Voter Turnout Prediction

Probabilistic Classification with GPR

Christian Riboldi - April 17, 2017

Table of Contents

The Dataset	3
The Problem	3
Exploration of the Dataset	4
Gaussian Process Regression	7
Algorithm Analysis	9

The Dataset

I acquired this dataset from a company called Optimus that does this sort of analysis for professional politicians. This set of data itself is originally from the L2 voter file combined with some commercial data. Because of all the in-house processing at this point Optimus would be the source of the data.

Anyone who is interested or invested in politics should be interested in this dataset and those similar to it. Voter turnout prediction software and analysis is a major industry in the political realm. Any professional political campaign will pay for this kind of analysis. Many companies perform these services through polling the current population, but those outcomes can be bias and inaccurate depending on the sample used, the questions asked, or even the company performing the analysis. Using machine learning algorithms on past voter turnout data is an objective way to approach this complex problem, that has the potential to get better results as it's training data increases year after year

The Problem

The question I'm asking is whether a voter will vote in a coming election or not. This problem is ultimately a binary classification problem that sorts potential voters into a prediction whether they will vote or not. However, I decided to solve this problem using Gaussian Process Regression for a number of reasons. While it's nice to have a specific binary value in the end for voting or not voting it's much more interesting to have a probability for how likely the voter is going to vote. One reason for this regressive approach is that with a probability in the future I could split these results into very unlikely, unlikely, likely, and very likely voters depending on their probabilities. Having a probability makes it easy to split the potential voters into further categories. Upon researching similar approaches to this problem this approach is called a supervised probabilistic classification algorithm.

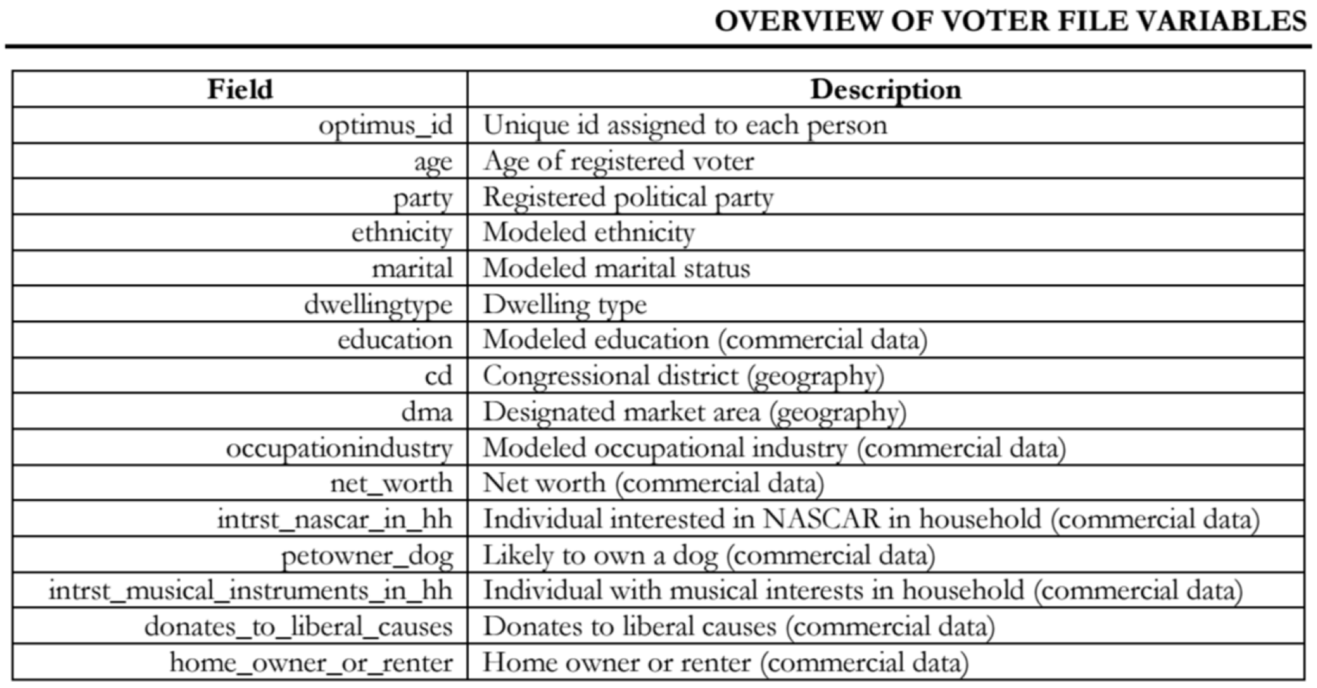
This problem is a supervised machine learning algorithm because the training data includes labels that shares each voter's voting history. The algorithm uses these labels to make intelligent predictions for voters with similar backgrounds.

In this problem, it's useful to have as much information about voting patterns historically as you can. I know, for instance that young voters that are younger than 22 weren't able to vote in past general elections because they were not 18. I don't really know how to solve this problem with the current data set because it seems a lot like a cold start problem to me. With additional data and testing I would be able to create a predictor for new and young voters based on their background information.

Another method that has been used to predict voter turnout is decision trees or random forests method. In this approach one would use the training data to create a decision tree based on attributes from a random subset of the data. The goal is to create a tree that matches the sample so that one could easily traverse the tree and reach a yes or no leaf node and be able to predict the turnout for one particular voter. One obvious challenge with this approach is that it’s easy to overfit the data, capturing in too much noise and not getting a very good decision tree. One way to combat this overfitting is to make a “forest” of decision trees and randomly select one tree to use for each sample. This randomness helps to prevent overfitting because each individual tree is a different representation of the data. I don’t know how random forests fare in terms of their RMSE but I can imagine that there is quite a bit of variability with the outcome depending on how effective each individual decision tree can get a random voter to the correct outcome.

Exploration of the Dataset

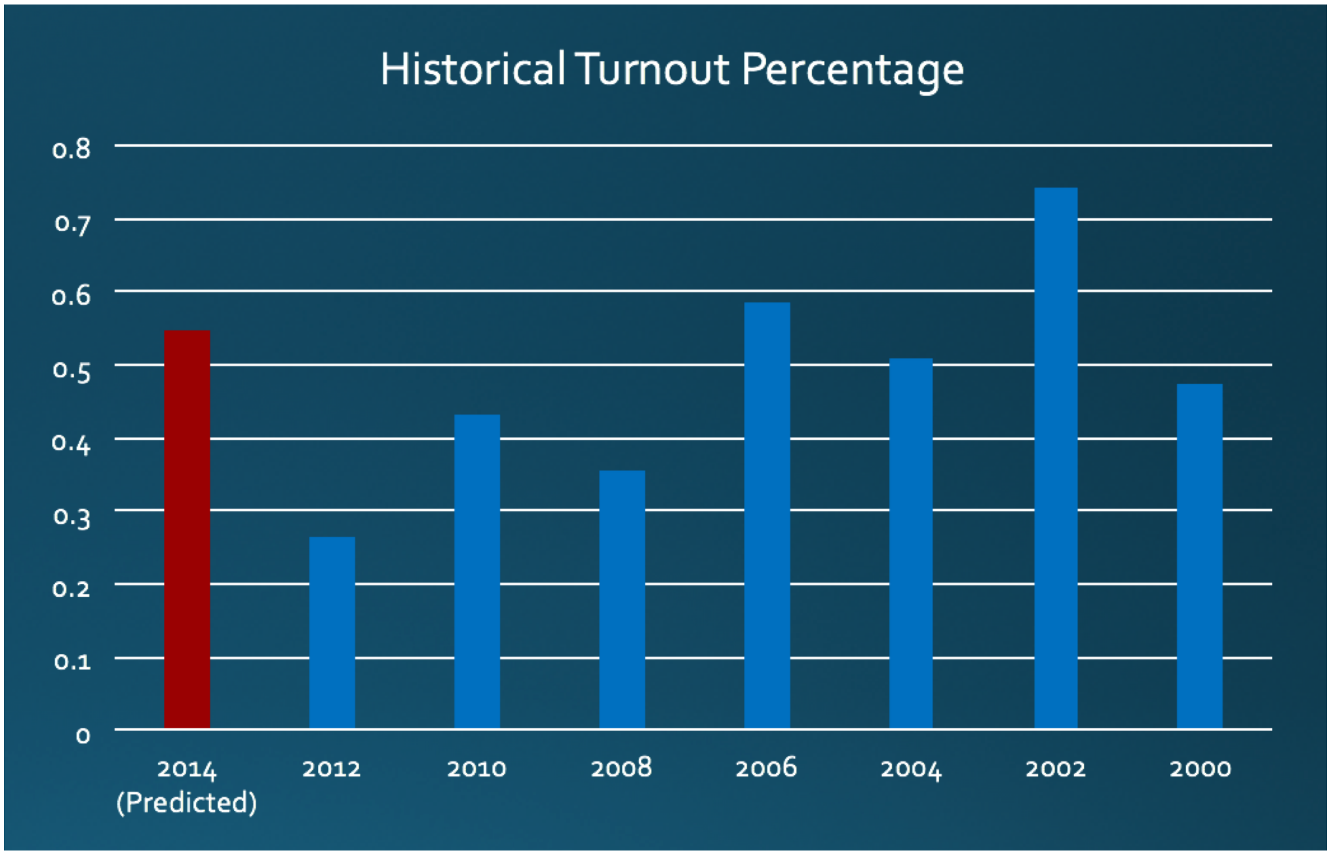
Before starting to code this is what I was able to find about the dataset. This first thing that I did was explore all of the labels and the data that I have access to. The data includes all of the following labels.

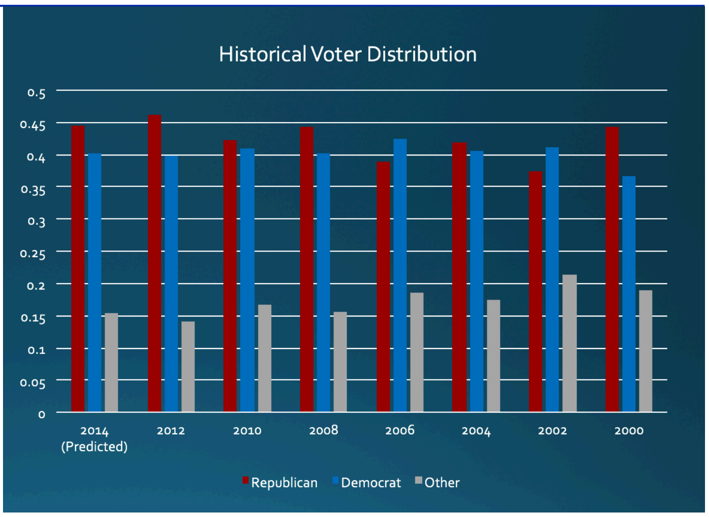


OVERVIEW OF VOTER FILE VARIABLES

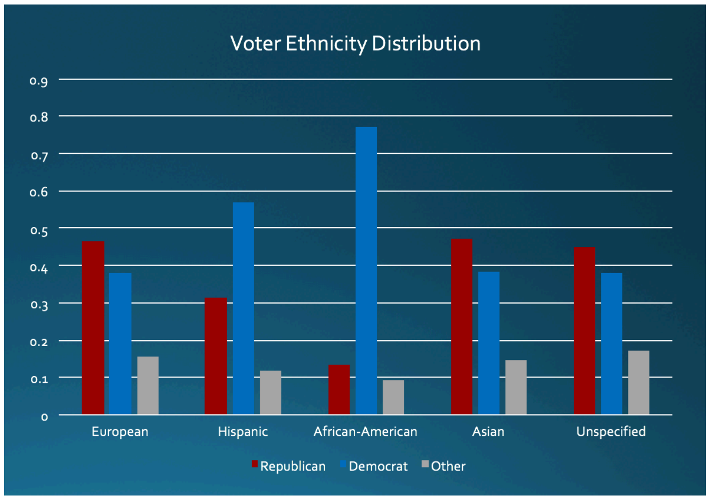
Field	Description
optimus_id	Unique id assigned to each person
age	Age of registered voter
party	Registered political party
ethnicity	Modeled ethnicity
marital	Modeled marital status
dwellingtype	Dwelling type
education	Modeled education (commercial data)
cd	Congressional district (geography)
dma	Designated market area (geography)
occupationindustry	Modeled occupational industry (commercial data)
net_worth	Net worth (commercial data)
intrst_nascar_in_hh	Individual interested in NASCAR in household (commercial data)
petowner_dog	Likely to own a dog (commercial data)
intrst_musical_instruments_in_hh	Individual with musical interests in household (commercial data)
donates_to_liberal_causes	Donates to liberal causes (commercial data)
home_owner_or_renter	Home owner or renter (commercial data)

The main thing that I did to explore the dataset was to create a few different charts that showed me who normally showed up to the voting booths. One of the most interesting pattern that I searched was the turnout rates of this same sample for each general election. I did see that every other election had a higher turnout than the election previous. This is pretty typical because there is a higher turnout for presidential elections than there is for a senator or house seat election. However, the surprising thing about this particular dataset was that the typical pattern for turnout was inverted with the presidential elections having a lower turnout than the primaries. I haven't really figured out why this patter was the way it was. I would have assumed it to be opposite. When I asked the people at Optumus about the data they actually clarified this pattern. They told me that some of the labels in the original had been changed. They mentioned specifically that the labels for general election and primary elections had been switched. It was pretty cool that I was able to find that pattern before finding out why it was there.

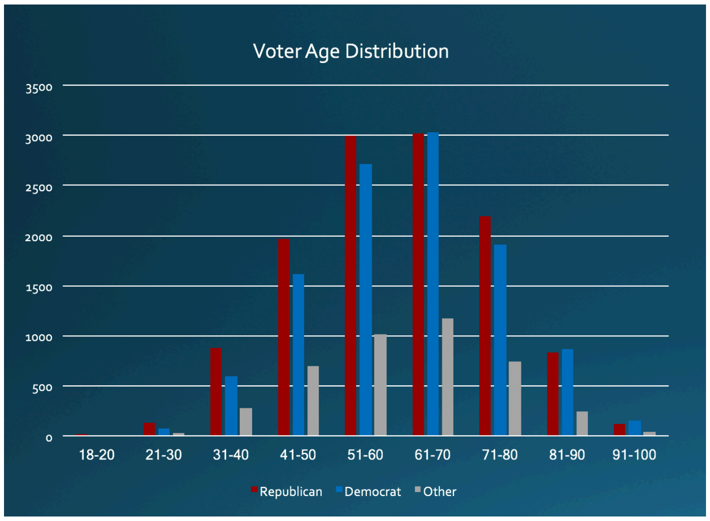




The second pattern that I thought was pretty interesting and that I was happy to see turn up in the predictions was that when considering ethnicity most ethnicities were fairly well balanced in this dataset leaning slightly towards Republican. This pattern was true among all ethnicities except Hispanics and African-Americans who leaned very heavily Democrat.



Another pattern that I found was when I plotted the age groups of those who turned up to vote. I saw that a voter was more likely to show up if he was quite a bit older with the mean around age 60. This result wasn't very surprising from the background information I know about how elections usually go. I was surprised to see that the data had such a well defined normal distribution. Especially because both major parties and even those of other party affiliations had the same Gaussian normal curve.



In that same vein of partisanship I wanted to see what the percentage of those who did show up year after year were from each party. This sample showed that again Republicans had a higher turnout than did Democrats and Other for all years except 2006 and 2002, when Democrats had a slight lead. This isn't very surprising because the dataset it from Utah, Nevada and a small sliver of Southern California. All in all this was a pretty normal finding, which I imagine wouldn't fluctuate too much election to election.

Gaussian Process Regression

Background

Gaussian Process Regression also known as kriging is used to infer continuous values from some given training data. Typically, the training data is put into a covariance matrix where a k function measures similarity between any two data points in the matrix. Because of the n^2 space complexity associated with this matrix it's difficult to create for large data sets. One solution for large scale data is to select or create landmark entries that are representative of a specific cluster of data. This would allow the covariance matrix to be much smaller while still striving to get a representative sample of entries from the data. Given the covariance matrix and a set of test data points estimation of each point is a simple matter of performing a few linear algebra operations of the matrix, the training values, and two matrices that compare the test data to each landmark column of the covariance matrix. Gaussian Process Regression has many applications in statistics and machine learning. Although this algorithm result in continuous values, those values could be probabilities which can then be used to categorize your predictions. This application is called Probabilistic Classification and is the approach we're using for this project.

Model

This version of Gaussian Process Regression doesn't use a specific model. The most modeling that takes place is when I split all the data points into different clusters. In order to cluster the data I used the sklearn.cluster library, and had to convert each of the data entries that I wanted to cluster into numerical data. In order to do that I created a map from the number to the term that each number represented. I converted partisanship, ethnicity, marital status, and education all into a numerical representation of those values. Ultimately it didn't matter what values I used as long as it was a consistent mapping for the clustering algorithm.

My approach

As mentioned above I used a Probabilistic Classifier made of a large-scale Gaussian Process Regression algorithm. In order to train my algorithm, I first had to format my data. The first thing I did was remove any unused data from my set. Ideally, I would use every entry but there were a number of columns that I either didn't think would be relevant or didn't want to focus on for this algorithm, some of these categories include is there an individual interested in music in the household, or is there an individual interested in NASCAR.

Once I had the basic columns that I wanted to use I needed to convert each of those values into a numerical data point so that the sklean clustering algorithm could use a comparator to cluster the data. As mentioned above I used a couple of maps and list comprehension to accomplish these conversions. I also created a new column called that calculated the probability that one particular voter showed up to the past general elections.

After cleaning the data, it was a simple matter of running it through the clustering algorithm. Once I had my clusters I created a representative landmark by taking one random entry in the cluster and setting its general election probability to the mean of it's corresponding cluster.

With the landmarks representing the data I was able to create the covariance matrix by simply running them through the k function that would compare two entries. One of the major improvements that I made over my lab in class was that I figured out how to parallelize the matrix creation using numpy meshgrids. It was one of the things that took the longest, but it was definitely worth the effort because I was able to run my entire algorithm in about 30 minutes, compared to several hours it would have taken before with the same number of landmarks.

Training and Test

Originally, I used all of my data as training data because I was trying to predict future turn out rather than improve upon the data that I already had. However, after the last lab I realized the huge benefit that creating test data for the RMSE formula. Once I was able to create that formula I was better able to tell which formula changes increased and decreased my accuracy. In order to make that split I decide to ask Optimus for the actual user data, I didn't know if they would send it to me but I figured it would be worth a shot. If they didn't send me the data I would have moved my prediction engine back to the previous election in order to simulate not knowing the future, while having the actual turnout numbers. When they got back to me they said that they didn't have the actual results anymore so I did push the predictions one general election. I decided to try and predict the 2012 general election instead of the 2014 general election for which I didn't have the results.

Algorithm Analysis

My first RMSE for this algorithm was .2666. At first I thought that was a really great result because the worst case scenario would be a solid 1 and I assumed that a random scenario would be about a .5. However, when I actually did create the completely random assignments I was a little disappointed because it consistently performed at a .23 average. I think that the best thing that I would be able to do to improve this number would be to play around with the k function, but it's one of those things that takes about 30 minutes to run. I'm able to create the covariance matrix in under 5 minutes, but calculating mu_test takes almost 30 minutes every time. That being said, I did decide to iterate on my design to try and tweak my k function. When I started changing my k function I started getting worse numbers (.323 for example), but in between each change I went back to my original formula. During one of my runs with my original formula I was able to get my numbers down to 0.19482 which I was happy with. To me that means that my random clusters probably has a larger influence on the accuracy than does my k function. In the future one way that I could test that theory is to create fixed clusters and then adjust my k function, because as it is right now it's not an effective measure of which k function is better and which is worse.

Overfitting the data is always a concern when using Gaussian Process Regression. Because I'm using all of my data to both train and validate that could be a real issue. However, I think that I'm okay because of how I create my covariance matrix. My covariance matrix isn't a basic random subset of the data, rather it tries to be a representative subset by using clusters. I don't know how the clustering algorithm works but I do know that it does create different clusters each time. So as long as I'm creating these clusters each time while I tune my algorithm it's not very likely that I will be overfitting the data.

I think that there might be a greater risk in overfitting if the covariance matrix were a full comparison for all of the data points. That being said one solution to avoid overfitting in the future would be cross validation. Instead of using all of my data as training I could split it into training and testing data. Then I could use the testing data as a measure of how well the algorithm is doing with no way to actually overfit the data because it's completely separate data.

Once I was able to calculate the RMSE, I iterated on this algorithm to tune the k matrix. In order to avoid overfitting I kept creating the landmarks each time, but that also made it very difficult to know if it was my changes to the k matrix that were making a difference or I just happened to have better landmarks on this iteration. The main progress I did make while iterating was that I was able to vectorize the matrix creation process. This led to drastic increases in performance when creating the covariance matrix, but it was still a struggle to do the final mu_test calculations because of the intense linear algebra calculations.

Obviously, I have a solution to this problem, but it's not necessarily the best solution. I was able to make a lot of progress towards a better solution, but there is still a lot of tinkering and adjusting that would have to happen in order for this algorithm to produce consistently accurate results. As with many machine learning algorithms I think that with more training data, and more iterations of tweaking I would be able to get this algorithm to work pretty effectively.