Cheating the System: Finding a Positional Voting System such that your favored candidate will *always* win

By Danny Liu

Harvey Mudd College

Math 189G – Mathematics of Voting

2016

Motivation:

The motivation of this study was inspired by Prof. Orrison's first lecture in the Mathematics of Voting class. Prof. Orrison began the class by asking the students to find a voting system such their designated candidate would always win in the election. The students were given preference schedules in the form of a number followed by what looked like a string of letters. An example of this is "4 ABC," and what this would mean is that four people ranked A > B > C as their preferred outcome in the general election. Within a few minutes, it was evident that most of the students in the class had decided on a positional voting system parameterized by a weighting vector w. However, it took way too long to figure out not only if there existed a weighting vector w such that a specified candidate would win but also what that vector would be if it existed. Hence, I set out to write an algorithm to determine what the weighting vector would be if it existed.

Goals:

At first, the goal of this project was to find just any weighting vector such that a specified candidate would win in the election. However, this poses the problem of what to do when there is more than one such vector and also what if no such vector existed for

your specified candidate to win. The latter problem was easily fixed since returning the 0 vector guarantees a tie and hence everyone wins! The former problem posed a more significant challenge but was still easily overcome through the idea of minimization. The idea is now as follows: since the Borda count is quite popular among voting theorists, I thought it would be best if our algorithm thus chose the weighting vector that minimized the distance between its equivalent Borda count vector.

The goal of this project is thus to write the following black-box algorithm.

Input:

1) A list of n candidates. Ex: [A, B, C, …]
2) A dictionary that maps the preference schedule as a string to how many voters actually input that schedule. Ex: {'ABC': 4, 'BCA': 1}
3) Candidate X, who you want to win in the election

Output:

The weighting vector $w \in R^n$ that not only allows candidate X to win under a positional voting system but also minimizes $\| w - B \|^2$, where B is the equivalent Borda count vector whose first and nth dimensions are the same as that of w's.

Method:

The main challenge of this algorithm if first to find any weighting vector that allows a specified candidate to win. The derivation of the technique is perhaps best illustrated by an example.

Example:

Suppose this is society's preference schedule: {'ABC': 4, 'BCA': 1}.

To compute how many points a candidate would receive under an arbitrary weighting vector, we first need to count up how many nth place votes each candidate receives. That is, how many first place votes did candidate A receive? How many second place votes? How many third place votes and so on. This is computed easily using Python and we will denote the vector computed for a particular candidate that candidate's *vote-count.* The vote-counts for each candidate in this example is {'A': [4, 0, 1], 'B': [1, 4, 0], 'C': [0, 1, 4]}, where [4, 0, 1] means that A received four first place votes, zero second place votes, and one third place vote. Given the vote count for a particular candidate and the weighting vector w, we can compute the number of points – or score – a candidate receives as the dot product of the vote-count for that particular candidate and w. Hence the score a candidate receives is given by $w \cdot v_n$ where v_n is the vote-count vector for candidate n.

If in this example we wanted to set B as the winner, the following two equations must hold true.

$$w \cdot v_b \geq w \cdot v_a$$

$$w \cdot v_b \geq w \cdot v_c$$

Rearranging both equations, we get that

$$w \cdot (v_b - v_a) \geq 0$$

$$w \cdot (v_b - v_c) \geq 0$$

This computation can be modeled by the matrix equation Ax >= b where b is the 0 vector, x is the weighting vector w, and the linear transformation matrix A is made up of the transpose of $v_b - v_a$ and $v_b - v_c$ as its rows. In this example, recall that v_a=[4, 0, 1], v_b=[1, 4, 0] and v_c=[0, 1, 4], so the matrix equation would be

$$\begin{bmatrix} -3 & 4 & -1 \\ 1 & 3 & -4 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Hence, given the preference schedule {'ABC': 4, 'BCA': 1}, any positional voting system that uses a weighting vector which satisfies the above equation will choose candidate B as its winner.

With the matrix equation out of the way, what's left now is to choose the weighting vector that minimizes the distance between itself and its equivalent Borda count vector. To compute this equivalent Borda count vector B, we set the first and nth dimensions of B the same as that of w. Since points in the Borda count should increase linearly, we compute the average distance, which is sum of the first and nth elements of w divided by the number of dimensions in w. We then set the n-ith element of B as $w_0 - i * average\_distance$. Hence, if $w \in R^5$ and w_0 is 2 and w_4 is -2, then B would be [2, 1, 0, -1, -2].

Solving this entire example now boils down to the following optimization problem.

Minimize $\| w - B \|^2$ subject to the following constraints:

1. $\begin{bmatrix} -3 & 4 & -1 \\ 1 & 3 & -4 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
2. $w_0 \geq w_1 \geq w_2$

To make the results a little cleaner, the additional constraint of $w_o = n - 1$ was also set, so that if there are n candidates, our resulting weighting vector will always reward n-1 points to the candidate ranked first by any voter.

Fortunately, there are pre-implemented algorithms for us to do such optimization problems. Since $\| w - B \|^2$ is non-linear, I used the Sequential Least Squares

Programing algorithm, or SLSQP for short. The SLSQP implementation in Python's scipy module takes in an optimization objective to minimize and a list of constraints. The optimization objective is the squared distance $\| w - B \|^2$ and the list of constraints was generated using the matrix A. For details of my implementation, please visit github.com/dannykliu/Mathematics-of-Voting-Cheating-the-System.

Results:

After implementing the entire black box algorithm in Python, I ran a number of tests to see if the output was close to what I expected. Note that since I do not know the actual correct weighting vector for any sample preference schedule given to the algorithm, I decided to give my algorithm the benefit of the doubt and to assume it was correct until proven otherwise.

Given the following inputs where the first input is the list of candidates, the second input is the preference schedule, and the third input is the specified winner, the returned weighting vector is on the right of the arrow → and the new scores of each candidate is listed below the input

(['A', 'B', 'C'], {'BCA': 4, 'CBA': 10, 'ABC': 1}, 'B') → [2.0, 1.77, 0.31]

New score: {'A': 6.34, 'B': 27.47, 'C': 27.39}. We see that B wins

(['A', 'B', 'C'], {'ABC': 4, 'BCA': 1}, 'B') → [2.0, 1.61, 0.32]

New score: {'A': 8.32, 'B': 8.44, 'C': 2.89}. We see that B wins yet again

(['A', 'B', 'C'], {'ABC': 4, 'BAC': 1}, 'B') → [2.0, 2.0, 0.31]

New score: {'A': 10.0, 'B': 10.0, 'C': 1.55}. B is not the sole winner, but in fact if you analyze the data closely, you'll notice that B's score is bounded above by A's score.

(['A', 'B', 'C'], {'ABC': 4, 'BAC': 1}, 'C') → [2.0, 2.0, 2.0]

What happened here? Well we tried to get a candidate who was ranked last by *every single* voter to win the election. Given the constraint that the individual weights of the weighting vector must not be decreasing, the only possible solution was thus to have all the weights be the same. The reason the algorithm returns a vector of twos and not zeroes is because of the constraint I added to make the data cleaner that $w_0 = n-1 = 3-1 = 2$.

Conclusion:

In conclusion, we have found that given any number of preference schedules, it is possible to write an algorithm to assure that your specified candidate will win or at least tie for first. To do so involves solving a non-linear optimization problem subject to a few constraints using the SLSQP algorithm. So next time an election is held with ranked preference and you happen to be on the voting committee deciding on which voting system to use, do give github.com/dannykliu/Mathematics-of-Voting-Cheating-the-System a look. The only dependency you will need is the Python Scipy module, which can either be downloaded through the command line using pip or comes pre-installed with Anaconda!