Campaigning for Congress after Redistricting

ESI 6341 - Introduction to Stochastic Optimization

James Diffenderfer

April 13, 2018

Abstract

The goal of this project is to develop a two-stage stochastic program model with recourse to consider how redistricting and organizing a political campaign affect the outcome of congressional elections for each political party involved. The first stage of the program will be used to determine how to redistrict a state. The following stages will correspond to how much money candidates should spend campaigning based on the factors of the predetermined voting districts and factors of their opponent's campaign. The model for the second stage of the program used to determine how each candidate should campaign will incorporate a variety of scenarios introducing uncertainty into the problem. Given a political party, the objective of the problem is to maximize the expected percentage of votes that go to that party across all districts in the state.

Contents

Introduction

United States Representatives are elected by voters in subsets of each state called districts. Congressional districts for each state are updated after the census is taken, roughly at the start of each decade, in order to balance the populations across districts. This process which takes place near the start of each decade is referred to as *redistricting*.

Drawing Congressional District Lines

Here, we outline the three main groups responsible for drawing *congressional* district lines:

State Legislature: In 37 states, the congressional district lines are determined by state legislature. Unfortunately, members of the state legislature fall into a political party (e.g., Republican, Democrat) which has historically influenced decisions when drawing district lines.

Single District States: In 7 states, there is only one congressional district due to the state population so there is no need to draw district lines.

Independent Commissions: In the remaining states, congressional districts are determined by independent commissions with limited participation and influence by elected officials.

There are several additional factors which also contribute to and provide restrictions on the process of redistricting:

Advisory Commissions: In Ohio and Rhode Island, commisions are appointed to advise the state legislature on where to draw congressional district lines. Non-legislators are sometimes invited to take place in forming an initial draft of the congressional districts. It should be noted that the state legislature is not required to make use of the advisory committee's suggestions. As an example, in Iowa the state legislature must either accept or reject new district lines proposed by a nonpartisan group without any modifications. If the Iowa legislature rejects two different proposals they are free to redistrict the state as they see fit.

Backup Commissions: In Connecticut and Indiana, backup commisions are appointed to review the initial draft of the districts determined by the state legislatures before it goes to a final vote. The members of the backup commision vary from state to state.

Subject to Governor Veto: In many of the states, new district lines are subject to review by the current Governor who has the option to veto the proposed districts.

Required Supermajority: In Connecticut and Maine, a two-thirds vote is required in each house to approve a redistricting plan.

Joint Resolution: In Connecticut, Florida, Maryland, Mississippi, and North Carolina, approval of new districts is required to pass a joint resolution, that is the new districts much be approved by the Senate and the House of Representatives and finally by the President of the United States.

Regulations on Congressional District Lines

In addition to the redistricting process handled by legislatures and commissions in each state, there are a set of guidelines and regulations that are to be followed during the redistricting process. Below are requirements that must be met when drawing *congressional* district lines in all states:

> **Equal Population**: It was decided, per *Wesberry v. Sanders* in 1964, that the population across all congressional districts in a state should be equal "as nearly as is practicable."
>
> **Race and Ethnicity**: In 1965, Congress enacted the *Voting Rights Act* to guard against attempts to prevent minorities from effectively voting through redistricting. The Voting Rights Act has had several amendments, the latest of which was in 2006. The two primary methods used to discriminate against minorities are often referred to as *cracking* and *packing*. Cracking is the practice of drawing district lines in order to split minorities into as many districts as possible while packing is the practice of drawing district lines in order to put minorities into as few districts as possible. Race and ethnicity can be considered when drawing districts but may not be used as "predominant" factors in the process (as ruled by the Supreme Court in *Bush v. Vera*, 1996).

In addition to the requirements set for all states, there are other constraints for congressional redistricting which are set on a state level:

> **Contiguity**: A district is considered to be *contiguous* if travel between any two points in the district can be made without crossing the boundary of the district. 23 states require that congressional districts must be contiguous "to the extent possible."
>
> **Political Boundaries**: Examples are county, city, or town lines. 19 states require that congressional districts follow political boundaries "to the extent practicable."
>
> **Compactness**: There is no widely accepted definition of compactness but the idea typically falls into three broader categories.
>
> > **Contorted Boundaries**: Districts with smoother boundaries are more compact.
> >
> > **Dispersion**: The degree to which a district spreads from its center. A district with few pieces extruding from the center is considered more compact.
> >
> > **Housing Patterns**: The idea of this restriction is that the district lines should not be drawn to specifically align with housing patterns.
>
> Finally, note that 18 states require that congressional districts must be compact.
>
> **Communities of Interest**: A *community of interest* is a region in a state with "[s]ocial, cultural, racial, ethnic, and economic interests common to the population of the area, which are probable subjects of legislation." These guidelines for identifying communities of interest were a result of the fairly recent case *Graham v. Thornburgh* in 2002. 13 states require that congressional districts must preserve communities of interest.
>
> **Political Outcomes**: There is little to no restrictions placed on redistricting to favor political outcomes in the United States Constitution. While the justices of the Supreme Court agreed in 2004 (per *Vieth v. Jubelirer*) that redistricting to favor a political party is unconstitutional,

a precedent on how to limit such behavior has not been set. Shockingly, only 8 states prohibit the redistricting of congressional districts in order to unfairly favor one political party over another.

Throughout the history of the United States, there have been conflicts over the redistricting process and the resulting district lines. In nearly every state constitution, there is a deadline for finishing the standard procedure for redistricting. States that do not complete the regular process by this deadline often lose their right to draw district lines for that period and either state or federal courts take on the responsibility of determining district lines prior to the upcoming election.

Due to the partisan nature many of the states have in place for drawing congressional district lines, state or federal courts were required to review and oversee the redistricting process in 42 states during the most recent redistricting process (in 2011). In two of these states, namely Florida and Texas, newly declared districts were determined to be unlawful due to unconstitutional partisan discrimination. Additionally, litigation was brought against Arizona, Maryland, North Carolina, and Virginia to challenge the districts drawn in 2011.

As a final note, it is not uncommon for the process of hearing these cases to take place over the span of several years before arriving at a conclusion. In recent news, the case *Benisek v. Lamone* challenging the congressional map in Maryland based on the "essentially non-contiguous structure" of the districts was originally filed in Noevmber of 2013 and is still ongoing with an argument set to take place before the Supreme Court on March 28, 2018.

Redistricting and Campaigning - Maximizing the Expected Percentage of Votes Earned Across Districts

In this section, we consider a two-stage stochastic program model for solving the problem of redistricting and campaigning for a congressional election in order to maximize the expected percentage of votes earned by our party across all districts. For the purposes of preventing the model from becoming overly complex, we will work under the assumption of a two party system taking place during each election (that is, two candidates will be the ballot for each district).

In order to model the problem using real data, a single state was selected over which to consider the optimization problem. After researching the requirements for redistricting and considering the data resources available, we created a state with 37 counties using population data on 37 counties from the state of Florida. In order to prevent the number of scenarios from growing too large, we decided that the state would be divided into three districts during the redistricting phase (which is the most computationally expensive part of the problem due to the contiguity constraint).

After selecting a state for the problem, we used the avaiable data to model a function that accurately predicts the percentage of votes a candidate will earn. To accurately model this function, we used data from the US Census Bureau Fact Finder, the Iowa Secretary of State website, the Office of the Clerk website, the website for the Federal Election Commission of the United States of America, and Ballotpedia. The final two sources provide detailed data on campaign expendetures while the first three are reliable sources for population data in the United States and information on voter registration for residents of Iowa.

Using the data obtained from the sources outlined above, we modeled a function which returns the percentage of votes which go to a given candidate based on 173 parameters:

- 165 from the US Census Bureau Fact Finder for population, age, gender, race, education, and income statistics
- 5 from the Federal Election Commission and Ballotpedia for campaign expenditures, incumbency statistics, and voting statistics
- 3 from the Florida Department of State and Iowa Secretary of State on voter registration information

The objective function was modeled as a neural network with two fully connected layers with ReLU activation functions followed by a sigmoid output layer (ensuring that the output is between 0 and 1). The model was coded and trained using the Tensorflow library for python. The data was preprocessed using the Pandas library for python. The code for the neural network is provided in the appendix.

Problem Model

We now present a model to maximize the expected percentage of votes earned across all districts by a single party as a two-stage stochastic program with recourse. We place the following assumptions on the model:

- Work from the perspective of a single political party in a single state.
- The opposing party spends the entirety of their budget on campaign expenditures.
- The counties in our state have the same shape and area.

In order to make the redistricting problem more reasonable to solve, we only consider enforcing the three most common requirements on the redistricting phase:

- There must be a **nearly equal population** across all districts.
- There must be a **nearly equal distribution of races** across all districts.
- All districts must be **contiguous**.

Note that we are not placing any restrictions on the percentage of voters registered under a certain party by district For the second stage of the program we will assume that we have a **discrete probability distribution** representing all possible campaign scenarios.

Data and Parameters

Given Data.

m	:	Number of districts in state (varies by state, we will take $m = 3$)
n	:	Number of counties in state (varies by state, we will take $n = 37$)
D	:	Index set $D = \{1, 2, \ldots, m\}$
C	:	Set of all counties in our state
R	:	Set of all races in our state
P	:	Set of all parties running in our state
s	:	Number of campaign scenarios (in our case $s = 24$)
L	:	Index set $L = \{1, 2, \ldots, s\}$

Parameters.

q : Total population of our state

q_{race} : Total population of a given race in our state, for all $race \in R$

p_{county} : Total population of given county, for all $county \in C$

$p_{race,county}$: Total population of given race in a given county, for all $race \in R$, $county \in C$

X_{county} : Population, age, gender, education, income, and voter registration parameters for a given county, for all $county \in C$

$\omega^{(\ell)}$: Probability of campaign scenario ℓ occurring, for all $\ell \in L$

$f_i^{(\ell)}$: Maximum amount of funds available to our party in campaign scenario, for $i \in D$ and $\ell \in L$

Stage One

Decision Variables. We have the following first-stage decision variable:

$S_{c,i}$: Set of population points used from County c in District i, $c \in C$, $i \in D$

X_i : Population, age, gender, education, income, voter registration, incumbency, and opposing party budget parameters for District i, for all $i \in D$

Constraints. In addition to contiguity requirements, we have the following first-stage constraints:

$\left|\sum_{c \in C} 1000|S_{c,i}|\frac{p_c}{q} - \frac{1}{m}\right| \leq 0.05, \quad i \in D$: Districts have nearly equal populations (in particular, population in each district is within five percent of one third of the state's population)

$\left|\sum_{c \in C} 1000|S_{c,i}|\frac{p_{r,c}}{p_c} - \frac{q_r}{q}\right| \leq 0.10, \quad i \in D, \; r \in R$: Percentage of each race in each district should be nearly identical across all districts (in particular, the percentage of each race in each district should be within ten percent of the state's percentage for each race)

$X_i = \sum_{c \in C} \frac{1000|S_{c,i}|}{p_c} X_c, \quad i \in D$: District parameters are a linear combination of parameters from each county in that district.

Stage Two

Decision Variables. We have the following second-stage decision variables:

$y_i^{(\ell)}$: Amount of money to be spent by our party's candidate in district i in campaign scenario ℓ, for all $i \in D$ and $\ell \in L$

IVRs. We have the following basic constraint on the second-stage decision variables:

$y_i^{(\ell)} \geq 0, \quad i \in D$: The amount of money spent on the campaign must be nonnegative

Constraints. We have the following constraints on the second-stage decision variables:

$y_i^{(\ell)} \leq f_i^{(\ell)}, \quad \ell \in L$: The amount of money spent on the campaign in each district must not exceed the amount of money available to the campaign

Objective

Ideally, we would like to maximize the expected number of districts that go to our party. Unfortunately, if we want to do this for all districts simultaneously, the objective function would be non-differentiable. This would add another layer of complication to the problem as solvers typically make use of first derivative information. In order to avoid this problem, we will instead solve the problem of maximizing the expected number of votes that go to our party across all districts.

We will let $\Phi_{X_i}(y)$ denote the function modeling the percentage of votes earned by our candidate in the congressional election with district parameters X_i. Before presenting the objective function as a whole, we first break down the components of the objective function.

First, the value given by $\Phi_{X_i}(y_i^{(\ell)})$ is the percentage of votes earned by our candidate in district i after redistricting when our candidate spent $y_i^{(\ell)}$ dollars on campaign expenditures. So the expected percentage of votes earned by our candidate in district i across all campaign scenarios is given by

$$\text{Expected percentage of votes earned in district } i = \sum_{\ell \in L} \omega_\ell \; \Phi_{X_i}\left(y_i^{(\ell)}\right).$$

Thus, the objective function is given by summing over all districts and maximizing

$$\max \quad \sum_{i \in D} \left[\sum_{\ell \in L} \omega_\ell \; \Phi_{X_i}\left(y_i^{(\ell)}\right) \right].$$

Now that we have completed the model of our problem, we will provide the data used for the various campaign scenarios followed by our methodology for determining a local solution of the problem and an analysis of our results.

Campaign Scenarios

For each district we considered two different campaign budget and incumbency scenarios each with a given probability.

District 1				
Scenario	Probability	Our Budget	Opponent Budget	Incumbent Party
1	0.7	1,500,000	1,000,000	Opponent
2	0.3	4,000,000	2,000,000	Opponent

District 2				
Scenario	Probability	Our Budget	Opponent Budget	Incumbent Party
1	0.4	2,000,000	1,000,000	Our Party
2	0.6	4,000,000	3,000,000	Our Party

District 3				
Scenario	Probability	Our Budget	Opponent Budget	Incumbent Party
1	0.8	3,000,000	2,000,000	Opponent
2	0.2	2,000,000	3,000,000	Our Party

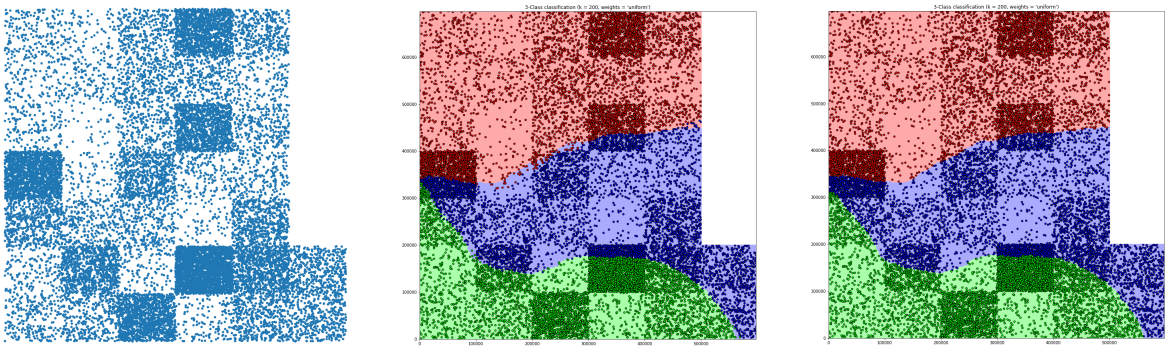
Using these independent distributions, we created a distribution with 8 outcomes in which the funding for the democrat candidate in all districts is a random variable. The main reason we do

not consider every district to have random variables for funding and incumbency is in order to keep the number of outcomes relatively small for the analysis of the solution.

Methodology

First, we developed a method for generating districts based on methods found in literature [REFERENCES]. As the methods found in the literature enforced the constraint that entire counties should be used when composing the districts and we did not choose to enforce this constraint in our problem, we were not able to directly use any of these existing methods for districting. However, we did use several ideas from methods in the literature.

First, we generated a geographical model of the state we planned on redistricting using population data from 37 counties in Florida available from the US Census Bureau. This can be seen in Figure 1a. Each point on the plot represents 1000 citizens in the state with population statistics equal to the mean of the population statistics for the county in which the point is located. Next, one initial point is chosen for each district. Then each district then takes turns choosing a point nearest to their initial point in order to keep the race percentages close to the mean race percentages for the entire state. We implemented this procedure using an algorithm we coded in python. This algorithm made use of the 'BallTree' data structure from the library sklearn with the standard Euclidean norm. When deciding which point to choose, we considered the four nearest points to our initial which had not yet been selected. From these four points, we selected the point that was closest and made the value of the variance from the mean of the race population smallest.



(a) State population distribution (b) Initialization of districts (c) k-NN classification

Figure 1: Visualization of redistricting procedure

After all of the points have been selected, k-nearest neighbors classification is performed to determine the boundaries of the districts and to enforce the contiguity requirement. In order to end up with districts that are contiguous, a high number of nearest neighbors is selected for the fitting the k-NN model. For our algorithm, we observed that using $k = 200$ with a uniform distrance typically resulted in three contiguous districts. The result of this step can be seen in Figure 1b. Finally, we use the k-NN model to classify the points in our state so that all of the points are correctly classified by district. This can be seen in Figure 1c. At the end of this algorithm, the decision variables $S_{c,i}$ presented in the model are determined. If these variables do not satisfy the first two constraints from stage one of the model, then the districts are destroyed and the process is restarted. If the first two constraints are satisfied and the districts are contiguous, then we

construct the decision variables X_i using the third constraint. The values X_i are then passed to the second stage to maximize the objective function.

Since we used a neural network with ReLU and Sigmoid activation layers to model the percentage of votes earned by our party, our objective function is nonconvex. As such, we are unable to guarantee that we can determine a global maximum but there are several methods available for finding a local maximum.

To maximize the objective function, we used a batch stochastic gradient projection algorithm modeled after a gradient projection algorithm by Hager and Zhang [REFERENCE]. Since we are only working with bound constraints in the second stage of the model, the algorithm was relatively short and was coded using Python. As our neural network was coded using the Neural Network API Keras on the library TensorFlow, we used the TensorFlow backend to perform the gradient evaluations. Pseudocode for the batch stochastic gradient projection algorithm, for minimization of a differentiable function g, is provided in Algorithm 1. Note that the pseudocode uses a function p which is a projection of a point onto the feasible set. In our problem, the output of the projection function p for the i component of scenario ℓ is defined as

$$p(y)_i^{(\ell)} = \begin{cases} f_i^{(\ell)} & : \ y_i^{(\ell)} > f_i^{(\ell)} \\ y_i^{(\ell)} & : \ y_i^{(\ell)} \in [0, f_i^{(\ell)}] \\ 0 & : \ y_i^{(\ell)} < 0 \end{cases}.$$

Algorithm 1 - Batch Stochastic Gradient Projection Algorithm

Choose $x_0 \in \mathbb{R}^n$. Initialize parameters ε_{stop}, $batch\_size$, and max_batch_iters.
for k $= 0, 1, 2, \ldots, max\_batch\_iters$ **do**
    if $\|p(x_k) - x_k\| \leq \varepsilon_{stop}$ **then**
        Return x_k and exit program.
    end if
    Set b to be an array of distinct indices between 1 and n of length $batch\_size$.
    Initialize $d_k = \mathbf{0} \in \mathbb{R}^n$.
    for j $= 1, 2, \ldots, batch\_size$ **do**
        $d_k[j] = \nabla g(x_k)[b[j]]$
    end for
    Set $x_{k+1} \leftarrow x_k + d_k$.
end for
if $k > max\_batch\_iters$ **then**
    Perform Hager and Zhang gradient projection algorithm with starting point x_{k-1}.
end if

In order to ensure convergence, if the search directions used from sampling components of the gradient do not satisfy the stopping criterion after some maximum number of iterations, the algorithm switches over to the standard gradient projection algorithm defined by Hager and Zhang which uses the full gradient in the search direction and an armijo line search to ensure the stopping criterion are satisfied.

Since the process of creating the districts is computationally expensive, we computed several local minimums for the second stage of the problem. Unfortunately, due to our approach for redistricting there was not a clear method for modifying the districts using gradients from the

objective function with respect to the variabels X_i in a way that would not violate the contiguity constraint. Ideally, we would be able to modify the districts without violating these constraints so that our party could also increase the percentage of votes earned by legally influencing the redistricting procedure. As we faced this limitation due to our model, we chose the local solution resulting in the largest percentage of votes from all of the second stage solutions we computed for a district constructed during the first stage.

Results

We ran the entire algorithm described in the methodology section eight times with the second phase computing three local maximums each time the algorithm ran. Here we present our results for two local maximums with the highest values resulting from different districts constructed in the first stage. The results from these solutions can be found in Figures 2 and 3.

Deviation from One-Third of Total Population		
District 1	District 2	District 3
1.385 %	0.740 %	0.645 %

Deviation from Mean of Each Race Population			
Race	District 1	District 2	District 3
0	4.829 %	6.365 %	1.842 %
1	4.500 %	5.163 %	0.946 %

Campaign Expenditures (in Dollars)			
Scenario	District 1	District 2	District 3
1,1,1	0.00	0.00	1,215,183.00
2,1,1	795,441.30	1,327,494.84	1,285,359.83
1,2,1	0.00	1,846,470.85	0.00
1,1,2	203,391.60	3,387,493.00	943,895.74
2,2,1	3,003,245.75	752,255.16	2,376,795.19
2,1,2	3,763,949.55	1,697,554.59	1,728,911.95
1,2,2	2,552,034.91	3,060,235.66	1,987,804.61
2,2,2	3,648,446.16	3,726,293.13	1,601,049.73

Expected Percentage of Votes Across All Districts
41.49

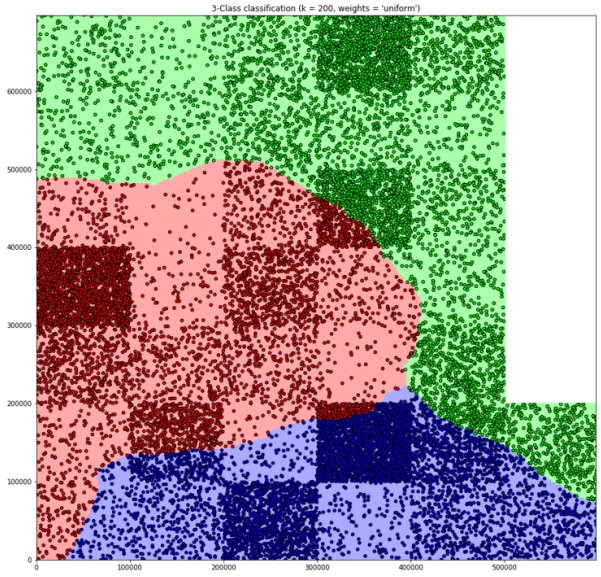


Figure 2: One local maximum obtained after redistricting

The values labeled 'Deviation from One-Third of Total Population' correspond to the left hand side of the first constraint in stage one of the model. As we required these values to be bounded above by 5%, we can see that they satisfy the constraints in all districts. The values labeled 'Deviation from Mean of Each Race Population' correspond to the left hand side of the second constraint in stage one of the model. All of these values satisfy the constraint as we required them to be bounded above by 10%. Observing the plot of the districts in both solutions, we observe that the regions are indeed contiguous as desired. Thus, we have that all of the first stage constraints were satisfied.

The second stage decision variables values at the local maximum are provided under the table labeled 'Campaign Expenditures (in Dollars).' The 'Scenario' column is an ordered triple where the ith number in the triple corresponds to the scenario for the ith district, for $i \in \{1, 2, 3\}$. The expenditures in each outcome satify the bound constraints provided in stage two of the model when compared to the campaign budget provided for each scenario.

Finally, the table labeled 'Expected Percentage of Votes Across All Districts' provides the value of the objective function times 100 divided by the number of districts to compute the expected percenatge of votes across all districts in all scenarios.

Deviation from One-Third of Total Population		
District 1	District 2	District 3
0.238 %	0.381 %	0.143 %

Deviation from Mean of Each Race Population			
Race	District 1	District 2	District 3
0	5.246 %	4.156 %	1.171 %
1	4.268 %	3.175 %	1.155 %

Campaign Expenditures (in Dollars)			
Scenario	District 1	District 2	District 3
1,1,1	0.00	0.00	436539.79
2,1,1	752,908.78	1,299,112.88	1,319,356.70
1,2,1	0.00	0.00	0.00
1,1,2	233,450.22	2,222,531.41	871,216.45
2,2,1	1,706,859.04	869,676.79	1,977,478.63
2,1,2	2,424,552.34	1,700,415.17	1,737,906.64
1,2,2	1,240,332.17	1,679,516.80	1,561,693.18
2,2,2	2,321,334.14	2,474,906.48	1,579,844.78

Expected Percentage of Votes Across All Districts
41.56

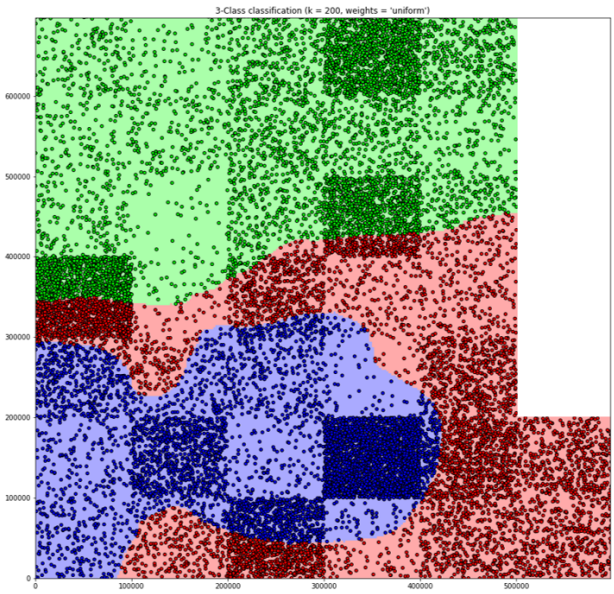


Figure 3: Another local maximum obtained with different districts

Out of all of the local solutions computed, the solution provided in Figure 3 was one of the best results for our party. One point of interest was that the solutions obtained often resulting in campaign expenditures in several districts in some outcomes to be zero dollars. This does not seem to result from an error in the solver, but rather as a result of the limited amount of campaign expenditure data readily available for training the neural network which was used in our objective function. Out of all of the local solutions we computed, the expected percentage of votes across all districts typically fell between approximately 39% and 42%.

Conclusion

This project considered an approach of considering the redistricting phase as a part of campaigning for a congressional election and how to effectively and legally influence redistricting and campaign in order to positively affect the outcome of an election. After considering this problem and developing the model and coding the algorithms presented here, there is still room for future work.

First, the optimization model should be updated to allow the districts determined from stage one to be modified by the gradients of the objective function. To do this, we should treat $\Phi_{X_i}(y)$ as a function of y and X_i. Then we can compute the partial derivatives of Φ with respect to each component of X_i. By modifying the algorithm used in redistricting, these partial derivatives can be used to make updates to the districts in the first stage of the model in order to increase the percentage of votes earned by our party across all districts.

Lastly, the neural network model should be trained on more data to obtain more accurate pre-

diction results. The primary difficulty faced when collecting congressional election and expenditure data to train the neural network with was that this data was not typically available in an easily modifiable file format. While ample amounts of data was available from the US Census Bureau in a csv fiel format, it was time consuming to convert the voting and campaign data from a pdf file format to a csv format resulting in a much smaller training set than what we believe the model would require. As a result, the gradients of the current model often indicate that our party should decrease campaign expenditures, even down to zero sometimes, to increase the percentage of votes obtained (at least from the perspective of the Democrat Party which is the party on which we based our model). This is counterintuitive since it seems that any election in which a candidate spends no money on campaign expenditures would result in a loss for that candidate.

References

[1] Census.gov. 2018. American FactFinder. Retrieved March 27, 2018, from https://factfinder.census.gov/faces/nav/jsf/pages/searchresults.xhtml?refresh=t

[2] Iowa Secretary of State. 2018. Election Results & Statistics. Retrieved March 27, 2018, from https://sos.iowa.gov/elections/results/

[3] Florida Division of Elections. 2018. Voter Registration - By County and Party. Retrieved March 27, 2018, from http://dos.myflorida.com/elections/data-statistics/voter-registration-statistics/voter-registration-monthly-reports/voter-registration-by-county-and-party/

[4] Federal Election Commision. 2018. Campaign Finance Data. Retrieved March 27, 2018, from https://www.fec.gov/data/

[5] Justin Levitt. 2018. All About Redistricting. Retrieved March 20, 2018, from http://redistricting.lls.edu/

[6] Kim, M. J. (2011). Optimization Approaches to Political Redistricting Problems. Ph. D. thesis, The Ohio State University, Columbus, OH.

[7] William W. Hager and Hongchao Zhang, An Active Set Algorithm for Nonlinear Optimization with Polyhedral Constraints, Science China Mathematics, ICIAM Special Issue, 59 (2016), pp. 1525-1542.

[8] Leveaux-Sharpe, C. (2001). Congressional Responsiveness to Redistricting Induced Constituency Change: An Extension to the 1990s. Legislative Studies Quarterly, 26(2), 275-286.

[9] "Voting Rights Act of 1965" (PL 89-110, 6 August 1965), 79 United States Statutes at Large, pp. 437-446. Available from http://www.gpo.gov/fdsys/pkg/STATUTE-79/pdf/STATUTE-79-Pg437.pdf

[10] Wesberry v. Sanders. (n.d.). Oyez. Retrieved March 20, 2018, from https://www.oyez.org/cases/1963/22

Appendix

Here we provide our code for modeling the percentage of votes earned by a candidate in a congressional campaign and the code for the two-stage model of the Redistricting and Campaigning Problem.

Neural Network Model using Tensorflow

```
# library for linear algebra tools
import numpy as np

# library for data processing tools
import pandas as pd

# keras for machine learning tools (API for Tensorflow)
import keras
from keras.models import Sequential
from keras.layers import Dense

# Create sequential model for neural network
nnet_model = Sequential()

# ----- Add first layer of network (Input layer) -----
# Number of neurons:     173
# Number of inputs:      173 (the number of features)
# Activation layer:      ReLU (Rectified Linear Unit)
# 'Dense' refers to the fact that the layer is totally connected
nnet_model.add(Dense(173, input_dim=173, activation='relu'))

# ----- Add second layer of network (Hidden layer) -----
# Number of neurons:       86
# Number of inputs:       173 (this is automatically determined since the model type is 'Sequential')
# Activation layer:       ReLU (Rectified Linear Unit)
nnet_model.add(Dense(86, activation='relu'))

# ----- Add final layer of network (Output layer) -----
# Number of neurons:       1
# Number of inputs:       86 (this is automatically determined since the model type is 'Sequential')
# Activation layer:       Sigmoid (This ensures out output is between 0 and 1 for percentage of votes)
nnet_model.add(Dense(1, activation='sigmoid'))

# Compile the neural network model
# Loss function:                  Mean Squared Error
# Gradient Descent algorithm:     adam
# Classification metric:          accuracy
nnet_model.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])

# Fit the model using the training data
nnet_model.reset_states();
nnet_model.fit(X_train, Y_train, verbose = 0, epochs=200, batch_size=10);
```

Program for Redistricting and Maximizing Expected Votes

```
# math for mathematical functions
import math

# numpy for linear algebra and sampling
import numpy as np

# pandas for data processing
import pandas as pd

# keras for neural network
from keras.models import load_model
from keras import backend

# Tensorflow for neural nerwork model gradients
import tensorflow as tf

# matplotlib for plotting
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.colors import ListedColormap

# sklearn for k-means clustering
from sklearn.cluster import KMeans

# sklearn for k-nearest neighbors
from sklearn.neighbors import NearestNeighbors
from sklearn import neighbors, datasets

# sklearn for balltree
from sklearn.neighbors import BallTree

# for k-d tree
from scipy import spatial

# scipy for optimization library
from scipy.optimize import minimize
from scipy.optimize import linprog

# itertools for generating scenarios
import itertools

# time for timing methods
import time

# Import population data
df_florida = pd.read_csv('CleanData/Florida-US-CENSUS-and-Voter-DATA.csv')
# Drop fist column (as it is 'Unnamed' column not containing real data)
df_florida.drop(df_florida.columns[0],axis=1,inplace=True)
# Save header names for data frame
df_florida_header_list = [i for i in list(df_florida.columns.values) if i not in ['County']]

# -------------------------- Functions associated with Redistricting ----------------------- #

# Function for generating possibly non-contiguous district
def construct_district(n_dist, n_neigh, pts, P, R, p_norm):
    # Create copy of P so P does not get destroyed
    Q = P[:]

    # Create copy of R so R does not get destroyed
    S = R[:]

    # Initialize n_race
    n_race = len(S)
```

```
# Create array of means of population race distributions
mu = []
for i in range(n_race):
    mu = np.append(mu, np.mean(S[i]))

# Initialize mean of race distribution for pts
R_means = np.zeros((n_dist, n_race))

# Initialize counter for each district
n_pts = np.ones(n_dist, dtype = int)

# Initialize list of remaining indices
rem_indices = [i for i in range(len(P))]

# Initialize x_dist, y_dist, and ind_dist used to store x and y coords of pts in each district
# and the indices of P belonging to each district
x_dist = [[] for i in range(n_dist)]
y_dist = [[] for i in range(n_dist)]
ind_dist = [[] for i in range(n_dist)]

while len(Q) > n_neigh:
    for i in range(n_dist):
        # Create KDTree
        tree = BallTree(Q) #, leaf_size = 100)

        # Set number of neighbors to use in update of indices and distances
        # (need to make sure we don't exceed avaiable number of points left)
        neighbors = min(n_neigh,len(Q)-1)
        # Determine distances and indices from nearest neighbor search
        dist, indices = tree.query(pts[i], k=neighbors)

        # Initialize index of point
        indices = indices.flatten()
        ind = indices[0]

        # Initialize min_dist
        min_dist = 100*n_race
        # Initialize temp mean array
        temp_mean = np.zeros(n_race)

        # Determine index which minimizes distances from mean of race distributions
        for j in range(neighbors):
            temp_ind = indices[j]
            # Update race means for test point
            for k in range(n_race):
                temp_mean[k] = ((n_pts[i]-1)*R_means[i][k] + S[k][temp_ind])/(n_pts[i])
            # Compute distance to mean for each race and sum
            temp_dist = np.mean(temp_mean - mu)
            # Check if distance from mean is smaller than current smallest value
            if temp_dist < min_dist:
                # Update index
                ind = temp_ind
                # Update min_dist
                min_dist = temp_dist
                # Update mean
                new_means = np.copy(temp_mean)

        # Update R_means
        R_means[i] = np.copy(new_means)

        # Append x and y values to district lists
        x_dist[i] = np.append(x_dist[i], Q[ind][0])
        y_dist[i] = np.append(y_dist[i], Q[ind][1])
```

```
            # Appending indices to index list
            ind_dist[i] = np.append(ind_dist[i], rem_indices[ind])

            # Update counter
            n_pts[i] = n_pts[i] + 1

            # Update pts[i]
            #for j in range(len(pts[i])):
            #    pts[i][j] = np.sum(pts[i][j])/n_pts[i]

            # Update Q by removing point added to district
            Q = np.delete(Q, ind, 0)

            # Update S by removing point added to district
            for k in range(n_race):
                S[k] = np.delete(S[k], ind, 0)

            # Update remaining indices by removing used index
            rem_indices = np.delete(rem_indices, ind)

    # Return x_dist and y_dist and exit function
    return x_dist, y_dist, ind_dist, R_means

# Function for creating array with number of points per county in each district
def county_per_district(n_dist, n_counties, county_arr, ind_arr):
    # Create array containing number of points from each county in each district
    Y_counties = np.zeros((n_dist, n_counties), dtype = int)

    j = len(ind_arr)
    k = len(county_arr)
    print("j,k: %lg, %lg" %(j,k))

    # Fill array with appropriate values
    for i in range(len(ind_arr)):
        Y_counties[int(ind_arr[i])][int(county_arr[i])] = Y_counties[int(ind_arr[i])][int(county_arr[i])] + 1

    # Return Y_counties
    return Y_counties

# Function for creating district data based on US CENSUS and Voter Registration Data for counties
def create_district_data(X_pops, X_medians, Y_counties, n_dist, n_counties, pop_scale):
    # Initialize array to store features for districts
    Y_pops = np.zeros((n_dist, np.shape(X_pops)[1]))
    Y_medians = np.zeros((n_dist, np.shape(X_medians)[1]))

    # Update medians by averaging medians
    Y_medians = np.matmul(Y_counties, X_medians)
    count_per_dist = np.matmul(Y_counties, np.ones(n_counties))

    # Fill population arrays
    for j in range(n_dist):
        Y_medians[j] = Y_medians[j] / (count_per_dist[j])
        #temp = np.zeros(np.shape(X_medians)[1])
        # Update populations and temp
        for i in range(n_counties):
            # Update populations
            Y_pops[j] = Y_pops[j] + (Y_counties[j][i] / pop_scale) * X_pops[i]
            # Update temp for averaging medians
            #temp = temp + Y_counties[j][i] * X_medians[i]

    # Return arrays containing population of each district and medians of each district
    return Y_pops, Y_medians
```

```
# ------------------- Functions for Classifying and Plotting Districts ------------------- #
# ------------ Initial classification of districts ------- #
def initial_district_classification(P, indk, district_number):
    # ------------------- Train nearest neighbors model for classification to enforce contiguity districts ------
    # Our features will be the set of ordered pairs P
    X_train = P
    # The target data will be the classes determined by partitioning of the points using the construct_districts
    Y_train = np.zeros(len(P), dtype = int)
    for i in range(num_districts):
        for j in range(len(indk[i])):
            t = int(indk[i][j])
            Y_train[t] = i

    # --------------- Initial Classification and View of Districts ------------------ #
    # Set the step size in the mesh (large since data points are fairly spread out)
    h = 2500

    # Set number of neighbors to use (larger number seems to force regions to be contiguous)
    n_neighbors = 200

    # Set norm to use when computing distance (using euclidean norm, seems to work well enough)
    nrm = 2

    # Create color maps for the classes
    cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
    cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

    for weights in ['distance', 'uniform']:
        # we create an instance of Neighbours Classifier and fit the data.
        clf = neighbors.KNeighborsClassifier(n_neighbors, weights=weights, p=nrm)
        clf.fit(X_train, Y_train)

        # Plot the decision boundary. For that, we will assign a color to each
        # point in the mesh [x_min, x_max]x[y_min, y_max].
        x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
        y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
        Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        plt.figure(figsize=(15,15))
        plt.pcolormesh(xx, yy, Z, cmap=cmap_light)
        plt.fill([501000,600000,600000,501000], [201000,201000,700000,700000], 'w', edgecolor='none')

        # Plot also the training points
        plt.scatter(X_train[:, 0], X_train[:, 1], c=Y_train, cmap=cmap_bold, edgecolor='k', s=20)
        plt.xlim(xx.min(), xx.max())
        plt.ylim(yy.min(), yy.max())
        plt.title("3-Class classification (k = %i, weights = '%s')" % (n_neighbors, weights))

        image_name = 'Finalized-Districts/Initial-Images/Districts-' + str(district_number) + '-' + weights + '.p
        plt.savefig(image_name)

    plt.show()

    # Return nearest neighbors model which used uniform distances
    return clf

# ------------ Final classification of districts ------- #
def final_district_classification(P, clf, district_number):
    # ------------------ Use Nearest Neighbors Classifier to Predict district for each population point ---------
    # Our features will be the set of ordered pairs P
    X_train = P
    # Use the model to finalize districts
```

```
Y_dist = clf.predict(X_train)

# Set number of neighbors to use (larger number seems to force regions to be contiguous)
n_neighbors = 200

# Initialize Districts and Races arrays for storing classified points
Districts = [[] for i in range(num_districts)]
Races = [[] for i in range(num_districts)]
for i in range(num_districts):
    Races[i] = [[] for j in range(num_races)]

# Set points and race percentages in each district
for i in range(len(Y_dist)):
    for j in range(num_districts):
        if j == Y_dist[i]:
            Districts[j] = np.append(Districts[j], X_train[i])
            for k in range(num_races):
                Races[j][k] = np.append(Races[j][k], R[k][i])
            break;

# Set filename for writing data
file_name = 'Finalized-Districts/Race-Distributions/Districts-' + str(district_number) + '-race-distribution.
file = open(file_name, 'w')

# Print statistics for state population
print("-------- Percentage of each Race across State --------")
file.write("-------- Percentage of each Race across State --------\n")
for i in range(num_races):
    print(" Percentage of Race %i:           %lg" %(i, mu_R[i]))
    file.write(" Percentage of Race %i:           %lg\n" %(i, mu_R[i]))
    file.write

print("\n")

# Determine race means and population distributions of new districts and print
for i in range(num_districts):
    print("---------------- District %i -----------------" %(i))
    file.write("---------------- District %i -----------------\n" %(i))
    print(" Population:                     %lg" %(len(Districts[i])))
    file.write(" Population:                     %lg\n" %(len(Districts[i])))
    for j in range(num_races):
        temp = np.mean(Races[i][j])
        print(" Percentage of Race %i:           %lg" %(j, temp))
        file.write(" Percentage of Race %i:           %lg\n" %(j, temp))
        print(" Deviation from Mean for Race %i: %lg" %(j, abs(temp - mu_R[j])))
        file.write(" Deviation from Mean for Race %i: %lg\n" %(j, abs(temp - mu_R[j])))

# Close file
file.close()

# -------------------- Final Plot of Districts and Population Denisity Points ---------------------- #
# Set the step size in the mesh
h = 2500

# Create color maps for the classes
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

for i in range(2):
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
    y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
```

```
        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        plt.figure(figsize=(15,15))
        plt.pcolormesh(xx, yy, Z, cmap=cmap_light)
        plt.fill([501000,600000,600000,501000], [201000,201000,700000,700000], 'w', edgecolor='none')

        # Plot also the training points
        if i == 0:
            plt.scatter(X_train[:, 0], X_train[:, 1], c=Y_dist, cmap=cmap_bold, edgecolor='k', s=20)
            plt.xlim(xx.min(), xx.max())
            plt.ylim(yy.min(), yy.max())
            plt.title("3-Class classification (k = %i, weights = 'uniform')" % (n_neighbors))
            image_name = 'Finalized-Districts/Final-Images/District-' + str(district_number) + '-uniform.png'
        else:
            #ax2 = plt.add_subplot(111, aspect='equal')
            #for p in [
            #    patches.Rectangle((0, 0), 20000, 20000, fill=False),
            #    patches.Rectangle((20000, 0), 20000, 20000, fill=False),
            #]:
            #    plt.add_patch(p)
            image_name = 'Finalized-Districts/Final-Images/Districts-' + str(district_number) + '-no-pts.png'

        # Save image
        plt.savefig(image_name)

    plt.show()

    return Y_dist



# ----------------- Function for generating district data from county data for objective function ---------------
def county_to_district(df_florida, df_florida_header_list, num_districts, num_counties, county_array, Y_dist, dis
    # Generate number of counties per district
    X_counties = county_per_district(num_districts, num_counties, county_array, Y_dist)
    #print(X_counties)

    # Create matrix of values from data frame corresponding to features encoding population values
    X_pops = df_florida.drop(['County', 'Total; Estimate; Median Household income (dollars)',
                              'Native; Estimate; Median Household income (dollars)',
                              'Foreign born; Estimate; Median Household income (dollars)',
                              'Foreign born; Naturalized citizen; Estimate; Median Household income (dollars)',
                              'Foreign born; Not a U.S. citizen; Estimate; Median Household income (dollars)'], a

    # Create matrix of values from data frame corresponding to features encoing median values
    X_medians = df_florida[['Total; Estimate; Median Household income (dollars)',
                            'Native; Estimate; Median Household income (dollars)',
                            'Foreign born; Estimate; Median Household income (dollars)',
                            'Foreign born; Naturalized citizen; Estimate; Median Household income (dollars)',
                            'Foreign born; Not a U.S. citizen; Estimate; Median Household income (dollars)']].val

    # Generate population and median district data
    District_pops, District_medians = create_district_data(X_pops, X_medians, X_counties, num_districts, num_coun

    # ---------------- Convert Population values to percentages (configure inputs for neural network) -----------
    # Concatenate population and median matrices (containing district information)
    X_complete = np.concatenate((District_pops, District_medians), axis = 1)

    # Create data frame from district data
    df_florida_p = pd.DataFrame(data=np.int_(X_complete), index=range(X_complete.shape[0]), columns=df_florida_he

    # Update values in data frame to reflect percentages so they can be used in neural network
    # List of headers to exclude as they do not correspond to percentage values
    drop_list = ['Total; Estimate; Total population', 'Native; Estimate; Total population',
```

```
            'Foreign born; Estimate; Total population',
            'Foreign born; Naturalized citizen; Estimate; Total population',
            'Foreign born; Not a U.S. citizen; Estimate; Total population',
            'Total; Estimate; Median Household income (dollars)',
            'Native; Estimate; Median Household income (dollars)',
            'Foreign born; Estimate; Median Household income (dollars)',
            'Foreign born; Naturalized citizen; Estimate; Median Household income (dollars)',
            'Foreign born; Not a U.S. citizen; Estimate; Median Household income (dollars)']

# List of headers of df_fl with values representing percentages
percentage_list = [i for i in df_florida_header_list if i not in drop_list]

# Create list of header types used in converting raw population values to percentages
header_types = ['Total; Estimate;', 'Native; Estimate;', 'Foreign born; Estimate;',
                'Foreign born; Naturalized citizen; Estimate;', 'Foreign born; Not a U.S. citizen; Estimate;']

# Convert each value from raw value to decimal (representing percentage)
for mystr in percentage_list:
    for header in header_types:
        if header in mystr:
            # Set 'totalstr' to name of header for total population
            totalstr = header + ' Total population'
            # Convert percentage to value
            df_florida_p[mystr] = df_florida_p[mystr] / df_florida_p[totalstr]
            break;

# ----------------------- Convert raw population values to percentages ---------------------- #
pop_list = ['Active Democrat Voters', 'Active Republican Voters', 'Active Voters with No Party',
            'Native; Estimate; Total population',
            'Foreign born; Estimate; Total population',
            'Foreign born; Naturalized citizen; Estimate; Total population',
            'Foreign born; Not a U.S. citizen; Estimate; Total population']

# Initialize string for data frame column containing total population values per district
totalpopstr = 'Total; Estimate; Total population'

for mystr in pop_list:
    # Convert population to percentage of population in the district
    df_florida_p[mystr] = df_florida_p[mystr] / df_florida_p[totalpopstr]

# Drop total population percentage as it is not needed anymore
df_florida_p.drop([totalpopstr], axis = 1, inplace = True)


# ------------------- Convert median income values to percentages of $100000 ---------------- #
income_list = ['Total; Estimate; Median Household income (dollars)',
               'Native; Estimate; Median Household income (dollars)',
               'Foreign born; Estimate; Median Household income (dollars)',
               'Foreign born; Naturalized citizen; Estimate; Median Household income (dollars)',
               'Foreign born; Not a U.S. citizen; Estimate; Median Household income (dollars)']

for mystr in income_list:
    # Convert population to percentage of population in the district
    df_florida_p[mystr] = df_florida_p[mystr] / 100000

# Create matrix of district values to be used in objective function
X_district = df_florida_p.values

# Preview values
df_florida_p.head()

# Remaining values required for objective function
rem_vars = ['Republican Candidate Spending', 'Democrat Candidate Spending',
                  'Party in Office', 'Incumbent Republican Candidate Running',
                  'Incumbent Democrat Candidate Running', 'Candidate Party']
```

```
    # Number of scenarios to consider
    n_rows = 6

    # Initialize data frame with these variables
    df_variables = pd.DataFrame(data=np.zeros((n_rows,len(rem_vars))), index=range(n_rows), columns=rem_vars)

    # Add column for scenario and district
    df_variables['District'] = [1, 1, 2, 2, 3, 3]
    df_variables['Scenario in District'] = [1, 2, 1, 2, 1, 2]
    df_variables['Scenario Probability'] = [0.7, 0.3, 0.4, 0.6, 0.8, 0.2]
    df_variables['Variable in District'] = ['Democrat Candidate Spending', 'Democrat Candidate Spending',
                                            'Republican Candidate Spending', 'Republican Candidate Spending',
                                            'Incumbent Candidate', 'Incumbent Candidate']

    # Initialize values in each column
    df_variables['Republican Candidate Spending'] = np.array([1000000, 2000000, 1000000, 3000000, 2000000, 300000
    df_variables['Democrat Candidate Spending'] = np.array([1500000, 4000000, 2000000, 4000000, 3000000, 2000000]
    df_variables['Party in Office'] = [0, 0, 1, 1, 0, 1]
    df_variables['Incumbent Republican Candidate Running'] = [1, 1, 0, 0, 1, 0]
    df_variables['Incumbent Democrat Candidate Running'] = [0, 0, 1, 1, 0, 1]
    df_variables['Candidate Party'] = np.ones(n_rows, dtype = int)

    # Generate all possible scenarios as binary lists
    scenario_list = list(itertools.product(*[(0, 1)] * 3))
    n_scenarios = len(scenario_list)

    # Initialize array to hold scenario values
    X_scenarios = np.empty((n_scenarios, num_districts, len(rem_vars) + X_district.shape[1]))

    # Initialize array to hold probability that each scenario occurs
    p_scenarios = np.empty(n_scenarios)

    # Fill array with values for all possible scenarios
    for i in range(n_scenarios):
        # Rows from data frame required for current scenario
        s_rows = [0,2,4] + np.asarray(scenario_list[i])
        # Probability of scenario i occuring
        p_scenarios[i] = np.prod(df_variables[['Scenario Probability']].iloc[s_rows].values)
        # Temp is array of variables from districts for current scenario
        temp = df_variables.drop(['District', 'Scenario in District', 'Variable in District', 'Scenario Probabili
        # Prepend district data with variable data for X_scenarios
        X_scenarios[i] = np.concatenate((temp, X_district), axis = 1)

    #print(p_scenarios)
    #print(X_scenarios)

    # Initialize names of binary files to save
    Xname = 'Numpy-Array-Files/Input-scenarios-Districts-' + str(district_number) + '.npy'
    pname = 'Numpy-Array-Files/Probability-scenarios-Districts-' + str(district_number) + '.npy'

    # Save both arrays to binary files
    np.save(Xname, X_scenarios)
    np.save(pname, p_scenarios)

    # Return number of scenarios and upper bounds for democrat spending
    return n_scenarios, X_scenarios[:,:,1].flatten()



# ------------------- Functions associated with minimizing Campign Expendetures ---------------------------- #
# ------------------------------ Objective function --------------------------------- #
def f (party_spending):
    # ----- Initialize variables to be used in evaluation ----- #
```

```
    # Set keras model filename location
    #keras_model_filename = 'Keras-Models/bin_model.h5'   # binary classification model
    #keras_model_filename = 'Keras-Models/dem_model_short_new.h5'
    keras_model_filename = 'Keras-Models/nnet_model.h5'
    # Set X_scens to contain the input data for all problem scenarios
    X_scens = np.load('Numpy-Array-Files/Input_scenarios.npy')
    # Set p_scens to contain the probability data for all problem scenarios
    P_scens = np.load('Numpy-Array-Files/Probability_scenarios.npy')
    # Set n_scens to be the number of scenarios
    n_scens = len(P_scens)
    # Set n_districts to be the number of districts
    n_districts = X_scens.shape[1]
    # Array to store values from evaluating keras_model
    district_outputs = np.array((n_scens, n_districts))

    # ----- Import new party spending values into scenario data ----- #
    # Update X_scens to reflect any changes in party spending (in this case changing amount of democrat spending)
    for i in range(n_scens):
        for j in range(n_districts):
            # Assign (n_districts*i + j)th element of party_spending to X_scens[i,j,1] ('1' is 'Democrat Spending
            X_scens[i,j,1] = party_spending[n_districts * i + j]

    # ----- Set up keras model to evaluate function ----- #
    model = load_model(keras_model_filename)

    # ----- Compute objective value as expected value of probability democrat candidate will win ----- #
    # Initialize value to 0
    value = 0
    for i in range(n_scens):
        # Initialize temp to 0
        temp = 0
        # Evaluate keras model for current scenario data
        arr_temp = model.predict(X_scens[i])
        #print("arr_temp = ", arr_temp)

        # Sum all district values up in ith scenario
        for j in range(n_districts):
            # Add district_output value for jth district to temp
            #temp = temp + arr_temp[j][1]      # this line is using binary classification model
            temp = temp + arr_temp[j]        # this line if predicting percentage of votes

        # Multiply temp by probability scenario i occurs then add to value
        value = value + P_scens[i] * temp

    return -value



# ------------------- Gradient of Objective function ----------------------- #
def g (party_spending):
    # ----- Initialize variables to be used in evaluation ----- #
    # Set keras model filename location
    #keras_model_filename = 'Keras-Models/bin_model.h5'   # binary classification model
    #keras_model_filename = 'Keras-Models/dem_model_short_new.h5'
    keras_model_filename = 'Keras-Models/nnet_model.h5'
    # Set X_scens to contain the input data for all problem scenarios
    X_scens = np.load('Numpy-Array-Files/Input_scenarios.npy')
    # Set p_scens to contain the probability data for all problem scenarios
    P_scens = np.load('Numpy-Array-Files/Probability_scenarios.npy')
    # Set n_scens to be the number of scenarios
    n_scens = len(P_scens)
    # Set n_districts to be the number of districts
    n_districts = X_scens.shape[1]
    # Set n_features to be the number of features
    n_features = X_scens.shape[2]
```

```
    # Array to store values from evaluating keras_model
    district_outputs = np.array((n_scens, n_districts))

    # ----- Import new party spending values into scenario data ----- #
    # Update X_scens to reflect any changes in party spending (in this case changing amount of democrat spending)
    for i in range(n_scens):
        for j in range(n_districts):
            # Assign (n_districts*i + j)th element of party_spending to X_scens[i,j,1] ('1' is 'Democrat Spending
            X_scens[i,j,1] = party_spending[n_districts * i + j]

    # ----- Set up keras model to evaluate gradient ----- #
    model = load_model(keras_model_filename)

    # Initialize output tensor for neural network model
    OutTensor = model.output

    # List of input variables for Tensor
    InVars = model.inputs
    #print(InVars)

    # Compute Gradients of model with respect to input variables
    Gradients = backend.gradients(OutTensor, InVars)
    #print(Gradients)

    # Evaluate Gradients
    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())

    # ----- Compute objective value as expected value of probability democrat candidate will win ----- #
    # Initialize grad to zero vector of length n_scenarios * n_districts
    grad = np.zeros(n_scens * n_districts)
    for i in range(n_scens):
        for j in range(n_districts):
            # Evaluate gradients at values for scenario i and district j
            eval_gradients = sess.run(Gradients, feed_dict={model.input: X_scens[i][j].reshape(1,n_features)})
            # Extract derivatives of objective with respect to Democratic and Republican Spending
            #repub_spending_derivative = eval_gradients[0][0][0]
            #dem_spending_derivative = eval_gradients[0][0][1]
            # Set (n_districts * i + j)th component of gradient value from tensorflow
            grad[n_districts*i + j] = - P_scens[i] * eval_gradients[0][0][1]

    # Close tensorflow session
    sess.close()

    return grad



# ------------------------------ Batch Objective function (more efficient computation) -------------------------
def f_batch (party_spending, model, district_number):
    # ----- Initialize variables to be used in evaluation ----- #
    # Set keras model filename location
    #keras_model_filename = 'Keras-Models/bin_model.h5'   # binary classification model
    #keras_model_filename = 'Keras-Models/dem_model_short_new.h5'
    keras_model_filename = 'Keras-Models/nnet_model.h5'
    # Set filnames needed to load X and p data
    Xname = 'Numpy-Array-Files/Input-scenarios-Districts-' + str(district_number) + '.npy'
    pname = 'Numpy-Array-Files/Probability-scenarios-Districts-' + str(district_number) + '.npy'
    # Set X_scens to contain the input data for all problem scenarios
    X_scens = np.load(Xname)
    # Set p_scens to contain the probability data for all problem scenarios
    P_scens = np.load(pname)
    # Set n_scens to be the number of scenarios
    n_scens = len(P_scens)
```

```
    # Set n_districts to be the number of districts
    n_districts = X_scens.shape[1]
    # Array to store values from evaluating keras_model
    district_outputs = np.array((n_scens, n_districts))

    # ----- Import new party spending values into scenario data ----- #
    # Update X_scens to reflect any changes in party spending (in this case changing amount of democrat spending)
    for i in range(n_scens):
        for j in range(n_districts):
            # Assign (n_districts*i + j)th element of party_spending to X_scens[i,j,1] ('1' is 'Democrat Spending
            X_scens[i,j,1] = party_spending[n_districts * i + j]

    # ----- Set up keras model to evaluate function ----- #
    #model = load_model(keras_model_filename)

    # ----- Compute objective value as expected value of probability democrat candidate will win ----- #
    # Initialize value to 0
    value = 0
    for i in range(n_scens):
        # Initialize temp to 0
        temp = 0
        # Evaluate keras model for current scenario data
        arr_temp = model.predict(X_scens[i])
        #print("arr_temp = ", arr_temp)

        # Sum all district values up in ith scenario
        for j in range(n_districts):
            # Add district_output value for jth district to temp
            #temp = temp + arr_temp[j][1]      # this line is using binary classification model
            temp = temp + arr_temp[j]        # this line if predicting percentage of votes

        # Multiply temp by probability scenario i occurs then add to value
        value = value + P_scens[i] * temp

    return -value



# ------------------- Batch Gradient of Objective function ----------------------- #
def g_batch (party_spending, Gradients, sess, batch):
    # ----- Initialize variables to be used in evaluation ----- #
    # Set keras model filename location
    #keras_model_filename = 'Keras-Models/bin_model.h5'   # binary classification model
    #keras_model_filename = 'Keras-Models/dem_model_short_new.h5'
    keras_model_filename = 'Keras-Models/nnet_model.h5'
    # Set filnames needed to load X and p data
    Xname = 'Numpy-Array-Files/Input-scenarios-Districts-' + str(district_number) + '.npy'
    pname = 'Numpy-Array-Files/Probability-scenarios-Districts-' + str(district_number) + '.npy'
    # Set X_scens to contain the input data for all problem scenarios
    X_scens = np.load(Xname)
    # Set p_scens to contain the probability data for all problem scenarios
    P_scens = np.load(pname)
    # Set n_scens to be the number of scenarios
    n_scens = len(P_scens)
    # Set n_districts to be the number of districts
    n_districts = X_scens.shape[1]
    # Set n_features to be the number of features
    n_features = X_scens.shape[2]
    # Array to store values from evaluating keras_model
    district_outputs = np.array((n_scens, n_districts))

    # ----- Import new party spending values into scenario data ----- #
    # Update X_scens to reflect any changes in party spending (in this case changing amount of democrat spending)
    for i in range(n_scens):
        for j in range(n_districts):
```

```
            # Assign (n_districts*i + j)th element of party_spending to X_scens[i,j,1] ('1' is 'Democrat Spending
            X_scens[i,j,1] = party_spending[n_districts * i + j]

    # ----- Compute objective value as expected value of probability democrat candidate will win ----- #
    # Initialize grad to zero vector of length n_scenarios * n_districts
    grad = np.zeros(n_scens * n_districts)
    for i in batch:
        j = i % 3               # District number
        k = int((i - j)/n_districts) # Scenario number
        # Evaluate gradients at values for scenario i and district j
        eval_gradients = sess.run(Gradients, feed_dict={model.input: X_scens[k][j].reshape(1,n_features)})
        # Extract derivatives of objective with respect to Democratic and Republican Spending
        #repub_spending_derivative = eval_gradients[0][0][0]
        #dem_spending_derivative = eval_gradients[0][0][1]
        # Set (n_districts * i + j)th component of gradient value from tensorflow
        grad[i] = - P_scens[k] * eval_gradients[0][0][1]

    return grad


# ----------- Stochastic Gradient Descent for Objective function ---------- #
def stoch_grad_proj(f_batch, g_batch, Gradients, sess, x, bounds, batch_size, n_scens, line_search, model, dist_n
    # Generate indices to keep for gradient descent
    batch = np.random.choice(n_scens, batch_size, replace = False)
    #print("Current batch: ", batch)

    # ------------ Compute gradient values --------- #
    # Initialize gradient values
    #start = time.time()
    gradient = np.copy(g_batch(x, Gradients, sess, batch))
    #end = time.time()
    #print("Gradient Time elapsed: ", end - start)
    # Set components of gradient not in current batch equal to zero
    for i in range(n_scens):
        if i not in batch:
            gradient[i] = 0

    # ------------ Line Search (if requested) --------- #
    # Initialize alpha parameter for line search
    alpha = 1
    # Check if line search was requested
    if line_search == True:
        # Initialize parameter for line search
        c = 0.5
        # Perform armijo line search
        while f_batch(x, model, dist_number) - f_batch(x - alpha * gradient, model, dist_number) < - alpha * c *
            # Reduce alpha size and try again
            alpha = 0.5 * alpha

    # Update x value
    xnew = x - alpha * gradient

    # ------------ Gradient Projection --------- #
    # Project values of x back onto bounds if necessary
    for i in batch:
        # Check if xnew is less than lower bound
        if xnew[i] < bounds[i][0]:
            #print("below lower")
            # Set xnew to lower bound
            xnew[i] = bounds[i][0]
        # Check if xnew exceeds upper bound
        elif xnew[i] > bounds[i][1]:
            #print("exceeds upper")
            # Set xnew to upper bound
            xnew[i] = bounds[i][1]
```

```
    # ---------- Return new x value ------------ #
    return xnew

# For every county in our state, generate uniformly distributed points in county representing 0.1 percent of popu
# Set number of districts
num_districts = 3
# Set number of counties
num_counties = 37
# Set number of races
num_races = 2

# Generate uniformly distributed population data
county_pop = df_florida[['Total; Estimate; Total population']].values
county_pop = county_pop.flatten()
#county_pop = np.random.permutation(county_pop)
#print(county_pop)
grid_scale = 100000 # x and y dimensions of each county
pop_scale = 1000  # number of people each point on the plot corresponds to


# Determine total population
total_pop = int(sum(county_pop))

# Initialize arrays for x, y coordinates and race percentages of counties
x = []
y = []
county_array = []
R = [[] for i in range(num_races)]

# --------------- Extract race percentages for each county ------------------ #
race_list = ['Total; Estimate; RACE AND HISPANIC OR LATINO ORIGIN - One race - White',
             'Total; Estimate; RACE AND HISPANIC OR LATINO ORIGIN - One race - Black or African American']

# Initialize data frames containing race data
df_races = df_florida[race_list].copy()

# Convert each value from raw value to decimal (representing percentage)
for mystr in race_list:
    # Convert percentage to value
    df_races[mystr] = df_races[mystr] / df_florida['Total; Estimate; Total population']

# Create array containing race distributions
R_dist = df_races.values.T
# To use randomized race data, use the following command to generate R_dist
#R_dist = np.random.dirichlet(5*np.ones(num_races), num_counties).T

# Initialize dimensions for state
dimen = math.ceil(math.sqrt(num_counties))

# Populate the arrays x, y, and R using randomly generated data
county_num = 0

for i in range(dimen):
    # Initialize population distribution data
    for j in range(dimen):
        num_points = int(county_pop[county_num]/pop_scale)
        t1 = np.random.uniform(grid_scale*i,grid_scale*(i+1),num_points)
        t2 = np.random.uniform(grid_scale*j,grid_scale*(j+1),num_points)
        x = np.append(x, t1)
        y = np.append(y, t2)
        county_array = np.append(county_array, county_num*np.ones(num_points))
        # Initialize race distribution data
        for k in range(num_races):
            temp = np.ones(num_points)*R_dist[k][j]
```

```
            R[k] = np.append(R[k], temp)
        # Update counties done counter
        county_num = county_num + 1
        # Check if all counties have been populated
        if county_num >= num_counties:
            break;
    # Check if all counties have been populated
    if county_num >= num_counties:
        break;

# Convert each numpy array in R to a list
for k in range(num_races):
    R[k] = R[k].tolist()

# Generate mean of race percentages in state
mu_R = []
for i in range(num_races):
    mu_R = np.append(mu_R, np.mean(R[i]))

# Combine x and y into vector of ordered pairs
P = np.column_stack((x,y))

# Determine population per district
points_per_dist = int(len(P)/num_districts)

# Use the following point to generate random starting points for districts
#pt = np.random.uniform(0,600000,(num_districts,2))
#print(pt)

# Set of points for generating districts
pts = np.array([[[428596.73182419, 565472.09203069], [406481.47389631,  76214.63431997], [529037.777425,   301312
                 [[103815.07274117, 523223.20943294], [127784.24278143, 367946.5601808 ], [405265.50666261, 126535
                 [[232498.50976297, 333876.44175454], [434856.49421111, 536007.79553189], [304201.82051287,  51653
                 [[264530.56995353, 453923.23875946], [439665.38813067, 440157.66435593], [129802.05652538, 421065
                 [[577596.93197487, 221554.65871974], [393323.71962208, 576595.73225033], [356491.77699861, 193253
                 [[ 10069.19911696,  27137.21980555], [541840.3264807,    4792.29529087], [134325.7114619,  582525
                 [[345564.19987199, 529750.04305769], [354862.82145347, 245082.66827711], [473278.58737241,   7581
                 [[123099.25101014,   6655.89064585], [ 18575.57679205, 245189.24132337], [321515.0611377,  530000

#print(pts)

# Number or nearest neighbors to use in initial construction of districts
number_of_neighbors = 4

# Set point name
district_number = 0
i = district_number

# Loop over all initial district points and generate local minimums
for i in range(8):
    # Generate district and return x, y coordinates, indices of points in P and mean of races in each district
    xdk, ydk, indk, r_mean = construct_district(num_districts, number_of_neighbors, pts[i], P, R, 1)

    # Generate nearest neighbors model
    nn_model = initial_district_classification(P, indk, i)

    # Generate district from nearest neighbor classification
    Y_dist = final_district_classification(P, nn_model, i)

    # Using district information, compile County data into District data
    n_scenarios, budget_max = county_to_district(df_florida, df_florida_header_list, num_districts, num_counties,

    # ----- Compute local minimum for district ----- #
    # Set number of decision variables
    n_dec_vars = n_scenarios * num_districts
```

```
# Define bound constraints on the variables
budget_bounds = np.stack((np.zeros(n_dec_vars), budget_max), axis = -1)

# ----- Set up keras model to evaluate gradient ----- #
# Initialize tensorflow session first
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())

# Initialize keras model from training
keras_model_filename = 'Keras-Models/nnet_model.h5'
model = load_model(keras_model_filename)

# Initialize output tensor for neural network model
OutTensor = model.output

# List of input variables for Tensor
InVars = model.inputs
#print(InVars)

# Compute Gradients of model with respect to input variables
Gradients = backend.gradients(OutTensor, InVars)
#print(Gradients)

# Set filename for writing data
file_name = 'Finalized-Districts/Local-Minimizer/Districts-' + str(i) + '-local-minimizer.txt'
file = open(file_name, 'w')

# Find three different local minimums
for j in range(3):
    # Set initial guess
    print("District %i - Point %i" %(i,j))
    #x0 = np.random.uniform(0, 1, X_scenarios.shape[0] * X_scenarios.shape[1])
    x0 = (3-j) * np.ones(n_dec_vars) / 3
    # Run stochastic gradient projection on objective function
    #start = time.time()
    for k in range(2001):
        # Most of the time run the stochastic gradient projection with line search
        if k % 100 != 0:
            x0 = stoch_grad_proj(f_batch, g_batch, Gradients, sess, x0, budget_bounds, 5, n_dec_vars, True, m
        else:
            x0 = stoch_grad_proj(f_batch, g_batch, Gradients, sess, x0, budget_bounds, 5, n_dec_vars, False,
        # Print out values every 400th iteration
        if k % 400 == 0:
            # End timer
            #end = time.time()
            # Print current statistics
            #print("Time elapsed: ", end - start)
            #print("x_", k, ":", x0)
            print("f(x_", k, ") =", f_batch(x0, model, i))
            # Reset timer
            #start = time.time()

    # Make filename to save solution point
    local_min_name = 'Numpy-Array-Files/' + str(i) + 'local_minimizer' + str(j) + '.npy'

    # Save final iterate
    np.save(local_min_name, x0)

    # Convert values in final iterate to dollar amounts by multiplying by 4000000
    x_dollars = 4000000 * x0

    # Print statistics for state population
    file.write("-------- District %i - Solution %i --------\n" %(i,j))
    file.write("x    = ")
```

```
    for l in range(len(x_dollars)):
        file.write(str(round(x_dollars[l], 2)))
        file.write(", ")
    file.write("\n")
    file.write("f(x) = ")
    file.write(str(f_batch(x0, model, i)[0]))
    file.write("\n\n")

# Close file
file.close()

# Close tensorflow session
sess.close()
```

Preprocessing Data Using Pandas

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load in

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import statsmodels as sm
import sklearn as skl
import sklearn.preprocessing as preprocessing
import sklearn.linear_model as linear_model
import sklearn.cross_validation as cross_validation
import sklearn.metrics as metrics
import sklearn.tree as tree
import seaborn as sns
from sklearn.preprocessing import StandardScaler

# keras for machine learning tools
import keras
from keras.models import Sequential, Model
from keras.layers import Dense
from keras import backend as k
from keras.layers import Input

# Tensorflow for neural nerwork model gradients
import tensorflow as tf

# Import plotting tools
import matplotlib.pyplot as plt

# Load scikit's random forest classifier library
from sklearn.ensemble import RandomForestClassifier

# Import tools for visualizing random forest classification results
from sklearn.tree import export_graphviz
import pydot

# sklearn for splitting data into training and testing
from sklearn.model_selection import train_test_split

# scikit tool for support vector machines
from sklearn import svm

# sklearn for linear regression
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score

# Set random seed
np.random.seed(0)

# ------------------------------------------------------------------------------ #
# ------------------------ Importing and Cleaning Data ------------------------- #
# ------------------------------------------------------------------------------ #


# ------------------------ Import US Census Data ----------------------------- #

# Import data
training_data = 'IOWA-Congress115-CENSUS-DATA/ACS_16_5YR_S0501_with_ann.csv'

# Create data frame of all data from congress since 2010
df_115_all = pd.read_csv(training_data, header=1)
```

```
# ------------------------ Cleaning US Census Data ---------------------------- #

# Remove first two columns of data set as they are not necessary for fitting data
df_115_all = df_115_all.drop(['Id', 'Id2'], axis=1)

# Create list of all header names
header_names = list(df_115_all.columns.values)

# Create empty list of header names for data we are going to remove
MoE_headers = [];

# Loop over header names and identify all headers with the string "Margin of Error"
for header in header_names:
    if "Margin of Error" in header:
        MoE_headers.append(header);

# Print list to check if it worked
#print(MoE_headers)

# Remove all elements from data set that correspond to Margin of Error in data as we will not be using them for t
df_115 = df_115_all.drop(MoE_headers,axis=1)

# Preview updated data frame
#df_115.head()

# Create a list of all features
feature_list = list(df_115)

# Print list of feautres
#print(feature_list)

# Combining testing and training data to begin cleaning process
df_all = df_115

# Print shape of data frame containing all data
#print("Train shape:", df_all.shape)

# Check if data frame contains any null values
#df_all.isnull().values.any()


# ---------------------- Encode Categorical Features as Numerical ------------------ #

# Add new column for Congress Number
df_all['Congress Number'] = 0;

# Loop over all rows of data frame to update 'Congress Number' value in df_all
for i in range(df_all.shape[0]):
    for j in range(108,116):
        # Set mystr to Congress Number + th (e.g. 115th)
        mystr = str(j) + 'th';
        # Check if Congress Number appears in Geography in df_all
        if mystr in df_all.loc[i,'Geography']:
            # Update 'Congress Number' value to j in df_all
            df_all.loc[i,'Congress Number'] = j;
            break;

# Preview updated data frame
#df_all.head()



# --------------------- Encode Geographical Information --------------------------- #

# Add new column for State
```

```
list_of_states = ['Alabama', 'Alaska', 'Arizona', 'Arkansas', 'California', 'Colorado', 'Connecticut',
                  'Delaware', 'Florida', 'Georgia', 'Hawaii', 'Idaho', 'Illinois', 'Indiana', 'Iowa',
                  'Kansas', 'Kentucky', 'Louisiana', 'Maine', 'Maryland', 'Massachusetts', 'Michigan',
                  'Minnesota', 'Mississippi', 'Missouri', 'Montana', 'Nebraska', 'Nevada', 'New Hampshire',
                  'New Jersey', 'New Mexico', 'New York', 'North Carolina', 'North Dakota', 'Ohio',
                  'Oklahoma', 'Oregon', 'Pennsylvania', 'Rhode Island', 'South Carolina', 'South Dakota',
                  'Tennessee', 'Texas', 'Utah', 'Vermont', 'Virginia', 'Washington', 'West Virginia',
                  'Wisconsin', 'Wyoming'];

# Create dictionary for list of states where keys are state names and values are indices (starting with 0)
states_dict = dict(zip(list_of_states, range(50)))

# Initialize 'State' values in data frame all to 0
df_all['State'] = 0;

# Loop over all rows of data frame to update 'Congress Number' value in df_all
for i in range(df_all.shape[0]):
    for state in list_of_states:
        # Check if State name appears in Geography in df_all
        if state in df_all.loc[i,'Geography']:
            # Update 'State' value to state in df_all
            df_all.loc[i,'State'] = states_dict[state];
            break;

# Create column for 'Congressional District'
df_all['Congressional District'] = 0;

# Update value of 'Congressional District' for each entry
for i in range(df_all.shape[0]):
    for j in range(1,5):
        # Set mystr to 'District j'
        mystr = 'District ' + str(j);
        # Set checkstr to Current Congressional District in data frame
        checkstr = str(df_all.loc[i,'Geography']);
        # Check if Congress Number appears in Geography in df_all
        if mystr in checkstr:
            # Update 'Congress Number' value to j in df_all
            df_all.loc[i,'Congressional District'] = int(j);
            break;

# Drop the Geography column now that all information has been extracted
df_all.drop(['Geography'], axis=1, inplace = True)


# -------------------- Create shorter version of the data (only age, gender, race, education) ------------------ #

# Copy data frame to make shorter version of data
df_short = df_all.copy()

# Create list of header names for df_fl
iowa_header_names = list(df_short.columns.values)

# Create empty list of header names for data we are going to remove
CENSUS_headers = [];

# Headers to drop (everything excluding gender, age, race, education, and median household income)
drop_headers = ['HOUSEHOLD TYPE', 'MARITAL STATUS', 'LANGUAGE SPOKEN AT HOME', 'EMPLOYMENT STATUS',
                'OCCUPATION', 'INDUSTRY', 'EARNINGS IN THE PAST 12 MONTHS', 'INCOME IN THE PAST 12 MONTHS',
                'POVERTY STATUS IN THE PAST 12 MONTHS', 'POVERTY RATES FOR FAMILIES', 'ROOMS',
                'VEHICLES AVAILABLE', 'SELECTED CHARACTERISTICS', 'SELECTED MONTHLY OWNER COSTS',
                'GROSS RENT AS A PERCENTAGE', 'Owner-occupied housing units', 'occupants per room',
                'Median number of rooms', 'Average household size', 'HOUSING TENURE', 'ccupied housing units',
                'Average number of workers', 'Median earnings (dollars)', 'CLASS OF WORKER',
                'Civilian employed population 16 years and over', 'Average family size']
```

```
# Loop over header names and identify all headers containing any string in drop_headers
for header in iowa_header_names:
    for drop in drop_headers:
        if drop in header:
            CENSUS_headers.append(header);
            break;

# Print list to check if it worked
#print(CENSUS_headers)

# Remove all elements from data set that do not correspond to age, gender, race, education, or median income
df_short = df_short.drop(CENSUS_headers, axis=1)
# Preview data
#df_short.head()


# --------------- Convert percentages in data frame to decimals ---------------- #

# Create data frame which has percentages instead of raw population numbers
df_short_p = df_short.copy()

# Create list of headers with values representing percentages
iowa_raw_header_list = list(df_short_p.columns.values)

# List of headers to exclude as they do not correspond to percentage values
drop_list = ['Total; Estimate; Total population', 'Native; Estimate; Total population',
             'Foreign born; Estimate; Total population',
             'Foreign born; Naturalized citizen; Estimate; Total population',
             'Foreign born; Not a U.S. citizen; Estimate; Total population',
             'Total; Estimate; Median Household income (dollars)',
             'Native; Estimate; Median Household income (dollars)',
             'Foreign born; Estimate; Median Household income (dollars)',
             'Foreign born; Naturalized citizen; Estimate; Median Household income (dollars)',
             'Foreign born; Not a U.S. citizen; Estimate; Median Household income (dollars)',
             'Congress Number', 'State','Congressional District']

# List of headers of df_fl with values representing percentages
percentage_list = [i for i in iowa_raw_header_list if i not in drop_list]

# Create list of header types used in converting percentages to raw population values
header_types = ['Total; Estimate;', 'Native; Estimate;', 'Foreign born; Estimate;',
                'Foreign born; Naturalized citizen; Estimate;', 'Foreign born; Not a U.S. citizen; Estimate;']

# Convert each value from percentage to decimal
for mystr in percentage_list:
    for header in header_types:
        if header in mystr:
            # Convert percentage to value
            df_short_p[mystr] = df_short_p[mystr] / 100
            break;

# Additional features to drop as they are not required for training
additional_drop_types = ['SCHOOL ENROLLMENT - Population 3 years and over enrolled in school',
                         'EDUCATIONAL ATTAINMENT - Population 25 years and over']

# Drop additional features
for mystr in additional_drop_types:
    for header in header_types:
        # Initialize header of feature to drop
        totalstr = header + ' ' + mystr
        # Drop feature from data frame
        df_short_p.drop([totalstr], axis = 1, inplace = True)

# Preview data
df_short_p.head()
```

```
# Choose data to use in remainder of cleaning process (if using all data comment out next line)
df_all = df_short_p.copy()

# If using df_all instead of df_short_p for data the following two lines should not be commented out (contain NaN
#df_all.drop(['Foreign born; Not a U.S. citizen; Estimate; EMPLOYMENT STATUS - Population 16 years and over - In
#df_all.drop(['Foreign born; Estimate; EMPLOYMENT STATUS - Population 16 years and over - In labor force - Armed



# ---------------------- Import Voter Information Data -------------------- #

# Initialize list of names of data frames
df_voters_names = [];

# Create data frame containing voter registration data for all election data
for i in [2010, 2012, 2014, 2016]:
    # Update name to use for data frame
    df_name = 'dfVoters' + str(i);
    # Update list of names of data frames
    df_voters_names.append(df_name)
    # Update name of data file to import
    data_file_name = 'IOWA-VoterRegistration-District-DATA/IOWA-VoterRegistrtion-District-DATA-' + str(i) + '.csv
    # Create data frame with name 'dfVoters' + str(i)
    locals()[df_name] = pd.read_csv(data_file_name)
    # Drop 'District' column from voters data frame and append to census data
    locals()[df_name].drop(['District'], axis = 1, inplace = True)
    # Rename some columns in voters data frame
    locals()[df_name].rename(columns = {'Democratic Active':'Active Democrat Voters',
                                        'Rebuplican Active':'Active Republican Voters',
                                        'No Party Active':'Active Voters with No Party'}, inplace = True)
    # Merge voter data with data from census
    locals()[df_name] = pd.concat([locals()[df_name], df_all], axis = 1)

# Create dictionary for new data frame names
voters_names_dict = dict(zip([2010, 2012, 2014, 2016], df_voters_names))

# Display the data from one of the years
locals()[voters_names_dict[2014]].head()

# Initialize list of party names
party_names = ['Republican', 'Democrat']
party_dict = dict(zip(range(2), party_names))



# --------------------------- Import Candidate Data -------------------------- #

# Initialize empty data frames to contain all candidate data from 2010 to 2016
df_names = []

for i in [2010, 2012, 2014, 2016]:
    # Update name to use for data frame
    df_name = 'dfFeatures' + str(i);
    # Update list of names of data frames
    df_names.append(df_name)
    # Create data frame with name 'dfVoters' + str(i)
    locals()[df_name] = pd.DataFrame()

# Create dictionary
df_features_dict = dict(zip([2010, 2012, 2014, 2016], df_names))

# Create data frame containing voter registration data for all election data
for i in [2012, 2014, 2016]:
```

```
    # Update list of names of data frames
    df_voters_names.append(df_name)
    # Update name of data file to import
    data_file_name = 'IOWA-ElectionSpending-Outcome-DATA/IOWA-ElectionSpending-Outcome-DATA-' + str(i) + '.csv'
    # Create temporary data frame df_temp
    df_year = pd.read_csv(data_file_name)
    # Drop 'Elected Party' and 'District' from df_temp
    df_year.drop(['Elected Party', 'District'], axis = 1, inplace = True)
    # Create column for 'Party'
    df_year['Candidate Party'] = 0
    # Create column for 'Year of Election'
    df_year['Year of Election'] = i
    # Update header names to reflect which candidate data supports
    for j in range(2):
        # Set df_temp to df_year
        df_temp = df_year.copy()
        # Set party to party name
        party = party_dict[j]
        # Rename 'Party' entries so they reflect which party the data pertains to
        df_temp['Candidate Party'] = party
        # Reset indices of df_temp and voters_names_dict[i] before concatenating to avoid creating NaNs
        df_temp.reset_index(drop=True,inplace=True)
        locals()[voters_names_dict[i]].reset_index(drop=True,inplace=True)
        # Merge df_temp with census and voter data by column
        df_temp = pd.concat([df_temp, locals()[voters_names_dict[i]]], axis = 1)
        # Merge voter data with data from census by row
        locals()[df_features_dict[i]] = pd.concat([locals()[df_features_dict[i]], df_temp])

locals()[df_features_dict[2014]].head(8)

# Initialize empty data frames to contain all candidate data from 2010 to 2016
df_names = []

for i in [2010, 2012, 2014, 2016]:
    # Update name to use for data frame
    df_name = 'dfTarget' + str(i);
    # Update list of names of data frames
    df_names.append(df_name)
    # Create data frame with name 'dfVoters' + str(i)
    locals()[df_name] = pd.DataFrame()

# Create dictionary
df_target_dict = dict(zip([2010, 2012, 2014, 2016], df_names))

# Create data frame containing target election results for all elections
for i in [2010, 2012, 2014, 2016]:
    # Update list of names of data frames
    df_voters_names.append(df_name)
    # Update name of data file to import
    data_file_name = 'IOWA-Votes-DATA/IOWA-Votes-DATA-' + str(i) + '.csv'
    # Create temporary data frame df_temp
    df_year = pd.read_csv(data_file_name)
    # Update header names to reflect which candidate data supports
    for j in range(2):
        # Set party to party name
        party = party_dict[j]
        # Set string to column of interest
        columnstr = 'Percentage of Votes for ' + party + ' Candidate'
        # Set df_temp to df_year
        df_temp = df_year[columnstr]
        # Rename columnstr to 'Percentage of Votes'
        df_temp.rename(columns = {columnstr:'Percentage of Votes (Republican then Democrat)'}, inplace = True)
        # Reset indices of df_temp before concatenating to avoid creating NaNs
        df_temp.reset_index(drop=True,inplace=True)
        # Merge target data with all targets for year
```

```
        locals()[df_target_dict[i]] = pd.concat([locals()[df_target_dict[i]], df_temp])
    # Create column for 'Year of Election'
    locals()[df_target_dict[i]]['Year of Election'] = i
    # Rename columnstr to 'Percentage of Votes'
    locals()[df_target_dict[i]].rename(columns = {0:'Percentage of Votes (Republican then Democrat)'}, inplace =



# -------------------- Data Frames for Features and Target ------------------------- #

# Initialize data frames
df_features = pd.DataFrame()
df_targets = pd.DataFrame()

# Create feature and target data frames
for i in [2012, 2014, 2016]:
    df_features = pd.concat([df_features, locals()[df_features_dict[i]]])
    df_targets = pd.concat([df_targets, locals()[df_target_dict[i]]])

# Display all features and targets for elections from 2012, 2014, and 2016
#df_features.head(24)
#df_targets.head(24)

# ----------- Configure data for modeling the percentage of votes earned using neural network ------------ #
# Create data frame of features and targets
df_complete = pd.concat([df_features.drop(['Year of Election', 'State', 'Congressional District', 'Congress Numbe
                        df_targets.drop(['Year of Election'], axis = 1)], axis = 1)

df_complete['Party in Office'] = df_complete['Party in Office'].replace(party_names, range(2))
df_complete['Incumbent Republican Candidate Running'] = df_complete['Incumbent Republican Candidate Running'].rep
df_complete['Incumbent Democrat Candidate Running'] = df_complete['Incumbent Democrat Candidate Running'].replace


# ----------------------- Convert raw population values to percentages ---------------------- #
pop_list = ['Active Democrat Voters', 'Active Republican Voters', 'Active Voters with No Party',
            'Native; Estimate; Total population',
            'Foreign born; Estimate; Total population',
            'Foreign born; Naturalized citizen; Estimate; Total population',
            'Foreign born; Not a U.S. citizen; Estimate; Total population']

# Initialize string for data frame column containing total population values per district
totalpopstr = 'Total; Estimate; Total population'

for mystr in pop_list:
    # Convert population to percentage of population in the district
    df_complete[mystr] = df_complete[mystr] / df_complete[totalpopstr]

# Drop total percentage as it is not needed anymore
df_complete.drop([totalpopstr], axis = 1, inplace = True)


# ------------------ Convert median income values to percentages of $100000 ---------------- #
income_list = ['Total; Estimate; Median Household income (dollars)',
               'Native; Estimate; Median Household income (dollars)',
               'Foreign born; Estimate; Median Household income (dollars)',
               'Foreign born; Naturalized citizen; Estimate; Median Household income (dollars)',
               'Foreign born; Not a U.S. citizen; Estimate; Median Household income (dollars)']

for mystr in income_list:
    # Convert population to percentage of population in the district
    df_complete[mystr] = df_complete[mystr] / 100000


# --------------- Convert campaign spending values to percentages of $4million ------------- #
df_complete['Republican Candidate Spending'] = df_complete['Republican Candidate Spending'] / 4000000
```

```
df_complete['Democrat Candidate Spending'] = df_complete['Democrat Candidate Spending'] / 4000000


# ---------------- Create data frames for Republicans and for Democrats -------------------- #
# Republican dataframe
df_republican = df_complete.copy()
df_republican.reset_index(drop=True,inplace=True)
# Democrat dataframe
df_democrat = df_complete.copy()
df_democrat.reset_index(drop=True,inplace=True)

# Initialize republican and democrat index drop lists
repub_drop_list = []
dem_drop_list = []

for i in range(df_complete.shape[0]):
    if 'Republican' != df_republican.iloc[i, 5]:
        repub_drop_list.append(i)
    if 'Democrat' != df_democrat.iloc[i, 5]:
        dem_drop_list.append(i)

df_republican.drop(repub_drop_list, inplace = True)
df_democrat.drop(dem_drop_list, inplace = True)

# Convert categorical features to nominal
df_complete['Candidate Party'] = df_complete['Candidate Party'].replace(party_names, range(2))
df_republican['Candidate Party'] = df_republican['Candidate Party'].replace(party_names, range(2))
df_democrat['Candidate Party'] = df_democrat['Candidate Party'].replace(party_names, range(2))

# Split data into training and testing sets for cross validation
train_data, test_data = train_test_split(df_complete, test_size=0.1)
train_repub, test_repub = train_test_split(df_republican, test_size=0.1)
train_dem, test_dem = train_test_split(df_democrat, test_size=0.1)

# View info on training and testing sets
print("------ Training Data Information ------")
train_data.info()

print("\n\n------ Testing Data Information ------")
test_data.info()

# ----- Format input and target data for age classification as numpy matrices -----
# Create numpy matricies for training data
X_train = train_data.drop(['Percentage of Votes (Republican then Democrat)'], axis=1).values
Y_train = train_data[['Percentage of Votes (Republican then Democrat)']].values
X_repub = train_repub.drop(['Percentage of Votes (Republican then Democrat)'], axis=1).values
Y_repub = train_repub[['Percentage of Votes (Republican then Democrat)']].values
X_dem = train_dem.drop(['Percentage of Votes (Republican then Democrat)'], axis=1).values
Y_dem = train_dem[['Percentage of Votes (Republican then Democrat)']].values

# Create numpy matricies for testing data
X_test = test_data.drop(['Percentage of Votes (Republican then Democrat)'], axis=1).values
Y_test = test_data[['Percentage of Votes (Republican then Democrat)']].values
X_test_repub = test_repub.drop(['Percentage of Votes (Republican then Democrat)'], axis=1).values
Y_test_repub = test_repub[['Percentage of Votes (Republican then Democrat)']].values
X_test_dem = test_dem.drop(['Percentage of Votes (Republican then Democrat)'], axis=1).values
Y_test_dem = test_dem[['Percentage of Votes (Republican then Democrat)']].values

# Update shape of Y_train and Y_test
Y_train = Y_train.reshape(Y_train.size,)
Y_test = Y_test.reshape(Y_test.size,)
Y_repub = Y_repub.reshape(Y_repub.size,)
Y_test_repub = Y_test_repub.reshape(Y_test_repub.size,)
Y_dem = Y_dem.reshape(Y_dem.size,)
Y_test_dem = Y_test_dem.reshape(Y_test_dem.size,)
```

```
# Divide Y_train and Y_test by 100 to get values between 0 and 1
Y_train = Y_train/100
Y_test = Y_test/100
Y_repub = Y_repub/100
Y_test_repub = Y_test_repub/100
Y_dem = Y_dem/100
Y_test_dem = Y_test_dem/100

# Confirming that the shape of the training matrices are correct
print("\n\nShape of X_train: ", X_train.shape)
print("Shape of Y_train: ", Y_train.shape)

# Confirming that the shape of the testing matrices are correct
print("\n\nShape of X_test: ", X_test.shape)
print("Shape of Y_test: ", Y_test.shape)

# Initialize number of features
n_features = X_train.shape[1]

# Check feature matrices for NaN
print("Checking X_train for NaNs")
for i in range(n_features):
    if np.isnan(X_train.T[i]).any():
        print("NaN in column:", i)

print("Checking X_test for NaNs")
for i in range(n_features):
    if np.isnan(X_test.T[i]).any():
        print("NaN in column:", i)

print("Checking X_repub for NaNs")
for i in range(n_features):
    if np.isnan(X_repub.T[i]).any():
        print("NaN in column:", i)

print("Checking X_test_repub for NaNs")
for i in range(n_features):
    if np.isnan(X_test_repub.T[i]).any():
        print("NaN in column:", i)
```