NAU NORTHERN ARIZONA UNIVERSITY
School of Informatics, Computing, and Cyber Systems

Software Testing Plan

Version 1

Date: November 8th, 2024
Team Name: FairyMander
Sponsor: Bridget Bero
Professor: Isaac Shaffer
Mentor: Vahid Nikoonejad Fard

Team Members:
Izaac Molina (Team Lead)
Dylan Franco
Jeysen Angous
Sophia Ingram
Ceanna Jarrett

Table of Contents

1 Introduction **3**
2 Unit Testing **4**
2.1 District Generator 4
2.1.1 .__init__() 4
2.1.2 ._load_state_gdf() 5
2.1.3 ._get_random_partition() 6
2.1.4 ._generate_maps() 6
2.1.4 ._run() and ._run_and_save() 7
2.2 Fairness Metrics 8
2.2.1 calc_*() 8
2.2.2 get_metric_dict() 10
2.3 Folium Converter and Pulling Current Congressional Districts 11
2.3.1 map_to_folium() 11
2.3.2 get_curr_district_file() 11
3 Integration Testing **12**
3.1 District Generation, Analysis, and Conversion tests 12
3.1.1 District generation and comparison between two generated plans 12
3.1.2 Comparison between generated district and current district plan 13
4 Usability Testing **13**
4.1 Background of End Users 13
4.2 Novelty of Product 14
4.3 Consequences of Bad Design 14
4.4 Testing Plan 14
6 Conclusion **15**

1 Introduction

U.S. congressional districts are used to elect members for the House of Representatives. As part of this process, each state must draw their own congressional district lines, updating them at minimum every 10 years following the release of the decennial census. Each state individually decides how the lines will be drawn, and in most states, the drawing is done by the state legislature. This often results in an issue in the redistricting process known as gerrymandering. Gerrymandering is the act of drawing district lines in a way that benefits certain political parties; this leads to underrepresentation and political manipulation. Representatives have the power to manipulate electoral outcomes during redistricting, thus guaranteeing their electoral success. This power poses a threat to our democracy, having the potential to make voting obsolete. In search of a more widespread solution, our team has developed an algorithm for generating voting districts. We aim to also complete a user-friendly website to educate citizens on how congressional districts can be created fairly by utilizing our algorithm. The results of running our algorithm on each state will be added to this website along with a description of what makes them fair. Overall, we are confident that this algorithm and website will provide a comprehensive and understandable way to promote civic engagement in the redistricting process.

As with any good software, we aim to test our product using standard software testing techniques. In software testing, there are three main kinds of tests: unit tests, which aim to test a highly specific functionality of the product, integration tests, which test systems within the product and how different components work together, and usability tests, which test how user friendly and accessible the product is. When combined, these techniques provide a way to assure a high quality product that fulfills the requirements and maintains high user friendliness.

We aim to implement these techniques to test our algorithm (and the package that surrounds it) as well as our website. A unit testing suite will be developed for our algorithm, fairness module, and folium converter. We will also implement an integration test for the general processing of our package, that is, the upload of state data, a proposed district plan via our algorithm based on this state data, and an interactable map output based on this plan. Finally, we will conduct usability testing for our website, as it is critical our site is responsive, easy to use, and understandable at a 5th grade reading level.

This testing regime fulfills our goal of developing a redistricting algorithm and presenting the results of this algorithm in an accessible manner. In testing the accuracy of our metrics and algorithm process, we are ensuring that we are delivering high quality results based on long withstanding metrics used to evaluate district fairness. On top of this, the nature of this project necessitates extensive usability testing to ensure that it can effectively reach a wide audience to inform and promote civic engagement. In this document, we outline our testing plan and how it will ensure the success and quality of our product.

2 Unit Testing

Unit testing is the "lowest" level of software testing, and consists of testing individual functionalities within a piece of software. For example, the testing suite for a calculator application would probably include separate unit tests for adding, subtracting, multiplying, etc. It is also important to note some key terms related to unit testing, such as equivalence classes, which are t, boundary values, which are cases at the various extremes of possible input, and coverage, which is a metric that determines how "covered" the codebase is by its unit tests. Overall, these tests are critical as they allow developers to ensure that key functionalities of the product are working as they should be.

Before moving on, we would like to note some key terms related to unit testing:

- **Equivalence (Eq.) classes:** test cases that belong in one "group" due to their similar conditions.
- **Boundary values:** test cases at the extremes of possible input.
- **Coverage:** a metric used to determine how "covered" a codebase is based on its unit testing suite. A codebase with better coverage is considered to be better tested.

These terms will be used throughout our discussion of unit testing, and are key for communicating our unit testing plans.

For our unit testing purposes, we will be using pytest, a common library used for testing python applications. This package is a standard in python software development, and perfectly fits our testing needs. It offers a robust suite of assertions which can be used to test a flexible range of functionalities. Additionally, utilities in its "mocker" offer useful ways to mock dependencies between modules. Finally, pytest also has built in tools for measuring coverage, ensuring our codebase will reach the coverage necessary for our testing plan.

Please note, in cases of erroneous input (i.e., the user enters "Hello World" when prompted to enter a float value) we will be using python typing to force specific types upon calling a method, allowing us to handle this output.

2.1 District Generator

The core functionality of our application relies on a DistrictGenerator class, which is used to load state data and produce viable district plans based on this data. This class consists of several functionalities which will be unit tested thusly:

2.1.1 .__init__()

When initializing a district generator, each parameter will have an acceptable range of values it can hold. We will unit test the exception handling to ensure that each parameter is in range. The

following table breaks down each parameter and the values that will be used to cover its test cases.

Parameter	**Description**	**Selected Value - Eq. class**
prefix : str	The two letter state abbreviation (i.e. az, ut, ny, etc.)	‘az’ - prefix is in possible state prefixes list ‘foo’ - prefix not in possible state prefixes list
deviation : float	The acceptable population deviation between districts (i.e. 0.05 for 5% population deviation). Ranges from (0.005-0.1)	0.05 - acceptable deviation 0.11 - deviation too high 0.004 - deviation too low
steps: int	The number of steps to run the algorithm for. Must be >= 0	1000 - acceptable step count -1 - step count too low
num_maps: int	The number of maps to keep, keeps the top (num_map) maps. Ranges from (1-10)	3 - acceptable num maps 0 - num maps too low 11 - num maps too high
opt_metric: str	Name of the strategy that will be used to optimize the districts. Accepts “compact” and “competitiveness”	“compact”- acceptable metric “foo” - not in possible metrics

By doing this, we help prevent issues in other methods that could arise from improper values during initialization.

2.1.2 ._load_state_gdf()

Loads a geopandas dataframe containing census block data corresponding to the prefix used in initializing the DistrictGenerator (‘az’, ut’, etc.) from our dataset

Selected Values and Eq. classes:

Equivalence Class	**Selected Values**
Dataframe is found	DistrictGenerator(prefix = “az”, deviation=0.05, steps = 1000, num_maps = 3, opt_metric=”compact”

The function should always return a dataframe because we already checked if the state had a valid prefix in the __init__() method.

2.1.3 ._get_random_partition()

Creates an initial random partition (a gerrychain class) of the state so the algorithm can be started, where each partition is within the deviation specified in initializing the DistrictGenerator, and the # of partitions == the # of state districts. Tallies metrics in the partition depending on the opt metric selected

Selected Values and Eq. classes:

Equivalence Class	**Selected Values**
Standard random partition - compact	DistrictGenerator(prefix = “az”, deviation=0.05, steps = 1000, num_maps = 3, opt_metric=”compact”)
Standard random partition - competitiveness	DistrictGenerator(prefix = “az”, deviation=0.05, steps = 1000, num_maps = 3, opt_metric=”compact”)
Random partition with low deviation boundary value (tighter constraints)	DistrictGenerator(prefix = “az”, deviation=0.008, steps = 1000, num_maps = 3, opt_metric=”compact”)
Random partition on state with islands - ensure islands do not break the partitioning process	DistrictGenerator(prefix = “ma”, deviation=0.05, steps = 1000, num_maps = 3, opt_metric=”compact”)

It is worth noting that the output of this method is nondeterministic. Depending on the state as well as the “luck” of the user, some partition configurations are not possible with certain initial random draws of the map, but are with others. Since this is impossible to discreetly determine, we will not be testing for this in our suite.

2.1.4 ._generate_maps()

Uses the initial random partition to step through different map configurations, keeping the top num_maps for the given optimization metric

Selected Values and Eq. classes:

Equivalence Class	**Selected Values**
Compact	DistrictGenerator(prefix = “az”, deviation=0.05, steps = 1000, num_maps = 3, opt_metric=”compact”)
Competitiveness	DistrictGenerator(prefix = “az”, deviation=0.05, steps = 1000, num_maps = 3, opt_metric=”compact”)

Low num_maps boundary value	DistrictGenerator(prefix = “az”, deviation=0.05, steps = 1000, num_maps = 1, opt_metric=”compact”)
High num_maps boundary values	DistrictGenerator(prefix = “az”, deviation=0.05, steps = 1000, num_maps = 10, opt_metric=”compact”)

A list of the opt_metrics found during the simulation will be kept as the simulation runs, which will be used to determine if the method succeeded in keeping the top “num_map” maps.

2.1.4 ._run() and ._run_and_save()

The “conductor” of the DistrictGenerator class, _run() is a public method for generating the districts and returning them

Equivalence Class	**Selected Values**
Compact	DistrictGenerator(prefix = “az”, deviation=0.05, steps = 1000, num_maps = 3, opt_metric=”compact”)
Competitiveness	DistrictGenerator(prefix = “az”, deviation=0.05, steps = 1000, num_maps = 3, opt_metric=”compact”)
Low deviation boundary value	DistrictGenerator(prefix = “az”, deviation=0.008, steps = 1000, num_maps = 1, opt_metric=”compact”)
High deviation boundary values	DistrictGenerator(prefix = “az”, deviation=0.08, steps = 1000, num_maps = 10, opt_metric=”compact”)

The resulting maps will be evaluated to determine that a correct number of districts were created and that their populations are within the deviation. Similarly to random_partition, we must also note here that the result of this function is nondeterministic.

run_and_save() performs run(), but saves the result to the local file system, creating the directory if it does not exist. In the tests, run() will be called using the first equivalence class.

Equivalence Class	**Selected Values**
Directory doesn't exist, is created	run_and_save(“map_dir”, “map_prefix”)

Directory exists, created from previous test	run_and_save("map_dir", "new_prefix")

2.2 Fairness Metrics

A critical component of our package is the fairness metrics, which are used to evaluate the maps we generate. It is essential that these metrics are accurate, as they will be used to make key decisions related to which maps we will show on our website. As such, we will be particularly focusing on the accuracy of this module, as failure to do so will not only negatively impact our project quality, but would be actively harmful in spreading misinformation.

2.2.1 calc_*()

The title of this section, "calc_*" refers to the functions we have developed to calculate various redistricting fairness metrics from a geopandas dataframe.

For each of these functions, there will be two "base" district plans, one where there is one district, and one where there are four districts. This ensures that each function can handle edge cases for states with only one district, an important equivalence class/boundary value. Unless specified otherwise in the table, these will be the values for the plans. Note that "eth-" denotes the citizen voting age population of an ethnicity in that district:

```
one_district_plan = {
        'District': [1],
        'geometry': [Polygon([(0, 0), (1, 0), (1, 1), (0, 1)])],
        'party_rep': [600],
        'party_dem': [400],
        'eth1_eur': [800],
        'eth1_aa': [100],
        'eth1_esa': [50],
        'eth1_hisp': [30],
        'eth1_oth': [20]
}
multiple_district_plan = {
        'District': [1, 2, 3, 4],
        'geometry': [
          Polygon([(0, 0), (2, 0), (2, 2), (0, 2)]),
          Polygon([(2, 0), (4, 0), (4, 2), (2, 2)]),
          Polygon([(0, 2), (2, 2), (2, 4), (0, 4)]),
          Polygon([(2, 2), (4, 2), (4, 4), (2, 4)])
        ],
        'party_rep': [600, 400, 300, 700],
```

```
        'party_dem': [400, 600, 700, 300],
        'eth1_eur': [800, 700, 900, 600],
        'eth1_aa': [100, 150, 50, 200],
        'eth1_esa': [50, 30, 70, 20],
        'eth1_hisp': [30, 40, 20, 10],
        'eth1_oth': [20, 80, 10, 70]
}
```

We summarize the testing plans for each of these functions in the table below, with additional equivalence classes specified if necessary:

Function	**Description**	**Additional Eq classes - selected values**
calc_avg_polsby_popper	Measure for district compactness. Ratio between district area and perimeter.	None
calc_avg_reock	Measure for district compactness. Ratio between district area and area of minimum bounding circle.	None
calc_efficiency_gap	Measures political competitiveness based on wasted votes	Dem advantage - 'party_rep': [200, 400, 800, 450], 'party_dem': [800, 600, 200, 550] Rep advantage - 'party_rep': [800, 600, 200, 550], 'party_dem': [200, 400, 800, 450]
calc_lobsided margin	Measures political competitiveness using the difference between a party's average vote share and its median vote share across all districts	Dem advantage - 'party_rep': [400, 450, 800, 850], 'party_dem': [600, 550, 200, 150] Rep advantage - 'party_rep': [600, 550, 100, 150], 'party_dem': [400, 450, 900, 850] One party wins, Dem - 'party_rep': [400], 'party_dem': [600] (One party win, rep is covered by the base plan w/

		one district)
calc_dissimilarity_indices	Measures district minority representation, each index indicates how spread out the minority population is across the district plan	None

2.2.2 get_metric_dict()

The testing for this function is quite simple, as the function just executes all the calc_* functions on the given dataframe, returning the results in a dict. We will run the base district plans from section 2.2.1 and ensure the values are returned properly in a dictionary format.

2.2.3 compare_maps()

In this function, two dataframes are compared for which metric has “better” metrics, which happens by performing each metric calculation on both maps and comparing them. We will mock the output of get_metric_dict(), where one returned dictionary will have better values than the other. These mocked dictionaries are as follows

```
map_a = {
    'Avg Polsby-Popper': 0.75,
    'Avg Reock': 0.65,
    'Efficiency Gap': -2.0,
    'Mean Median Difference': -1.5,
    'Lopsided Margin': 4.0,
    'Dissimilarity Indices': {
        'eth1_aa': 0.20,
        'eth1_esa': 0.25,
        'eth1_hisp': 0.30,
        'eth2_81': 0.18,
        'eth1_oth': 0.22
    }
}

map_b = {
    'Avg Polsby-Popper': 0.60,
    'Avg Reock': 0.50,
    'Efficiency Gap': -3.5,
    'Mean Median Difference': -2.0,
    'Lopsided Margin': 6.0,
    'Dissimilarity Indices': {
```

```
            'eth1_aa': 0.25,
            'eth1_esa': 0.28,
            'eth1_hisp': 0.35,
            'eth2_81': 0.20,
            'eth1_oth': 0.25
        }
}
```

Equivalence Class	**Selected Values**
Map 1 > Map 2	Map 1 = map_a, Map 2 = map_b
Map 2 > Map 1	Map 1 = map_b, Map 2 = map_a
Map 1 == Map 2	Map 1 = map_a, Map 2 = map_a

2.3 Folium Converter and Pulling Current Congressional Districts

Since these are both fairly small modules, they have combined in this section. We will unit test our folium converter, which produces an interactive map based for a district plan, as well as our functionality for pulling a state's current congressional district data.

2.3.1 map_to_folium()

Given a state prefix and a geopandas dataframe, converts the dataframe into a folium visualization. These data frames are quite large, so descriptions for the selected values will need to be generalized for the sake of brevity

Equivalence Class	**Selected Values**
Current district plan	Current AZ congressional district map
Generated district plan	AZ congressional district map generated from our DistrictGenerator module
Map has islands	Current MA congressional district map
Invalid state plan	state prefix = 'foo'

2.3.2 get_curr_district_file()

Given a state prefix, loads the current congressional district data for that state and returns it as a geopandas dataframe

Equivalence Class	Selected Values
State data exists	prefix = ‘az’
State data doesn’t exist	prefix = ‘foo’

3 Integration Testing

Integration testing involves testing the interactions between modules in a system to ensure they all work together properly. Taking our calculator example from earlier, while unit tests would look at specific functionalities like adding/subtracting/etc., an integration test might look like a simulated input of different operations (i.e. add, then subtract, then multiply …) to ensure that each operation is compatible with the others and that all operations function appropriately together. This is essential to ensuring a system functions as expected and that the system as a whole can achieve intended results.

Our application has a fairly straightforward system, involving uploading from our data, using this data to generate districts, evaluating the districts, then potentially saving them. As such, we do not require an overly robust integration test suite. Nevertheless, we will develop an integration test suite for this core system flow, with checks at each step to ensure an expected result is obtained. It is worth noting that due to the nature of our algorithm, the results are hard to exactly predict as each run will produce nondeterministic results depending on the initial algorithm configuration. Regardless, we will ensure that the data can be uploaded, then converted into a district plan within the given population deviation, properly evaluated for metrics, then saved into the file system.

3.1 District Generation, Analysis, and Conversion tests

In this section, we outline the integration tests we will use to test our python package functionality. This will be presented as a series of steps

3.1.1 District generation and comparison between two generated plans

The first tests will test “standard” district generation and evaluation, i.e. with middle-range population deviation and on an average sized state with no islands This will be ran with both “compact” and “competitiveness” as the optimization metric

1. Initialize DistrictGenerator(prefix = “az”, deviation=0.01, steps = 2000, num_maps = 1, opt_metric=”compact”)
2. DistrictGenerator.run() is called to generate districts
3. Run get_metric_dict on the generated district plan to get district metrics.
4. Repeat steps 1-3 using “competitiveness” for the opt_metric

5. Run compare_maps() to get the district comparison result.
6. Convert the results from both maps to a folium map using map_to_folium()

We will then run the same tests on “pa” to essentially “stress test” the system, as “pa” is a large, highly populated state.

3.1.2 Comparison between generated district and current district plan

Similar to above, but involves loading an comparing to a current district plan

1. Initialize DistrictGenerator(prefix = “az”, deviation=0.01, steps = 2000, num_maps = 1, opt_metric=”compact”)
2. DistrictGenerator.run() is called to generate districts
3. Run get_metric_dict on the generated district plan to get district metrics.
4. Load the current az district map
5. Run compare_maps() to get the district comparison result.
6. Convert the results from both maps to a folium map using map_to_folium()

4 Usability Testing

Usability testing is a crucial step in any software design process as it is focused on the interactions between the software itself and the end user while ensuring that end users can easily access the functionality provided by the software. Moreover, usability testing is used to evaluate the user experience with the website to ensure that the system meets the needs and expectations of its users. The primary goals of usability testing are to identify any usability issues, gather feedback regarding the user interface, and most importantly ensure the software meets the needs and expectations of the end users. To develop an extensive strategy for our software’s usability testing, the following important factors need to be taken into consideration.

4.1 Background of End Users

Our website “FairyMander”, whose goal is to encompass an educational aspect of the process of redistricting while making the website clear and easily understandable, is specifically designed for both the fifth-grade reading level and members of the committee. Given that the website’s aim is educational, many detailed steps have been taken into consideration in making the application’s written content suitable for the fifth-grade reading level. This results in not only helping current voters understand how redistribution is done along with gerrymandering, but it will also make the application useful in an educational context, enabling the youth to learn about our democracy. Lastly, our target audience consists of individuals who also want to view a side-by-side comparison of the Current Congressional Districts in addition to the FairyMandered

Algorithm with the differences in Demographic Distribution in a pie chart form and Fairness Metrics for district comparisons in all fifty states.

4.2 Novelty of Product

Our website is intended to teach individuals about the redistricting process along with gerrymandering. Unlike other web applications, FairyMander's goal is to focus on the educational aspect which is achieved through an interactive map when hovered over a state, the state's name, population, and districts are viewable. Furthermore, the website is designed for users to access any state's information with ease while providing them with beneficial educational opportunities and taking user friendliness and ease of navigation into account.

4.3 Consequences of Bad Design

Poor website design including poor navigation, slow-loading pages, broken links, bad content, and inconsistency will lead to many unintended consequences including errors, driving audiences away, and negative feedback. Therefore, is it extremely important that our web application is designed to be user-friendly while keeping in mind that each page should load quickly and function smoothly which most importantly includes fast loading speeds without any delays when hovering over or clicking on a state.

4.4 Testing Plan

The purpose of usability testing is to have an extensive understanding of the user experience with FairyMander, making sure it meets the needs of both the user and the client.

We have decided to have some friends of ours test the website's functionalities and provide us with feedback on what they liked and disliked about the website. The main purpose of the testing phase is to make sure everything on the website works and functions as expected. The first step of any testing process is to load the website and ensure that the landing page loads correctly and smoothly with all navigation links included. The tester then has the option to select a state from either the hamburger drop-down menu or an interactive map, alternatively by scrolling down the tester can read some basic information about the redistricting process and learn more information about it. The tester then selects a state of their choice and upon clicking on that state, they are provided with a side-by-side comparison of the current congressional districts and the FairyMandered algorithm. Scrolling down, a pie chart of each state is viewable as well along with Fairness Metrics for both current districts and our redrawn districts. Lastly, the tester has the option to select another state or can view the glossary where a more in-depth definition will be provided for a specific term.

6 Conclusion

To conclude, FairyMander is confident that our presented testing plan will result in a bug free, high quality product. As we have outlined throughout this document, our suite will thoroughly test the core functionalities of our product using unit and integration tests, as well as the ease of use of our product via usability testing. Our team has put great effort into ensuring that all equivalence classes and relevant systems have been covered in the described testing suite, along with connecting with human testers to ensure a highly usable product. In doing so, we will confirm that the product is effective, reliable, and presents redistricting plans in an approachable way. Testing is an essential part of the software development process, and we are excited to further prove the quality of our product through fulfilling this testing plan.