Education Investigation Final Report

Nadia Malik, Milena Stepanova, Joshua Nicholson
Github: https://github.com/nmalikn/FinalProject.git

1. **The goals for your project including what APIs/websites you planned to work with and what data you planned to gather (10 points)**

Our project group, Education Investigation, initially aimed to focus on educational APIs and the correlation between educational level, region, and crime. As such our initial list for APIs included the National Center for Education Statistics (NCES), National Census Bureau, and Crimeometer.

We initially wanted the NCES API to collect educational performance metrics including test scores, graduation rates, GPA, and demographics including student ethnicity and gender. For the National Census Bureau API we wanted to collect demographic data including income levels by ethnicity, poverty rates, and educational attainment. Lastly for the NAEP Crimeometer API we wanted to collect crime data including location of incidents, socioeconomic status of the location, and the ethnicity and socioeconomic status of the offenders.

Once this data was collected and stored in our database, we wanted to explore relationships between different factors: Correlate educational performance metrics (test scores, graduation rates) with demographic factors (race, income levels, educational level) to identify patterns of educational success. And secondly to analyze correlations between demographic factors and crime rates to understand socioeconomic influences on crime. Overall we would have investigated if there's a correlation between educational performance/demographics and crime rates, to determine potential links between education quality, socioeconomic factors, and crime levels in different areas.

2. **The goals that were achieved including what APIs/websites you actually worked with and what data you did gather (10 points)**

Due to unforeseen issues with the Crimeometer API, we switched out of the crime focus and further into education. We decided to focus on the goal of investigating the relationship between educational level, subject (math, english, history, science) scores, race, gender, and socioeconomic status for students in the state of Michigan specifically.

To do so we gathered data from three APIs. From Michigan's Open Data Portal (data.Michigan.gov) we retrieved information on the Michigan Per Capita Personal Income categorized by race and gender. This data provided us with insights into the income distribution among different demographic groups within Michigan. Next, we accessed subject-specific scores for Michigan students from The Nation's Report Card (NationsReportCard.gov). This included

scores across multiple subjects such as math, writing, history, and science, categorizing these scores by race and gender, allowing us to analyze academic performance disparities among different demographics. Lastly from the National Census Bureau API (api.census.gov), we accessed Michigan's socioeconomic information by congressional district, including income levels and poverty percentages, segmented by different racial groups. This data was crucial for understanding the socioeconomic context in which students were learning and the potential impact of economic factors on academic performance.

By gathering these specific datasets, we were able to conduct an analysis on the relationships between income, educational level, subject scores, race, gender, and socioeconomic status among students in Michigan. Our hypothesis was that lower test scores amongst certain student demographics, such as black students, who seemed to have lower scores from the Nation's Report Card, was due to worse socioeconomic conditions the students' faced. Through our graphs, which will be seen later in this report, we found that the Michigan per capita personal income was lower than the national level. We also found that there appears to be a slight correlation between higher proportion of non-white residents in a county and poverty rate in Michigan. Once we found that black male students and black female students in Michigan scored lowest on the Nation's Report Card for the state, we determined that this was because of the previous economic indicators that we found.

3. **The problems that you faced (10 points)**

Aside from the initial issue with the Criminology API there were a few issues we had to work out. One of the largest issues was with the NationsReportCard.gov API. Firstly, we had to manually find enough years/unique subjects of data to hit the 100 items minimum looking through the NAEP Data Explorer: https://www.nationsreportcard.gov/ndecore/landing. Then, due to the formatting of the different data (nested dictionaries of each subject and then each student and student score) we had to convert the JSON into an overall dictionary for the whole API. The NationsReportCard.gov API also has two duplicate strings that we needed to keep in the database- the subject name and the gender+race of the student- which were both turned into integers with an id assigned to every possibility, 1-12 for every subject and 0-5 for every gender+race combination.

Another temporary issue we faced was in the NCB API where we accidentally used DROP TABLE IF EXISTS *Tablename* as we were following examples from in-class discussion before realizing it was throwing an error where it was wiping out the table with every run. After looking over discussion material more closely we realized that it had to be CREATE TABLE IF NOT EXISTS Tablename which then made and maintained our database properly to follow the project guidelines.

Finally, there were minor bugs and errors throughout the project, but as with any code project a mixture of rotating files between the team so each individual can try and resolve them, attending office hours at least twice a week, and looking back on our discussion and in-class code made the final project go by smoothly.

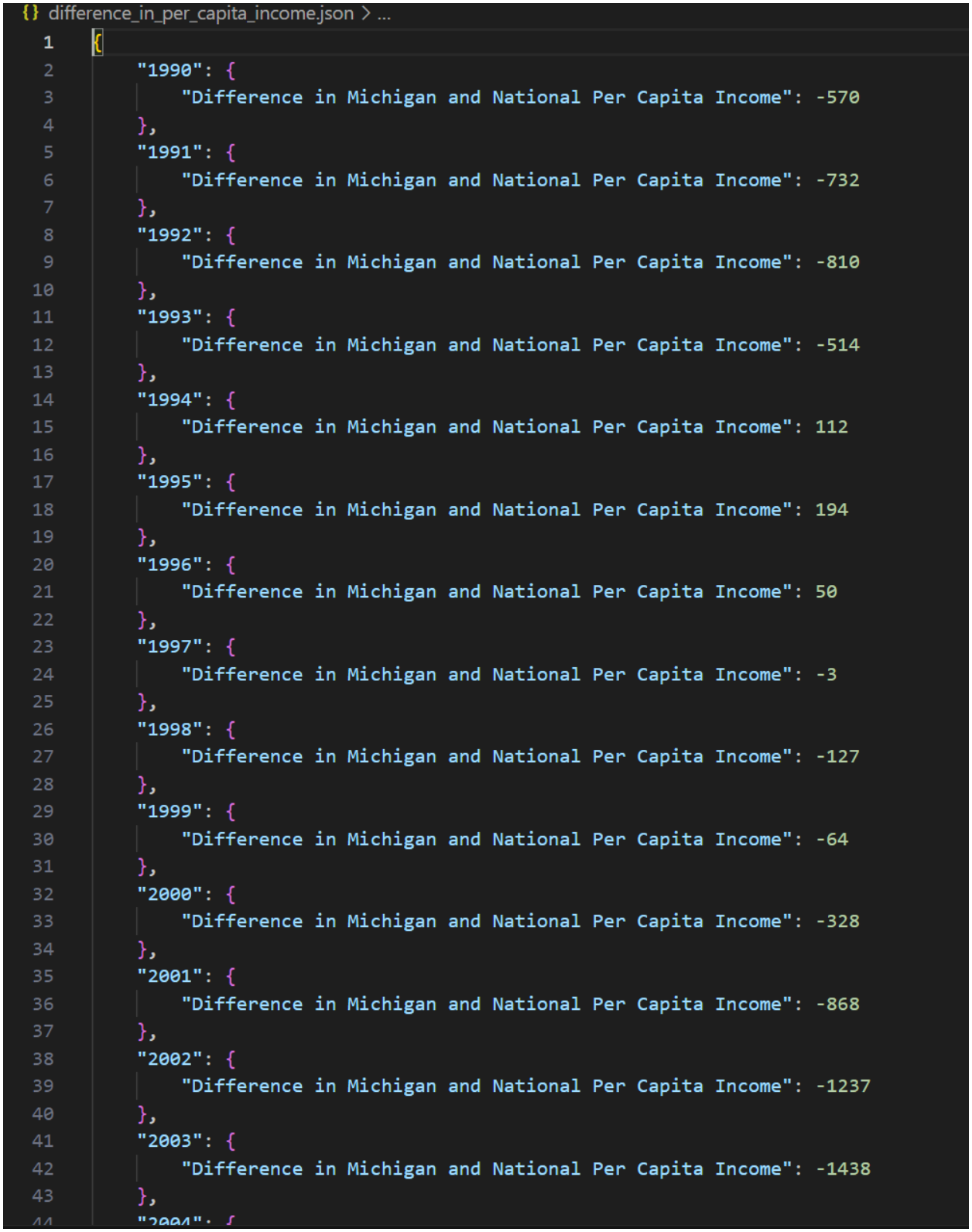
4. **The calculations from the data in the database (i.e. a screen shot) (10 points)**

{} average_poverty_rate_MI_counties.json > ...

```json
{
    "Average Poverty Rate of Michigan Counties": 12.3,
    "National Average Poverty Rate": 11.5
}
```

{} NCES_average_scores_by_race_gender.json > ...

```json
{
    "Black+Male": {
        "Average Score": 189.10917132432593
    },
    "Black+Female": {
        "Average Score": 195.3229611870341
    },
    "Hispanic+Male": {
        "Average Score": 214.762726920351
    },
    "Hispanic+Female": {
        "Average Score": 217.9623523411877
    }
}
```



Then we also have our non-calculation JSON files

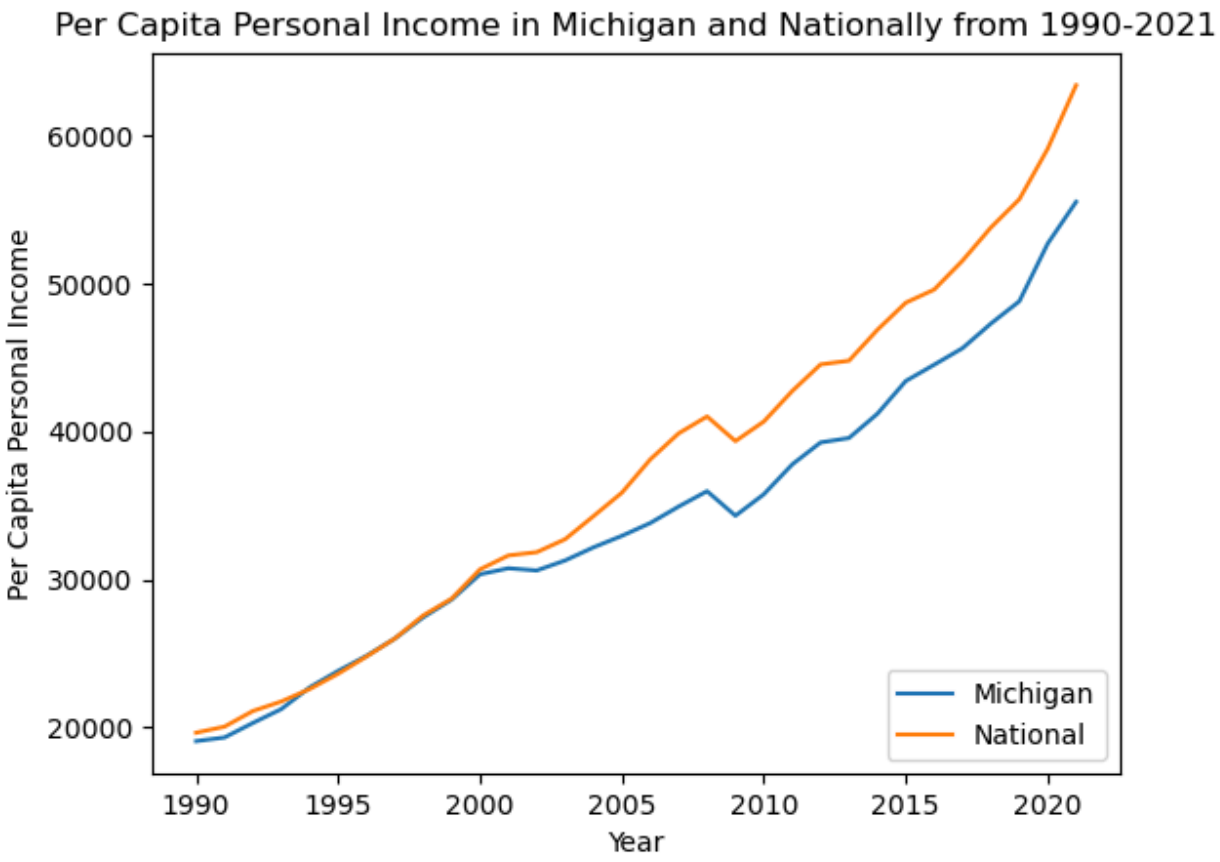
michigan_poverty_rates.json > ...

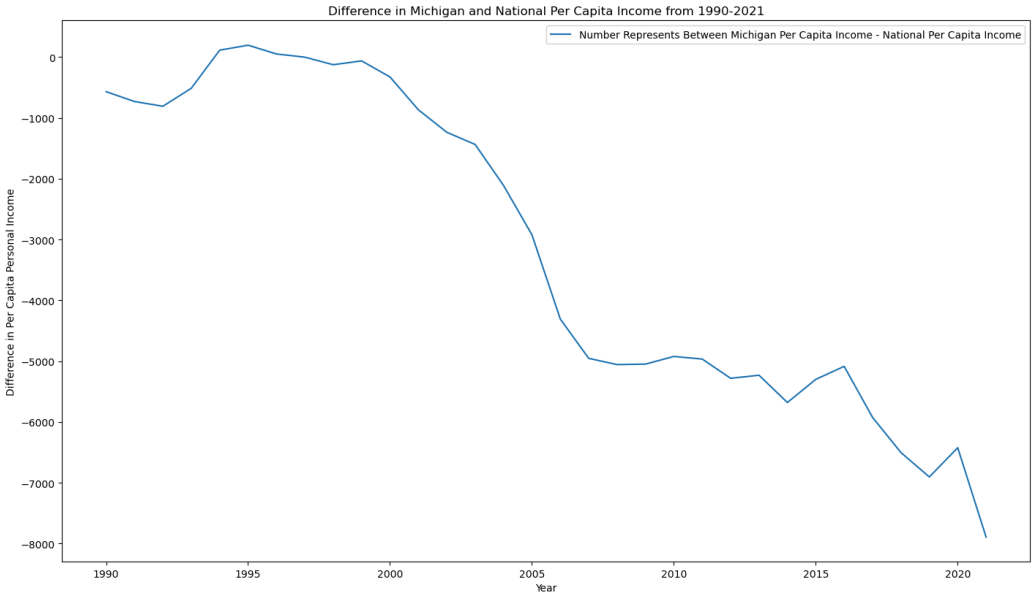

```json
{
    "Bay County": {
        "Population": 44664,
        "White Percentage": 89.6,
        "Black Percentage": 1.5,
        "Poverty Rate": 15.4
    },
    "Berrien County": {
        "Population": 63311,
        "White Percentage": 73.1,
        "Black Percentage": 13.4,
        "Poverty Rate": 14.9
    },
    "Calhoun County": {
        "Population": 53388,
        "White Percentage": 76.9,
        "Black Percentage": 11,
        "Poverty Rate": 13.2
    },
    "Genesee County": {
        "Population": 167950,
        "White Percentage": 72.4,
        "Black Percentage": 19.5,
        "Poverty Rate": 16.4
    },
    "Ingham County": {
        "Population": 118485,
        "White Percentage": 68.5,
        "Black Percentage": 10.8,
        "Poverty Rate": 16.5
    },
    "Jackson County": {
        "Population": 63368,
        "White Percentage": 82.5,
        "Black Percentage": 6.1,
        "Poverty Rate": 12.8
    },
    "Kalamazoo County": {
        "Population": 108397,
        "White Percentage": 76.7,
        "Black Percentage": 10.6,
        "Poverty Rate": 11.8
    },
```

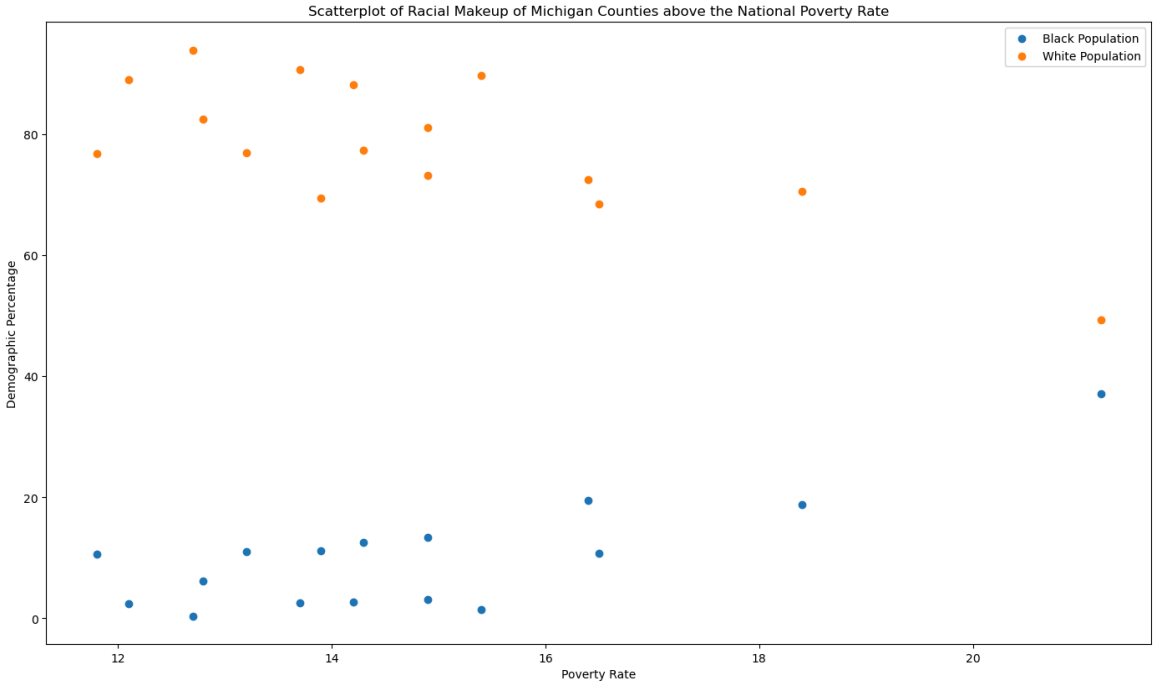
per_capita_data.json > ...

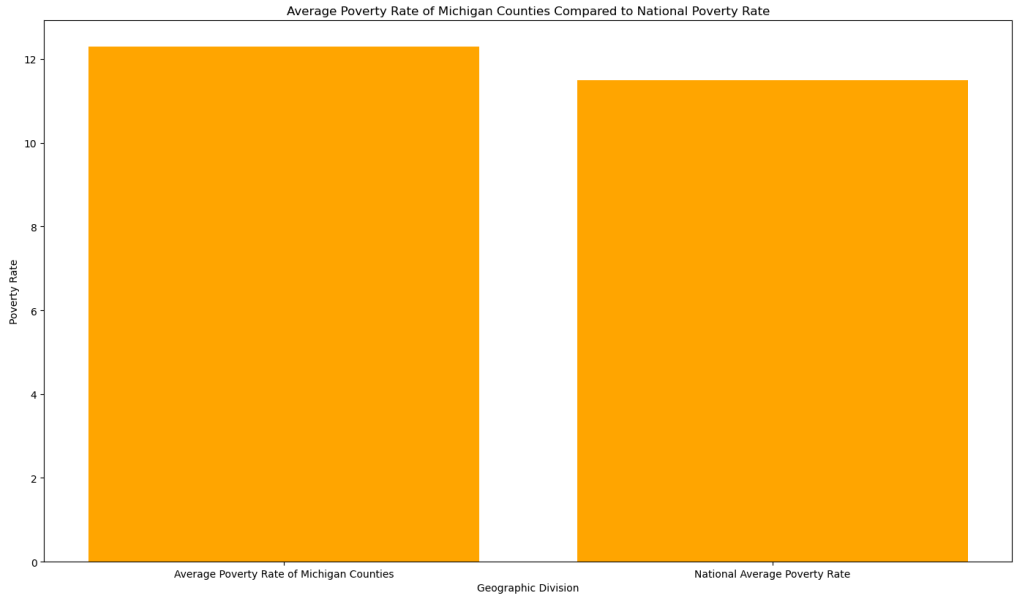

```json
{
    "1990": {
        "Michigan Per Capita Income": 19051,
        "National Per Capita Income": 19621
    },
    "1991": {
        "Michigan Per Capita Income": 19298,
        "National Per Capita Income": 20030
    },
    "1992": {
        "Michigan Per Capita Income": 20280,
        "National Per Capita Income": 21090
    },
    "1993": {
        "Michigan Per Capita Income": 21219,
        "National Per Capita Income": 21733
    },
    "1994": {
        "Michigan Per Capita Income": 22687,
        "National Per Capita Income": 22575
    },
    "1995": {
        "Michigan Per Capita Income": 23801,
        "National Per Capita Income": 23607
    },
    "1996": {
        "Michigan Per Capita Income": 24821,
        "National Per Capita Income": 24771
    },
    "1997": {
        "Michigan Per Capita Income": 25990,
        "National Per Capita Income": 25993
    },
    "1998": {
        "Michigan Per Capita Income": 27430,
        "National Per Capita Income": 27557
    },
    "1999": {
        "Michigan Per Capita Income": 28629,
        "National Per Capita Income": 28693
    },
    "2000": {
```

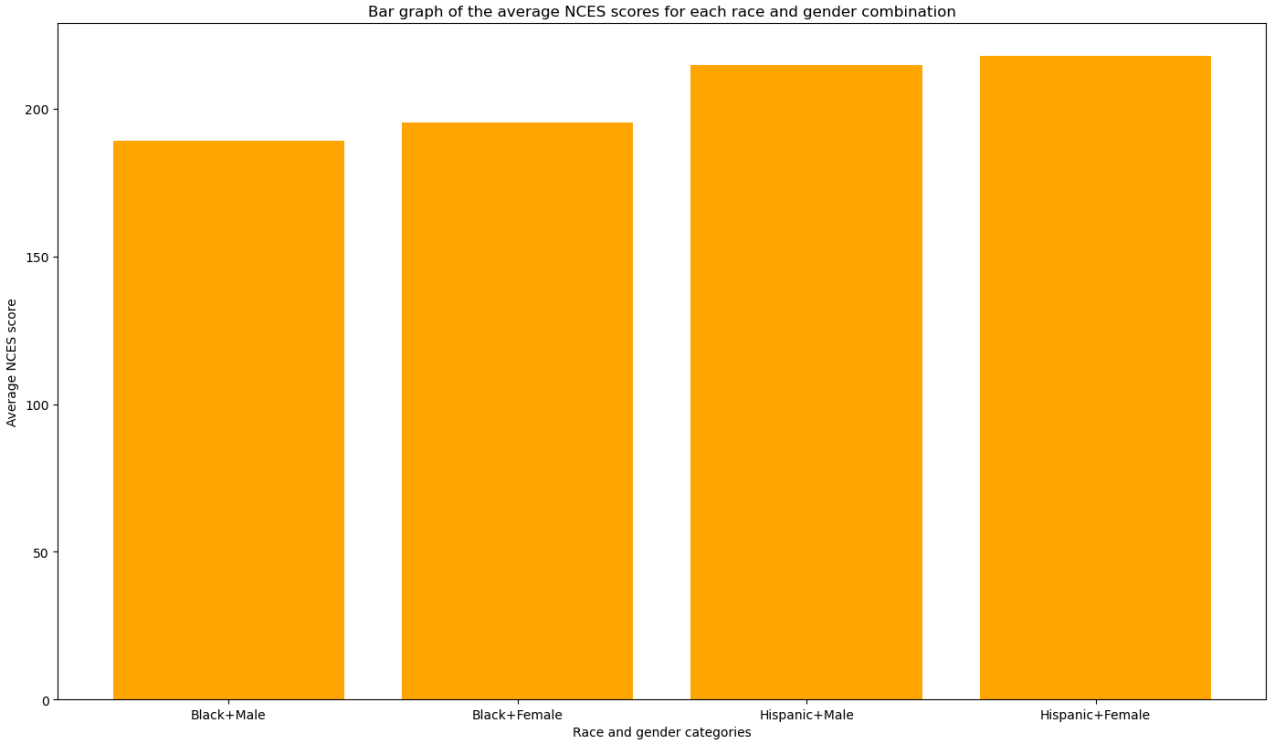
5. The visualization that you created (i.e. screen shot or image file) (10 points)




Difference in Michigan and National Per Capita Income from 1990-2021
Number Represents Between Michigan Per Capita Income - National Per Capita Income
Difference in Per Capita Personal Income
Year
0
−1000
−2000
−3000
−4000
−5000
−6000
−7000
−8000
1990
1995
2000
2005
2010
2015
2020


Scatterplot of Racial Makeup of Michigan Counties above the National Poverty Rate
Black Population
White Population
Demographic Percentage
Poverty Rate
0
20
40
60
80
12
14
16
18
20





6. Instructions for running your code (10 points)

Our code makes use of multiple different files, all of which create their own JSON files which need to be called in our visualizations. For this reason, our files must be run in a specific order a set number of times.

First, you will need to run each of our API.py files. These are NAEPapi, NCBapi, and NCESapi (all named after the three APIs we used for our investigation). Because we made sure to only add 25 items at a time, you will need to run the NCES file 6 times, the NAEP file twice, and the NCB file, which took demographic information for every county in the United States, at least 34 times, or until you see a print statement saying that all data has been added.

Once you have completed this task, you will need to run our file calculate_data.py. This will pull data from our databases and store the corresponding calculations in six json files, five of which we use in our visualizations. After running this file, you should see michigan_census_data.json, michigan_poverty_rates.json, per_capita_data.json, average_poverty_rate_MI_counties.json, difference_in_per_capita_incomes.json, and NCES_average_score_by_race_gender.json.

Finally, you will run our visualizations file, which is labeled as such. This file will pull all of our previously stored json data and make five visualizations. Because of lengthy labels, you should put our visualizations on full screen to get the best viewing experience.

7. **Documentation for each function that you wrote. This includes describing the input and output for each function (20 points)**

We will go in order of files:

NAEP.api:

1) The first function is scrape_data(). This function takes no arguments, but accesses the url for the Michigan State government data site and returns the data as a json.
2) The second function is set_up_database(data_name). This function establishes the path for our database and makes its name the input variable, data_name. It returns conn, which will be used later to create our cur.
3) Our third function is make_database(cur, conn). As labeled, this function executes the cur to make a main table for our per capita personal income data. It returns nothing, but commits on the conn.
4) The fourth function is add_data_to_table(data, cur, conn), which runs through the input variable data and adds the items to the main table through cur.execute(). This function also limits the amount of entries each time to 25, while eventually making a conn.commit().
5) The final function is main(), where we run all of the functions. This accepts and returns nothing, but assigns the return of scrape_data() to a variable data. If this was successful, we then assign the return of set_up_database("FinalProject.db") to conn, which we then use to create a cur object. After this, our main() function executes make_database and add_data_to_table.

NCB.api:

1) Before we made any functions, we assigned our query url to a variable named query_url. We also gathered various parameters and assigned them to easy-to-understand variables

which described their function. Although we gathered many, we only used total_population, pop_percent_white, pop_percent_black, and poverty_percent.

2) Our first function was scrape_data_counties(). This function accessed our query_url using our parameter variables to return the data in a json format. Specifically, it returns a json object with the total population, proportion of the population that is white and the proportion that is black, and the percentage of population in poverty for every county in the country. The county name included its state as well, which we will use later.
3) Our second function is scrape_state_names(). This also accepts nothing, but runs a query to just get the name of every state in the country (as classified by the census bureau data, which included Washington D.C and Puerto Rico in this query). We then returned this data, which was also in json format.
4) Our third function is the same in all of the API files, set_up_database(data_name). As stated earlier, this establishes a .db file with the name entered as an argument at the location of our FinalProject folder. We run this every time to create the database if it does not exist.
5) The fourth function, make_state_table(state_names, cur, conn), is used to create a key table to avoid duplicate string data. Because we will be putting multiple counties for each state in the main table, we use this function to assign each state a key id starting at 0. We take in the cur, conn for the database and state_names, which is the return of our scrape_state_names(). The function creates the table if it doesn't exist and commits it using conn.commit().
6) The fifth function does the same thing for counties, make_county_table(county_data, cur, conn). It takes in the return of our function scraping county data and assigns the name of the county a unique id, which we then commit with conn.commit(). This function adds hundreds of counties at once, but this was necessary, since we assumed that there would be repeat county names throughout the country. This is vindicated by this table having 679 items, as opposed to the 848 entries in the main table. If we added only 25 counties at a time using this function, while adding 25 items to the main table, we could run into an instance where the county hasn't yet been assigned a unique id. If we wrote our code so that data wouldn't be added to the main table until the county_id table was filled, the process of creating the database would be unreasonably long. Therefore, we added all 679 entries to this table at once.
7) The sixth function, make_main_table(cur, conn), creates our main table US_Demographic_Info.
8) def_main_data(county_data, cur, conn), our seventh function, takes in the same county data as make_county_table. First, the function gets the number of ids in the table, making that number the starting count and the number 25 from that as the end count. While the starting count is less than the end count, we access the county_data to add each element into the table. We pull the county and state ids for each place to ensure no duplicate string data. After inserting all of the data into our main table, we use conn.commit().
9) Finally, we have a main() function that lays out how the code file should run. We assign the return values of scrape_data_counties() and scrape_state_names() to their own variables. If this was successful, we run set_up_database("FinalProject.db") and create a cur object

using the conn. Finally, the main() function runs make_county_table, make_state_table, make_main_table, and add_main_data in that order.

NCESapi:

1) First we made scrape_data to call the API key and get at least 100 items of student data across 13 subject scores for the public middle and high school district of Michigan, focusing on recent years. If the request fails, it prints a message indicating the failure. After fetching data for all subjects, the function makes a file path to save the JSON file within the FinalProject directory. It then writes the combined_data dictionary to the JSON file. Finally, it calls the rewrite_json_data function to process the fetched data and convert it into a more structured dictionary format before returning it.
2) Next is rewrite_json_data(combined_data) which as seen takes the combined_data.json and converts it into a dictionary since the combined JSON wasn't able to commit 25 items to SQL at a time due. Here we also iterate over each subject and its associated list of students, taking relevant data such as grading scale, year, value, and varValueLabel (gender and race keyword) for every student. We then map the varValueLabel to a corresponding integer value using a predefined dictionary label_to_value, which represents gender and race combinations. It organizes this data into a new dictionary new_dict, where each entry consists of a key formed by a combination of subject_id, scale, year, value, and gender_race_id, and a corresponding list of student IDs unique to each student. The function returns this dictionary containing the transformed student data.
3) The set_up_database(data) creates or establishes a connection to a SQLite database named 'FinalProject.db' depending on if it already exists/ran. It returns both the cursor object (cur) and the connection object (conn).
4) Next, the function, create_gender_race_table is responsible for creating and populating a secondary SQLite table to store gender and race information since these are strings that would have been repeated in our main database. It takes three parameters: data, cur, and conn, where data is not used, cur is the cursor object for executing SQLite commands, and conn is the connection to the SQLite database. Inside the function, a list gender_race_id is defined, containing unique combinations of race and gender strings. It has six combinations including "White+Male", "Black+Male", "Hispanic+Male", "White+Female", "Black+Female", and "Hispanic+Female". The function then drops the existing table gender_race_id if it already exists to ensure a fresh start. Afterward, it creates a new table named gender_race_id with two columns: id as the primary key (auto-incrementing integer) and gender_race as a text field to store the unique combinations. Next, it iterates over the gender_race_id list, and for each entry, it inserts a row into the table with the id corresponding to its index in the list and the gender_race value being the respective combination of race and gender. Finally, it commits the changes to the database. It does not return anything.
5) Next is the creation of our main database, the create_main_table function begins by ensuring that a table named "NCESdata " exists in the SQLite database. This table is where the data will be stored. If the table doesn't exist, it creates one with columns for id, subject_id, year, score, and gender_race_id. It then opens the JSON file (filename), reads

its contents, and loads the data into a Python dictionary using the json.loads() function. This JSON data likely contains information about students, including their subject, year, score, and gender/race. The function queries the database to count the existing records in the NCESdata table. This count is crucial for determining the starting point for inserting new records since we have to insert 25 items at a time until we display all 100. Based on the current count of records and the desired group size (25), the function calculates the end count. Using a while loop to ensure we dont go out of bounds, the function iterates from the current record count (id_count) to the end count. For each iteration, it extracts student information from the Python dictionary (new_data), constructs an SQL INSERT query with the student values. After inserting each batch of 25 records, the function commits the changes to the database using conn.commit(). This ensures that the inserted records are permanently saved. It does not return anything.

6) The TestAllMethods(unittest.TestCase) has a setUp method executed before each test method in the class. Here, it calls the set_up_database function to establish a connection to the SQLite database and obtain a cursor. Then, it calls the create_gender_race_table function to create the "gender_race_id" table in the database. It follows with a tearDown method executed after each test method in the class to clean up and reset the testing environment to its initial state. Lastly it closes the database connection (self.conn.close()), ensuring that resources are properly released after running each test.
7) After all that, we have our main() function which calls the scrape_data function to fetch data from an API. If data is successfully retrieved then it proceeds to the next steps. It also calls the set_up_database function to initialize a connection to an SQLite database and obtain a cursor (cur) and connection (conn). If data was retrieved successfully, it assumes that the database setup will also be successful. It then calls the create_gender_race_table function to create a table named "gender_race_id" in the database. This table contains IDs and corresponding gender/race labels. Lastly it calls the create_main_table function to create the main table named "NCESdata" in the database. This table is intended to store data related to subjects, years, scores, and gender/race. Once the table is created, it populates it with data obtained from the JSON file "combined_data.json".

Calculate_data:

1) First, we created a variable national_poverty_rate, where we assigned the value 11.5. This is the national poverty rate, and will be important for our calculations later.
2) Our first function, access_table(db_name), accesses the database with the input name and returns a conn object, which we will use to create a cur object.
3) Our next function, get_michigan_data(cur, conn), accesses the data from our main US_Demographic_Info table and only returns values for the state of Michigan. Once we've fetched the id for Michigan, we run a cur.execute to select all of the data from US_Demographic_Info with an equivalent state_id. We also join the county_id table to get the county name rather than id for each item. Once we've fetched this data, we assign it to michigan_data. We then create an empty dictionary, which we fill with the key being each

county name and the value being another dictionary with its population and demographic information. We return this dictionary.

4) Our next function, out of order, is make_json(data, filename). This function creates a json file using the filename argument and writes in the data argument as a json using json.dump.
5) The next function, get_highest_poverty_rates(cur, conn), does almost all the same things as the get_michigan_data function. However, it adds in an additional parameter, which is all data where the poverty_percentage is greater than national_poverty_rate. We then create and return a similar dictionary to what we return in get_michigan_data.
6) Our next function is the actual calculation for the census bureau, average_poverty_rate(cur, conn). This function takes in data representing all of the Michigan counties and finds the average of their poverty rates. It then places this in a dictionary alongside the national poverty rate, which we independently found to be 11.5.
7) Our next function accesses the per capita income data. get_per_capita_income_data(cur, conn) takes in our cur object and executes it to return the per capita income data for michigan and nationally in each year. We then go through the fetched data to assign it to a nested dictionary where the keys are each year and the values are a dictionary with the two per capita income results.
8) We then do the calculations for the per capita income data using get_difference_in_incomes(cur, conn). This function takes the same data as the previous function, but this time it subtracts the national per capita income from the michigan per capita income for each year. We then store this value in a dictionary with each year as a key. A positive value means that the Michigan per capita income was greater than the national per capita income, while a negative value means that it was less than the national income.
9) Our final function is parse_NCES_data(cur, conn). Much like the other functions, this selects all of the data from our NCES table joined on the id for gender_race combinations. We also put a parameter that the score should not equal 999, since we believe this was an error with the initial API. We then create a data_dict, where we assign a list of scores to each gender_race combination. Once we've done this, we create another dictionary, average_dict, which creates an embedded dictionary where the key is each gender_race combination and the value is a dictionary with the average score. We return this dictionary.
10) After we created all of these functions, we wrote a main() function that runs the file. First, it creates a cur object out of the return value of access_table. It then runs all four of our functions and returns the data to four different variables. Finally, we run the make_json function four times to create four different json files out of the data.

Visualizations:

Finally, we have our visualizations file.

1) First, we made a function read_json(filename) which reads in the data from each file and returns the json object.
2) Next, we made parse_percapita_data(data), which accesses a data variable and creates a line graph with two separate lines for the per capita personal income of Michigan and Nationally over the years, specifically from 1990-2021.

3) We then plot the calculation with percapita_difference(data). This takes a dictionary with the difference for each year and similarly plots it on a line graph, this time only containing the one line.
4) We then plot the calculation for the census bureau with MI_average_poverty(data). This takes our dictionary with the average poverty rate for Michigan counties, as well as the national rate, and plots both values in a bar graph to easily compare them.
5) After that, we use our poverty data in parse_poverty_data(data). This takes in our poverty data and creates a scatterplot of the poverty rates in each Michigan county with their demographic information. We did this by making two scatters on the plot, one with only the white percentages and one with the black percentages, which were paired by their indexes in a list.
6) Our final visualization uses parse_nces_data(data). This takes in our nces data and makes a bar graph, where the gender_race categories are on the x axis and the average score is the y axis.
7) We used main() to run all of our functions in order. First, it reads in all three of the relevant json files using read_json. It then runs all of the visualization functions in order.

8. You must also clearly document all resources you used. The documentation should be of the following form (20 points)

Date	**Issue Description**	**Location of Resource**	**Result (did it solve the issue?**
Throughout 4-10 to 4-26	General issues debugging and getting correct syntax. Re-visiting instructions on how to make Matplotlib charts.	W3 Schools: https://www.w3schools.com/	Yes
4-17	NCESapi was unable to properly load in 25 items at a time- it either loaded all 100 or did the first item 25 times.	ChatGPT	No. This issue was solved by our team alone.
Throughout 4-10 to 4-26	General issues making sure we created, called, and updated our SQL databases properly during our project	SQL Documentation: https://www.sqlite.org/docs.html	Yes
4-10	NCESapi did not have enough items within the apis getting called, and it was not clear which subjects/years have been collected. Accessed the	NAEP Data Explorer: https://www.nationsreportcard.gov/ndecore/landing	Yes

	NAEP Data Explorer to manually search for data collected in public schools in Michigan that we could call.		