Group Project Report
Team 79
Pranay Raj Kapoor, Vipin Jain, and Jeff Grant

1. Additional Feature

Our additional feature returns the highest and lowest property market value for the zip code with the highest full vaccination rate per capita. We do this in four steps:

1. We uses the the getFullVaccinatedPerCapita() method from feature 2, which uses the data from the population.txt and covid_data (.csv or .json) datasets to create a map the zip codes analyzed as the key and the vaccination rate for that zip code as the value.
2. We then iterate over that map to find the zip code with the highest vaccination rate.
3. We iterate over the list created by our PropertiesDataReader and for each PropertiesData object in the list, we get the value for market value. We compare it to the highest and lowest market values for that zip encountered to that point (stored in the local variables *max* and *min*); if it is greater than the maximum or less than the minimum we assign the value to the appropriate variable
4. We create and return a HashMap with the zip code as the key and an ArrayList with the maximum and minimum market values as the value.

Because we used the modularity of our code to use a methods from feature 2 in feature 6, our testing of feature 6 also included our testing of feature 2. We also needed to test the functionality of our Reader classes for each data set:

For feature 2, we used JUnit tests to check the functionality of the creation of the map from the two lists of data using the method on smaller lists of population and covid data we created that included regular and edge cases.

For feature 6, we used JUnit tests to check the functionality of iterating over the map of zip codes to find the zip code with the highest vaccination rate and then iterating over a list of PropertiesData to match them to the zip code and compare their maximum and minimum values using a smaller map of zip code and a smaller list of PropertiesData objects we created with sample data that included regular and edge cases.

To test the PopReader we compared our results of reading the given file in feature 1 to the stated correct result given in the project descriptions. We also used JUnit tests with sample data that contained edge cases to check the size of the resulting map and that the populations were mapped to the correct zip codes.

To test the JSONReaderCovidData and the CSVReaderCovidData we used JUnit tests with sample data that contained edge cases to check the size of the resulting list and that the CovidData objects contained the correct zip codes, number of partially and full vaccinated residents, and timestamps. We also compared the size of the outputs of the JSONReaderCovidData and CSVReaderCovidData with the covid_data files to identify any discrepancies between them.

To test the PropertiesReader, we used JUnit tests with sample data that contained edge cases to check the size of the resulting list and that the PropertiesData objects contained the correct zip codes, market values, and livable areas.

2. Use of Data Structures

In our program we used the following data structures: TreeMap, ArrayList, and HashMap.

1. TreeMap

   We used a TreeMap in our implementation of PopulationReader. We chose a TreeMap because we wanted to assign the number of vaccinated (partial or full based on user input) to each zip code and because we knew that we would want to access the data in sorted order (by zip code), and because we wanted to associate the ratio of people fully vaccinated with each zip code. We considered creating a List or Set of objects in a new PopulationData class to store the zip code, population, and vaccination rates, as an ArrayList or HashSet can use get() with O(1) complexity, but decided that this would use up more memory and would not lend itself to being easily sorted. Similarly, although ArrayLists HashSets can add elements with O(1) time while the TreeMap adds elements with O(log n) time, there is no efficient way to sort the elements of an ArrayList or a HashSet (since there is no native sorting method for these data structures, the data would have to be put into a completely new data structure before it could be sorted). Thus, for our purposes in feature 2, we decided that a TreeMap was the most efficient data structure to use.

2. ArrayList

   For the PropertiesReader, we again used an ArrayList. We considered using a Map implementation here that would map a zip code to an Array of Lists of market values and livable areas, but we thought that doing this would best be left to the processing tier and therefore chose a List data structure. Additionally, getting values from a Map implementation is less efficient than getting values from a List. We again chose an ArrayList over a LinkedList because of its efficiency in using the get() method to iterate over the list, since we do not anticipate removing any items from the list.

3. HashMap

   For the JSONCovidDataReader and CSVCovidReader class, we decided to store the information from the CSV and JSON files as a HashMap. We wanted to use a Map so that we could store the data in a way where the zip code would be associated with a list of CovidData objects that contain lists of partial and full vaccination. We decided to use a HashMap instead of a TreeMap as the zip codes with which we are associating the data are already stored in a TreeMap created by PopulationReader, so adding an entry to a TreeMap O(log n), would be redundant and less efficient than adding to a HashMap O(1). Using a HashMap with an entry containing a list for each zip code is more efficient than

using a list of all CovidData objects, as in feature 2, we can get the value for the given key (zip code) in the map and iterate through just the CovidData objects that are associated with that zip code to get the total number of people vaccinated instead of iterating over all CovidData objects created by the JSONCovidDataReader or CSVCovidReader.