Electoral Engineering through Simulation

October 19, 2012

Author:
Michael Fowlie

Supervisor:
Mark C. Wilson

COMPSCI 380 - Undergraduate Project in Computer Science
Semester 2, 2012
University of Auckland

Contents

1 Introduction

Election outcomes have two properties that we are interested in, fairness/proportionality and stability/governability. Under different voting rules (and different parameters for these rules), election outcomes have different tendencies for values for these properties. Both of these properties are desirable however it is well understood that there is a trade-off between the two.

(Carey and Hix, 2011) uses data from 610 elections in 81 countries. It categorised voting rules based on District Magnitude (See Section 5). We investigate this trade-off in the context of artificially generated societies and in particular look for non linearity.

In particular we use the STV (See Section 6.1) and DP (See Section 6.2.1) methods and compare results by varying the District Magnitude (See Section 5).

It is claimed that the trade-off between these two properties is non linear (Carey and Hix, 2011). As District Magnitude increases, usually fairness increases at the expense of stability; however as we demonstrate this is not always the case. In particular in some cases the opposite effect occurs.

If there is non linearity in the trade-off, it follows that there must be an arguably ideal point though there is no clear definition of a desirability function as some may value one property more so than the other.

2 Methodology

2.1 Hypothesis

Our hypothesis H_1: There is a non-linear relationship between mean scores for fairness and stability under the different voting systems discussed in Section 6 and various parameters for these voting systems. So that our results to be comparable with (Carey and Hix, 2011), we use the Gallagher Index (See Section 3.1) as a measure of fairness and the ENP Index (See Section 3.2.1) as a measure of stability. We also consider the case using the stability index discussed in Section 3.2.2.

2.2 Procedure

2.2.1 Generating Data

To control for the different artificial societies and different voting systems, we consider these in separate runs of the simulator.

In each run of the simulator, we generate k elections for each value of the property that we vary. This is usually District Magnitude (See Section 5). Each election is generated with a fixed number of seats and so as the District Magnitude changes the number of Districts also changes. This also forces us to only consider District Magnitudes that are an integer divisor of the number of seats. We consider the case with 120 seats.

Each election is generated independently, for example with the Spatial Model we define points for each party and each electorate in the Space of Issues in each election.

2.2.2 Statistical Analysis

We fit a regression curve to the measure of fairness and a regression curve to the measure of stability, using DM (see Section 5) as the explanatory variable.

See Section 8 for technical notes.

2.3 Tools

The simulator itself consists of a library written in C# 5.0 along with scripting code written in C# and F# 2.0. Simulations are performed on Amazon EC2 c1.xlarge instances with Intel ®Xeon ®E5410 processors and a workstation with an Intel ®Pentium ®G620.

Data from (Vowles et al., 2008) was imported and processed with scripts written in both SAS and C#.

The resulting dataset is then analysed using R 2.15.1.

3 Measures

Fairness and Stability are measured in a variety of ways, described below.

We actually measure the *opposite* of the desirable properties, that is, we measure unfairness and instability.

Note that in our plots we normalize measures so that the point (0,0) is optimal. For the measures 3.2.1 and 3.2.2 this involves subtracting 1 from the values.

3.1 Measures of Unfairness

We define $\delta_i = v_i - s_i$ where v_i is the proportion of votes of party i and s_i is the proportion of seats of party i.

We use the Loosemorehanby Index $D = \frac{1}{2}\sum_{i=1}^{m} |\delta_i|$

and the Gallagher Index $D = \sqrt{\frac{1}{2}\sum_{i=1}^{m} \delta_i^2}$. (Benoit, 2000)

3.2 Measures of Instability

3.2.1 Effective Number of Parties

As a measure of ungovernability, we use the Effective Number of Parties. $ENP = \frac{1}{sum_{i=1}^{m} p_i^2}$ (Laakso and Taagepera, 1979) where p_i is the portion of seats that party i has.

3.2.2 ENP using Shapley Shubik Power Index

As for the purposes of governability, only power (Shapley and Shubik, 1954) is relevant (as opposed to the portion of seats), it seems reasonable to modify the ENP formula to use political power instead of the proportion of seats, by party. $ENP = \frac{1}{sum_{i=1}^{m} SSPI_i^2}$

For example a parliament with a party with 0.51 of the seats has the same amount of governability as one with a party with 0.99 of the seats. In both of these cases the party in question has all of the power as defined by the Shapley Shubik power index, however the ENP formula distinguishes between these two cases.

However this measure has, in some cases, the interesting effect of measuring an increase of stability when reducing the threshold under MMP. Consider the following election (See Figure 1) where v_i denotes the number of votes of party i, p_i denotes the Shapley Shubik power of party i with a 10% threshold, and q_i denotes the Shapley Shubik power of party i with a zero threshold. Note that a *lower* value for ENP is considered more stable.

3.2.3 Entropy of Shapley Shubik Power

We also consider the entropy of the Shapley Shubik index. $\sum_{i=1}^{m} SSPI_i \cdot \ln(SSPI_i); SSPI_i \neq 0$

4 Artificial Societies

In each society we define m parties and $M = m!$ preference orders, where a preference order is an order over all parties. We define n as the total number of voters in a given district.

4.1 Polya Eggenberger (Urn) Model

In each district we sample n preference orders, each representing a single voter, from the Polya Eggenberger distribution with parameter a where a represents the homogeneity of preferences in a given district. We assume districts are independent. As we assume independence of districts and there is no bias in favour of any particular party, as the number of districts approaches ∞ the distribution of votes to parties will approximate a uniform distribution - thus this model is not very realistic.

4.1.1 Urn Process

The Polya Eggenberger distribution can be understood via intuition from the Urn Process. The Urn contains multiple balls, each of a different colour (where a colour represents a preference order). To sample from the Urn we select a ball from the Urn uniformly at random, record it, and place the ball and a copies of the ball back into the Urn. We define the weight of a preference order (colour of ball) to be equivalent to the number of balls in the Urn, and the weight of the Urn to be equal to the total weight of all colours of balls in the Urn. However, it is important to note that a (and thus weight) need not be an integer.

4.1.2 Parameter selection

We use both the case with a constant a for all districts, and the case for a random a, where we first define $b = U(0, 1)$ and then let $a = b/(1 - b)$

Votes	p_i	q_i
47	0.3333	0.6667
26	0.3333	0.1667
26	0.3333	0.1667
01	0.0000	0.0000
$sum_{i=1}^{m} SSPI_i^2$	0.3333	0.5000
$\frac{1}{sum_{i=1}^{m} SSPI_i^2}$	3	2

Figure 1: Under some elections, increasing the number of parties increases stability as measured as described in section 3.2.2

4.1.3 Sampling

A naive approach to sampling a value x from the Polya Eggenberger distribution would be to store the weights of each preference order in a vector w then select a value uniformly at random. $y = U(0, \sum_{i=1}^{M} w_i)$ then select the minimum value of x such that $\sum_{i=1}^{x}(w_i) \geq y; x \leq M$. We would then adjust the weight for the selected preference order w_x by adding a.

This has time complexity of $O(M)$ per sample, or $O(n \cdot M)$ to sample a single district, and space complexity $O(M)$. Note that $M = m!$ can be quite large. We use a more efficient method to sample, by replacing x with a compressed data structure (See Section 9) with worst case space complexity $O(min(n \cdot \ln(M), M))$ and worst case time complexity $O(n \cdot \ln M)$ (per district) which is better than $O(n \cdot m \ln m)$

4.2 Spatial Model

Our model has a parameter z which is a vector representing the standard deviation in each dimension in the space of issues. Where $z_j \geq z_{j+1}$

We have a parameter d which represents the standard deviation of population means of districts as a portion of the standard deviation of voters.

We define a $|z|$ dimensional space called the space of issues. Each voter and each party are represented by a point in this space. Voters prefer parties with a lower distance between a voters point and a given party's point. In each party i we select $p_{i,j} = N(0, z_j)$ where p_i represents the point in the space of issues of party i. We also consider the case in which two of the parties are instead represent the traditional major left and right parties. Population means for voters in a district k is selected $\mu_{k,j} = N(0, z_j \cdot d)$ We assume districts are independent. In each district k, a voter n is selected $v_{n,j} = N(\mu_{k,j}, z_j)$

Parameter selection

An obvious class of parameter values to select are all non-zero variances set to be equal, with 1, 2 or 3 non-zero dimensions. In particular two dimensions is interesting as the dimensions can refer to economic liberty and social liberty.

We fit parameters to this model, obtained from real world data, via the method discussed in Section 10.

4.3 Preference Swapping Model

The Preference Swapping Model takes a set of real world elections as a parameter and creates a cluster of similar elections around each real world election and extends upon the 2011 referendum simulator (Pritchard and Wilson, 2011). The model works by evolving an input election by first inferring the preference orders for each vote and switching preferences with some probability. The probability of switching between 1st place and 2nd place may be different from switching between 1st place and 3rd place, for example. As we are only concerned with 1st place preferences, we do not consider the case where another pair of preferences also switches. We also use the case where we apply the output of our model as the input to another stage in our model (i.e. so that we model 2 or more steps into the future). Each stage is calculated in the same way with the exception that we only infer preference orders from the first preferences in the first step.

We define s_i to be the probability that a voter switches a it's 1st and ith preference, where s_i is selected uniformly at random between 0 and $max(s_i)$ where $max(s_i)$ is a parameter to our model.

It doesn't make much sense for $s_i < s_{i+1}$ however even with $max(s_i) > max(s_{i+1})$ (as we would expect) our model can generate values of s_i and s_{i+1} such that $s_i < s_{i+1}$. We consider both the case where we apply no additional restrictions on the model and the case where $s_i \geq s_{i+1}$

Parameter selection

We use the 2002, 2005, 2008 and 2011 elections in New Zealand as starting points and infer preference orders and the probability of switching preference orders from the NZES 2008 (Vowles et al., 2008).

We infer preference orders from the survey by assuming a voter's first preference is the party voted for and that the remaining parties are ordered in descending order based on the scores that voters were asked to rate parties by.

In the case where voters gave the same score to more than one party, we consider all possible orders of parties with the same score with the weight for each score reduced accordingly.

We normalise the weights for all preference orders where voters voted for a given party to 1, then treat the normalised weights as a probability that a voter for a given party has a particular preference order.

We assume that given a voter voted for a particular party, the probability it has a particular preference order is independent of it's district and from election to election.

We select s_i to be the proportion of voters who's vote in 2005 (the previous election) was equal to their ith current preference.

5 District Magnitude

District Magnitude represents the number of seats in a particular district. (Carey and Hix, 2011) uses the median district magnitude to group collections of voting rules and their parameters for analysis. In some systems (e.g. SM, see Section 6.2.2) the district magnitude varies from district to district. Using the median would group SM in the same category as FPP which is undesirable.

6 Voting Rules

6.1 Single Transferable Vote

In this system we take into account the complete preference orders of voters. We define a number of equal sized districts and define a quota, such that $quota = (\frac{n}{DistrictMagnitude+1}) + 1$. (Where n is the number of voters in a given district)

All parties need to obtain *quota* votes to obtain a seat. The process of allocating votes to parties, in each district, works as follows:

- 1. Assign a weight of 1.0 to all votes.
- 2. Calculate the sum of weights, by party, for all votes. Select the party to assign the weight to by the highest non-excluded and non-elected party in a voter's preference order.

- 3. Each party that obtains a weight greater than or equal to the quota is elected. We reduce the weights of all votes that voted for a given elected party by multiplying each by $m = \frac{Weight for votes for this party - quota}{Weight for votes for this party}$, thus there is *quota* weight removed.
- 4. If no party was elected in Step 3, we exclude the party that obtained the least weight in Step 2.
- 5. We loop back to Step 2. until enough parties are elected.

6.2 Proportional Systems

We use the following two voting systems which are special cases of a more general system which is proportional in each district. At one extreme of both of these systems is FPP (First Past the Post), and the other extreme, pure proportional. In each district under these systems, we apply some apportionment method (see Section 7) to select the number of seats by party, and aggregate the results. These systems only take into account first preference votes.

6.2.1 District Proportional

In this special case, we define multiple districts of equal size. Each voter only votes under one district.

6.2.2 Supplementary Member

In this special case, we define a number of districts of size 1 and a single large district of some larger size. Voters are assumed to each vote in one of the smaller districts and to all also vote in the larger district. We assume voters vote the same way for both the small district and the large district.

7 Apportionment Methods

7.1 St Lague

7.2 Hill

7.3 Jefferson

7.4 Hamilton

8 Technical Notes on Statistical Analysis

9 Efficiently Sampling from a Discrete Probability Distribution

We define w_i to be the weight of element i in the probability distribution, where $Pr(X = i) = \frac{w_i}{\sum w}$ We take advantage of the fact that most elements in the probability distribution have a weight of 1. However this structure does not actually store w directly.

The structure is similar to a binary search tree, however instead of containing elements it contains *ranges* where leaf nodes (i.e. nodes with a range length of 1) represent the elements of w. Every element in the tree contains a precomputed parameter representing the weight of the element, where we define the weight of a non-leaf node (that is, a node that doesn't represent a single element) to be the sum of the weights of it's left and right child. Much of the tree is not ever written to or read from, and so we can lazily load nodes on demand. This lazy loading saves us a significant amount of space for a large $|w|$.

When sampling from the distribution, we select a value y uniformly at random $y = U(0, \sum w)$ and call a recursive search function on the root node, as defined by the below psuedocode:

```
Node RecursiveSearch(Node n, double y)
{
        if (n.length == 1)
                return n;

        double skipAmount = 0.0;

        if (getLeftChild(n) != null)
        {
                if (getLeftChild(n).weight>=y)
                {
                        return RecursiveSearch(getLeftChild(), y);
                }
                else
                {
                        skipAmount = getLeftChild(n).weight;
                }
        }

        return RecursiveSearch(getRightChild(n), y - skipAmount);
}
```

Once the node in question is found, we return the sampled value from the distribution and update the weight of the node by the requested amount. When increasing the weight of an element, we need to increase the weight of the parent node (and the parent's parent, up to the root etc.) which takes in the worst case $\ln |w|$ worst case time.

10 Fitting real world data to the Spatial Model

It is of interest to calculate parameters for the Spatial Model (See Section 4.2) that are reasonably like the real world. To do so we attempt to fit real world data to the spatial model.

We assume that there is a relationship between the parameter z and the proportion of variance explained by each principal component of the results per party by district.

We attempt to validate this hypothesis by simulating a large number of elections under our model, several for each of many different values of z and compare it to the values obtained when

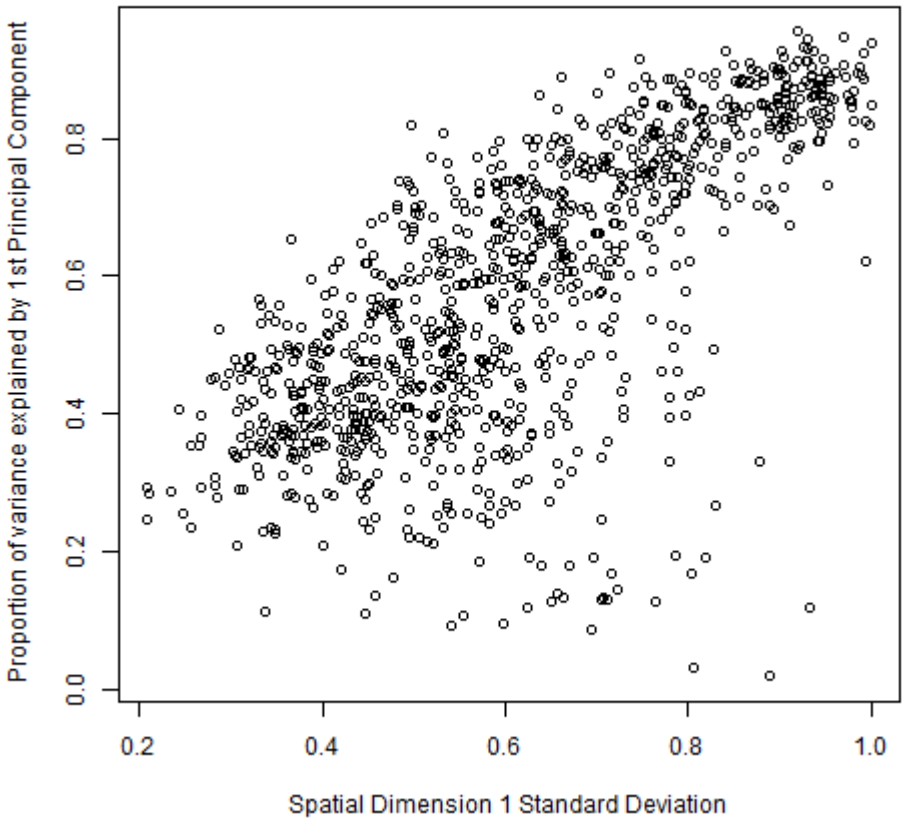

Figure 2: Values of z_0 vs proportion of variance explained by the first principal component

applying the formula to the variance from principal components of the election results. See Figure 2.

We fitted a regression line and found extremely strong evidence $p - value < 2 \times 10^{-}16$ against H_0 that there was no relationship between the two variables. The fit had high error $R^2 = 0.49$ however this error can be mitigated by taking a large number of samples from artificial societies for a given z value and comparing distances to a reasonably large set of real world elections.

We select z as to minimise the error of the model in describing a set of real world elections. We recursively select smaller and smaller ranges that we predict contains the target value of z to minimise the error as follows:

We define $y = \sum_i {z_i}^2$

- 1. Generate several values uniformly at random within the range of y values that we believe the target y to be in.

- 2. Find a value of z with y close to our target and update the values of y with the actual values from our values of z
- 3. For every value of z we simulate a number of elections and compute the mean error.
- 4. Select all the results (by y value) that have the minimum mean absolute error, and discard the rest.
- 5. Calculate what the underlying distribution of y values that the y values that generated our current results is (assuming a Normal distribution)
- 6. Calculate new upper and lower bounds for the target y value. If these bounds are within our target error margin we select mean, otherwise we loop with these new bounds.
- 7. Select a z value from a number of generated candidates with the target y that minimises the error.

11 Conclusion

The outcomes from a series of simulation runs are plotted below. In the case of each run, four plots are provided. Points in the top left plot represent individual simulations in the given run. Points in the top right plot represent means predicted, in accordance with our regression model, for a given District Magnitude. The bottom left plot may be slightly misleading as points plotted here are not necessarily near results from actual simulations. Each point represents a single District Magnitude and the position along each axis refers to the 95th percentile value for simulations using this District Magnitude. The purpose of this plot is to demonstrate the worst expected values for each of the two properties plotted. Points in the bottom right plot represent means, from actual results, per District Magnitude.

As can be seen from the below plots, we found curvature in our results, however the shape of the curvature varies between models and between voting systems.

Of particular interest is the shape of the curvature in Figure 4 for very high district magnitude.

Although fairness tends to increase at the expense of stability with an increase in District Magnitude, we found that extremely high values of District Magnitude under STV have consistently worse results than lower values of District Magnitude. In our simulations this applies to $DM > 30$. This applies to both the Spatial Model and the Polya Eggenberger model.

Interestingly with the District Proportional rule, we found almost the opposite effect. Likewise increasing District Magnitude tends to trade-off an increase in fairness for a decrease in stability, but extremely high values of District Magnitude tends to have higher stability than slightly lower values with gains in fairness.

We find that the choice of measurements has a non trivial effect on our results. For example in Figure. 4, it is debatable whether Green (DM 2-10) or Blue (DM>10) is optimal; as around $DM = 4 - 6$ the trade-off is very close to linear, but at the extremes, clearly the Blue values $DM = 10 - 29$ are optimal in the case that we favour fairness over stability and very small values $DM = 2 - 3$ are optimal if we favour stability over fairness.

By plotting the same simulation using ENP of the Shapley Shubik Power Index (See Section 3.2.2) (Figure 5) we find an even stronger effect that we observed on each side of $DM = 4 - 6$. In particular $DM = 3$ seems to do very well and $DM = 10$ has much better 95th percentile performance than $DM = 8$, in both fairness and stability.

The Loosemorehanby Index doesn't seem to have much of an effect on our results. (See Figure 4 vs. Figure 6)

Clearly a District Magnitude of 1 (Red) tends to do very poorly. We agree with (Carey and Hix, 2011) that low magnitude systems tend to do better if we value each property equally, however slightly increasing or decreasing District Magnitude from this range has great effects.

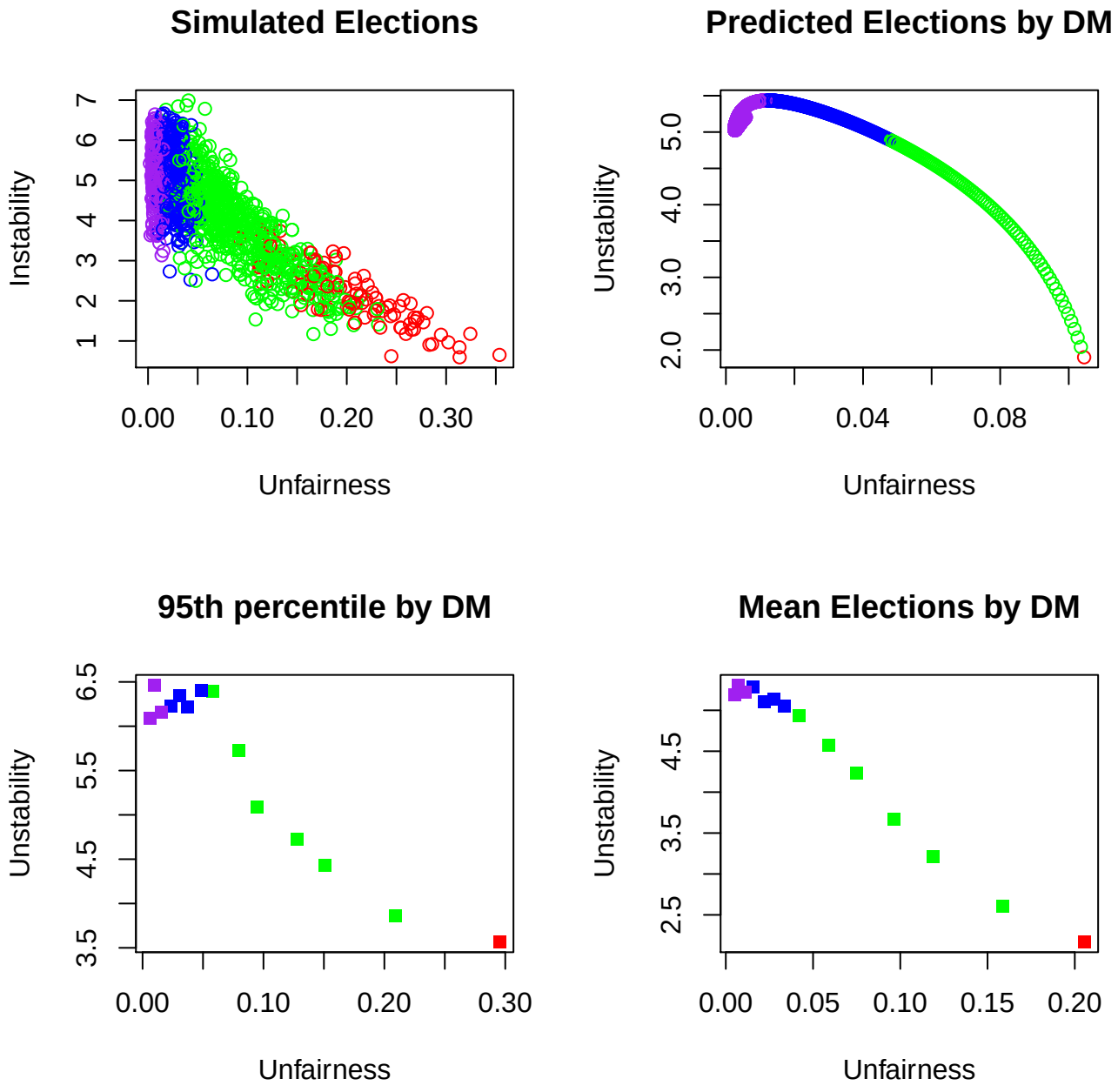

Figure 3: Results from simulated elections with the Spatial Model and District Proportional rule. Plotted using the Gallagher Index vs. ENP. DM=1 (Red) DM=2-9 (Green) DM=10-29 (Blue) DM>30 (Purple)

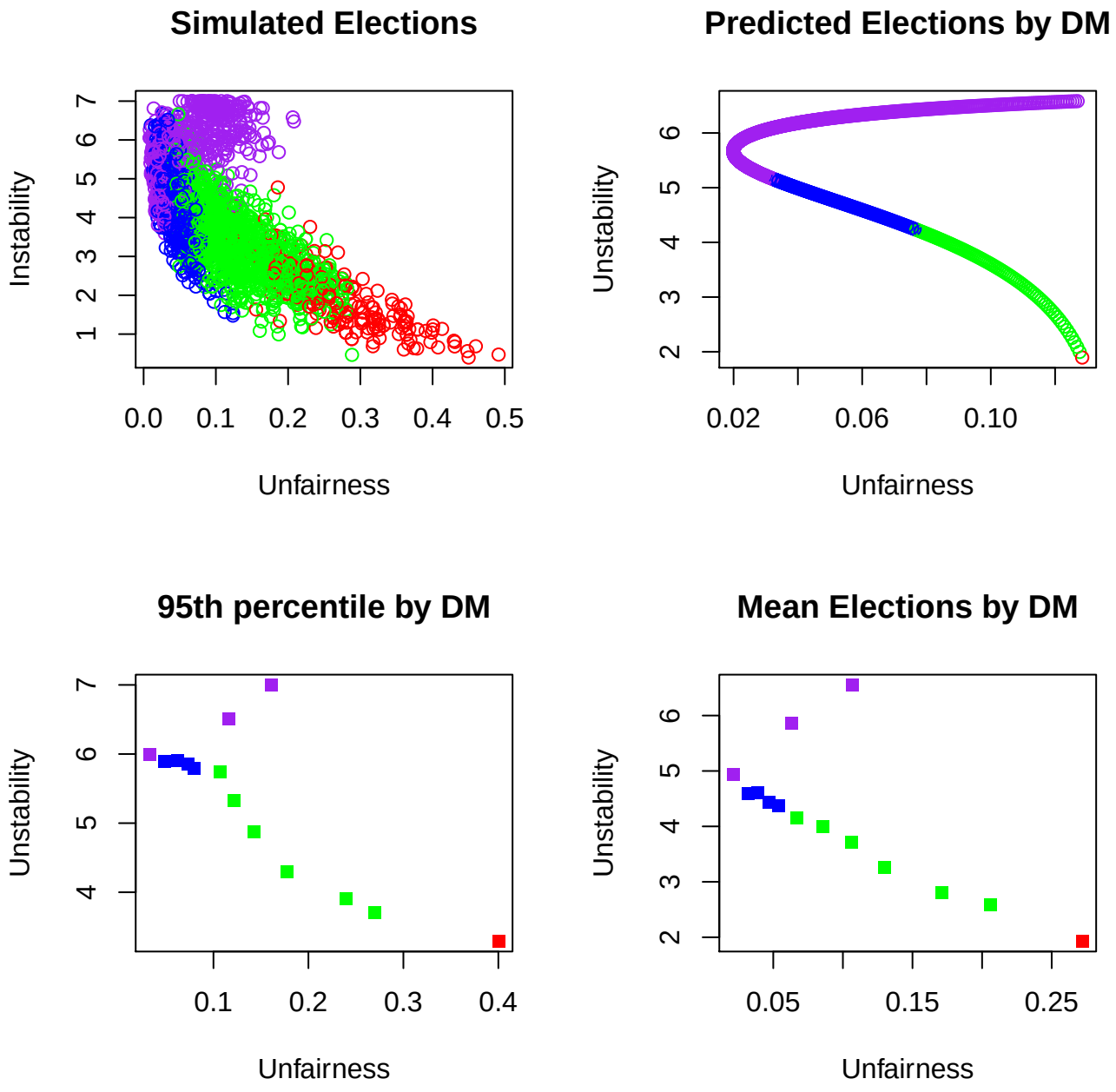

Figure 4: Results from simulated elections with the Spatial Model and STV rule. Plotted using the Gallagher Index vs. ENP. DM=1 (Red) DM=2-9 (Green) DM=10-29 (Blue) DM>30 (Purple)

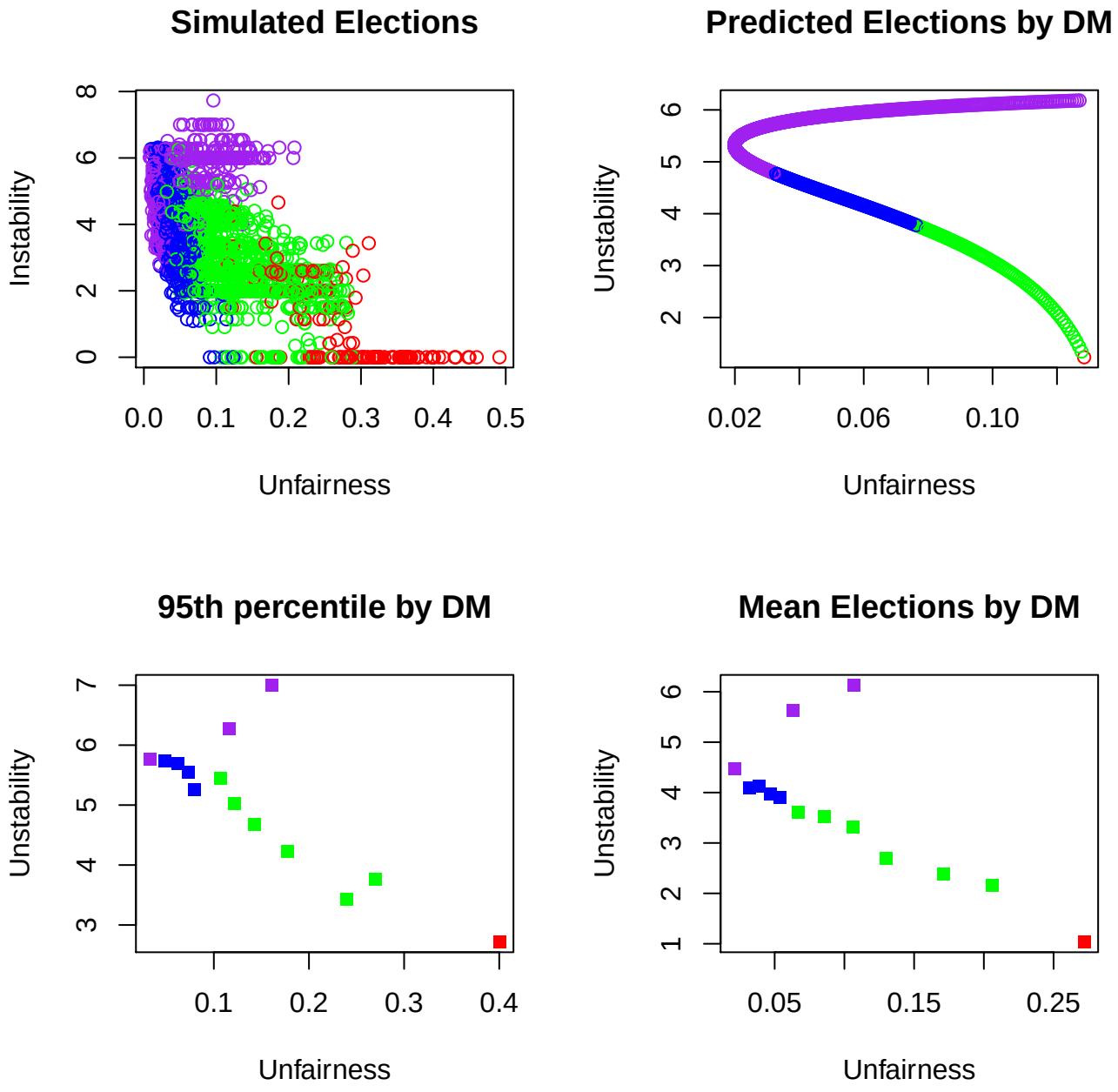

Figure 5: Results from simulated elections with the Spatial Model and STV rule. Plotted using the Gallagher Index vs. ENP using SSPI. DM=1 (Red) DM=2-9 (Green) DM=10-29 (Blue) DM>30 (Purple)

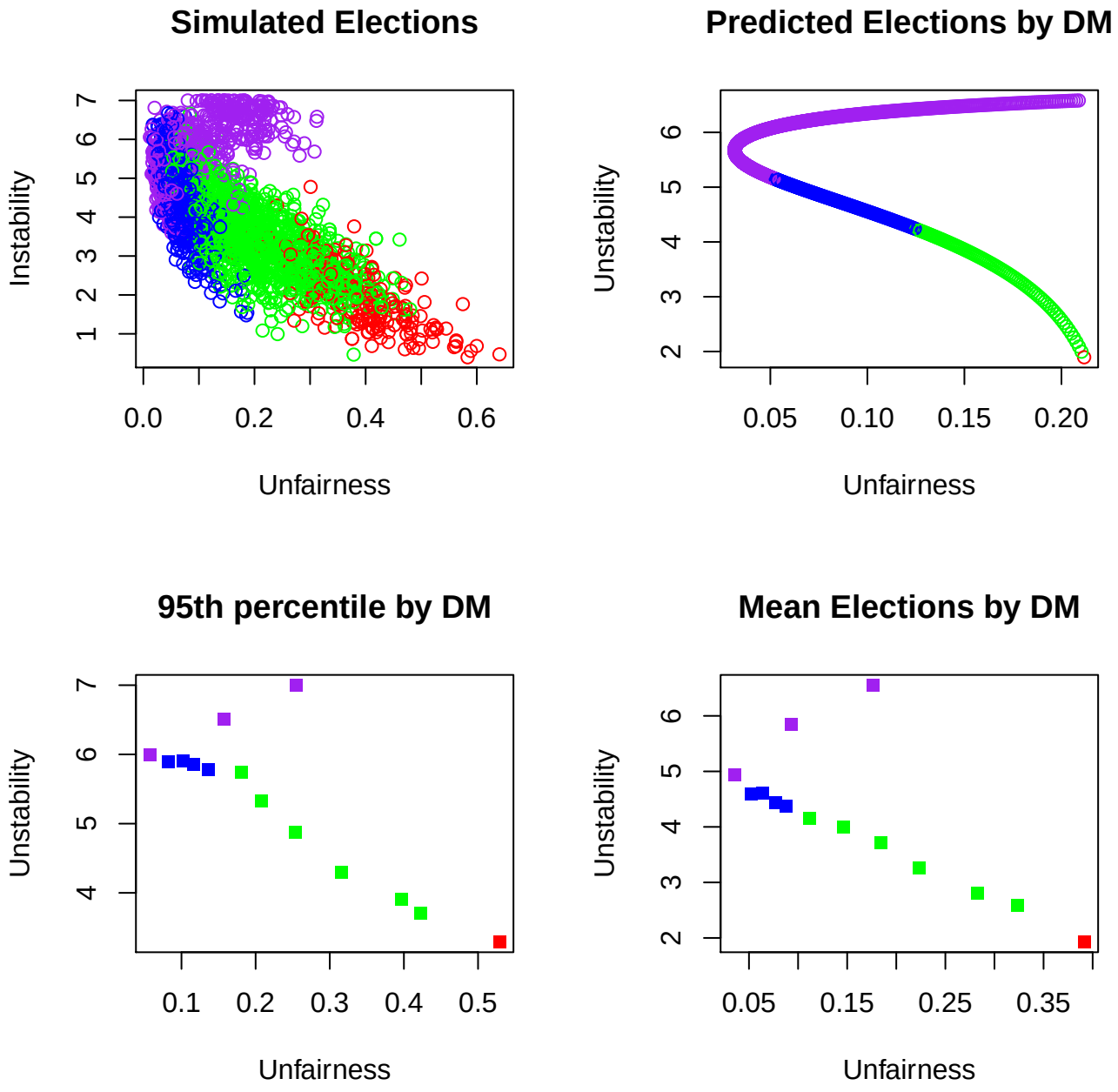

Figure 6: Results from simulated elections with the Spatial Model and STV rule. Plotted using the Loosemorehanby Index vs. ENP. DM=1 (Red) DM=2-9 (Green) DM=10-29 (Blue) DM>30 (Purple)

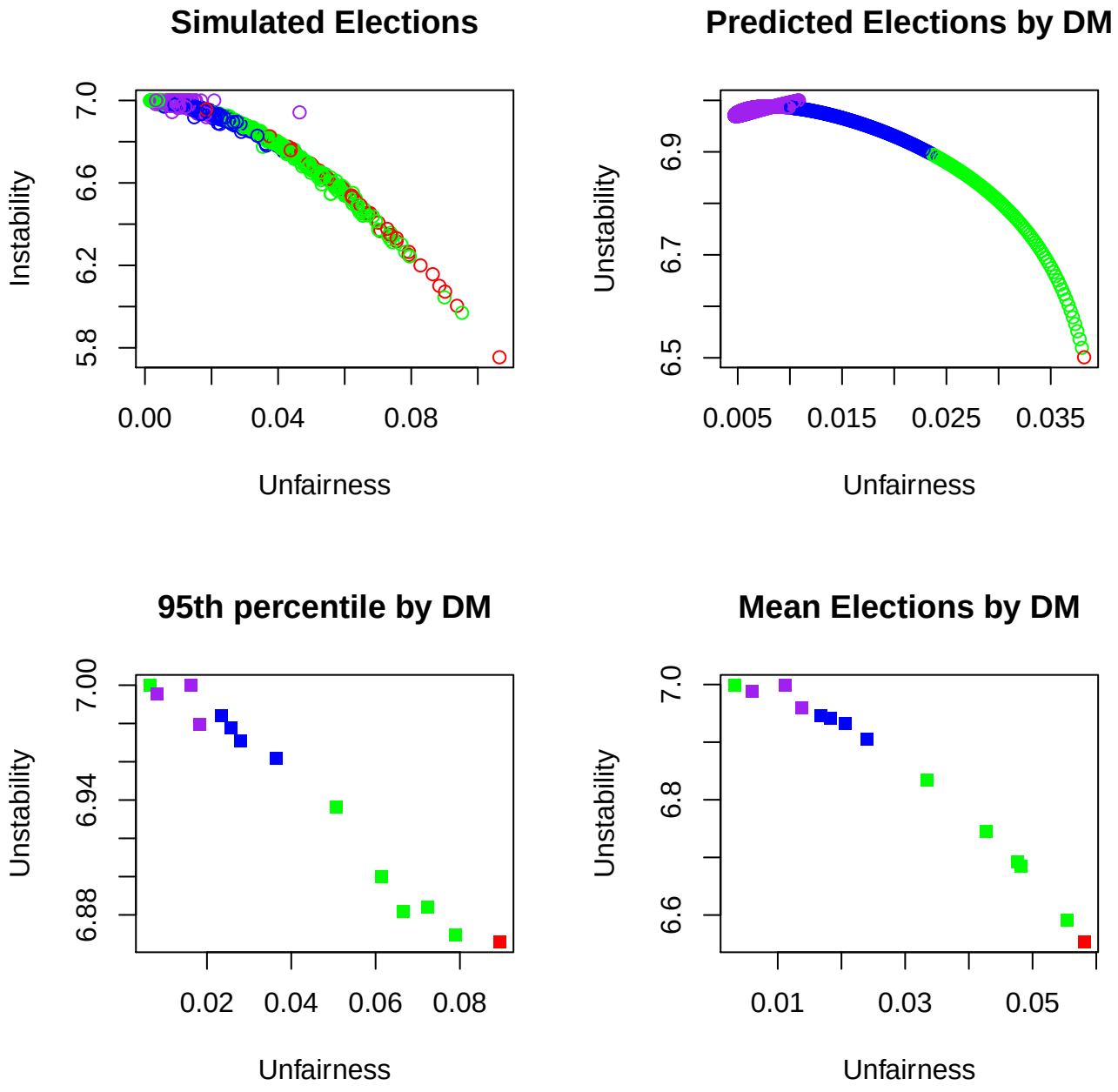

Figure 7: Results from simulated elections with the Polya Eggenberger Model and STV rule. Plotted using the Gallagher Index vs. ENP. DM=1 (Red) DM=2-9 (Green) DM=10-29 (Blue) DM>30 (Purple)

References

Benoit, K. (2000). Which electoral formula is the most proportional? a new look with new evidence.

Carey, J. and Hix, S. (2011). The electoral sweet spot: Low-magnitude proportional electoral systems. *American Journal of Political Science*.

Laakso, M. and Taagepera, R. (1979). Effective number of parties: A measure with application to west europe. *Comparative Political Studies*.

Pritchard, D. G. and Wilson, D. M. C. (2011). 2011 referendum simulator. http://www.stat.auckland.ac.nz/ geoff/voting/.

Shapley, L. and Shubik, M. (1954). A method for evaluating the distribution of power in a committee system. *American Political Science Review*.

Vowles, J., Miller, R., Banducci, S., Sullivan, A., Karp, J., and Curtin, J. (2008). 2008 election study. http://www.nzes.org/exec/show/2008.

A FSharp Sourcecode

```
open CS380
open CS380.VotingRules

open System
open System.Threading.Tasks;

let constrain (x:double, lower:double, upper:double) =
    Math.Min(Math.Max(x, lower), upper);

let r() =
    Rand.NextDouble();

// Hypothesis testing of ENI (Effective Number of Issues)

let numberOfParties = 8
let electorateCount = 120
let voteCount = 5000
let simCount = 2000

// We set the limits on ENI to between 1 and numberOfParties-1
let sampleENI() =
    1.0 + (r() * (double)(numberOfParties - 2))

let sampleRandVector() =
    let seq1 = Seq.toList(seq { for i in 1 .. numberOfParties do yield
        r() })
    seq { for x in seq1 do yield x }

let sampleUnitVector() =
    let seq1 = sampleRandVector()
    let sum = Seq.sum(seq1)
    seq { for x in seq1 do yield x / sum }

let sumOfSquares s =
    Seq.sum(seq { for x in s do yield x * x });

let calcEni s =
    1.0 / sumOfSquares(s)

let calcEntropy s =
    Seq.sum(seq { for x in s do yield if x = 0.0 then 0.0 else x * Math
        .Log(x) });
```

```
let normalize s =
    let total = Seq.sum(s)
    seq { for x in s do yield x / total }

let normalizeDistance s =
    let dist = sumOfSquares(s)
    let result = seq { for x in s do yield x / sqrt(dist) } |> Seq.
       toArray
    let dist2 = sumOfSquares(result)
    result

let abs (x:float) =
    Math.Abs(x)

let sampleVectorWithENIUncertainty = 0.0000001;

let sampleVectorWithProperty eni prFunc normFunc =
    let rec evolve (x:seq<float>) =
        let oldEni = prFunc x
        let currentError = abs(oldEni - eni)
        let slowFactor = 1.0 / currentError
        let transformVector = seq { for x in sampleRandVector() do
           yield x + slowFactor } |> Seq.toArray
        let candidate = Seq.zip x transformVector |> Seq.map (fun((a,b)
           ) -> a * b) |> normFunc

        let newEni = prFunc candidate

        if (abs(newEni-eni) < abs(oldEni-eni)) then
            if abs(newEni - eni) < sampleVectorWithENIUncertainty then
                candidate
            else
                evolve candidate
        else
            evolve x

    let start = normFunc(sampleRandVector())
    Seq.sort(evolve start)

let sampleVectorWithENI eni =
    sampleVectorWithProperty eni calcEni normalize

let sampleVectorWithEntropy entropy =
    sampleVectorWithProperty entropy calcEntropy normalize

let sampleVectorWithSum sum =
```

```
    sampleVectorWithProperty sum Seq.sum normalizeDistance

//let rec sampleVectorWithENI eni =
//    let seq1 = Seq.toList(sampleRandVector())
//    let sum = Seq.sum(seq1);
//    let seqNorm = seq { for x in seq1 do yield x / sum };
//    let sumOfSquares = Seq.sum(seq { for x in seqNorm do yield x * x
   });
//    let actualENI = 1.0 / sumOfSquares;
//
//    if (Math.Abs(actualENI - eni) < 0.1) then
//        seqNorm
//    else
//        sampleVectorWithENI (eni)

// Transforms the entire set of real numbers to the range 0 to 1
let invLogistic x =
    let y = Math.Exp(x)
    y / (1.0 + y)

// Transforms the entire set of real numbers to valid ENI values.
let transformToENISpace x =
    let y = invLogistic (x)
    1.0 + y * (float)(numberOfParties - 3)

// Given a function float->float, we estimate it's root (i.e. argument
   that makes it return zero)
// However the function has ***random error***, i.e. so when comparing
   comparisons with it we can only be sure with some probability that
   it is correct.
// Use Newton Raphson with slight modifications.

let newtonRaphsonUncertainty = 0.0001;
let estimateRoot f =
    let rec evolve x count =        // x is the estimation of the
       parameter. count is how many times recently we "did badly"
       estimating. The higher count is, the slower we progress
        let delta = r()
        let valueAtX = f(x)
        let valueAtXd = f(x+delta)
        let derivative = valueAtXd - valueAtX
        let slowingCoefficient = 1.0 / (1.0 + Math.Pow((float)count,
           2.0))
        let candidateX = x - (slowingCoefficient * (x/derivative))
        let valueAtCandidate = f(candidateX)
```

```
            System.Console.Write("Error ")
            System.Console.Write(valueAtX)
            System.Console.Write(" ")
            System.Console.WriteLine(valueAtCandidate)

            if (abs(valueAtCandidate)<abs(valueAtX)) then
                System.Console.WriteLine("Keep")
                let newCount = if valueAtCandidate * valueAtX < 0.0 then
                    count + 1 else 0

                if abs(candidateX) < newtonRaphsonUncertainty then
                    candidateX
                else
                    evolve candidateX newCount
            else
                System.Console.WriteLine("Discard")
                evolve x (count + 1)
        evolve 1.0 0

    let rec doENISimulations rule z =
        let results : SimulationResults array = Array.zeroCreate(simCount)
        let sqr = seq { for x in z do yield x * x }
        let ss = Seq.sum(sqr)
        let effectiveENI = 1.0 / ss;
        let zSum = Seq.sum(z);

        ignore (Parallel.For (0, simCount, (fun (iteration:int) ->
            let society = new SpatialArtificialSociety()
            society.PartyCount <- numberOfParties;
            society.Dimensions <- Seq.toArray(z);
            society.ElectorateCount <- electorateCount;
            society.DistrictMagnitude <- 1;
            society.SetupElection();
            let districts = seq { for index in 1..society.ElectorateCount
                do yield society.SampleElectorate(voteCount)}
            let situation = new VotingSituation();
            situation.PartyCount <- numberOfParties;

            let processDistrict d =
                let ev = new ElectorateVotes();
                ev.Magnitude <- 1;
                ev.VoteCounts <- d;
                ev;

            situation.SetElectorates(seq { for district in districts do
                yield processDistrict(district) });
```

```
        let simResults = SimulationResults.ComputeSimulation(situation,
            rule)
        results.[iteration] <- simResults;
        ignore null )))

    if System.Double.IsNaN(Seq.average((seq { for x in results do yield
        x.EffectiveNumberOfPCAVars }))) then
        // We got NAN for our answer. Ugh.
        // Try again.
        doENISimulations rule z
    else
        results

// Find the ENI given ENV
let solveForENV env =
    let rule = new STVVotingRule();

    let eniFunc (eni:float) =
        //Console.Write("Estimate ")
        //Console.WriteLine(eni)

        let z = sampleVectorWithENI eni
        let results = doENISimulations rule z
        Seq.average((seq { for x in results do yield x.
            EffectiveNumberOfPCAVars })) - eni

    let rootFunc x =
        x |> transformToENISpace |> eniFunc

    let root = estimateRoot rootFunc
    let eniEstimate = root |> transformToENISpace
    eniEstimate

// Find the ENI given ENV
let solveForProperty env prFun lower upper =
    let rule = new FPPVotingRule();

    let rec solve lowerLimit upperLimit =
        // Generate some ENI values randomly.
        let targetCandidates = seq { for x in 1..50 do yield ((r() * (
            upperLimit-lowerLimit))+lowerLimit) } |> Seq.toArray

        // Generate values of Z with ENI close to our target ENI
        let zSeq = seq { for eniTarget in targetCandidates do yield
            sampleVectorWithENI(eniTarget) } |> Seq.toArray
```

```
        // Find out the actual ENI of the Z values. (Note we can't
           quickly generate Z exactly equal to our target)
        let candidateENI = seq { for z in zSeq do yield prFun z } |>
           Seq.toArray

        // Do many simulations for each ENI value.
        let simResults = seq { for z in zSeq do yield doENISimulations
           rule z } |> Seq.toArray

        // Calculate mean ENV for each ENI value.
        let candidateENV = seq { for results in simResults do yield Seq
           .average(seq { for simRun in results do yield simRun.
           EffectiveNumberOfPCAVars }) }
        let mappedResults = Seq.zip candidateENI candidateENV |> Seq.
           toArray
        let sortedResults = mappedResults |> Seq.sortBy(fun (a,b) ->
           abs(b - env))
        let bestResults = sortedResults |> Seq.take 10

        let resultMean = seq { for (a, b) in bestResults do yield a }
           |> Seq.average
        let resultSD = seq { for (a, b) in bestResults do yield Math.
           Pow(a-resultMean,2.0) } |> Seq.average |> Math.Sqrt
        let resultSE = resultSD / Math.Sqrt((float)(bestResults |> Seq.
           length))
        let newLower = constrain(resultMean - (resultSE * 1.96), 1.0,
           ((float)(numberOfParties-1)))
        let newUpper = constrain(resultMean + (resultSE * 1.96), 1.0,
           ((float)(numberOfParties-1)))

        if resultSE < 0.1 then
            resultMean
        else
            solve newLower newUpper
    solve lower upper

let findEniForEnv env =
    solveForProperty env calcEni 1.0 ((float)(numberOfParties-1))

//let hypothesisTest1 =
//    let eni = sampleENI()
//    Console.Write("eni ");
//    Console.WriteLine(eni);
//
//    let strm = System.IO.File.Open("enisim.csv", System.IO.FileMode.
   OpenOrCreate)
```

```
//    strm.SetLength((int64)0)
//    let report = new CSVReportWriter<SimulationResults>(new CSVWriter
    (strm))
//
//    report.AddColumn("Loosemorehanby", fun a -> a.LoosemoreHanbyIndex
    .ToString())
//    report.AddColumn("Gallagher", fun a -> a.GallagherIndex.ToString
    ())
//    report.AddColumn("EffectiveNumberOfParties", fun a -> a.
    EffectiveNumberOfParties.ToString())
//    report.AddColumn("EffectiveNumberOfPCAVars", fun a -> a.
    EffectiveNumberOfPCAVars.ToString())
//    report.AddColumn("Governability", fun a -> a.Governability.
    ToString())
//    report.AddColumn("Round", fun a -> a.Properties.["zIndex"])
//    report.AddColumn("ENI", fun a -> a.Properties.["ENI"])
//    report.AddColumn("Entropy", fun a -> a.Properties.["Entropy"])
//    report.AddColumn("SpatialSum", fun a -> a.Properties.["Sum"])
//
//    //let zSeq = seq { for i in 1..10 do yield sampleVectorWithENI (
    eni) |> Seq.toArray } |> Seq.toArray
//    //let zSeq = seq { for i in 1..10 do yield
    sampleVectorWithEntropy (-1.4) |> Seq.toArray } |> Seq.toArray
//    let zSeq = seq { for i in 1..10 do yield sampleVectorWithSum
    (1.1) |> Seq.toArray } |> Seq.toArray
//
//    let rule = new FPPVotingRule();
//
//    let indicies = seq { for i in 1..Seq.length(zSeq) do yield i } |>
     Seq.toArray
//    for (z,index) in Seq.zip zSeq indicies do
//        let results = doENISimulations rule z
//
//        let average:double = Seq.average((seq { for x in results do
    yield x.EffectiveNumberOfPCAVars }))
//
//        for result in results do
//            result.Properties.Add("zIndex", index.ToString())
//            result.Properties.Add("ENI", (calcEni z).ToString())
//            result.Properties.Add("Entropy", (calcEntropy z).ToString
    ())
//            result.Properties.Add("Sum", (Seq.sum z).ToString())
//            report.WriteLine (result)
//        Console.WriteLine(average)
//
//    report.Close()
```

```
//    printfn "Test Complete"


let simulatedElectionCount = 50

let simulateElectionsByModel (societyGen:unit->
    ArtificialSocietyGenerator) (ruleFactory:unit->VotingRule) (filename
    :string) =
    let strm = System.IO.File.Open(filename, System.IO.FileMode.
        OpenOrCreate)
    strm.SetLength((int64)0)
    let report = new CSVReportWriter<SimulationResults>(new CSVWriter(
        strm))

    report.AddColumn("lijphart", fun a -> a.LijphartIndex.ToString())
    report.AddColumn("raes", fun a -> a.RaesIndex.ToString())
    report.AddColumn("loosemorehanby", fun a -> a.LoosemoreHanbyIndex.
        ToString())
    report.AddColumn("gallagher", fun a -> a.GallagherIndex.ToString())
    report.AddColumn("enp", fun a -> a.EffectiveNumberOfParties.
        ToString())
    report.AddColumn("pca", fun a -> a.EffectiveNumberOfPCAVars.
        ToString())
    report.AddColumn("governability", fun a -> a.Governability.ToString
        ())
    report.AddColumn("entropy", fun a -> a.EntropyIndex.ToString())
    report.AddColumn("entropyseatprop", fun a -> a.EntropySeatPropIndex
        .ToString())

    let performSimulation() =
        let society = societyGen()
        society.PartyCount <- numberOfParties;
        society.ElectorateCount <- electorateCount / society.
            DistrictMagnitude;

        society.SetupElection();
        let districts = seq { for index in 1..society.ElectorateCount
            do yield society.SampleElectorate(voteCount)}
        let situation = new VotingSituation();
        situation.PartyCount <- numberOfParties;

        let processDistrict d =
            let ev = new ElectorateVotes();
            ev.Magnitude <- society.DistrictMagnitude;
            ev.VoteCounts <- d;
            ev;
```

```
            situation.SetElectorates(seq { for district in districts do
                yield processDistrict(district) });

            let rule = ruleFactory()
            let simResults = SimulationResults.ComputeSimulation(situation,
                rule)
            simResults

        let results : SimulationResults array = Array.zeroCreate(
            simulatedElectionCount)

        ignore (Parallel.For (0, simulatedElectionCount, (fun (iteration:
            int) ->
            results.[iteration] <- performSimulation()
            System.Console.WriteLine(iteration)
            )))

        for result in results do
            report.WriteLine(result)

        report.Close()
        System.Console.Write("Test ")
        System.Console.Write(filename)
        System.Console.WriteLine(" Complete")

    [<EntryPoint>]
    let main args =
        printfn "Hypothesis Testing"
        //printfn "Test solve for env"

    //    let ukSpatial = SpatialFile.FromStream("ukspatial.csv")
    //    let ukENV = ukSpatial.GetENV()
    //
    //    let eniEstimate = findEniForEnv ukENV
    //    printfn "Estimate complete"
    //    Console.WriteLine(eniEstimate) //6.21
        //hypothesisTest1

        let districtMagnitudes = [| 1; 2; 3; 4; 5; 6; 8; 10; 12; 15; 20;
            30; 60; 120 |]

        //let districtMagnitudes = [| 10; 12; 15 |]

        let stv() = new STVVotingRule() :> VotingRule
```

```
//     let pd() =
//         let rule = new ProportionalByDistrictVotingRule()
//         rule.Apportionment <- ApportionmentMethod.StLague
//         rule :> VotingRule
//
//     let pdHamilton() =
//         let rule = new ProportionalByDistrictVotingRule()
//         rule.Apportionment <- ApportionmentMethod.Hamilton
//         rule :> VotingRule
//
//     let pdHill() =
//         let rule = new ProportionalByDistrictVotingRule()
//         rule.Apportionment <- ApportionmentMethod.Hill
//         rule :> VotingRule
//
//     let pdJefferson() =
//         let rule = new ProportionalByDistrictVotingRule()
//         rule.Apportionment <- ApportionmentMethod.Jefferson
//         rule :> VotingRule


//     for dm in districtMagnitudes do
//         let spatialSocietyFactory() :ArtificialSocietyGenerator =
//             let society = new SpatialArtificialSociety()
//             society.Dimensions <- [| 1.0; 1.0; |]
//             society.DistrictMagnitude <- dm;
//             society :> ArtificialSocietyGenerator
//
//         simulateElectionsByModel spatialSocietyFactory stv ("
    spatial2_" + dm.ToString() + "_stv.csv")

//     for dm in districtMagnitudes do
//         let spatialSocietyFactory() :ArtificialSocietyGenerator =
//             let society = new SpatialArtificialSociety()
//             society.Dimensions <- [| 1.0; 1.0; |]
//             society.DistrictMagnitude <- dm;
//             society :> ArtificialSocietyGenerator
//
//         simulateElectionsByModel spatialSocietyFactory pd ("spatial2_
    " + dm.ToString() + "_pd.csv")

//     let dm = 10;
//     let spatialSocietyFactory() :ArtificialSocietyGenerator =
//         let society = new SpatialArtificialSociety()
//         society.Dimensions <- [| 1.0; 1.0; |]
//         society.DistrictMagnitude <- dm;
```

```
//        society :> ArtificialSocietyGenerator
    //simulateElectionsByModel spatialSocietyFactory pd ("
       apportionmentStLague_" + dm.ToString() + "_stv.csv")
    //simulateElectionsByModel spatialSocietyFactory pdHamilton ("
       apportionmentHamilton_" + dm.ToString() + "_stv.csv")
    //simulateElectionsByModel spatialSocietyFactory pdHill ("
       apportionmentHill_" + dm.ToString() + "_stv.csv")
//    simulateElectionsByModel spatialSocietyFactory pdJefferson ("
   apportionmentJefferson_" + dm.ToString() + "_stv.csv")

    for dm in districtMagnitudes do
        let urnSocietyFactory() : ArtificialSocietyGenerator =
            let society = new UrnArtificialSociety()
            society.AlphaGenerator <- fun () ->
                let beta = r()
                beta / (1.0 - beta)

            society.DistrictMagnitude <- dm;
            society :> ArtificialSocietyGenerator

        simulateElectionsByModel urnSocietyFactory stv ("urn_" + dm.
           ToString() + "_stv.csv")

//    for dm in districtMagnitudes do
//        let urnSocietyFactory() : ArtificialSocietyGenerator =
//            let society = new UrnArtificialSociety()
//            society.AlphaGenerator <- fun () ->
//                let beta = r()
//                beta / (1.0 - beta)
//
//            society.DistrictMagnitude <- dm;
//            society :> ArtificialSocietyGenerator
//
//        simulateElectionsByModel urnSocietyFactory pd ("urn_" + dm.
   ToString() + "_pd.csv")

    0 // return an integer exit code
```

B R Sourcecode

```
datafiles.df = data.frame(
        datasource = c(
                "C:/Users/Michael/Documents/Visual Studio 11/Projects/
                    CS380/HypothesisTesting/bin/Debug/results2/pd_
                    spatial2_results.csv",
                "C:/Users/Michael/Documents/Visual Studio 11/Projects/
                    CS380/HypothesisTesting/bin/Debug/results4/
                    spatial2stv.csv",
                "C:/Users/Michael/Documents/Visual Studio 11/Projects/
                    CS380/HypothesisTesting/bin/Debug/results5/urnstv.
                    csv"
        ),

        name=c(
                "pd_spatial2",
                "stv_spatial2",
                "stv_urn"
        ),stringsAsFactors=FALSE
)

for(datasetId in 1:nrow(datafiles.df))
{
        for (stabilityId in c(0,1))
        {
                for (fairnessId in c(0,1))
                {
                        results.df = read.table(datafiles.df$datasource
                            [datasetId], sep = ",", skip = 0, header=T)
                        datasetName = datafiles.df$name[datasetId]

                        results.df$X <- NULL
                        results.df[1:5,]

                        if (fairnessId==0)
                        {
                                results.df$fairness = results.df$
                                    gallagher;
                        }

                        if (fairnessId==1)
                        {
                                results.df$fairness = results.df$
                                    loosemorehanby;
                        }
```

```
if (stabilityId==0)
{
        results.df$stability = results.df$enp
            -1;
}

if (stabilityId==1)
{
        results.df$stability = (1/results.df$
            governability)-1;
}

plot.new()
pdf(paste("C:/Users/Michael/Documents/Visual␣
    Studio␣11/Projects/CS380/Report/images/",
    datasetName,"␣",ifelse(fairnessId==0,"
    gallagher","loosemorehanby"),"␣",ifelse(
    stabilityId==0,"enp","enpsspi"), ".pdf",sep=
    ""))

split.screen(c(2,2))

screen(1)
results.df$col1 = "purple"
results.df$col1[results.df$dm<30] = "blue"
results.df$col1[results.df$dm<10] = "green"
results.df$col1[results.df$dm==1] = "red"
rOrder = order(rnorm(nrow(results.df)))
plot(results.df$fairness[rOrder], results.df$
    stability[rOrder], xlab="Unfairness", ylab="
    Instability",col=results.df$col1[rOrder],
    main="Simulated␣Elections")
results.df$col1 <- NULL

fairness.fit = lm(log(fairness)~I(dm)+I(dm^2),
    data=results.df)
summary(fairness.fit)

#Fit stability
stability.fit = lm(stability~I(dm)+I(log(dm))+I
    (dm^2), data=results.df)

summary(stability.fit)

dummy.df = data.frame(dm=(10:1200)/10)
```

```
dummy.df$fairness = exp(predict(fairness.fit,
    dummy.df))
dummy.df$stability = predict(stability.fit,
    dummy.df)
dummy.df$isPredicted = TRUE;
dummy.df$type=1


dummy.df$col1 = "purple"
dummy.df$col1[dummy.df$dm<30] = "blue"
dummy.df$col1[dummy.df$dm<10] = "green"
dummy.df$col1[dummy.df$dm==1] = "red"

screen(2)
randOrder = order(rnorm(nrow(dummy.df)))
plot(dummy.df$fairness[randOrder], dummy.df$
    stability[randOrder], xlab="Unfairness",
    ylab="Unstability", main="Predicted
    Elections by DM", col=dummy.df$col1[
    randOrder])

#Calculate extremes
#Note this plot may be misleading. Points on
    this plot do not necessarily correspond to
    actual predicted elections.
extreme.df = data.frame(dm=unique(results.df$dm
    ))
for(dm in extreme.df$dm)
{
        extreme.df$fairness05[extreme.df$dm==dm
            ] = quantile(results.df$fairness[
            results.df$dm==dm], 0.05)
        extreme.df$fairness95[extreme.df$dm==dm
            ] = quantile(results.df$fairness[
            results.df$dm==dm], 0.95)
        extreme.df$stability05[extreme.df$dm==
            dm] = quantile(results.df$stability[
            results.df$dm==dm], 0.05)
        extreme.df$stability95[extreme.df$dm==
            dm] = quantile(results.df$stability[
            results.df$dm==dm], 0.95)
}

screen(3)

extreme.df$col1 = "purple"
```

```
                extreme.df$col1[extreme.df$dm<30] = "blue"
                extreme.df$col1[extreme.df$dm<10] = "green"
                extreme.df$col1[extreme.df$dm==1] = "red"
                plot(extreme.df$fairness95, extreme.df$
                    stability95, xlab="Unfairness", ylab="
                    Unstability", main="95th percentile by DM",
                    pch=15,col=extreme.df$col1)

                unknown = rep(NA,length(unique(results.df$dm)))
                actualmeans.df = data.frame(dm=unknown,
                    fairness=unknown, stability=unknown)
                actualmeans.df$dm = unique(results.df$dm)
                actualmeans.df$type=0
                actualmeans.df$isPredicted=FALSE

                for (dm in actualmeans.df$dm)
                {
                        actualmeans.df$fairness[actualmeans.df$
                            dm == dm] = mean(results.df$fairness
                            [results.df$dm==dm])
                        actualmeans.df$stability[actualmeans.df
                            $dm == dm] = mean(results.df$
                            stability[results.df$dm==dm])
                }

                actualmeans.df$col1 = "purple"
                actualmeans.df$col1[actualmeans.df$dm<30] = "
                    blue"
                actualmeans.df$col1[actualmeans.df$dm<10] = "
                    green"
                actualmeans.df$col1[actualmeans.df$dm==1] = "
                    red"

                screen(4)
                plot(actualmeans.df$fairness, actualmeans.df$
                    stability, xlab="Unfairness", ylab="
                    Unstability", main="Mean Elections by DM",
                    pch=15, col=actualmeans.df$col1)

                dev.off()
        }
    }
}
```