)}{\partial\theta_i} \\ &= \mu_\ell(x;\theta)\Big[I(\ell\in b_{il})\bar{d}_i(x;\theta) - I(\ell\in b_{ir})d_i(x;\theta)\Big]\frac{\partial f_i(x;\theta)}{\partial\theta_i}. \end{aligned} \tag{13}$$

Plugging it into the expression of $\partial L(\theta,\pi;x,y)/\partial\theta_i$, we could yield the following result

$$\frac{\partial L(\theta,\pi;x,y)}{\partial\theta_i} = \sum_{\ell\in\mathbb{L}}\frac{\mu_\ell(x;\theta)\pi_\ell(y)}{P_T(y|x;\theta,\pi)}\Big[I(\ell\in b_{ir})d_i(x;\theta) - I(\ell\in b_{il})\bar{d}_i(x;\theta)\Big]\frac{\partial f_i(x;\theta)}{\partial\theta_i}. \tag{14}$$

11.2 Gradient w.r.t π

Similarly, we could derivate the gradient of $L(\theta,\pi;x,y)$ w.r.t π. Here, for $\pi_\ell(y)$

$$\begin{aligned} \frac{\partial L(\theta,\pi;x,y)}{\partial\pi_\ell(y)} &= -\frac{1}{P_T(y|x;\theta,\pi)}\frac{\partial P_T(y|x;\theta,\pi)}{\partial\pi_\ell(y)} \\ &= -\frac{\mu_\ell(x|\theta)}{P_T(y|x;\theta,\pi)}. \end{aligned} \tag{15}$$

11.3 Updating Rule

Let

$$\begin{cases} A_\ell(x,y) &= \frac{\pi_\ell(y)\mu_\ell(x|\theta)}{P_T(y|x;\theta,\pi)}, \quad \ell \in \mathbb{L}, \\ A_{il}(x,y) &= \sum_{\ell\in\mathbb{L}} A_\ell(x,y) I(\ell \in b_{il}), \quad i \in \mathbb{I}, \\ A_{ir}(x,y) &= \sum_{\ell\in\mathbb{L}} A_\ell(x,y) I(\ell \in b_{ir}), \quad i \in \mathbb{I}. \end{cases}$$

The parameter π will be updated based on the following rule

$$\begin{aligned} \pi_\ell^{(t+1)}(y) &= \frac{1}{Z_\ell^{(t)}} \sum_{(x,y')\in D} I(y'=y) \frac{\pi_\ell^{(t)}(y)\mu_\ell(x|\theta)}{P_T(y|x;\theta,\pi^{(t)})} \\ &= \frac{1}{Z_\ell^{(t)}} \sum_{(x,y')\in D} I(y'=y) A_\ell^{(t)}(x,y). \end{aligned} \tag{16}$$

The gradient w.r.t θ_i

$$\frac{\partial L(\theta,\pi;x,y)}{\partial \theta_i} = \left[A_{ir}(x,y) d_i(x;\theta_i) - A_{il}(x,y)\bar{d}_i(x;\theta)\right] \frac{\partial f_i(x;\theta)}{\partial \theta_i}, \quad i \in \mathbb{I} \tag{17}$$

could also be computed from bottom to up, and we could update θ_i in consequent steps.

11.4 Steps

- $\theta^{(t)}$, $\pi^{(t)}$
- $d_i(x;\theta^{(t)}) = \sigma(f_i(x;\theta_i^{(t)}))$
- $\mu_\ell(x;\theta^{(t)}) = \prod_{i\in\mathbb{I}} d_i(x;\theta^{(t)})^{I(\ell\in b_{il})} \bar{d}_i(x;\theta^{(t)})^{I(\ell\in b_{ir})}$
- $P_T(y|x;\theta^{(t)},\pi^{(t)}) = \sum_{\ell\in\mathbb{L}} \mu_\ell(x;\theta^{(t)}) \pi_\ell^{(t)}(y)$
- $A_\ell^{(t)}(x,y) = \frac{\mu_\ell(x;\theta^{(t)})\pi_\ell^{(t)}(y)}{P_T(y|x;\theta^{(t)},\pi^{(t)})}$
- *Batch*: $\pi_\ell^{(t+1)}(y) = \frac{1}{Z_\ell^{(t)}} \sum_{(x,y')\in D} I(y'=y) A_\ell^{(t)}(x,y)$
- $A_{il}^{(t)}(x,y) = \sum_{\ell\in\mathbb{L}} A_\ell^{(t)}(x,y) I(\ell \in b_{il})$

- $A_{ir}^{(t)}(x,y) = \sum_{\ell \in \mathbb{L}} A_{\ell}^{(t)}(x,y) I(\ell \in b_{ir})$
- *Stochastic*: $\theta_i^{(t+1)} = \theta_i^{(t)} - \alpha \Big[A_{ir}^{(t)}(x,y) d_i(x;\theta^{(t)}) - A_{il}^{(t)}(x,y) \bar{d}_i(x;\theta^{(t)}) \Big] \frac{\partial f_i(x;\theta)}{\partial \theta_i}$
- *(Mini-)Batch*: $\theta_i^{(t+1)} = \theta_i^{(t)} - \alpha \sum_{(x,y) \in S \subset D} \Big[A_{ir}^{(t)}(x,y) d_i(x;\theta^{(t)}) - A_{il}^{(t)}(x,y) \bar{d}_i(x;\theta^{(t)}) \Big] \frac{\partial f_i(x;\theta)}{\partial \theta_i}$

12 Neural Network

Artificial neural network is one promising technique to train a classifier. Based on specific loss function, we have to derivate the gradients of the neural network on each layer w.r.t weights. We introduce the detailed information on the gradients computing in multi-layer neural network learning.

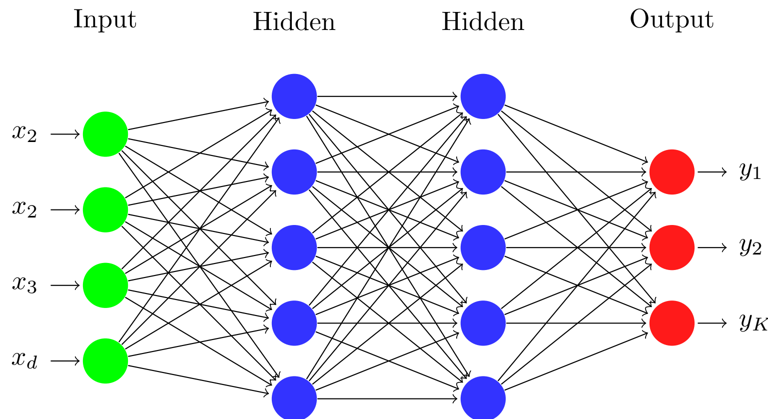

Fig. 4: A Neural Network of Four Layers

Considering a neural network with multiple layers $G = \{L_0, L_1, \ldots, L_K\}$, where L_0 is the input layer, L_1 is the first hidden layer, and L_K is the output layer. Each layer contains some nodes $L_k = \{n_{k1}, n_{k2}, \ldots, n_{kd_k}\}$, and $d_k = |L_k|$ is the number of nodes that parks at L_k, $k = 0, 1, \ldots, K$. Let X be the input space, Y be the output space. Obviously, $X \in \mathbb{R}^{d_0}, |Y| = d_K$. Moreover, let Θ be the parameter space, and the neural network can be formulated as

$$f(\cdot;\Theta) : X \mapsto Y.$$

The parameter space is hierarchical, with parameters distributed over the edges connecting layers close to each other. We denote $w_{ij}^k \in \mathbb{R}$ the weight of the edge connecting $n_{k-1,j} \in L_{k-1}$ to $n_{ki} \in L_k$. Each layer has one activation function, which in general a sigmoid function, denote as $\sigma_k, k = 0, 1, \ldots, K$, and $\sigma_0(x) = x$.

We introduce the classical back-prop algorithm on a fully connected multi-layerred neu-

ral network. The algorithm contains two primary steps: forward feed propagation and back-propagation. At the phase of forward propagation, an instance x is feed and propagated over the network until reaching the output layer. The network produces a prediction of the input instance, which most likely does not consistent with the true label y of the input instance. At the phase of back propagation, the network will propagate the loss of the prediction to the ground-truth label, iteratively adjust the weights of each layer from the output to the input.

The input x_{ki} of node $n_{ki} \in L_k$ is a weighted sum of the outputs $z_{k-1} \in \mathbb{R}^{d_{k-1}}$ of the prior layer L_{k-1}, i.e.

$$x_{ki} = \sum_{j=1}^{d_{k-1}} z_{k-1,j} w_{ij}^k, k = 1, 2, \ldots, K,$$

and the output z_{ki} of n_{ki} is a sigmoid transformation of x_{ki} with $z_{ki} = \sigma_k(x_{ki})$.

12.1 Forward Propagation

Given an instance $(x, y) \in X \times Y$, we have the forward propagation procedure

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{d_0} \end{bmatrix} \overrightarrow{\sigma_0} \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_{d_0} \end{bmatrix} \overset{W^1}{\overrightarrow{\sigma_1}} \begin{bmatrix} z_{11} = \sigma_1(x_{11}) = \sigma_1(\sum_{j=1}^{d_0} z_j w_{1j}^1) \\ z_{12} = \sigma_1(x_{12}) = \sigma_1(\sum_{j=1}^{d_0} z_j w_{2j}^1) \\ \vdots \\ z_{1d_1} = \sigma_1(x_{1d_1}) = \sigma_1(\sum_{j=1}^{d_0} z_j w_{d_1,j}^1) \end{bmatrix} \cdots$$

$$\cdots \overset{W^k}{\overrightarrow{\sigma_k}} \begin{bmatrix} z_{k1} = \sigma_k(x_{k1}) = \sigma_k(\sum_{j=1}^{d_{k-1}} z_{k-1,j} w_{1j}^k) \\ z_{k2} = \sigma_k(x_{k2}) = \sigma_k(\sum_{j=1}^{d_{k-1}} z_{k-1,j} w_{2j}^k) \\ \vdots \\ z_{kd_k} = \sigma_k(x_{kd_k}) = \sigma_k(\sum_{j=1}^{d_{k-1}} z_{k-1,j} w_{d_k,j}^k) \end{bmatrix} \overset{W^{k+1}}{\overrightarrow{\sigma_{k+1}}} \cdots$$

$$\cdots \overset{W^K}{\underset{\sigma_K}{\rightarrow}} \begin{bmatrix} z_{K1} = \sigma_K(x_{K1}) = \sigma_K(\sum_{j=1}^{d_{K-1}} z_{K-1,j} w_{1j}^K) \\ z_{K2} = \sigma_K(x_{K2}) = \sigma_K(\sum_{j=1}^{d_{K-1}} z_{K-1,j} w_{2j}^K) \\ \vdots \\ z_{Kd_K} = \sigma_K(x_{Kd_K}) = \sigma_K(\sum_{j=1}^{d_{K-1}} z_{K-1,j} w_{d_K,j}^K) \end{bmatrix}$$

and define the generic error function between the prediction of the network and the ground-truth label y of x

$$e(x,y;\Theta) = e\Big(f(x;\Theta), y\Big) = e\Big(z_{K1}, \dots, z_{Kd_K}; y_1, \dots, y_{d_K}\Big).$$

12.2 Back Propagation

To minimize the loss function, we utilize the stochastic gradient descend method. Therefore, the gradients of $e(x,y;\Theta)$ w.r.t $\Theta = \{W^K, W^{K-1}, \dots, W^1\}$ should be computed and then weights are updated based on delta-rule, where $W^k \in \mathbb{R}^{d_k \times d_{k-1}}$ represents all the weights of edges connecting L_{k-1} and L_k with the following matrix form

$$W^k = \begin{bmatrix} w_{11}^k & w_{12}^k & \cdots & w_{1,d_{k-1}}^k \\ w_{21}^k & w_{22}^k & \cdots & w_{2,d_{k-1}}^k \\ \vdots & \vdots & \ddots & \vdots \\ w_{d_k,1}^k & w_{d_k,2}^k & \cdots & w_{d_k,d_{k-1}}^k \end{bmatrix}, \forall k = 1, \dots, K.$$

Considering the complex structure of the network, we derivate the gradients with chain rule from L_K to L_{K-1} until L_1.

Let's see the weight w_{ij}^K between $n_{K-1,j}$ and n_{Ki}. The weight is independent to the other weights in L_K, but dependent on the output z_{Ki} and therefore the input x_{Ki}, then we can obtain the gradient w.r.t w_{ij}^K

$$\frac{\partial e(x,y;\Theta)}{\partial w_{ij}^K} = \frac{\partial e(x,y;\Theta)}{\partial z_{Ki}} \frac{\partial z_{Ki}}{\partial x_{Ki}} \frac{\partial x_{Ki}}{\partial w_{ij}^K} = \frac{\partial e(x,y;\Theta)}{\partial z_{Ki}} \sigma_K'(x_{Ki}) z_{K-1,j}, \tag{18}$$

with the facts that $z_{Ki} = \sigma_K(x_{Ki})$, $x_{Ki} = \sum_{j=1}^{d_{K-1}} z_{K-1,j} w_{ij}^K$ and

$$\begin{cases} \partial z_{Ki}/\partial x_{Ki} &= \sigma_K'(x_{Ki}), \\ \partial x_{Ki}/\partial w_{ij}^K &= z_{K-1,j}. \end{cases}$$

Similarly, we obtain the gradient w.r.t w_{ij}^k

$$\frac{\partial e(x,y;\Theta)}{\partial w_{ij}^k} = \frac{\partial e(x,y;\Theta)}{\partial z_{ki}} \frac{\partial z_{ki}}{\partial x_{ki}} \frac{\partial x_{ki}}{\partial w_{ij}^k} = \frac{\partial e(x,y;\Theta)}{\partial z_{ki}} \sigma_k'(x_{ki}) z_{k-1,j}. \tag{19}$$

There is one quite special aspect in calculation of the gradient w.r.t w_{ij}^k is the derivate $\partial e(x,y;\Theta)/\partial z_{ki}$. The node n_{ki} is the very and the only node, from which the weight w_{ij}^k affects the error function $e(x,y;\Theta)$. Along the route propagating forward, the weight w_{ij}^k affects z_{ki}, then all nodes in L_{k+1}, therefore $z_{k+1,1}, z_{k+1,2}, \ldots, z_{k+1,d_{k+1}}$, and each of which can be viewed as a function of w_{ij}^k. Following the chain rule

$$\frac{\partial e(x,y;\Theta)}{\partial z_{ki}} = \frac{\partial e(x,y;\Theta)}{\partial z_{k+1,1}} \frac{\partial z_{k+1,1}}{\partial z_{ki}} + \frac{\partial e(x,y;\Theta)}{\partial z_{k+1,2}} \frac{\partial z_{k+1,2}}{\partial z_{ki}} + \cdots + \frac{\partial e(x,y;\Theta)}{\partial z_{k+1,d_{k+1}}} \frac{\partial z_{k+1,d_{k+1}}}{\partial z_{ki}}, \tag{20}$$

and given the gradients $\partial e(x,y;\Theta)/\partial z_{k+1,1}, \partial e(x,y;\Theta)/\partial z_{k+1,2}, \ldots, \partial e(x,y;\Theta)/\partial z_{k+1,d_{k+1}}$ over all outputs of L_{k+1}, we obtain the gradients of $e(x,y;\Theta)$ w.r.t each weight in G.

We expand these terms and make the above equations more explicit. Based on the relations between these outputs of L_k and L_{k+1}

$$\begin{cases} x_{k+1,j} &= \sum_{i=1}^{d_k} z_{ki} w_{ji}^{k+1}, \\ z_{k+1,j} &= \sigma_{k+1}(x_{k+1,j}), \end{cases}$$

we have the derivates

$$\frac{\partial z_{k+1,j}}{\partial z_{ki}} = \sigma_{k+1}'(x_{k+1,j}) \frac{\partial x_{k+1,j}}{\partial z_{ki}} = \sigma_{k+1}'(x_{k+1,j}) w_{ji}^{k+1}, \forall j = 1, 2, \ldots, d_{k+1},$$

and $\partial e(x,y;\Theta)/\partial z_{ki}$ in L_k is a linear combination of all the gradients $\partial e(x,y;\Theta)/\partial z_{k+1,j}$ in L_{k+1} with importances $\alpha_{ij}^k = \sigma_{k+1}'(x_{k+1,j}) w_{ji}^{k+1}, \forall j = 1, 2, \ldots, d_{k+1}$.

Back-propagation procedure propagates the gradients of the error function to the outputs of L_{d_K} backward to the last hidden layer $L_{d_{K-1}}$ in a form of linear combination, then the cumulated gradients are propagated again to the layer before its home layer, so on and so forth, until the gradients signals are propagated to the input nodes. Meanwhile, the weight vectors of each layer are updated using these local cumulated gradients with certain updating rule.

12.3 Hyperparameters

The number of layers $K > 2$, the number of hidden nodes $\{d_1, d_2, \ldots, d_{K-1}\}$, the activation functions $\{\sigma_1, \sigma_2, \ldots, \sigma_K\}$, the weight updating rule δ, the number of iterations m and the learning rate η to update weights, the error function e, the penalty coefficient λ (if regularization is available), and some other variables in some specific methods, such as the batch-size b in the mini-batch gradient descent method, even the optimization algorithms, are all hyperparameters require careful tuning.

12.3.1 Activation Function σ

The most commonly used activation functions include the rectifier, logistic function, the softmax function, and the gaussian function. The logistic regression model can be implemented using a neural network of an input layer, one output node with a logistic activation function. The back propagation algorithm for calculating a gradient has been rediscovered a number of times, and it is a special case of a more general technique called *automatic differentiation* in the reverse accumulation mode.

12.3.2 Error Function e

The error function can be smooth or non-smooth based on its application and the learning algorithm. The most popular error functions include *the squared error function*, *the*

misclassification function, and *the cross entropy error function*. The non-convexity of the error function was long thought to be a major drawback, however[12] argued that it is not in many practical problems.

13 Evolutionary Strategy

Whereas RL algorithms like Q-learning and policy gradients explore by sampling actions from a stochastic policy, Evolution Strategies (ES) derives learning signal from sampling instantiations of policy parameters[28] (OpenAI).

Let F denote the objective function acting on parameters θ. ES algorithms represent the population with a distribution over parameters $P_\psi(\theta)$ – itself parameterized by ψ and proceed to maximize the average objective value $\eta(\psi) = \mathbb{E}_{\theta \sim P_\psi(\theta)} F(\theta)$ over the population by searching for ψ with stochastic gradient ascent. Similar to RL, NE takes gradient steps on ψ with estimator

$$\Delta_\psi \eta(\psi) = \mathbb{E}_{\theta \sim P_\psi(\theta)}[F(\theta) \Delta_\psi \log P_\psi(\theta)].$$

If F is the return function in RL, θ will be the parameter of a policy function π_θ. Let $P_\psi(\theta)$ be a multivariate Gaussian with mean ψ and fixed covariance $\sigma^2 I$, we can write η in terms of a mean parameter vector θ: $\eta(\theta) = \mathbb{E}_{\epsilon \sim N(0,I)} F(\theta + \sigma\epsilon)$, and optimize η with the gradient estimator

$$\Delta_\theta \eta(\theta) = \frac{1}{\sigma} \mathbb{E}_{\epsilon \sim N(0,I)}[F(\theta + \sigma\epsilon)\epsilon].$$

14 Preference Recommendation

Preference recommendation is made based upon the data collected using OPRA. OPRA provides a default ranking to users or voters, but they may change the ranking when the recommended rankings are not their true preferences. It will takes them amount of time to adjust from the recommended rankings to their favorite rankings before the submission.

Suppose the number of users involved into the voting is large enough, we could estimate the probability distribution of various rankings over the same candidates appeared in the poll. *Assuming that all voters reveal their real preferences during the poll*, we expect that the system could recommended each voter a ranking as close to his or her true preference as possible, such that the overall operating time before casting a vote could reduce drastically.

It is an optimization problem, searching an optimal ranking π^* that requires a minimum expected operating time:

$$\pi^* = \underset{\pi \in \Pi[m]}{\arg\min} \sum_{\sigma \in \Pi[m]} P(\sigma) L(\pi, \sigma; \theta),$$

where $P(\sigma)$ is the probability distribution of possible rankings, and it can be estimated based on the existing voting outcomes to date. If the number of voters is large enough, the estimation can be very accurate. $L(\pi, \sigma; \theta)$ is the cost (e.g. time) of voting for a voter, given her true preference is σ and the recommended preference ranking is π. The problem consists of two sub-problems, one is the estimation of the distribution of preference rankings $P(\sigma)$, another one is the estimation of voting cost.

The recommendation model is built upon two assumptions:

- all voters reveal their true preferences in voting
- the discrepancy between the true and the recommended rankings determines the voting cost; other factors, e.g. the characteristic differences of voters are ignored

The log data we can collect from OPRA including the time duration associated to different combinations of a truth preference ranking and the recommended preference ranking.

We will design some recommendation models and verify them using the collected data. If the method is proved to be effective, it will be evaluated with some paid experiments. The final product should be a well-designed web-interface, and we outsource the experiments using **Amazon Turk** and collect rich data to verify our conclusion.

Given N historical voting entries $D = \{\sigma_i, \pi_i, t_i\}_{i=1}^{N}$, with a triple represents the true preference ranking σ_i, the recommended initial rankings π_i, and the required voting time t_i for voter v_i. Also, we have trained a time predictor $T(\sigma, \pi; \theta^*)$ to fit current data set D which minimizes the voting time on average

$$\theta^* = \arg\min_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} L(T(\sigma_i, \pi_i; \theta), t_i).$$

We can train the predictor as a regression problem or a classification problem by discretising the continuous time. There is a brute force method with m small number of candidates for full rankings. It's to traverse over $m!$ rankings, compute the voting cost for each pair of initial and final rankings. Then, the initial ranking with minimum average voting cost for all possible final rankings are selected as the recommendation. However, it's very hard for large m, or partial rankings.

How about a special voting rule which aggregates those historical votes such that it satisfies some good properties?

15 Hyper-parameter Optimization

Deep neural network involves setting up the architecture of the network, including the depth of the network, choice of connectivity per layer (convolutional, fully connected, drop out et al), choice of optimization algorithms (SGD, Adam) and recursive choice of parameters (learning rate, momentum, eta).

Hyperparemeter optimization refers to automatically finding a good setting of these parameters. *Stochastic gradient descent* method is good enough to optimize the continuous parameters. It's more challenge to optimize the discrete parameters, and existing approaches are the *Bayesian optimization*, *multi-armed bandit* algorithm and *random search*.

15.1 Bayesian Optimization

Bayesian optimization approach assumes that there exists a prior distribution of the loss function, and keeps updating the prior distribution based on new observations. Each new observation is selected according to a query function, which balances exploration and exploitation such that either the new observation gives us better outcome or provides us with more information about the loss function. The most popular Bayesian optimization packages is Spearmint.

15.2 Multi-armed Bandit

Hyperband is a general version of the successive halving algorithm, an extension of the Multi-armed bandit algorithm on randomly selected configurations.

15.3 Random Search

15.4 Spectral Approach

The spectral approach is built upon harmonic analysis of Boolean functions, and tries to fit a *sparse polynomial function* to the discrete, high dimensional function which maps hyper-parameters to loss, and then optimize the resulting sparse polynomial.

First quasi-polynomial time algorithm for learning decision trees under the uniform distribution with polynomial sample complexity, the first improvement in over two decades.

References

1. Breiman, L. (1993). *Classification and regression trees*. CRC press.

2. Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., and Hullender, G. (2005). Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96. ACM.

3. Burka, D., Puppe, C., Szepesváry, L., and Tasnádi, A. (2016). And the winner is... chevalier de borda: Neural networks vote according to bordas rule.

4. Chaslot, G., Bakkes, S., Szita, I., and Spronck, P. (2008). Monte-carlo tree search: a new framework for game ai. In *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 216–217. AAAI Press.

5. Chechik, G., Shalit, U., Sharma, V., and Bengio, S. (2009). An online algorithm for large scale image similarity learning. In *Advances in Neural Information Processing Systems*, pages 306–314.

6. Clark, C. and Storkey, A. (2014). Teaching deep convolutional neural networks to play go. *arXiv preprint arXiv:1412.3409*.

7. EĞECIOĞLU, Ö. and Giritligil, A. E. (2013). The impartial, anonymous, and neutral culture model: A probability model for sampling public preference structures. *The Journal of Mathematical Sociology*, 37(4):203–222.

8. Golea, M., Bartlett, P., Mason, L., and Lee, W. S. (1998). Generalization in decision trees and dnf: Does size matter? *Advances in Neural Information Processing Systems*, pages 259–265.

9. Ittner, A. and Schlosser, M. (1996). Non-linear decision trees-ndt. In *ICML*, pages 252–257. Citeseer.

10. Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. (2016). Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*.

11. Kontschieder, P., Fiterau, M., Criminisi, A., and Rota Bulo, S. (2015). Deep neural decision forests. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1467–1475.

12. LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

13. Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

14. Liu, T.-Y. (2011). *Learning to rank for information retrieval*, volume 13. Springer.

15. Maddison, C. J., Huang, A., Sutskever, I., and Silver, D. (2014). Move evaluation in go using deep convolutional neural networks. *arXiv preprint arXiv:1412.6564*.

16. Marivate, V. and Littman, M. (2013). An ensemble of linearly combined reinforcement-learning agents. In *Proceedings of the 17th AAAI Conference on Late-Breaking Developments in the Field of Artificial Intelligence*, pages 77–79. AAAI Press.

17. Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937.

18. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

19. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

20. Murthy, S. K., Kasif, S., and Salzberg, S. (1994). A system for induction of oblique decision trees. *Journal of artificial intelligence research*.

21. Park, Y. and Sklansky, J. (1990). Automated design of linear tree classifiers. *Pattern Recognition*, 23(12):1393–1412.

22. Pemmaraju, S. and Skiena, S. S. (2003). *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica®*. Cambridge university press.

23. Procaccia, A. D., Zohar, A., Peleg, Y., and Rosenschein, J. S. (2007). Learning voting trees. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 22, page 110. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

24. Procaccia, A. D., Zohar, A., Peleg, Y., and Rosenschein, J. S. (2009). The learnability of voting rules. *Artificial Intelligence*, 173(12):1133–1149.

25. Qin, T., Liu, T.-Y., and Li, H. (2010). A general approximation framework for direct optimization of information retrieval measures. *Information retrieval*, 13(4):375–397.

26. Rodriguez, A. and Laio, A. (2014). Clustering by fast search and find of density peaks. *Science*, 344(6191):1492–1496.

27. Sakar, A. and Mammone, R. J. (1993). Growing and pruning neural tree networks. *IEEE Transactions on Computers*, 42(3):291–299.

28. Salimans, T., Ho, J., Chen, X., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

29. Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

30. Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395.

31. Sutton, R. S. and Barto, A. G. (1998). Reinforcement learning: An introduction. 1998. *A Bradford Book*.

32. Tenmoto, H., Kudo, M., and Shimbo, M. (1998). Piecewise linear classifiers with an appropriate number of hyperplanes. *Pattern Recognition*, 31(11):1627–1634.

33. van Hasselt, H. P., Guez, A., Hessel, M., Mnih, V., and Silver, D. (2016). Learning values across many orders of magnitude. In *Advances in Neural Information Processing Systems*, pages 4287–4295.

34. Walsh, T. (2015). Generating single peaked votes. *CoRR*, *abs/1503.02766*.

35. Xia, L. (2013). Designing social choice mechanisms using machine learning. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 471–474. International Foundation for Autonomous Agents and Multiagent Systems.

36. Xia, L. and Conitzer, V. (2008). Generalized scoring rules and the frequency of coalitional manipulability. In *Proceedings of the 9th ACM conference on Electronic commerce*, pages 109–118. ACM.

37. You, K. and Fu, K.-S. (1976). An approach to the design of a linear binary tree classifier. In *LARS Symposia*, page 132.