TEST PLAN

Our project can effectively be split into three parts: a data engineering component (represented by the "extractor," "statepoll," and "countrypoll" modules), a data science component (represented by the "regularizer" and "models" modules), as well as a frontend component (represented by the "simulator" and "interface" modules).

In general, when we used OUnit, our testing strategy was as follows:

1. Based on specifications from all of the associated ".mli" files, we would start by writing black box tests, while thinking of potential corner cases that might arise.

2. After our first round of black box testing, we would run bisect and analyze our coverage statistics. If our coverage was less than 90%, we would look at the Bisect report to create glass box tests, improving the coverage of our program.

For the modules tested with OUnit, this strategy demonstrates correctness of the programs–by combining black box tests and glass box tests, we ensure that behavior is exactly as how the user would expect it, while also ensuring that the entirely of our code is behaving as us (the implementers) expect.

The data engineering component was directly tested with OUnit. The "extractor" module was designed to function as a means of processing user-fed data into data that will be processed by our data science component, so testing it with dummy data and OUnit was fairly simple. Furthermore, the "statepoll" and "countrypoll" modules–which convert extracted data to the custom data structures that we defined–also were designed to make unit testing the best option.

It is worth mentioning that the "save_data_locally" method in countrypoll had a component of manual testing involved. The test designed using OUnit was to ensure that calling the method would not throw any exceptions but, in order to actually ensure that data was formatted the way we intended, there was an element of manual testing.

The data science component of the project comprises much of the simulation logic. We have 2 models that we use for simulation: the first is the Naive Bayes Model, and the second is the Logistic Regression model. Both are implemented natively by us without the usage of any library functions. Hence, we must test all components of these models. For example, for each subcomponent of the logistic regression operation (including computing gradients, running gradient descent, and making an inference), we compare outputs from the model against hand calculations, in addition to assessing properties of the outputs. This proved to be helpful for validating the correctness of common yet complicated algorithms

like gradient descent: we knew what the output was supposed to be since we could run calculations by hand (or with another objectively correct gradient descent program), and we also knew what the output was supposed to look like in general. Through this manner, we were able to test several aspects of the ML models thoroughly and attain confidence that the models were indeed working correctly. The data science portion of the project was tested automatically using OUnit, and the above testing plan specifications apply, which allowed us to maximize coverage. The automated testing code is located in "models_tests.ml".

Another core portion of the data science part of this project are the "regularizer" modules, which add randomness to the simulation logic for the Naive Bayes model. Much of the regularizer is implemented in the "do_computations" function, which does not take any inputs and therefore is challenging to test directly. Instead, we observe the performance of the regularizer through manual testing of the election simulator as a whole. We observed after running 50 different simulations with the Naive Bayes model that outcomes varied to a healthy degree, with different candidates winning in different simulations and the vote tally between candidates changing a lot. At the same time, however, the model still usually behaves according to high-level expectations (for example, it is very rare that a hard "blue" state like California gets allocated to Donald Trump). Because Naive Bayes is deterministic, we can conclude that the regularizer is indeed doing its job of adding randomness to the simulation. Hence, we conclude that this portion of the code is correct. Some portions of the regularizer were tested automatically with OUnit, and these tests are available in "regularizer_tests.ml".

There were more modules tested through manual testing. These are the "file", "interface" "local", "model" and "simulator" modules. These modules were tested through the use of main.ml, where the testing inputs were inputted through the terminal. These modules weren't practical to test through OUnit due to many functionalities requiring tedious file inputs and outputs. Given the flexibility that the frontend demands in regards to user input, unit testing was not feasible, so our strategy in regards to testing our frontend component was centered around manual testing. In these cases, we focused carefully on taking all the possible decisions a user could take, and ensuring that no exceptions were thrown. We even went ahead to violate frontend specifications, posing as a somewhat destructive user, and found that we were able to handle these cases well.

Training Data for the elections, which is under `data-extraction/state-data`, is sourced from the "U.S. President 1976–2020" dataset: https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/42MVDX

Overall, our extensive testing led to approximately 88.32% line coverage in bisect. This strong metric leads us to conclude that our program is, as a whole, according to specifications and correct.