Earth Engine Project: Congestion and Crash Severity along the Philadelphia Vision Zero High Injury Network

By Ian Schwarzenberg

CPLN 670

Fall 2018

Link to application: and full code:

https://code.earthengine.google.com/abe4d512adc4c692b50e419854e7c193

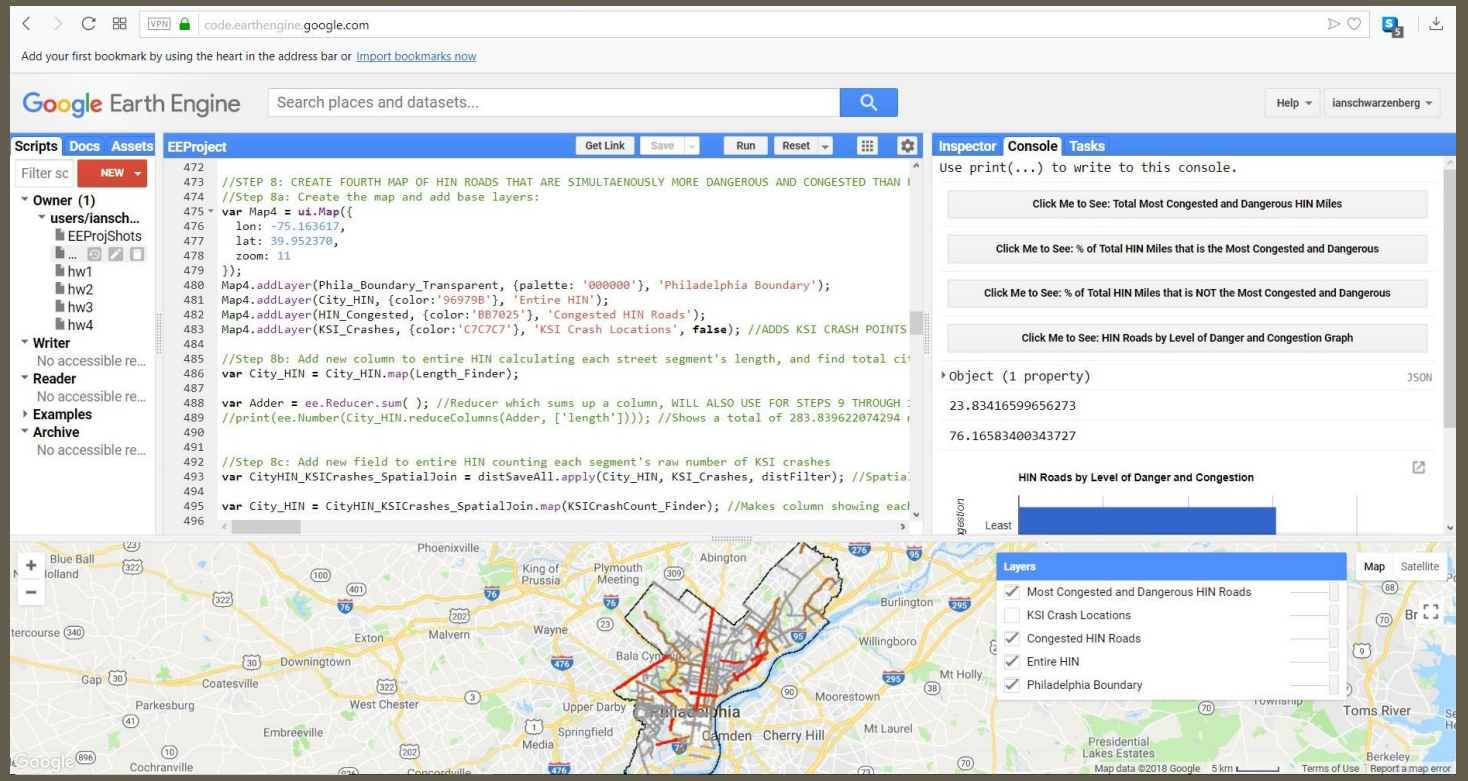
TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

Introduction



Map 4 Overview – Shows Congested HIN and Most Congested *and* Dangerous HIN

PURPOSE

Aims: The aims of this application are to let users easily:

- 1) Find which Philadelphia High Injury Network (HIN) roads are consistently congested across the busiest hours of the day. The HIN was selected for study instead of all Philadelphia streets since the HIN consists of the city's most dangerous roads.
- 2) Find which of those consistently congested HIN roads are also more dangerous than the rest of the HIN. This would show the HIN roads that are simultaneously the most congested *and* the most dangerous out of all other roads on the network.
- 3) See if there are any common characteristics among the HIN roads that are simultaneously the most congested and dangerous which could potentially give hints as to why they were both highly congested and dangerous. This involves examining the widths of those streets, the wealth of the neighborhoods surrounding those streets, the proximities of those roads to rail stations, and the proximities of those roads to areas of commercial activity.

Motivation: After being hit by a car while crossing a street two years ago, I was inspired to pursue the career direction of working to make streets safer for all users. Doing this turns a life event which could prevent me from living life to the fullest into a life event that enables me to work towards making transportation networks safer, so that less people will have experiences like mine. Analyzing data about street safety is a good way to work towards improving transportation safety for everyone, since this helps planners identify the most dangerous locations where solutions can have the most impact. I am also currently interning at the Delaware Valley Regional Planning Commission (DVRPC), where I am using GIS to analyze Philadelphia area road congestion data. All of this sparked in my interest in comparing Philadelphia congestion data to local crash data to see if there were a majority of especially dangerous HIN roads that are also congested.

METHODS OVERVIEW

- Joining road congestion data to the city HIN by street name
- Mapping functions over that result to create new columns in it averaging congestion data for the busiest hours.
- Filtering the result of the busiest hour column creation to select out the most consistently congested roads.
- Spatially joining killed and severe injury (KSI) crashes to the original HIN in its entirety to get the average HIN KSI crash rate.
- Filtering out congested HIN roads with KSI crash rates higher than the HIN average to get the HIN roads that are simultaneously the most congested *and* the most dangerous out of any other HIN road.
- Intersecting city street centerlines to the most congested and dangerous HIN roads using a special intersection join method to find which of the most congested and dangerous HIN roads are considered wide.
- Joining median household income data to census tract polygons
- Filtering the census tracts with the joined median household income data to find poor city census tracts
- Intersecting the most congested and dangerous HIN roads with the poor census tracts using a special intersection join to see which of the most congested and dangerous HIN road segments pass through poor areas.

METHODS OVERVIEW

- Creating a quarter mile buffer around rail station points.
- Intersecting the most congested and dangerous HIN roads with those rail station buffers using a special intersection join method to find which of the most congested and dangerous HIN roads pass within walking distance of the rail stops.
- Intersecting the most congested and dangerous HIN roads with commercial corridor polygons to find which of those roads pass through commercial corridors.
- Creating buttons which reveal statistics and graphs that portray information to the user about conditions on the most congested and dangerous HIN roads.
- *Eliminating* Google Earth Engine's default map and *replacing* it with eight separate maps detailing the analysis. Only one map shows at a time which is determined by the user.
- Creating prompts which let the user choose which of the eight maps to see, detailing each map's respective data and buttons.

DATA SOURCES

Data Sources:

- City HIN vector line dataset (created in 2017): Philadelphia office of Transportation and Infrastructure Systems (oTIS)
- Congestion data (from 2017): INRIX, but provided to me through the Delaware Valley Regional Planning Commission (DVRPC)
- Crash data (from 2013-17): Pennsylvania Department of Transportation (PennDOT)
- Philadelphia Boundary: Philadelphia Department of Planning and Development (DPD)
- Philadelphia 2010 Census tracts shapefile: DPD
- Median household income data: US Census American Community Survey 2012-16
- Street Centerlines line shapefile: Philadelphia Streets Department
- Rail Stations point shapefile: Southeastern Pennsylvania Transportation Authority (SEPTA), DVRPC
- Commercial Corridors: DPD

DEFINITIONS

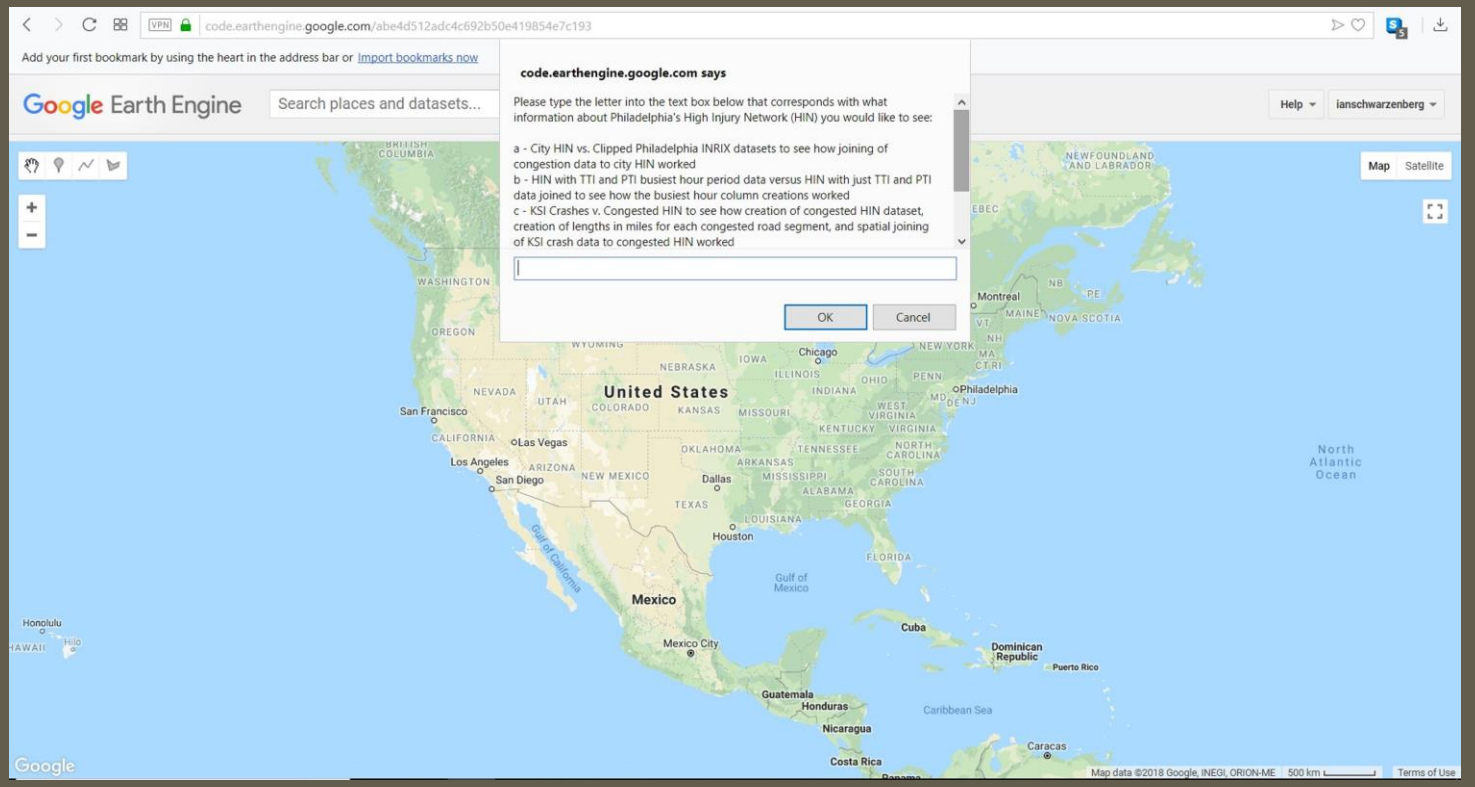
- **HIN:** The Philadelphia Office of Transportation and Infrastructure Systems (oTIS) created a vector line shapefile called the Philadelphia High Injury Network, which consists of Philadelphia roads which they found to have the highest rates of KSI crashes per mile of roadway in the entire city from 2012-16.[1] I am limiting my analysis just to this network because this network allows the city to target their limited resources to the most dangerous roads. I will follow this mindset for my analysis as well.
- **KSI Crash:** Simply any crash that results in a fatality or severe injury.[1]
- **ATT:** Actual Travel Time, the time it takes to drive down a road *with* congestion at a certain time.[2]
- **FTT:** Free-Flow Travel Time, the time it takes to drive down a road *without* congestion at a certain time.[2]
- **PT:** Planning Time, the time a motorist would plan to travel down the length of a road segment at a certain time, with or without congestion.[2]
- **TTI:** Travel Time Index, a measure of congestion which is simply calculated by dividing the ATT by the FTT of a road segment for a certain time. 1.5 is consistently used as a cutoff by engineers for this, since this means if a road segment has a TTI of 1.5 or above, it takes at least 50% longer to drive down a road with congestion than without it.[2]
- **PTI:** Planning Time Index, another measure of congestion which is calculated by dividing the 95th percentile PT for a road segment by its FTT. Unlike TTI, there is not cutoff consistently used by engineers and planners for this to mark a road as congested. However, the Delaware Valley Regional Planning Commission (DVRPC) uses 3, which allows them to use limited resources to target the roads motorists have to devote the most time to traveling down.[2] Therefore, I am using this PTI cutoff in my analysis as well.

[1] http://visionzerophl.com/uploads/attachments/cj84hcz3u00vvkcd6ebnmvo3q-2017-vision-zero-high-injury-network.pdf
[2] Told by DVRPC Senior Planner Tom Edinger to Ian Schwarzenberg on 9/21/18.

DEFINITIONS

- **Philadelphia Area Busiest Hours:** The DVRPC aggregates TTI and PTI data into four different busiest hours: 6-10 AM (morning rush hour), 7-8 AM (morning peak hour), 3-7 PM (evening rush hour) and 7-8 AM (evening peak hour). They do this so they can examine where road are most congested at the busiest times, further enabling them to target limited resources to the most congested roads at the busiest times.[2] I will be following the DVRPC's example by focusing my analysis on congestion during these four timeframes on weekdays.
- **Consistently congested HIN roads:** HIN roads that will come up as congested across all of the busiest hour periods. These are the road segments I am interested in the most because these are the most consistently congested roads.
- **Most congested and most dangerous HIN roads:** HIN roads that will come up as congested across all of the busiest hour periods, but also have KSI crash rates that are higher than the entire HIN's average KSI crash rate.
- **Wide Street:** Any city street that is classified as a major or minor arterial by the Philadelphia Streets Department.
- **Poor Census Tracts:** Any Philadelphia census tract with a median household income of less than $39,370.153846153844 (this is the median Philadelphia census tract median household income which I found through my analysis).
- **Rail Stations:** Any SEPTA Market Frankford Line subway station, Broad Street Line station, or regional rail station.
- **Areas of Commercial Activity:** Any commercial corridor as defined by the DPD.

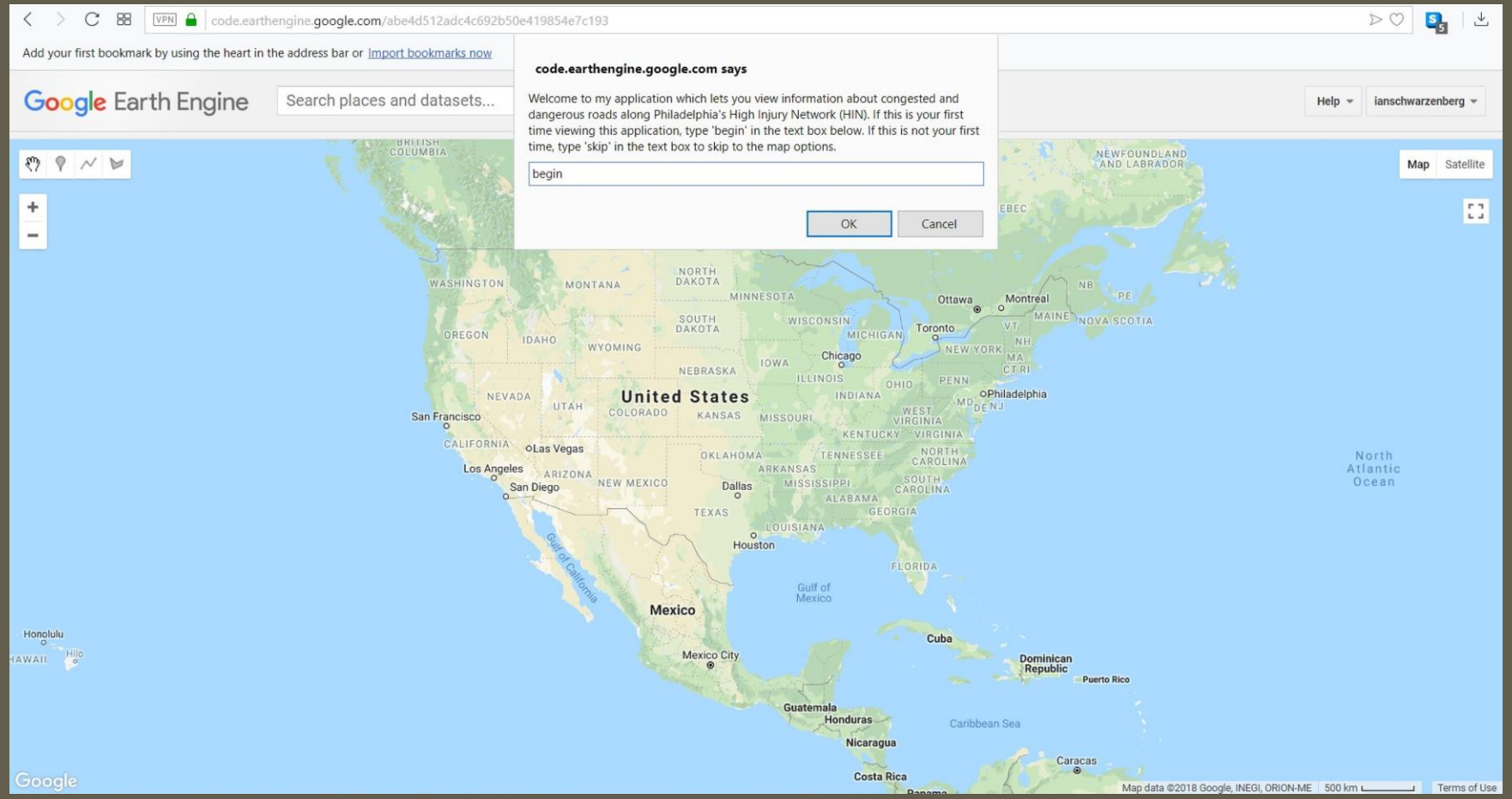
User Interface



Part of Second Prompt – One of most Important Parts of User Interface

INTRODUCTORY PROMPTS

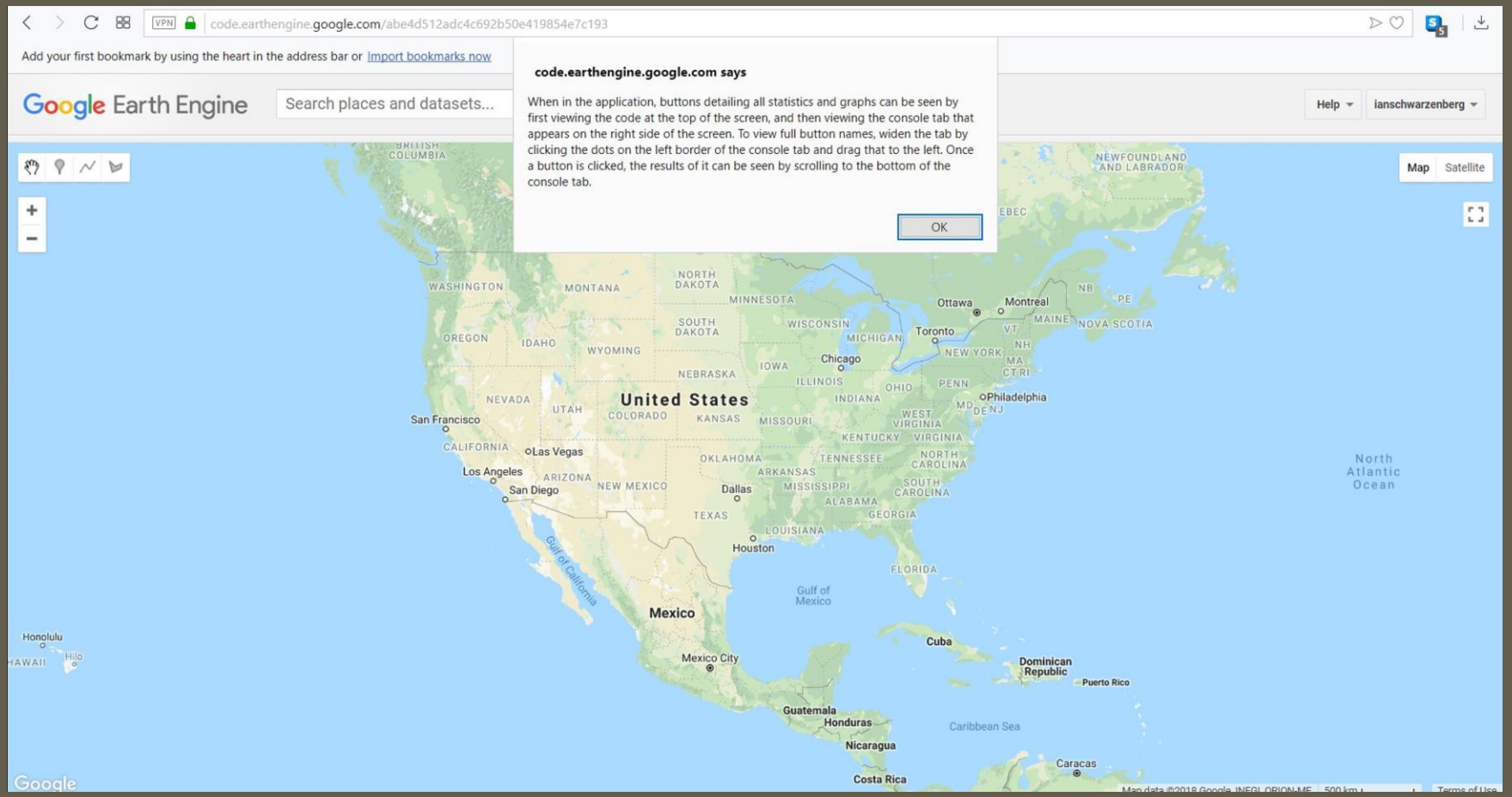
- **First Prompt:** When the user first starts the application, a prompt pops up asking the user if this is their first time viewing the application. If the user types 'begin' to indicate that it is their first time, they will then go through a set of instructions detailing how to use the application. If this is not their first time using the application, then they can type 'skip' which will then bypass the instructions and lead them to the final introductory prompt which will let them choose which data to view.



First Prompt

INTRODUCTORY PROMPTS

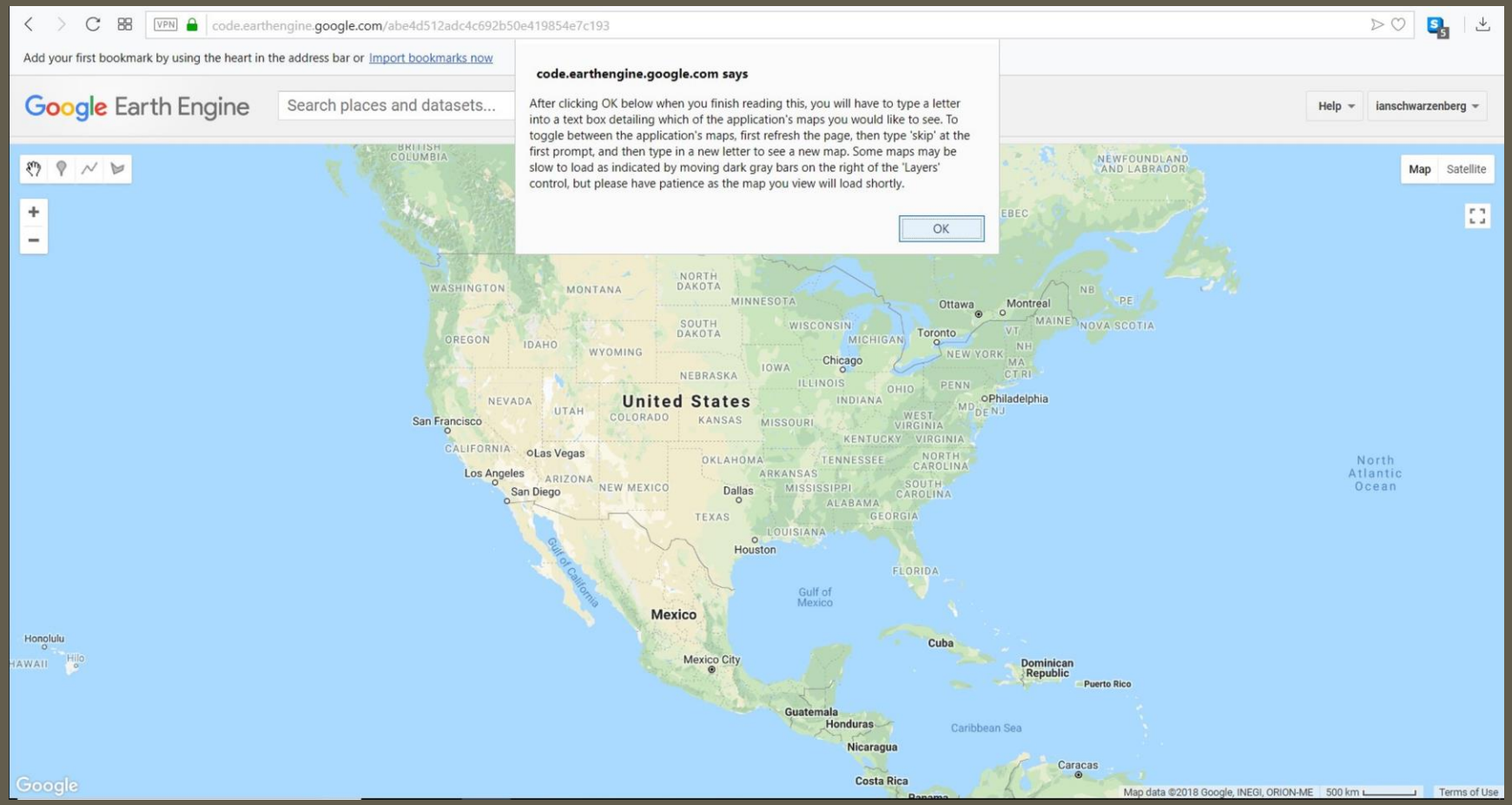
- **Application Instructions:** When the user indicates that this is the first time using the application, one of two alerts then pops up detailing the first part of how to use the site. This first alert tells the viewer how to access the application's graphs and statistics by clicking buttons that will appear once they view the application's code. This message is intended for users who are not familiar with the general Google Earth Engine user interface.



First Instructions Alert

INTRODUCTORY PROMPTS

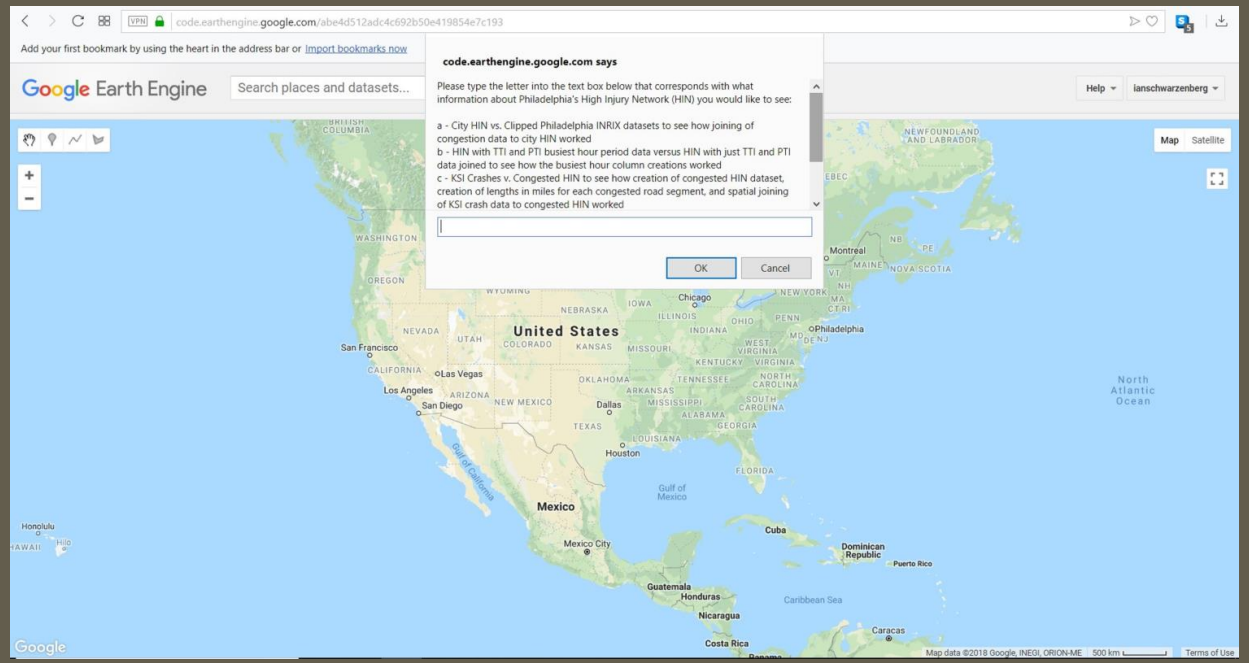
- **Application Instructions:** When the user finishes viewing the first alert, they then come across and the second and final instructional alert which tells them how to view and switch between the application's eight maps. It also warns them about how some maps may be slow to load at first, but politely tells them to be patient, as all maps load shortly.



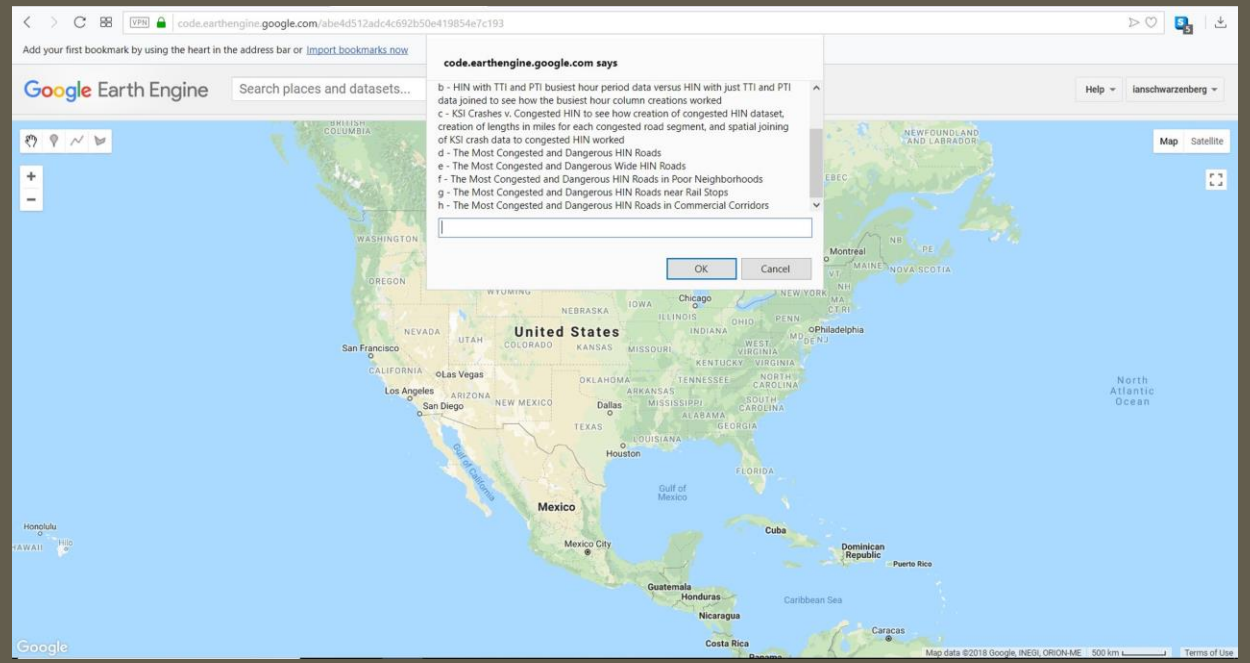
Second Instructions Alert

INTRODUCTORY PROMPTS

- **Second Prompt:** When the user finishes viewing the two instruction alerts, they then reach the final prompt which asks them to choose which map to view. They must type in a single lowercase letter that corresponds with the map they want to view. Rather than have all buttons and map layers be put on the one default map that automatically comes with Google Earth Engine, my application instead *replaces* that one default map with *eight new maps,* with each map having its own data and buttons. This is an alternative to having all of the application's data and buttons appear at once, which can overwhelm the viewer. This second prompt thereby serves as a gateway to the application, with the instruction alerts aiming to serve as a leadup to this gateway. Returning users who want to view other maps can simply type 'skip' at the first prompt to take them directly to this second one, since they already went through and know the instructions. This way, they do not have to waste time reading over what they already read before, and can go straight to their next map.



Top of Second Prompt



Bottom of Second Prompt

IF-ELSE STATEMENTS

- **Code that Creates Prompts:**

```
var FirstPrompt = prompt("Welcome to my application which lets you view information about congested and dangerous roads along Philadelphia's High Injury Network (HIN). If this is your first time viewing this application, type 'begin' in the text box below. If this is not your first time, type 'skip' in the text box to skip to the map options."); //First prompt where user can either go through all instruction if it's their first time using this, or skip to choosing maps if it is not

if(FirstPrompt=='begin'){ //Means "if the user types 'begin' in first prompt, then everything in this if statement will happen"

alert("When in the application, buttons detailing all statistics and graphs can be seen by first viewing the code at the top of the screen, and then viewing the console tab that appears on the right side of the screen. To view full button names, widen the tab by clicking the dots on the left border of the console tab and drag that to the left. Once a button is clicked, the results of it can be seen by scrolling to the bottom of the console tab."); //Shows instructions

alert("After clicking OK below when you finish reading this, you will have to type a letter into a text box detailing which of the application's maps you would like to see. To toggle between the application's maps, first refresh the page, then type 'skip' at the first prompt, and then type in a new letter to see a new map. Some maps may be slow to load as indicated by moving dark gray bars on the right of the 'Layers' control, but please have patience as the map you view will load shortly.");

var SecondPrompt = prompt("Please type the letter into the text box below that corresponds with what information about Philadelphia's High Injury Network (HIN) you would like to see:\n\na - City HIN vs. Clipped Philadelphia INRIX datasets to see how joining of congestion data to city HIN worked\nb - HIN with TTI and PTI busiest hour period data versus HIN with just TTI and PTI data joined to see how the busiest hour column creations worked\nc - KSI Crashes v. Congested HIN to see how creation of congested HIN dataset, creation of lengths in miles for each congested road segment, and spatial joining of KSI crash data to congested HIN worked\nd - The Most Congested and Dangerous HIN Roads\ne - The Most Congested and Dangerous Wide HIN Roads\nf - The Most Congested and Dangerous HIN Roads in Poor Neighborhoods\ng - The Most Congested and Dangerous HIN Roads near Rail Stops\nh - The Most Congested and Dangerous HIN Roads in Commercial Corridors");

//After viewing the two alerts with the instructions, the user then gets brought to this second prompt where they enter in the letter of the map they would like to see

}


else if(FirstPrompt=='skip'){ //This makes it so if the user types 'skip' in the first prompt, they simply get brought to the second prompt where they can choose which map to see

var SecondPrompt = prompt("Please type the letter into the text box below that corresponds with what information about Philadelphia's High Injury Network (HIN) you would like to see:\n\na - City HIN vs. Clipped Philadelphia INRIX datasets to see how joining of congestion data to city HIN worked\nb - HIN with TTI and PTI busiest hour period data versus HIN with just TTI and PTI data joined to see how the busiest hour column creations worked\nc - KSI Crashes v. Congested HIN to see how creation of congested HIN dataset, creation of lengths in miles for each congested road segment, and spatial joining of KSI crash data to congested HIN worked\nd - The Most Congested and Dangerous HIN Roads\ne - The Most Congested and Dangerous Wide HIN Roads\nf - The Most Congested and Dangerous HIN Roads in Poor Neighborhoods\ng - The Most Congested and Dangerous HIN Roads near Rail Stops\nh - The Most Congested and Dangerous HIN Roads in Commercial Corridors");

}
```

IF-ELSE STATEMENTS

- **Code that Leads Users to View Individual Maps:**

```
if(SecondPrompt=='a'){ //Means "if user writes 'a' in that text box"...

ui.root.clear(); //MEANS THE DEFAULT MAP WILL BE ELIMINATED. Found out how to do from https://developers.google.com/earth-engine/ui_widgets

ui.root.add(Map1); //ADDS MAP TO THE BOTTOM HALF OF THE SCREEN TO REPLACE THE DEFAULT MAP WHICH WAS ELIMINATED IN THE LINE ABOVE. Found out how to do from
https://developers.google.com/earth-engine/ui_widgets

print(Map1Button1); //THIS MAKES THE BUTTONS ONLY FOR THAT STEP PRINT OUT WHEN THE USER SELECTS a

print(Map1Button2);

}

else if(SecondPrompt=='b'){

ui.root.clear();

ui.root.add(Map2);

print(Map2Button1);

print(Map2Button2);

}

else if(SecondPrompt=='c'){ //Means "if user writes 'b' instead of a or any other letter in that text box"...

ui.root.clear();

ui.root.add(Map3);

print(Map3Button1);

}
```

IF-ELSE STATEMENTS

- **Code that Leads Users to View Individual Maps:**

```
else if(SecondPrompt=='d'){
ui.root.clear();
ui.root.add(Map4);
print(Map4Button1);
print(Map4Button2);
print(Map4Button3);
print(Map4Button4);
}

else if(SecondPrompt=='e'){
ui.root.clear();
ui.root.add(Map5);
print(Map5Button1);
print(Map5Button2);
print(Map5Button3);
print(Map5Button4);
}
```

IF-ELSE STATEMENTS

- **Code that Leads Users to View Individual Maps:**

```
else if(SecondPrompt==‘f'){
ui.root.clear();
ui.root.add(Map6);
print(Map6Button1);
print(Map6Button2);
print(Map6Button3);
print(Map6Button4);
}

else if(SecondPrompt==‘g'){
ui.root.clear();
ui.root.add(Map7);
print(Map7Button1);
print(Map7Button2);
print(Map7Button3);
print(Map7Button4);
}
```

IF-ELSE STATEMENTS

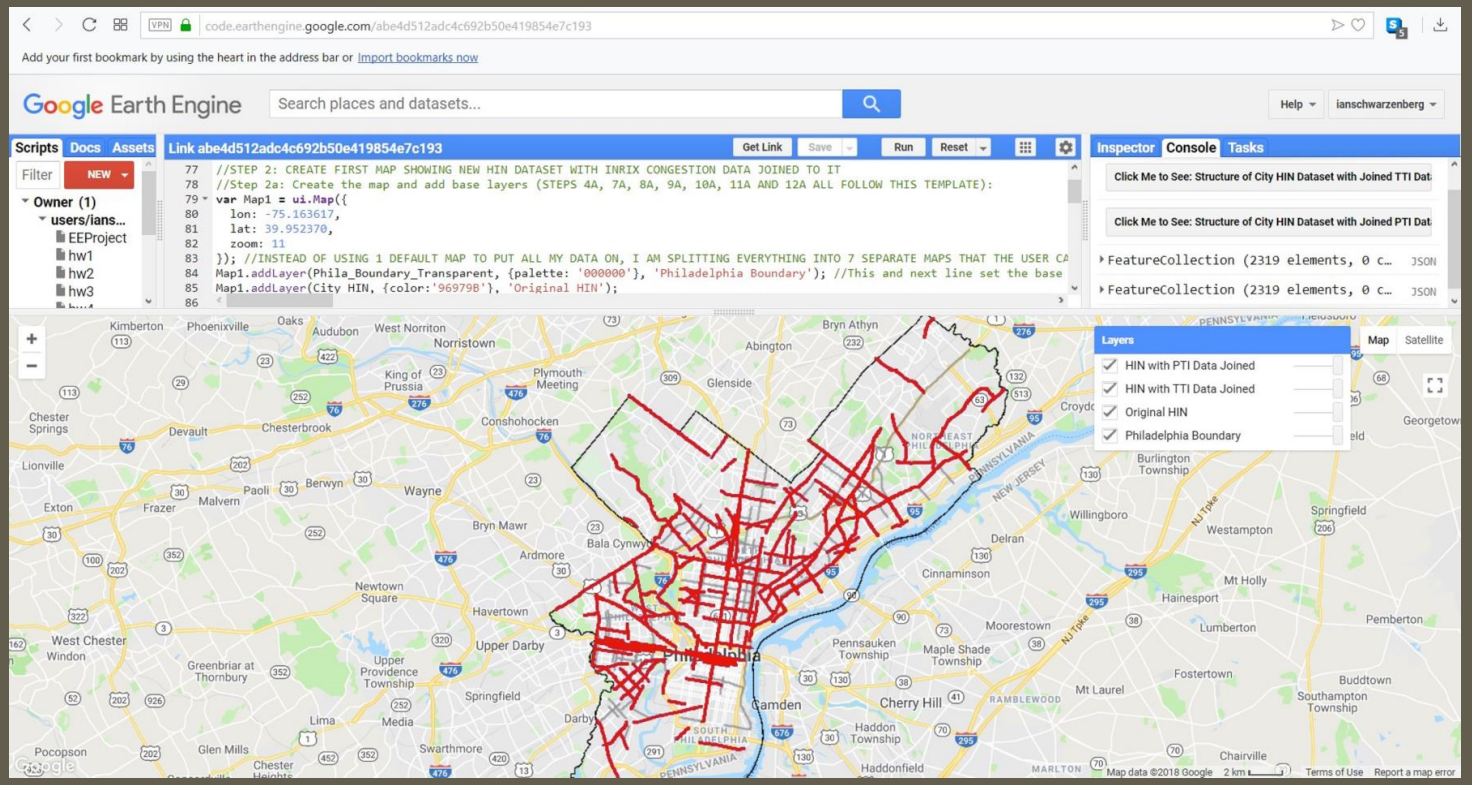
- **Code that Leads Users to View Individual Maps:**

```
else if(SecondPrompt=='h'){
ui.root.clear();
ui.root.add(Map8);
print(Map8Button1);
print(Map8Button2);
print(Map8Button3);
print(Map8Button4);
}
```

CODE EXPLANATION

- **UI Code Explanation:** A series of if-else statements are used to create the user interface:
  - After the first prompt is introduced using the FirstPrompt variable and prompt() command, an if-else statement is created. The *if* part makes it so if a user types 'begin' in the first prompt, it takes them through the series of instructions inside the brackets of that if-else statement. The series of instructions are presented using the alert() command. This is different from prompt() because with prompt(), users have to type in an answer to continue. On the other hand, users just have to click OK with alerts.
  - The *else* part of the statement makes it so if the user typed 'skip' at the first prompt, they automatically get taken to just the second prompt which enables them to choose which map to view. This second prompt is put inside the brackets of the else part of that statement.
  - Then, a *new* if-else statement is created based on the second prompt and eight maps. The if parts determine which letter the user chooses in the second prompt. If that letter is chosen, the default map first gets *eliminated* using ui.root.clear(). "Root" means what was originally there, and "clear" gets rid of it. Then, the default map gets *replaced* with the map the user chose using ui.root.add(). This adds the new map to the root. Then, the buttons for that individual map get printed in the console tab using print().
  - The "else" part means that if a user chose one letter instead of another, then the letter they chose will have its map displayed, as opposed to all the other letters' corresponding maps. All of this essentially means: "if the user chose to see the map for letter a, then only the layers and buttons for a's map will be displayed. Or else, if the user chose to see the map for letter b, then only the layers and buttons for b's map will be displayed., etc."

Map 1/Choice a



Map 1 Overview

SETUP

- The goal of this process and map creation is to make 2 variables: 1) a variable of HIN roads with their hourly TTIs, and 2) a variable of HIN roads with their hourly PTIs.
- The first step was to create the first map that *replaces* the default map. The ui.Map({}) command creates the map to replace the default, and sets its default center and zoom level:

```
var Map1 = ui.Map({
  lon: -75.163617,
  lat: 39.952370,
  zoom: 11
});
    });
});
```

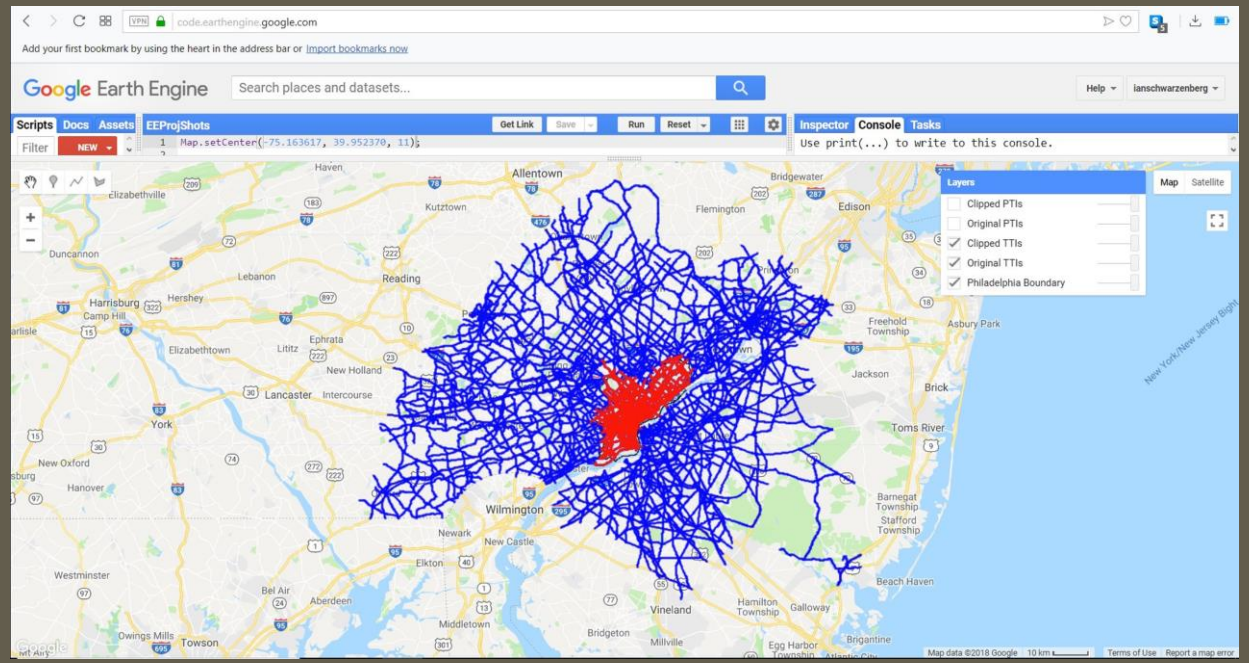
- The second step in creating this map was to load in the Philadelphia boundary, polygon, the City HIN, TTI and PTI lines. Firstly, I made all of the shapefiles into Google Fusion Tables using https://shpescape.com. Secondly, I made the Philadelphia boundary transparent by first creating an empty image using ee.Image.byte(). The "byte" part of the command allows for something specific to be "painted" to it. The empty.paint({}) command is then used to assign the filled Philadelphia boundary polygon to this empty image, thereby making it transparent:

```
var TTIs  = ee.FeatureCollection('ft:1ujHCjJFiyKIQamrVrxiUr2jyi7mlLsakPyCHGDjB');
var PTIs  = ee.FeatureCollection('ft:13mAvrsxdLr4bSTSbbMwhAfTAg2a2DBMo0bg_UyYP');
var City_HIN =  ee.FeatureCollection('ft:1edGX7-c7OBUVTyV7vEzcKqKxgTcXaQPjb1wWAT4e');
var Phila_Boundary = ee.FeatureCollection('ft:1A2rSnITNWDIUUL0A3Z5C8JGZDCRI0QVsnkocUmEl');
var empty = ee.Image().byte();
var Phila_Boundary_Transparent = empty.paint({
  featureCollection: Phila_Boundary,
  color: 1,
  width: 1.25
});
```

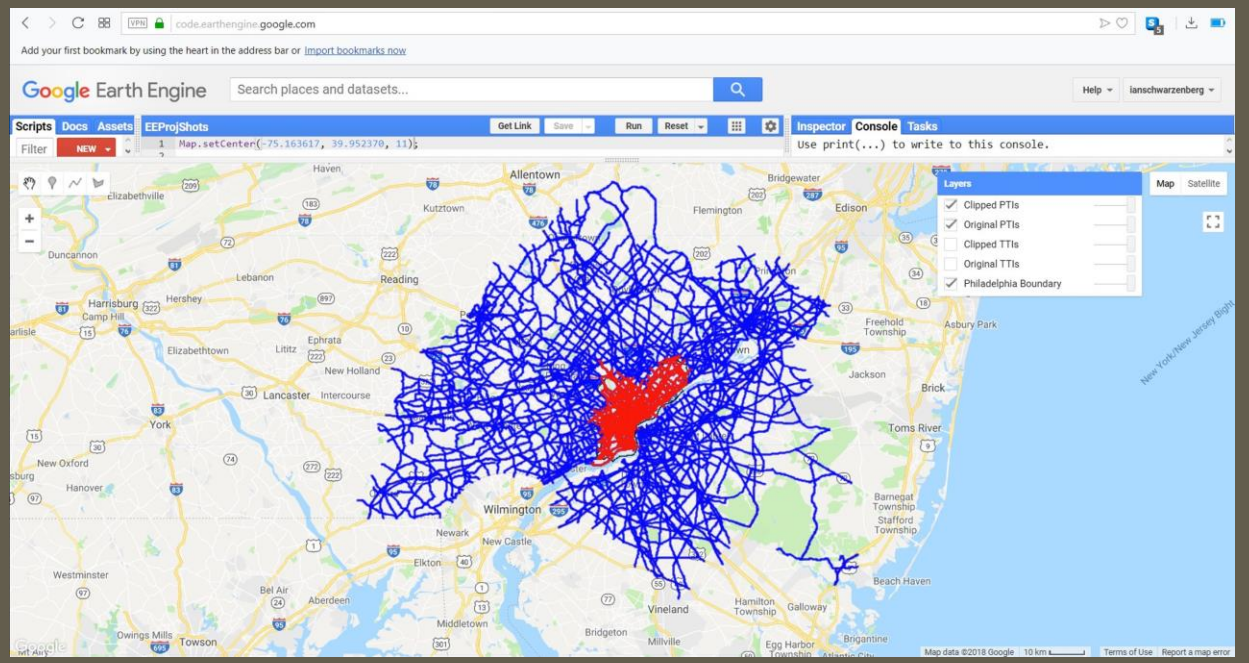
CLIPPING

- The next step was to clip the TTI and PTI line datasets to the Philadelphia boundary. The DVRPC had this data for the entire Philadelphia metropolitan area, and I only wanted it for Philadelphia. The clip is done by first establishing a filter using ee.Filter.bounds(). Then, two new variables are established, one with just the TTIs for all major Philadelphia roads, and one with just PTIs for the same roads:

```
var Phila_Filter = ee.Filter.bounds(Phila_Boundary);

var Phila_TTI = TTIs.filter(Phila_Filter);

var Phila_PTI = PTIs.filter(Phila_Filter);
```



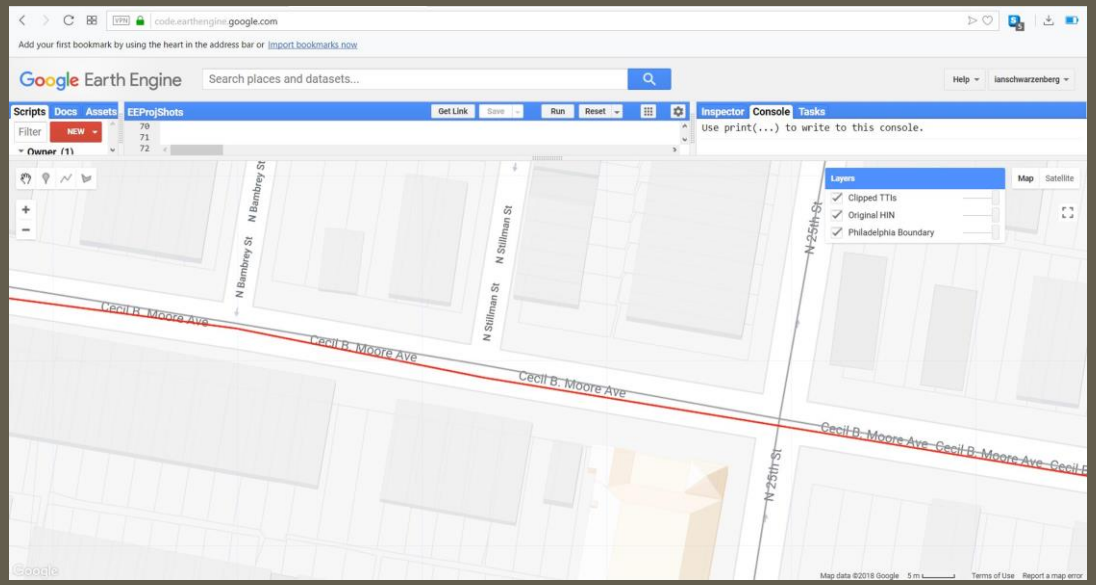
Original TTI Data vs. TTI Data Clipped to Philadelphia Boundary



Original PTI Data vs. PTI Data Clipped to Philadelphia Boundary

INNER JOINING

- The next step was to do an inner join of the TTI and PTI data to the city HIN by street name. This is because the TTI and PTI lines did not overlap directly with the city HIN, preventing the use of an intersection to select out TTI and PTI lines that overlapped.



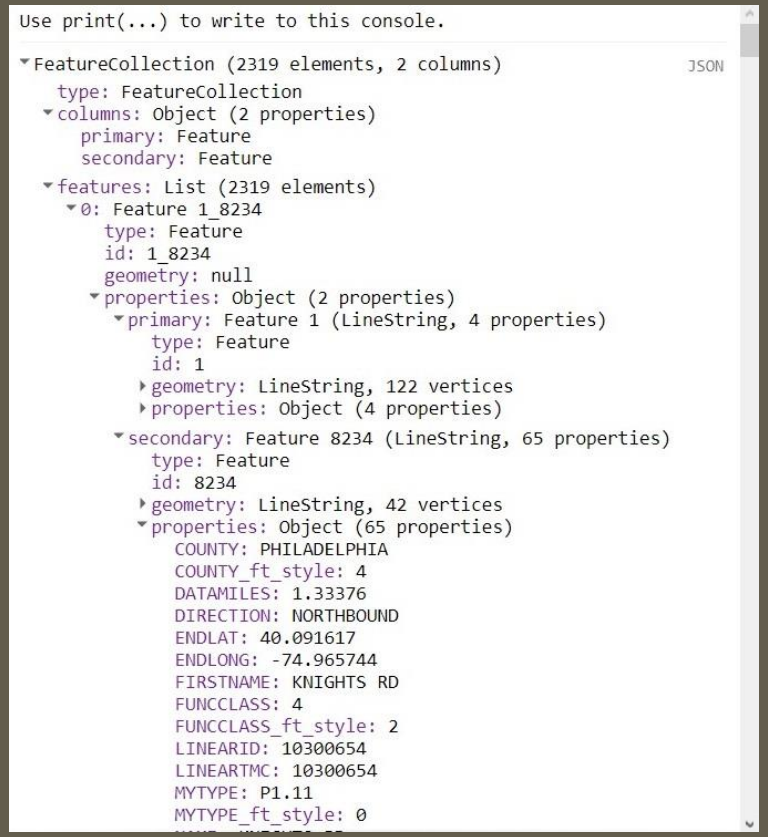
INRIX Lines and City HIN Lines not Overlapping

- The primary features of the inner join were the city HIN since these are the lines I wanted to keep, and the secondary features were TTIs and PTIs. The fields of each dataset representing street names were used to link each TTI and PTI road to its corresponding city HIN road, representing the common identifier between the two datasets:

```
var TheFILTER = ee.Filter.equals('stname', null, 'ROADNAME', null ); //Joins INRIX data to City HIN by their common street names

var TheJOIN = ee.Join.inner('primary','secondary'); //Establishes the join

var HIN_with_TTIs = TheJOIN.apply(City_HIN, Phila_TTI, TheFILTER); //Applies the join, city HIN is primary dataset since I want each city HIN line to have corresponding TTI data
```

INNER JOINING

- The join output produces this, which cannot be mapped because it is *divided* into two parts: its primary and secondary features. This creates problems with Earth Engine reading the features' geometries. In addition, the result of the join contains many columns that are unnecessary for the further analysis. To get around this, a new variable will have to be created that is mappable and that extracts just the columns that I want from the inner join result.



Inner Join Result *before* Mapping over Collection

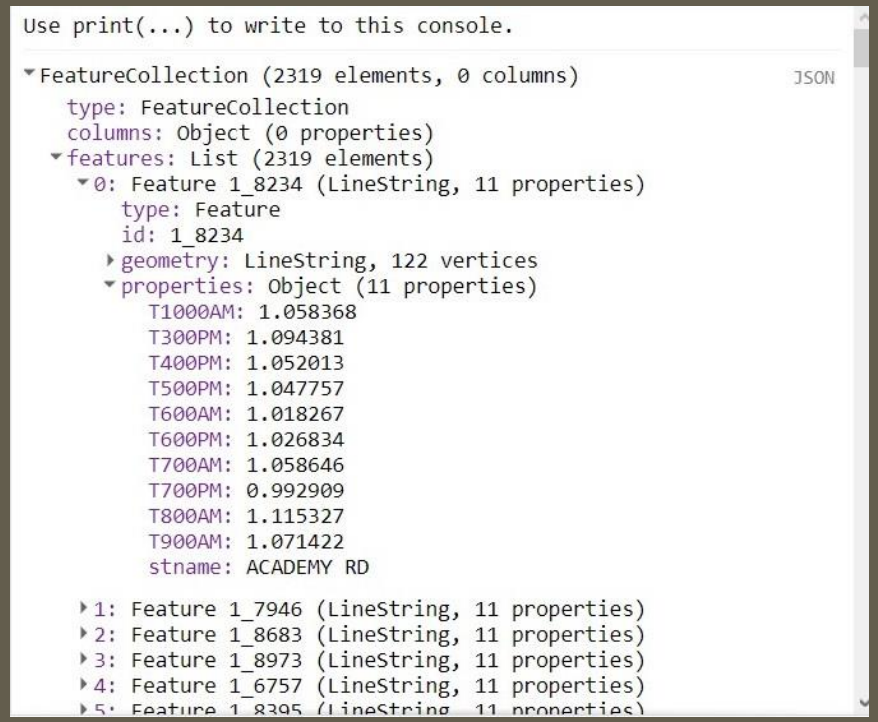
INNER JOINING

- In order to create the new mappable variable that has just the columns I want, I must create a *function* which extracts just the columns I want from the inner join result. This is also called *mapping over the collection,* as evidenced by the HIN_with_TTIs.map() in this example. Firstly, the function establishes two variables just to use in the function which extracts all features from the inner join's primary and secondary feature collections using feature.get(). The function then extracts all the columns I want from the primary and secondary inner join features: HIN street names, their corresponding TTI and PTI data for the busiest hours from 6 AM to 7 PM, and each street line's geometry string. The function then returns a new feature collection using return ee.Feature() which just those columns I defined. This return line also names the columns:

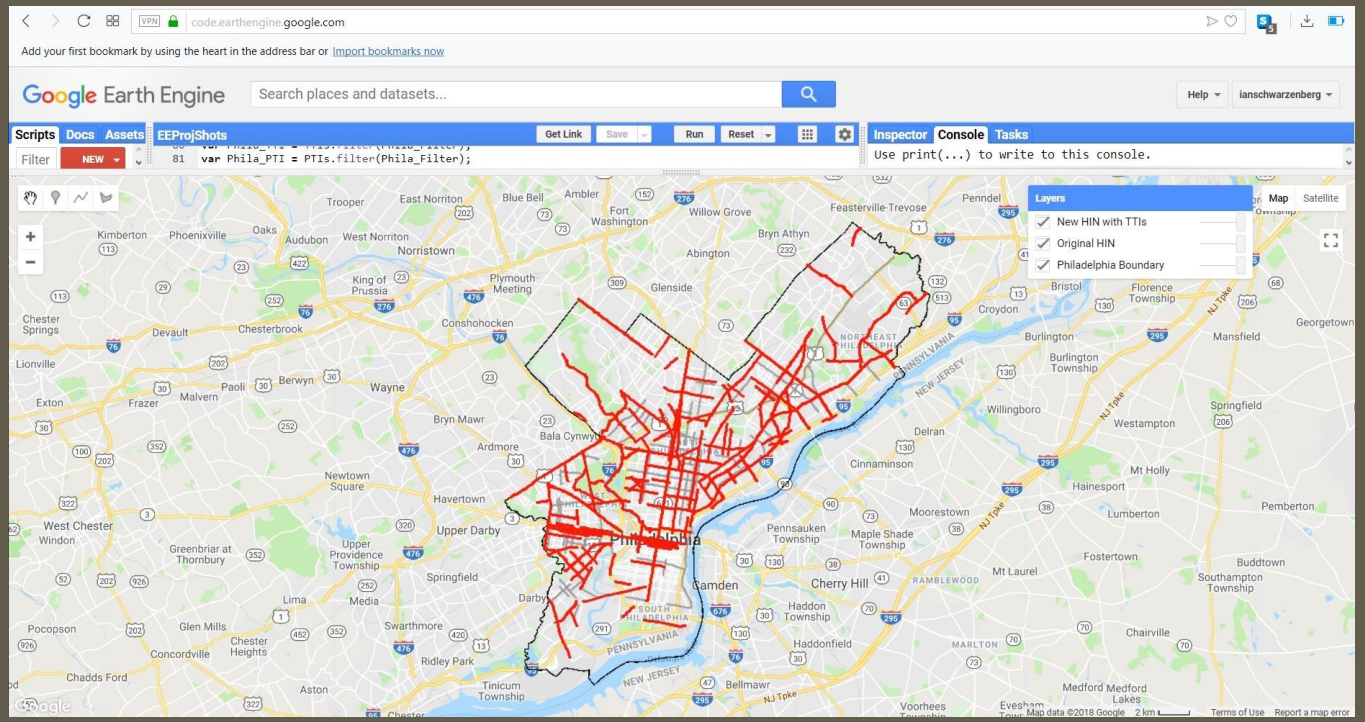
```
var HIN_with_TTIs = HIN_with_TTIs.map(function(element){
    var PRIMARY       = ee.Feature(element.get('primary'));
    var SECONDARY     = ee.Feature(element.get('secondary'));
    var stname        = PRIMARY.get('stname');
    var T600AM        = SECONDARY.get('T600AM');
    var T700AM        = SECONDARY.get('T700AM');
    var T800AM        = SECONDARY.get('T800AM');
    var T900AM        = SECONDARY.get('T900AM');
    var T1000AM       = SECONDARY.get('T1000AM');
    var T300PM        = SECONDARY.get('T300PM');
    var T400PM        = SECONDARY.get('T400PM');
    var T500PM        = SECONDARY.get('T500PM');
    var T600PM        = SECONDARY.get('T600PM');
    var T700PM        = SECONDARY.get('T700PM');
    var geom          = ee.Feature(element.get('primary')).geometry();
    return ee.Feature(geom, {'stname':stname, 'T600AM':T600AM,
    'T700AM':T700AM, 'T800AM':T800AM, 'T900AM':T900AM, 'T1000AM':T1000AM, 'T300PM':T300PM,
    'T400PM':T400PM, 'T500PM':T500PM, 'T600PM':T600PM, 'T700PM':T700PM
    });
});
```

INNER JOINING

- The results of this show how the inner join result is now a mappable feature collection overlapping perfectly with the city HIN despite not having done so before. This is because the city HIN lines were kept as the primary features, but the congestion data was transferred over to the city HIN from the secondary features. The results of this also show how only the columns that I wanted were retained in the new feature collection. This also shows how some city HIN segments were left out. This is because INRIX only collects congestion data for roads it defines as being significant to make the data collection process less overwhelming.[1] As a result, some city HIN roads do not have congestion data because INRIX sees them as less significant, presenting a limitation to the project.



Inner Join Result *after* Mapping over Collection



Inner Join Result Mapped

[1] Told by DVRPC Senior Planner Tom Edinger to Ian Schwarzenberg on 9/21/18.

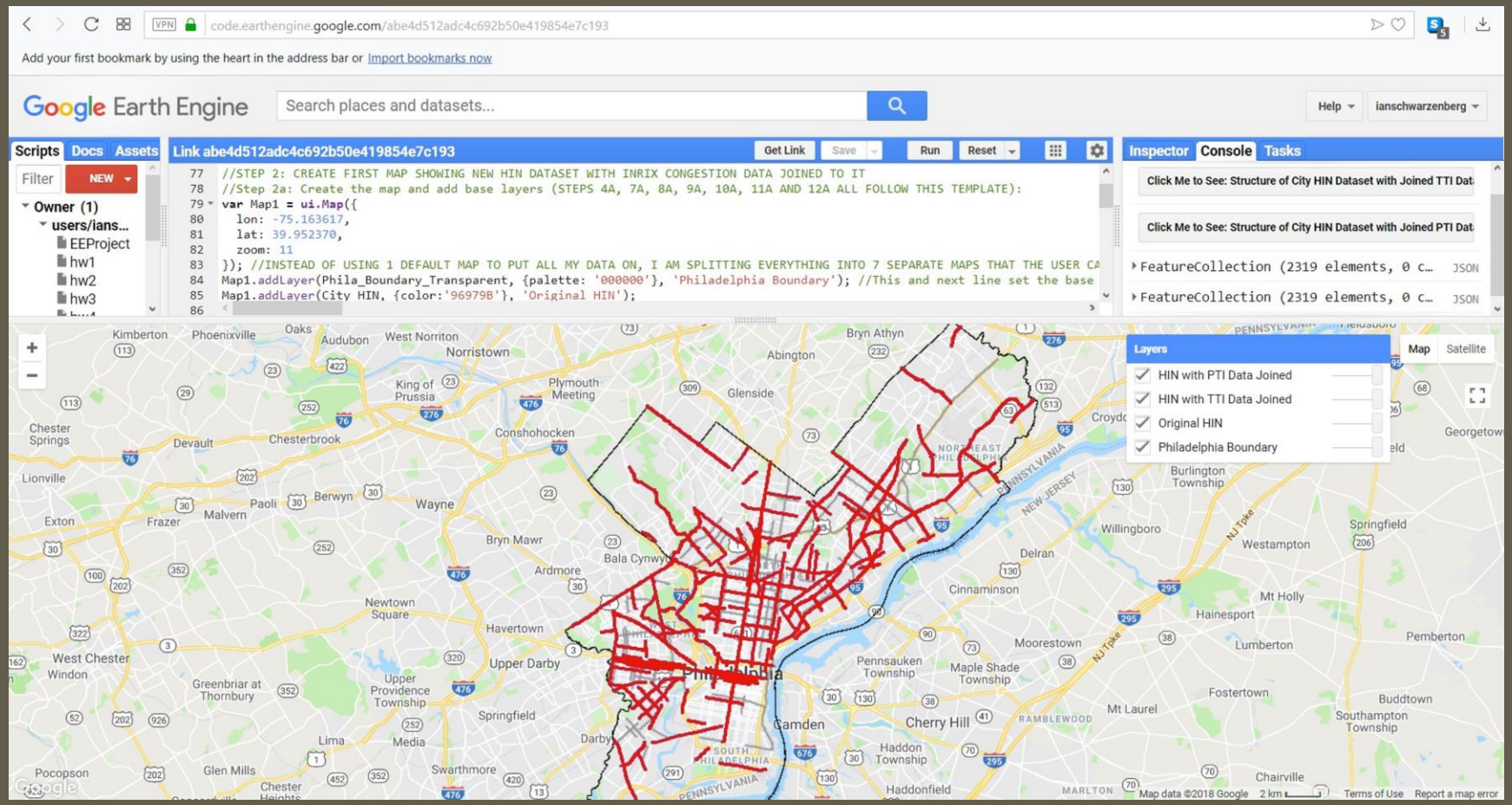
BUTTONS

- The exact same inner joining and mapping-over-collection process that was taken for the TTI dataset was also taken for the PTI dataset, since INRIX gave the TTI and PTI data the exact same properties. This allowed for the code used to make the inner join and mapping-over-collection function to be used as templates for the HIN with PTIs dataset creation.
- After this, two *buttons* were created to let the user view the structures of the datasets previously described before and after the mapping-over-collection process. The ui.Button({}) command is used to create the buttons which appear in the console tab. First the buttons are named using label:. Then, a function is created called onClick which then has a command to simply print the structures of the datasets. onClick allows for those print commands to be executed once the user *clicks* on the button:

```
var Map1Button1 = ui.Button({
  label: 'Click Me to See: Structure of City HIN Dataset with Joined TTI Data', //Names button
  onClick: function() {
    print(HIN_with_TTIs); //Shows the line of code that will be executed on the back end when user clicks button
  }
});
var Map1Button2 = ui.Button({
  label: 'Click Me to See: Structure of City HIN Dataset with Joined PTI Data',
  onClick: function() {
    print(HIN_with_PTIs);
  }
});
```

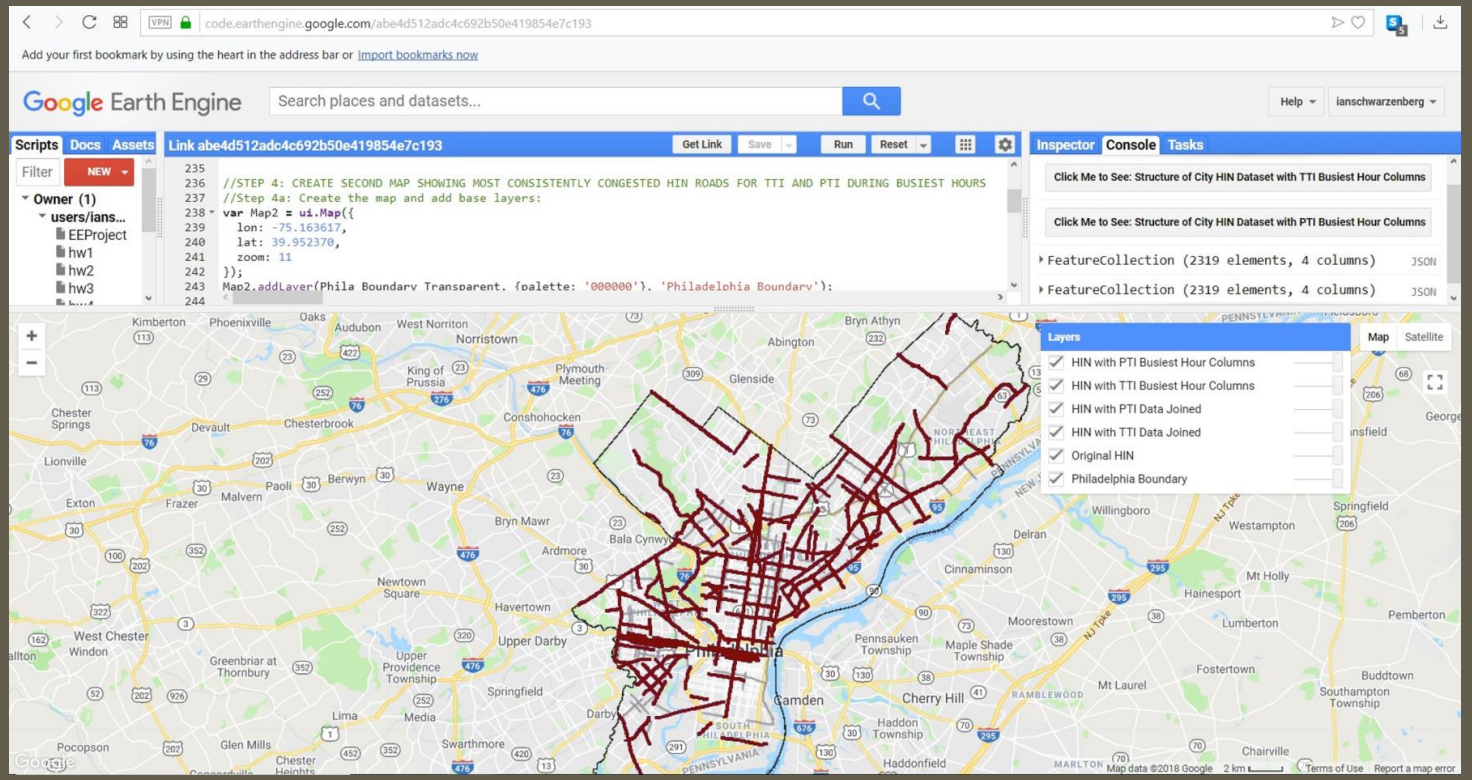
FINAL MAP

- Once all of those steps were completed, the final map was made. Because this and all of the application's other maps replace Earth Engine's default map, the Map.addLayer() command *cannot* be used. This is because "Map" in that instances refers to the default map. Therefore, *Map1*.addLayer() had to be used whenever layers were added to the map, since Map1 was already defined in the ui.Map({}) command used earlier to create the map. The purpose of this map is simply to show the user the end result of joining congestion data to the HIN.



Map 1 Final

Map 2/Choice b



Map 2 Overview

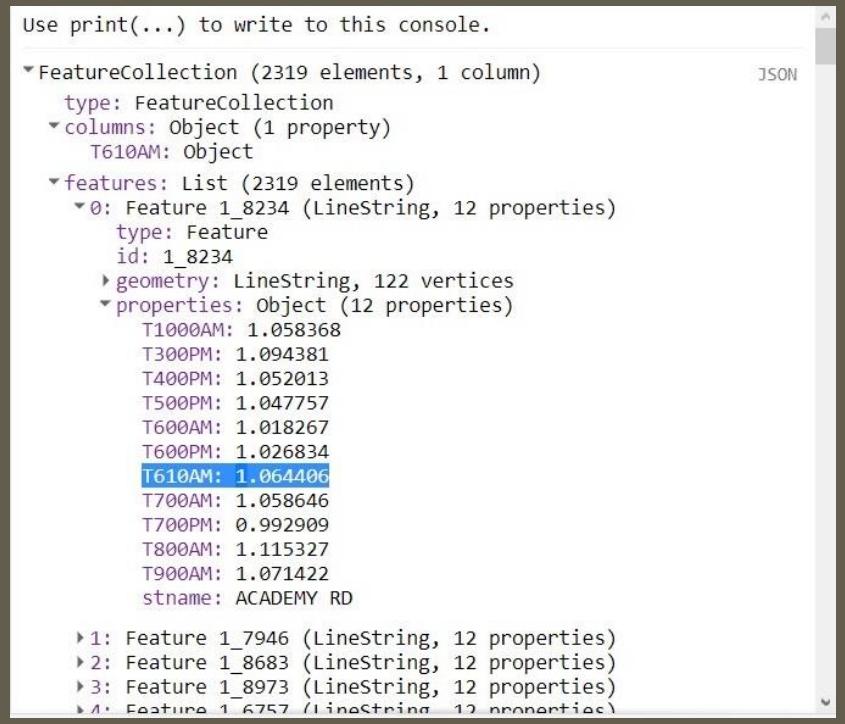
BUSY HOUR AVERAGING

- At this point in the analysis process, I now have: 1) a variable of HIN roads with their hourly TTIs, and 2) the same exact variable as 1), but for PTIs.
- The goal of this process and map creation is to make 2 variables: 1) a variable of HIN roads with their hourly TTIs and 4 new columns showing their average TTIs for the busiest timeframes, and 2) the same exact variable as 1), but for PTIs.
- After defining Map2 using the ui.Map({}) command the same way as was done in Map1 and in all the other application's maps, the next step in making this map was to create functions which make new columns that find the average TTI or PTI for specific busy timeframes. Four different functions that do this were created: One for the morning rush hour, one for the morning peak hour, one for the evening rush hour and one for the evening peak hour. The code example below shows the function created for the morning rush hour. The function first extracts the numeric TTI or PTI feature values for the hours that are necessary for the calculation using ee.Number(feature.get()). In this instance, the TTI and PTI columns that are extracted range from 6 to 10 AM, since these hours make up the morning rush hour time period. The function then establishes a mean reducer using ee.Reducer.mean() to prepare for the averaging calculation. The function creates a new numeric column for the collection being mapped, in this example called 'T610AM' to indicate average TTI or PTI for that time frame. This is done by making a numeric list of the individual hour columns using ee.Number(ee.List([]), and then applying the mean reducer to that. The function then finally returns a new feature of the final new mean column, in this case called 'T610AM', using return feature.set(). This function template can notably be mapped over both TTI and PTI collections, since INRIX gave the same exact hourly column names to both their TTI and PTI datasets. The function template below was repeated for the other three busiest hours for both TTI and PTI:

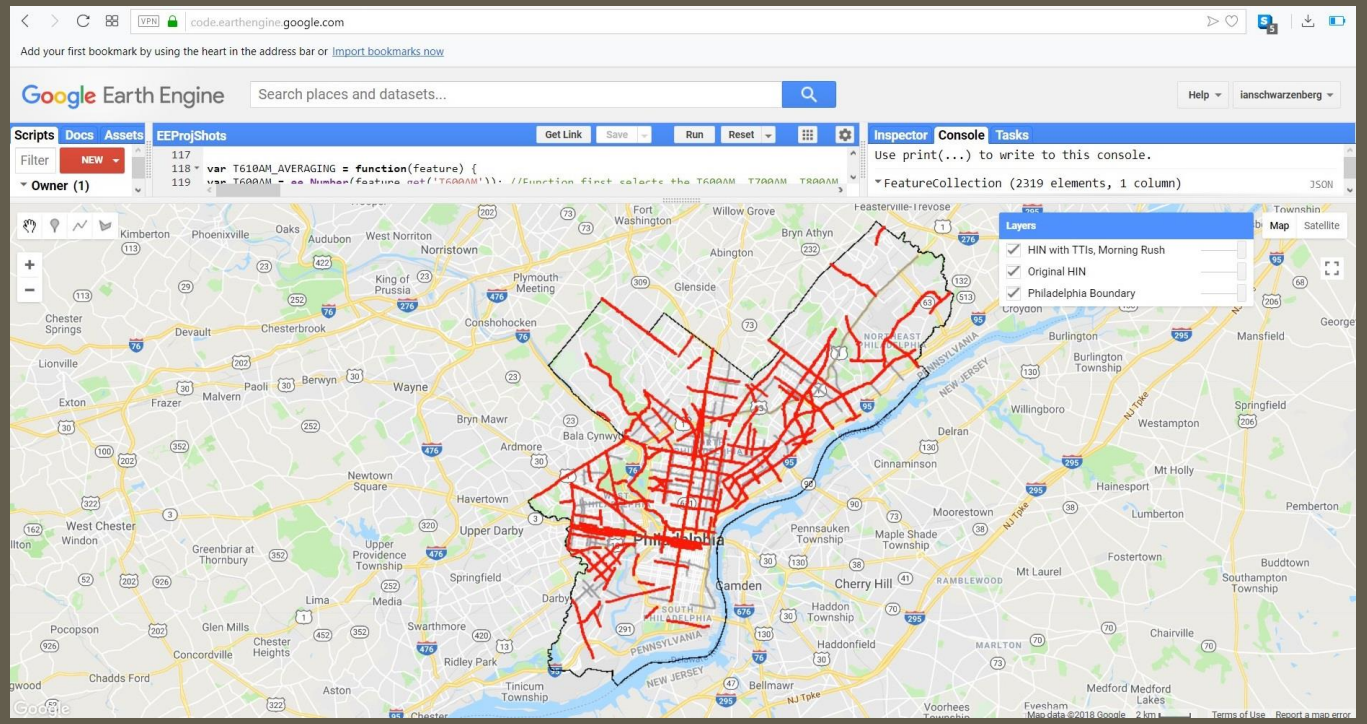
```
var T610AM_AVERAGING = function(feature) {
var T600AM = ee.Number(feature.get('T600AM')); //Function first selects the T600AM, T700AM, T800AM, T900AM and T1000AM columns
var T700AM = ee.Number(feature.get('T700AM'));
var T800AM = ee.Number(feature.get('T800AM'));
var T900AM = ee.Number(feature.get('T900AM'));
var T1000AM = ee.Number(feature.get('T1000AM'));
var mean = ee.Reducer.mean(); //Function then creates mean reducer
var T610AM = ee.Number(ee.List([T600AM, T700AM, T800AM, T900AM, T1000AM]).reduce(mean)) //Function then finds mean of T600AM, T700AM, T800AM, T900AM and T1000AM columns using mean reducer
return feature.set({ 'T610AM' : T610AM}) //Function then creates new column called TTI610AM and fills it in with the means of T600AM, T700AM, T800AM, T900AM and T1000AM columns
};

var HIN_with_TTIs_WithMorningRush = HIN_with_TTIs.map(T610AM_AVERAGING);
var HIN_with_PTIs_WithMorningRush = HIN_with_PTIs.map(T610AM_AVERAGING);
```

BUSY HOUR AVERAGING

- The results of this example show how the city HIN with joined TTI data now has a new column with the average TTI for just that morning rush hour, and how it is also mapped.



Column Creation Result



Column Creation Result Mapped

FINAL MAP ITEMS

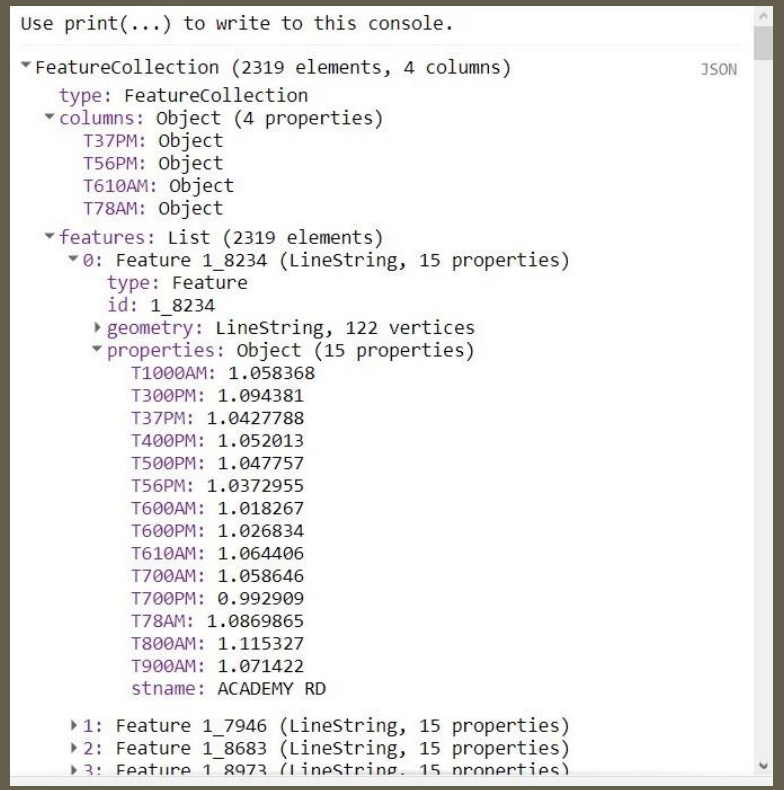
- Once separate variables were created for both TTI and PTI that included busy hour average columns, two final variables were created for this map: One variable showing the final HIN with joined TTI data and busy hour columns, and one variable showing the exact same but for PTI. This was done by mapping these column-creator functions over each collection mapping result in succession. In this example, once a variable was created showing HIN streets' TTIs with a morning rush hour average TTI column, then a new variable was created adding a new column to that exact same dataset, but with a evening rush hour average TTI added on, and so forth. At the end of this example, a new variable gets created with four new columns, each one showing an HIN road's average TTI for each busy timeframe. This exact same template was repeated for PTIs as well:

```
var HIN_with_TTIs_WithMorningRush = HIN_with_TTIs.map(T610AM_AVERAGING);

var HIN_with_TTIs_WithEveningRush = HIN_with_TTIs_WithMorningRush.map(T37PM_AVERAGING);

var HIN_with_TTIs_WithMorningPeak = HIN_with_TTIs_WithEveningRush.map(T78AM_AVERAGING);

var HIN_with_TTIs_Final = HIN_with_TTIs_WithMorningPeak.map(T56PM_AVERAGING);
```

- Two buttons were also created to show the user the structure of the datasets exactly

like in the picture to the right:

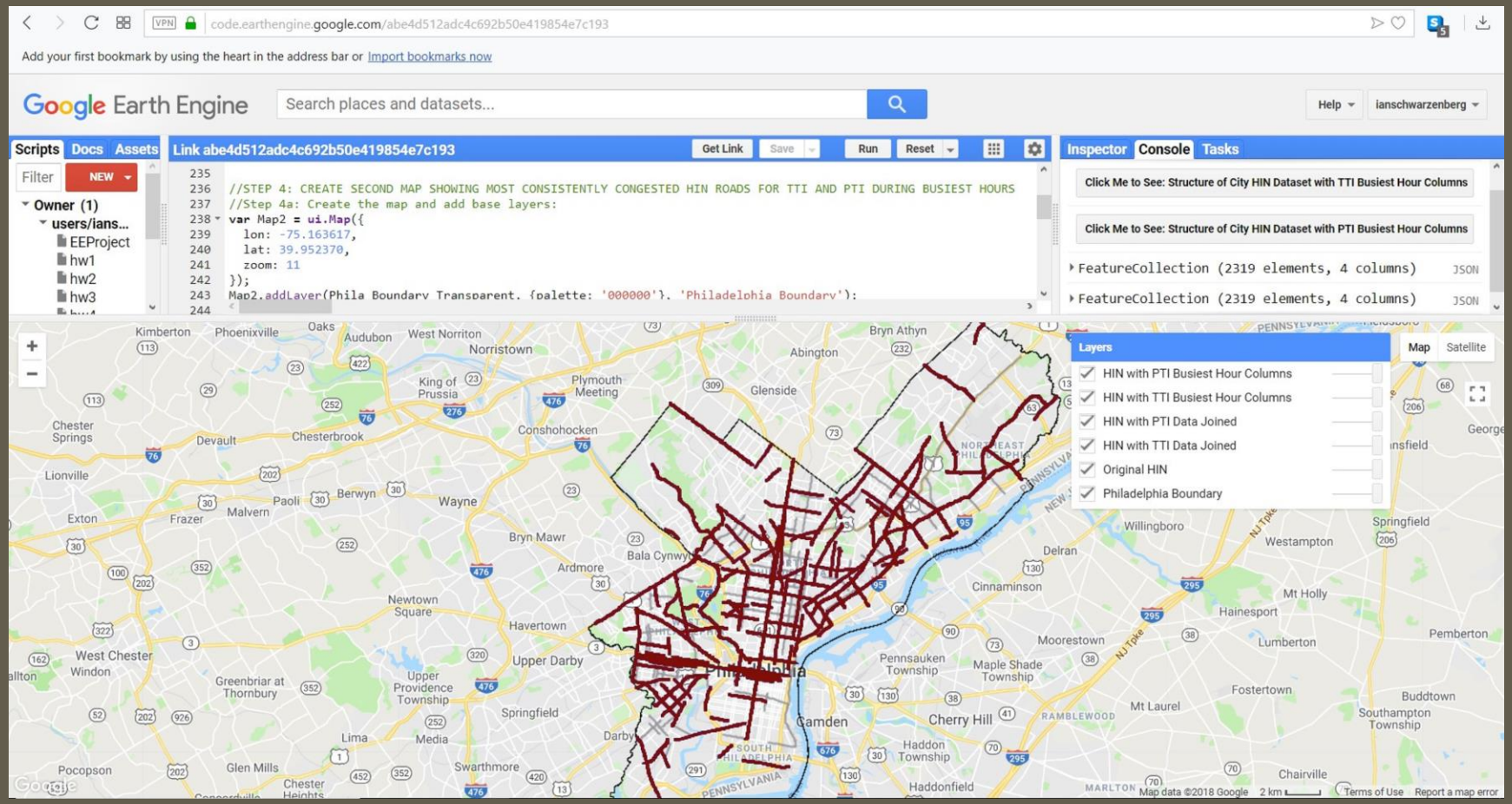
```
var Map2Button1 = ui.Button({
  label: 'Click Me to See: Structure of City HIN Dataset with TTI Busiest Hour Columns',
  onClick: function() {
    print(HIN_with_TTIs_Final);
  }
});

var Map2Button2 = ui.Button({
  label: 'Click Me to See: Structure of City HIN Dataset with PTI Busiest Hour Columns',
  onClick: function() {
    print(HIN_with_PTIs_Final);
  }
});
```



All Columns Creation Result

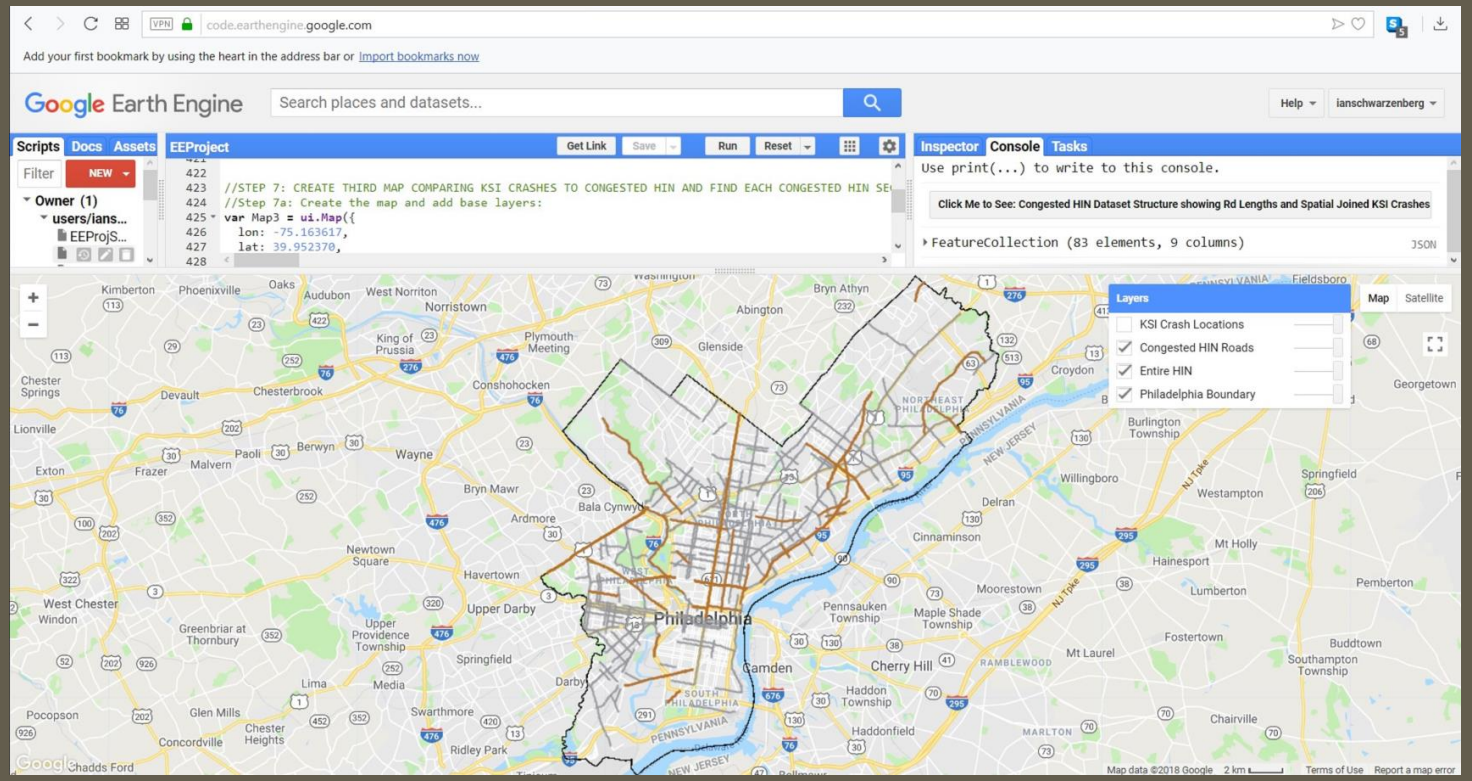
FINAL MAP

- Once the busiest hour column creation was completed, for both TTIs and PTIs, and once the button creation process was completed, the final map was made. The purpose of this map is simply to show the user the end result of the process of averaging each HIN road's TTI and PTI data by the busiest timeframes.



Map 2 Final

Map 3/Choice c



Map 3 Overview

INTERSECTING AND FILTERING

- At this point in the analysis process, I now have: 1) a variable of HIN roads with their hourly TTIs and 4 new columns showing their average TTIs for the busiest timeframes, and 2) the same exact variable as 1), but for PTIs.
- The goal of this process and map creation is to: 1) make a new variable of just the HIN roads that are congested across *all* of the busiest timeframes in terms of *both TTI and PTI*, 2) create a column in that variable showing each *congested* HIN road's KSI crash *count,* and 3) create a column in that variable showing each *congested* HIN road's KSI crash *rate*
- The first step after establishing the map with ui.Map({}) was to establish a *special intersection method* to use for the rest of the project. At first, I used feature.intersection() whenever I wanted to intersect shapefiles with one another in Earth Engine. However, I ran into a problem where the variables resulting from this method could not be read by Earth Engine as mappable feature collections. As a result, I created my own intersection method to use for the rest of the project, since that problem did not occur with this. This method first uses a distance filter, with distances of 0, which tests whether the geometry of a primary feature overlaps with the geometry of a secondary feature. This method then uses a saveAll join, which saves only the elements of a primary feature with geometries that overlap with the geometry of a secondary feature. I call the resulting field from this process 'JOINED' and the distance field 'DISTANCE'.

```
var intersection_Filter = ee.Filter.withinDistance( 0, '.geo', null, '.geo' ); //Establishes filter where primary features within 0 meters of secondary features (intersecting features)

//get selected

var intersection_Join = ee.Join.saveAll( 'JOINED', null, true,'DISTANCE' ); //Uses saveAll join to retain elements of primary feature that intersect with secondary feature
```

INTERSECTING AND FILTERING

- The next step was to then filter the city HIN with TTIs and PTIs variables created in the previous map using feature.filterMetadata() to select out HIN roads marked as congested during the busiest times. This meant using that command to select out roads with TTIs over 1.5 and roads with PTIs over 3 in the busiest hour average columns created in the previous map.
- The next step was to then intersect the rush and peak hour congested HIN roads variables using the special intersection method (intersection_Join.apply(primary feature name, secondary feature name, intersection_Filter)) to get the HIN roads congested during morning and evening. For example, intersecting HIN roads with TTIs above 1.5 during morning rush hours with ones that fit the same criteria for morning peak hours results in a variable containing the HIN roads consistently congested throughout morning.
- The next step was to then intersect the morning and evening congested HIN roads variables with each other to get one variable showing HIN roads consistently congested throughout the day in terms of TTI, and one that fits the same criteria but for PTI.
- The next step was to then intersect the road congested throughout the day in terms of TTI with the ones in terms of PTI to get the final most congested HIN variable:

```
var HIN_TTI_Congested_MorningPeak = HIN_with_TTIs_Final.filterMetadata( 'T78AM', 'greater_than', 1.5 ); //Filters all HIN roads by their average TTI for morning peak hour
var HIN_TTI_Congested_Morning = intersection_Join.apply(HIN_TTI_Congested_MorningRush, HIN_TTI_Congested_MorningPeak, intersection_Filter); //"Intersects" using my intersection method
var HIN_TTI_Congested_EveningRush = HIN_with_TTIs_Final.filterMetadata( 'T37PM', 'greater_than', 1.5 ); //Filters all HIN roads by their average TTI for evening rush hour
var HIN_TTI_Congested_EveningPeak = HIN_with_TTIs_Final.filterMetadata( 'T56PM', 'greater_than', 1.5 ); //Filters all HIN roads by their average TTI for evening peak hour
var HIN_TTI_Congested_Evening = intersection_Join.apply(HIN_TTI_Congested_EveningRush, HIN_TTI_Congested_EveningPeak, intersection_Filter);
var HIN_TTI_Congested_MorningandEvening = intersection_Join.apply(HIN_TTI_Congested_Morning, HIN_TTI_Congested_Evening, intersection_Filter);

var HIN_PTI_Congested_MorningRush = HIN_with_PTIs_Final.filterMetadata( 'T610AM', 'greater_than', 3 );
var HIN_PTI_Congested_MorningPeak = HIN_with_PTIs_Final.filterMetadata( 'T78AM', 'greater_than', 3 );
var HIN_PTI_Congested_Morning = intersection_Join.apply(HIN_PTI_Congested_MorningRush, HIN_PTI_Congested_MorningPeak, intersection_Filter);
var HIN_PTI_Congested_EveningRush = HIN_with_PTIs_Final.filterMetadata( 'T37PM', 'greater_than', 3 );
var HIN_PTI_Congested_EveningPeak = HIN_with_PTIs_Final.filterMetadata( 'T56PM', 'greater_than', 3 );
var HIN_PTI_Congested_Evening = intersection_Join.apply(HIN_PTI_Congested_EveningRush, HIN_PTI_Congested_EveningPeak, intersection_Filter);
var HIN_PTI_Congested_MorningandEvening = intersection_Join.apply(HIN_PTI_Congested_Morning, HIN_PTI_Congested_Evening, intersection_Filter);

var HIN_Congested = intersection_Join.apply(HIN_TTI_Congested_MorningandEvening, HIN_PTI_Congested_MorningandEvening, intersection_Filter); //Final congested HIN layer
```

SPATIAL JOINING

- To assess danger along both the congested and non-congested HIN, the next step was to create a spatial join method to prepare for spatially joining KSI crashes to congested and non-congested HIN roads.
- Firstly, another distance filter was created, but this time with a distance of 15.24 meters instead of 0 meters. This measure was chosen because I wanted to capture all points within 50 feet of an HIN street line, since 15.24 meters = 50 feet. I assumed that 100 feet was the width of a typical street, so 50 feet in either direction of a street line would capture all KSI crashes happening on it. I chose a maximum error of 0 to get the best results.
- Secondly, another saveAll join was used so the HIN street segments could capture all KSI crash points that met the criteria of being 50 feet away from them. This is why the matches key was set to 'points' for the saveAll join.
- Therefore, the point of this spatial join method is to join all KSI crashes within 50 feet of both congested and non-congested HIN street segments to those segments so that each HIN street segment could get a *count* of KSI crashes happening on it. This is important for calculating each street's KSI crash *rate* shortly:

```
var distFilter = ee.Filter.withinDistance({
  distance: 15.24,
  leftField: '.geo',
  rightField: '.geo',
  maxError: 0
}); // Define a spatial filter, with distance 15.24 meters (15.24 meters = 50 feet, width of a typical street in either direction of a street centerline to capture all crashes happening on those
//streets).

var distSaveAll = ee.Join.saveAll({
  matchesKey: 'points',
  measureKey: 'distance'
}); // Define a saveAll join, since each street gets all points joined to it saved, making up the spatial join
```

COLUMN FUNCTIONS

- Then, a function which creates a new column with each HIN road segment's crash count was created. This first extracts the list called "points" which results from the spatial join through ee.List(feature.get('points')). It then creates a new variable inside the function called 'KSICrshCnt' which counts the number of items in the "points" list for each HIN segment, thereby getting each street's number of KSI crashes. It does this using *list*.length(), which details the number of items in the list. This is *different* from *feature*.length(), which gets the length in distance of each feature's geometry. The function then returns a new column with the KSI crash counts using return feature.set({}):

```
var KSICrashCount_Finder = function(feature) {
  var points = ee.List(feature.get('points')); //Part of KSI crash count column creation function which extracts the list called "points" containing the points joined to each street line segment
  var KSICrshCnt = ee.Number(points.length()); //Part of KSI crash count column creation function which counts the number of items in the "points" list using "length()" ("length" means NUMBER OF
  //ITEMS IN THE LIST, AS OPPOSED TO "DISTANCE" LENGTH), detailing each street segment's KSI crash count
return feature.set({ 'KSICrshCnt' : KSICrshCnt}) //Creates the actual KSI crash column
};
```

- Then, a function which finds each HIN road segment's length in miles and makes a new column out of that was created. This first takes each feature's length using feature.length() and divides it by 1609.344 to get each feature's length in miles. This is because 1 mile = 1609.344 meters. The function then returns a new column called 'length' with the miles equivalent of each road's length.

```
var Length_Finder = function(feature) {
  var length = ee.Number(feature.length().divide(1609.344)); //Calculates each feature's length. Divides the meter length value by 1609.344 to get miles equivalent Found out how to calculate
  //length from https://www.earthdatascience.org/tutorials/basic-polygon-operations-google-earth-engine/
  return feature.set({ 'length' : length}) //Creates the actual length column but does not apply length calculations to entire dataset until next line
};
```

COLUMN FUNCTIONS

- Then, a function which calculates each HIN street segment's KSI crash *rate* was created. This function first extracts the numeric 'KSICrshCnt' column created in the previous function from the feature collection being mapped over using ee.Number(feature.get()). It then does the exact same thing as it did for the KSI crash count column, but with the length column instead. It then creates an entirely new column called 'KSICrshRte' by dividing the KSICrshRte column by the length in miles column to get each HIN street's KSI crash rate, or number of KSI crashes per mile. It then finally returns a feature collection with the KSI crash rate column added to it using return feature.set({}):

```
var KSICrashRate_Finder = function(feature) {
  var points = ee.Number(feature.get('KSICrshCnt')); //Part of KSI crash rate per mile column creation function which extracts the KSI crash count column
  var lengthmi = ee.Number(feature.get('length')); //Part of KSI crash rate per mile column creation function which extracts the road length in miles column
  var KSICrshRte = ee.Number(points.divide(lengthmi)); //Part of KSI crash rate per mile column creation function which divides each road's number of KSI crashes by each road's length in miles,
  //to get each road's KSI crash rate per mile
return feature.set({ 'KSICrshRte' : KSICrshRte}) //Creates the actual KSI crash rate per mile column
};
```

- All three of these functions are simply tools which get applied to variables in order to produce each function/tool's results in them. This means that none of this column creation actually happens until the functions are applied to the variables themselves. For now, the commands in these functions are essentially instructions for Earth Engine to follow once these functions are applied to the variables themselves.

CRASH DATA

- Then, the data for all crashes is loaded in to Earth Engine. Since PennDOT breaks their crash data into separate CSVs by years, I first loaded in each CSV, then merged all the years together by twos to get a final crashes CSV from 2013-2017.
- I then exported that final crashes CSV variable to Google Drive in order to make that a fusion table shapefile that could be mapped in Earth Engine. Once exported, I opened that CSV in Google Fusion Tables and made it read the table's latitude and longitude columns as a "two-column" location. This way, Earth Engine could read this fusion table as a point shapefile when loaded back into Earth Engine.
- I then loaded in this new fusion table back into Earth Engine, so that now it could be read as a point shapefile of all crash locations regardless of severity:

```
var Crashes13CSV = ee.FeatureCollection('ft:1dpTKRh1MOzgiZCXz7_Znsu1tUg4GHGUJug_Q-57m'); // Loads in Philadelphia 2013 crashes CSV (Source: Pennsylvania Department of Transportation)
var Crashes14CSV = ee.FeatureCollection('ft:1M4_85ksqykxxhMiG4gs3IBsA3W49U5oPyqjnRGoV'); // Loads in Philadelphia 2014 crashes CSV (Source: Pennsylvania Department of Transportation)
var Crashes15CSV = ee.FeatureCollection('ft:1TC0DJv52j2F5Y-g27sn0yPAYCSiRKt_wi_4D4QXG'); // Loads in Philadelphia 2015 crashes CSV (Source: Pennsylvania Department of Transportation)
var Crashes16CSV = ee.FeatureCollection('ft:1Ea4aVCOpd4tOT4SYi-2jNVYGvRkgLqWK6LGhltRX'); // Loads in Philadelphia 2016 crashes CSV (Source: Pennsylvania Department of Transportation)
var Crashes17CSV = ee.FeatureCollection('ft:14ALBJyASt4tjWEfFBByTHfODwKEXUrbGvsouomms'); // Loads in Philadelphia 2017 crashes CSV (Source: Pennsylvania Department of Transportation)

var CrashesCSV = Crashes13CSV.merge(Crashes14CSV);
var CrashesCSV = CrashesCSV.merge(Crashes15CSV);
var CrashesCSV = CrashesCSV.merge(Crashes16CSV);
var CrashesCSV = CrashesCSV.merge(Crashes17CSV);

//Export.table.toDrive({
//  collection: CrashesCSV,
//  description:'Crashes_All',
//  fileFormat: 'CSV'
//}); //Creates a shapefile out of the final crashes CSV, would not export to SHP due to field names being longer than 10 characters, so I exported it as a CSV to my drive, found out how to do
//from https://developers.google.com/earth-engine/exporting. Made this command in comments so that it does not run again

//Created new fusion table in Google Drive from this CSV
//Made Google read the table's latitude and longitude columns as a "two-column" location in Google Sheets so Earth Engine could read this as a point shapefile
//(found out how to do from https://support.google.com/fusiontables/answer/175922?hl=en)


var Crashes_All = ee.FeatureCollection('ft:12EFFwZsDvX2W6W7II9k5vJFccMD0tVTxwDgj2D60'); //Now Google Earth reads the 2013-17 crash points in as a shapefile
```

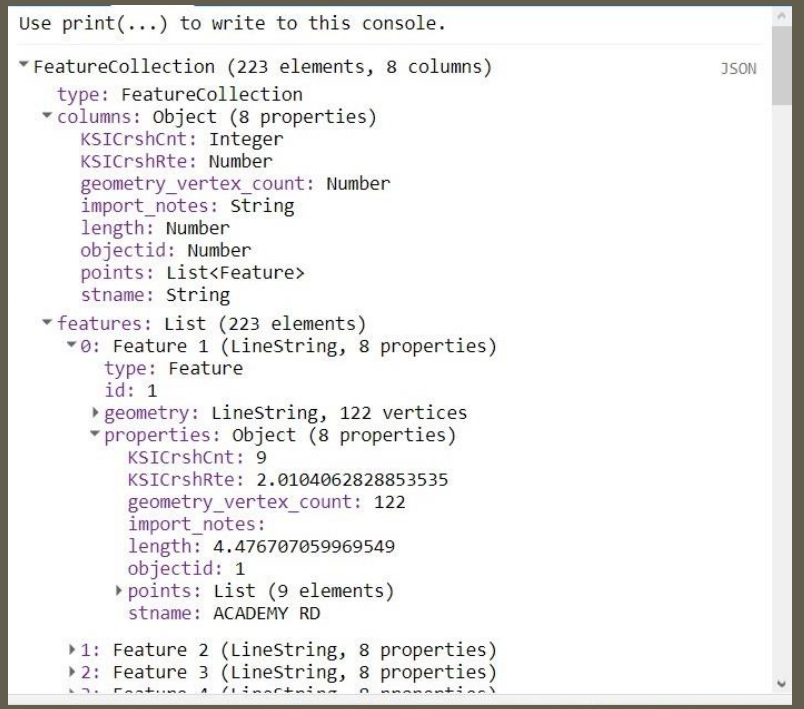
KSI CRASHES

- Then, I filtered out which crashes qualified as KSI. According to PennDOT's metadata for their crash CSVs, the max severity level column indicates crashes' levels of severity. Using feature.filterMetadata(), I selected out all crashes with codes that indicated that they resulted in fatalities or serious injuries with certainty. After selecting out fatal and serious injury crashes in separate variables, I merged them to get my final KSI crashes (killed and severe injury) variable:

```
var Killed_Crashes = Crashes_All.filterMetadata('MAX_SEVERITY_LEVEL', 'equals', 1);
var SevereInjury_Crashes = Crashes_All.filterMetadata('MAX_SEVERITY_LEVEL', 'equals', 2);
var KSI_Crashes = Killed_Crashes.merge(SevereInjury_Crashes); //Merges killed and severe injury crashes for CONGESTED HIN to get FINAL KSI CRASHES ON CONGESTED HIN
Map3.addLayer(KSI_Crashes, {color:'C7C7C7'}, 'KSI Crash Locations', false); //ADDS KSI CRASH POINTS TO MAP, false makes it so the layer is not turned on automatically
```

- Then, I applied my three functions and spatial join created just before this to the entire city HIN. This gave each city HIN road its KSI crash rate. I spatially joined the crashes to the streets, then created the three new columns:

```
var City_HIN = City_HIN.map(Length_Finder);
var CityHIN_KSICrashes_SpatialJoin = distSaveAll.apply(City_HIN, KSI_Crashes, distFilter); //Spatially joins KSI crashes to all of HIN
var City_HIN = CityHIN_KSICrashes_SpatialJoin.map(KSICrashCount_Finder); //Makes column showing each HIN street's raw KSI crash count
var City_HIN = City_HIN.map(KSICrashRate_Finder);
```



All Columns Creation Result

KSI CRASHES

- Then, I found the average KSI crash rate for the entire city HIN, regardless of congestion levels. The point of this is to eventually select out congested HIN roads that have KSI crash rates higher than the average one for the entire HIN, thereby selecting out congested roads that are more dangerous than the rest of the HIN:

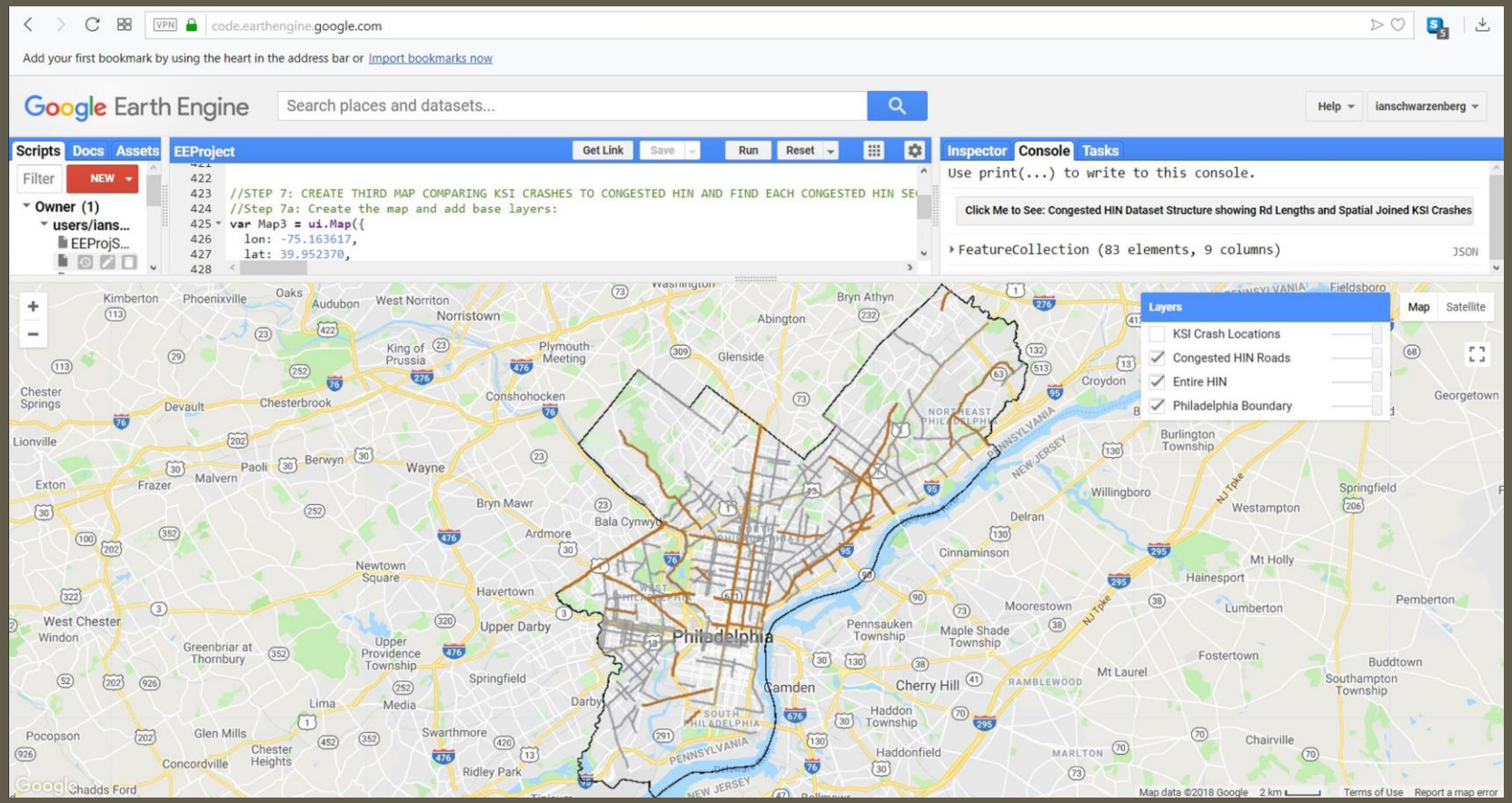
```
var mean = ee.Reducer.mean();
var CityHIN_AvgKSICrashRate = ee.Number(City_HIN.reduceColumns(mean, ['KSICrshRte'])); //Turns CityHIN_AvgKSICrashRate from a dictionary to a number using ee.Number() to "cast" the result as a
//number
//print(CityHIN_AvgKSICrashRate); //Average KSI crash rate for entire HIN is 3.900499074286767 KSI crashes per mile according to the output of this line
```

- I then applied my three functions and spatial join to my congested HIN dataset so each of those roads has a KSI crash rate. I also created a button to let users see the structure of the congested HIN dataset with all of those columns :

```
var HIN_Congested = HIN_Congested.map(Length_Finder); //CREATES COLUMN IN HIN_Congested SHOWING EACH ROAD'S LENGTH IN MILES
var HINCongested_KSICrashes_SpatialJoin = distSaveAll.apply(HIN_Congested, KSI_Crashes, distFilter); // Apply the spatial join for congested HIN segments
var HIN_Congested = HINCongested_KSICrashes_SpatialJoin.map(KSICrashCount_Finder); //CREATES COLUMN IN HIN_Congested SHOWING EACH ROAD'S NUMBER OF KSI CRASHES HAPPENING ON IT
var HIN_Congested = HIN_Congested.map(KSICrashRate_Finder); //CREATES COLUMN IN HIN_Congested SHOWING EACH ROAD'S KSI CRASH RATE PER MILE

var Map3Button1 = ui.Button({
  label: 'Click Me to See: Congested HIN Dataset Structure showing Rd Lengths and Spatial Joined KSI Crashes',
  onClick: function() {
    print(HIN_Congested);
  }
});
```

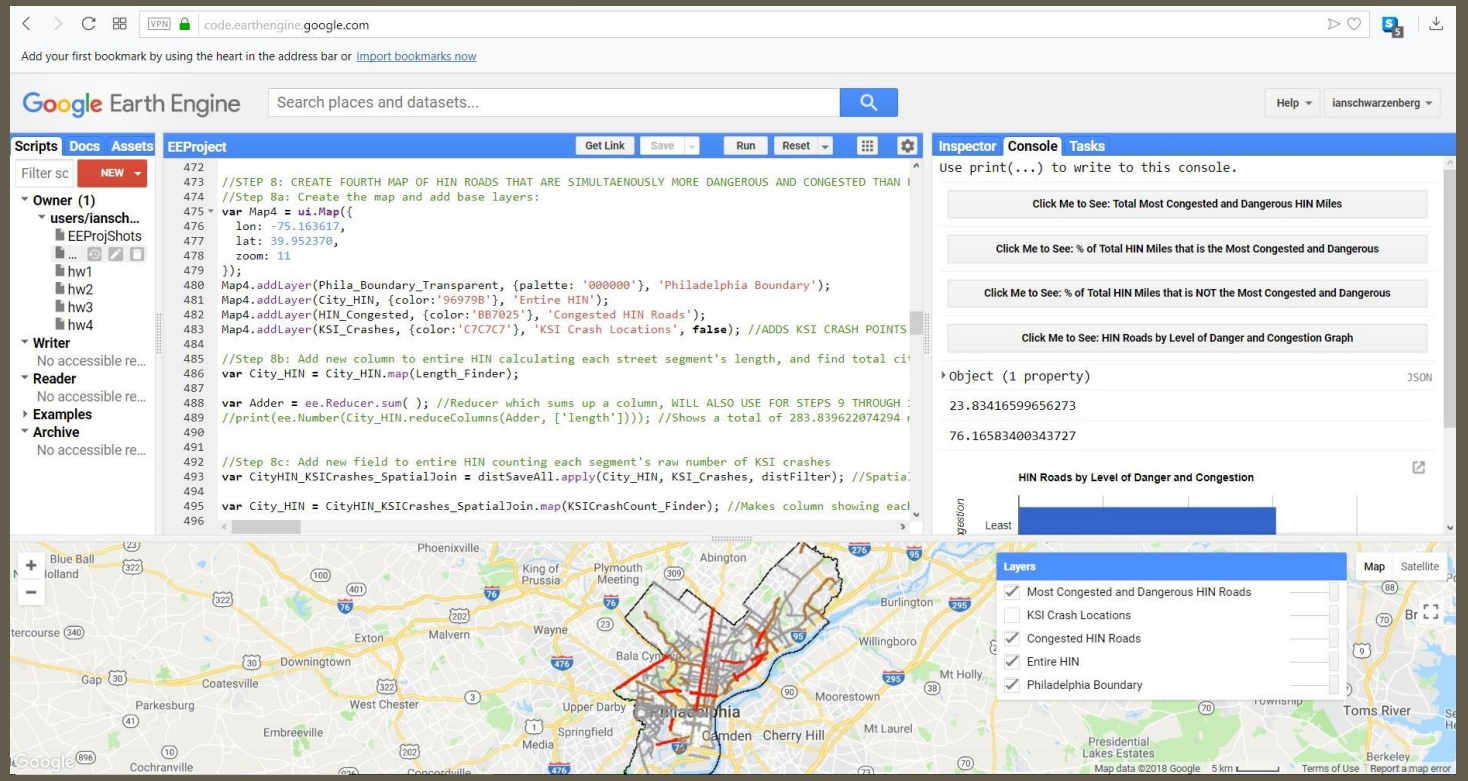
FINAL MAP

- Once that button was created, the final map was made. The purpose of this map is simply to show the user the end result of finding: which HIN roads are most congested, the process of finding each HIN road's length in miles, the process of finding each road's KSI crash count and crash rate. Displaying the roads that are simultaneously the most congested and dangerous compared to the rest of the network is reserved for the next map. There is only a button to let users see the congested roads' structure because learning safety figures on the congested HIN roads is ultimately what is most important, as opposed to on the entire network.



Map 3 Final

Map 4/Choice d



Map 4 Overview

STATISTICS

- At this point, I have the congested HIN roads with their KSI crash information. My goal is to find the most dangerous *and* congested roads.
- After establishing the map, I used feature.filterMetadata() to select out the congested HIN segments with KSI crash rates over the average city HIN KSI crash rate to get the HIN roads that are simultaneously the most congested and dangerous in the entire network:

```
var HIN_CongestedandDangerous_Roads = HIN_Congested.filterMetadata( 'KSICrshRte', 'greater_than', 3.900499074286767 );
Map4.addLayer(HIN_CongestedandDangerous_Roads, {color:'FE1004'}, 'Most Congested and Dangerous HIN Roads'); // MAPS FINAL CONSISTENTLY CONGESTED AND DANGEROUS ROADS
```

- I then wanted to find various statistics about the congested HIN roads: 1) the total mileage of HIN that is the most congested and dangerous, 2) the percentage of total HIN road miles that was marked as the most congested and dangerous, 3) the percentage of total HIN road miles that was *not* marked as the most congested and dangerous in comparison, and 4) a bar chart showing the HIN roads by level of danger and congestion to let the user visually compare the percentages in 2) and 3). To find these statistics, I had to first find the total number of miles of HIN road, regardless of congestion. To do this, I created a reducer which I called 'Adder' which finds the sums of columns using ee.Reducer.sum():

```
var Adder = ee.Reducer.sum( ); //Reducer which sums up a column, WILL ALSO USE FOR STEPS 9 THROUGH 12
//print(ee.Number(City_HIN.reduceColumns(Adder, ['length']))); //Shows a total of 283.839622074294 miles of HIN road
```

- I then created three buttons which showed the statistics described in 1), 2) and 3). I programmed each button's onClick function so that the user would be executing the calculations to find those statistics themselves by clicking the buttons, thereby enhancing the user experience of the application. The examples below show the buttons that calculate 2) and 3):

```
var Map4Button2 = ui.Button({
  label: 'Click Me to See: % of Total HIN Miles that is the Most Congested and Dangerous',
  onClick: function() {
    print(ee.Number((67.65080668920355/283.839622074294)*100)); //Equates to 23.83416599656273%
  }
});

var Map4Button3 = ui.Button({
  label: 'Click Me to See: % of Total HIN Miles that is NOT the Most Congested and Dangerous',
  onClick: function() {
    print(ee.Number(100-23.83416599656273)); //Subtracts 100% from the percent found in Map4Button2 to get the calculation for Map4Button3
  }});
```

GRAPH

- I then wanted create the graph representing 4). First, I created a variable that would contain the chart data. This particular graph contained two columns created by a list in the variable called 'cols': One column of string observations detailing each road's danger and congestion level (least or most, as determined by the row list), and a numeric column detailing each type of HIN road's percentage of the entire HIN (around 76% and 24% for least ad most, as determined by the row list). The rows list assigns the observations for each column name in the column list:

```
var HIN_CongestedandDangerous_Roads_Chart_Data = { //Creates variable containing a table of the data to be charted
  cols: [{id: 'Level',  label: 'Danger and Congestion Level', type: 'string'}, //Creates a string column detailing HIN road types
         {id: 'Pct', label: '% of HIN Road Mileage', type: 'number'}], //Creates a numerical column detailing each HIN road type's percentage of all HIN road miles
  rows: [{c: [{v: 'Least'}, {v: 76.16583400343727}]}, //Assigns value of 76.16583400343727% for the Other HIN road type for the % of HIN Road Mileage column
         {c: [{v: 'Most'}, {v: 23.83416599656273}]}], //Assigns value of 23.83416599656273% for the Most Congested and Dangerous HIN road type for the % of HIN Road Mileage column
};  //Creates dataset to chart
```

- In short, the code above does Earth Engine's equivalent of creating this table to be made into a bar chart shortly for this example:

Danger and Congestion Level	% of HIN Road Mileage
Least	76.16583400343727
Most	23.83416599656273

- I then created a variable called options which detailed: the graph's title, y-axis ("vAxis", or vertical axis") title, x-axis ("hAxis", or horizontal axis") title, and how the graph will have no legend by giving a position of 'none':

```
var options = {
  title: 'HIN Roads by Level of Danger and Congestion', //Creates graph title
  vAxis: {title: 'Danger and Congestion Level'}, //Creates y-axis title (vAxis means "vertical axis" which is the y-axis)
  legend: {position: 'none'}, //Makes it so that there is no legend
  hAxis: {title: '% of HIN Road Mileage'} //Creates x-axis title (hAxis means "horizontal axis" which is the x-axis)
};
```

GRAPH

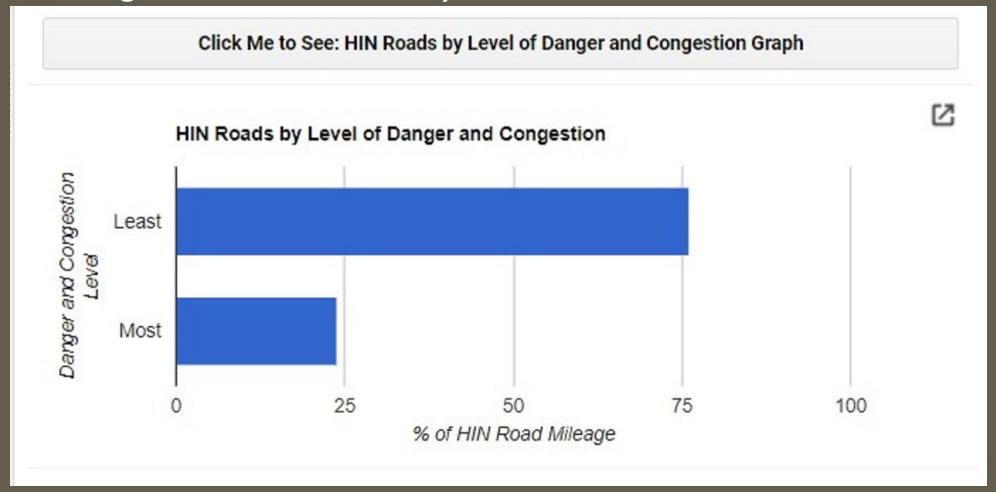
- I then finally made the graph using the new.uiChart() command. This command sets the data to be graphed as the chart data variable I defined before, and makes it plot as a 'BarChart' with the graph title, and axes titles I defined in my "options" variable before:

```
var HIN_CongestedandDangerous_Roads_Chart = new ui.Chart(HIN_CongestedandDangerous_Roads_Chart_Data, 'BarChart', options); //Creates variable with the chart in it and makes it a bar chart
```

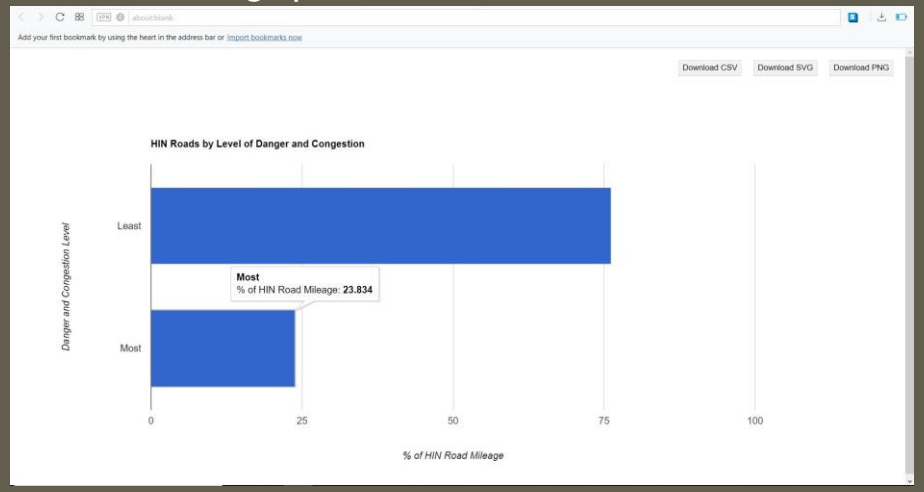
- I then created a button which would let the user view this graph when they click on it using its onClick function that prints the graph. I followed this entire statistic button and graph creation button template for the rest of the project:

```
var Map4Button4 = ui.Button({
  label: 'Click Me to See: HIN Roads by Level of Danger and Congestion Graph',
  onClick: function() {
    print(HIN_CongestedandDangerous_Roads_Chart);
  }
}); //This button instead will show the graph created above
```

- When the user clicks on the graph button, they are able to click on a square with an arrow button on the type right of the graph to let them view the graph in a bigger setting. Once blown up, the user can download the graph, showing how Earth Engine makes the data open. Earth Engine also automatically makes it so the user can move their cursor above the graph's bars to view more detailed information.



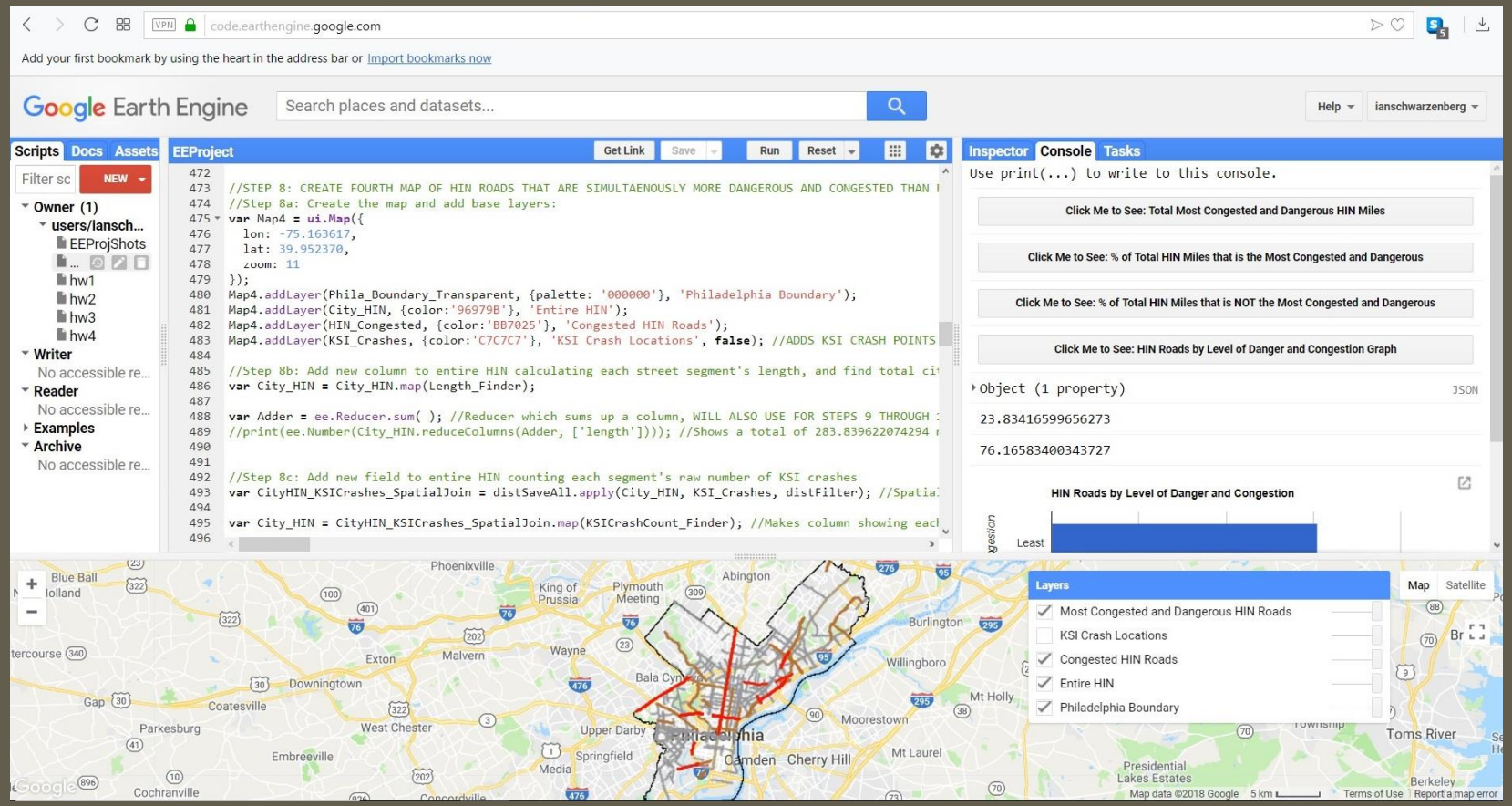
Graph in Application Console Tab with Expansion Button on Top Right



Expanded Graph in Separate Tab

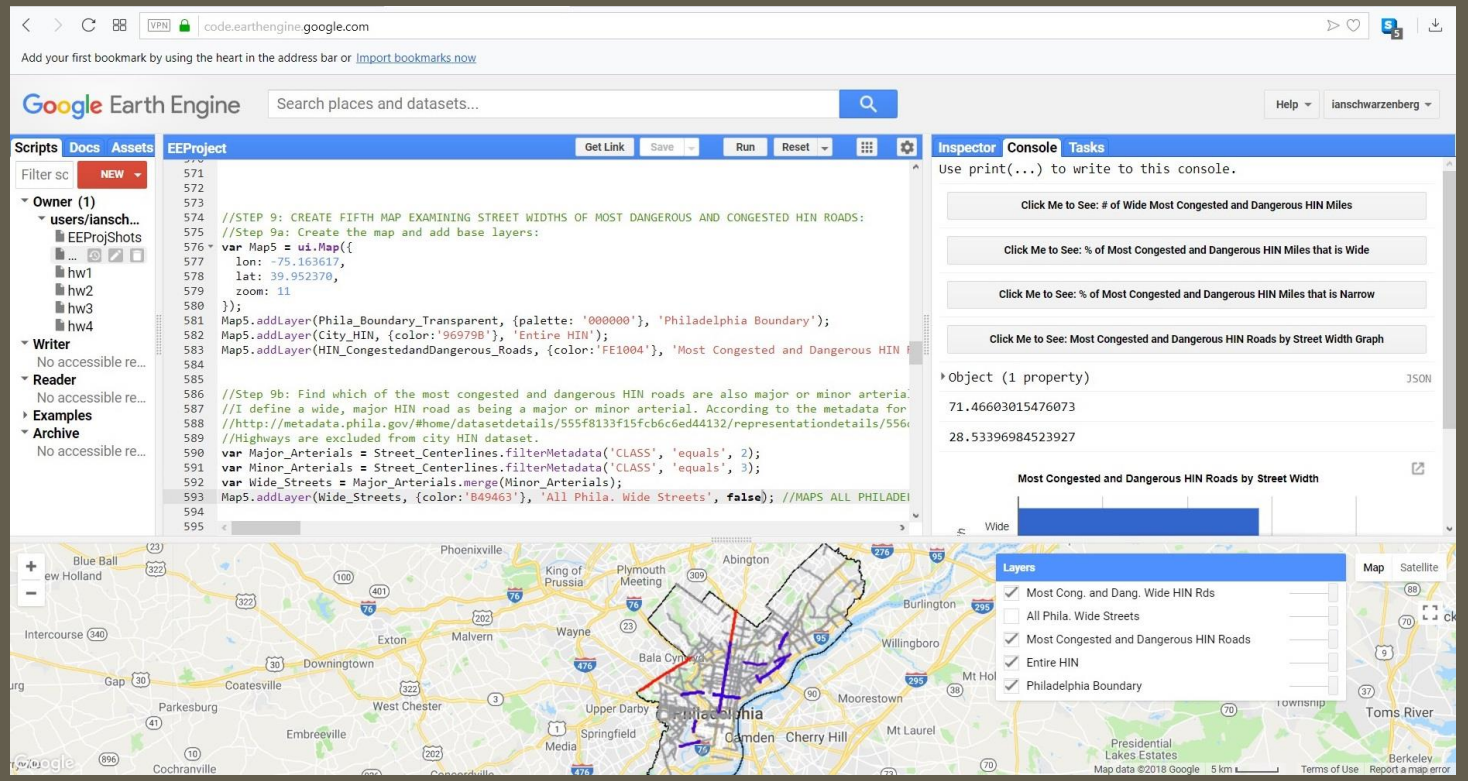
FINAL MAP

- Once those four statistics and graph buttons were created, the final map was made. One purpose of this map is to show the user which HIN roads are simultaneously the most congested and dangerous out of all other HIN roads, making them the most problematic. Another purpose of this viewing option is to show the user basic statistics about those roads in an interesting manner that lets the user calculate the statistics and make the graphs themselves, to further engage them in the app and prepare for comparing these roads to conditions like street width, neighborhood wealth, proximity to train stops and proximity to commercial corridors shortly.



Map 4 Final

Map 5/Choice e



Map 5 Overview

WIDE STREETS

- At this point in the analysis process, I now have: 1) a variable of the most congested HIN roads, 2) a variable of the most congested *and* dangerous HIN roads, and 3) basic statistics and feature collections detailing 1) and 2) displayed through buttons. The goal of this fifth map is to start examining characteristics of the most congested *and* dangerous HIN streets to see why they might be this way, specifically checking their *street widths*. If most of the streets end up being defined as "wide", then this could reveal potential reasons as to why these roads are especially dangerous and congested. For example, if a road is particularly wide, it could have more cars that travel down it, where they can collide with each other. Wider streets can also mean pedestrians would have harder times crossing, increasing their chances of being hit by those vehicles.
- The first step after establishing the map with ui.Map({}) was to load in the Philadelphia city street centerlines, regardless of their position on the HIN, and select out all Philadelphia city streets, *regardless* of whether those were HIN or regular streets, that are major or minor arterials using feature.filterMetadata(). After putting major and minor arterials in their own variables, I then merged them to get my final wide streets variable:

```
var Street_Centerlines = ee.FeatureCollection('ft:1I-o1O1Lf89XoSA0TFUGi_SB71PrNg1IOaSS0MQFM') // Loads in Philadelphia street centerlines which contain street width information (source:
//Philadelphia Streets Department)
var Major_Arterials = Street_Centerlines.filterMetadata('CLASS', 'equals', 2);
var Minor_Arterials = Street_Centerlines.filterMetadata('CLASS', 'equals', 3);
var Wide_Streets = Major_Arterials.merge(Minor_Arterials);
Map5.addLayer(Wide_Streets, {color:'B49463'}, 'All Phila. Wide Streets', false); //MAPS ALL PHILADELPHIA WIDE STREETS REGARDLESS OF THEIR POSITION ON THE HIN
```

- Using my special intersection method (intersection_Join.apply(primary feature name, secondary feature name, intersection_Filter)), I then intersected all Philadelphia 'wide' streets with the most congested and dangerous HIN roads to see which of those were also considered 'wide'. Doing this essentially selects out which most congested and dangerous HIN road segments are also wide:

```
var HIN_CongestedandDangerous_WideRoads = intersection_Join.apply(HIN_CongestedandDangerous_Roads, Wide_Streets, intersection_Filter);
Map5.addLayer(HIN_CongestedandDangerous_WideRoads, {color:'2703FF'}, 'Most Cong. and Dang. Wide HIN Rds'); //MAPS THE MOST DANGEROUS CONGESTED HIN ROADS WHICH ARE ALSO WIDE
```

STATISTICS

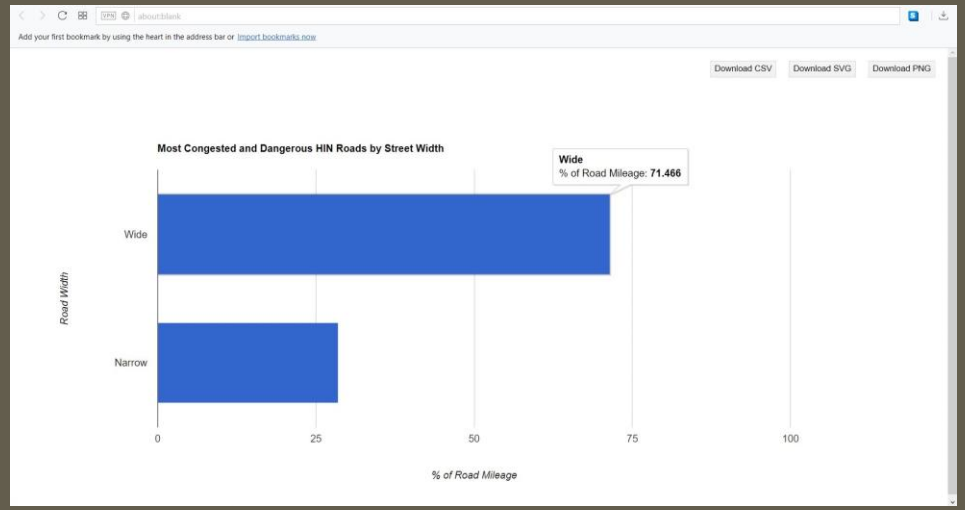
- I then created three buttons detailing statistics about the most congested and dangerous roads' street widths: 1) the *number* of miles of the most congested and dangerous road that is also wide, 2) the *percent* of most congested and dangerous road that is also wide, and 3) the *percent* of most congested and dangerous road that is *narrow* (narrower than major or minor arterial) in order to enable the user to get a more visual sense of the proportion of wide most congested and dangerous HIN streets:

```
var Map5Button1 = ui.Button({
  label: 'Click Me to See: # of Wide Most Congested and Dangerous HIN Miles',
  onClick: function() {
    print(ee.Number(HIN_CongestedandDangerous_WideRoads.reduceColumns(Adder, ['length'])));
  }
});

var Map5Button2 = ui.Button({
  label: 'Click Me to See: % of Most Congested and Dangerous HIN Miles that is Wide',
  onClick: function() {
    print(ee.Number((48.3473459084451/67.65080668920355)*100));
  }
});

//print('Narrow Roads: % of Most. Cong. Dang. HIN Mi.', ee.Number(100-71.46603015476073));  //Shows 28.5339698452% of the most congested and dangerous HIN road mileage is NOT wide. Total mileage of HIN road
//that is most congested and dangerous taken from step 8f

var Map5Button3 = ui.Button({
  label: 'Click Me to See: % of Most Congested and Dangerous HIN Miles that is Narrow',
  onClick: function() {
    print(ee.Number(100-71.46603015476073));
  }
});
```

GRAPH

- I then finally made the graph and corresponding button to further enable the user to see the proportion of most congested and dangerous streets that are wide, using the exact same button and graph creation method templates from previous maps:

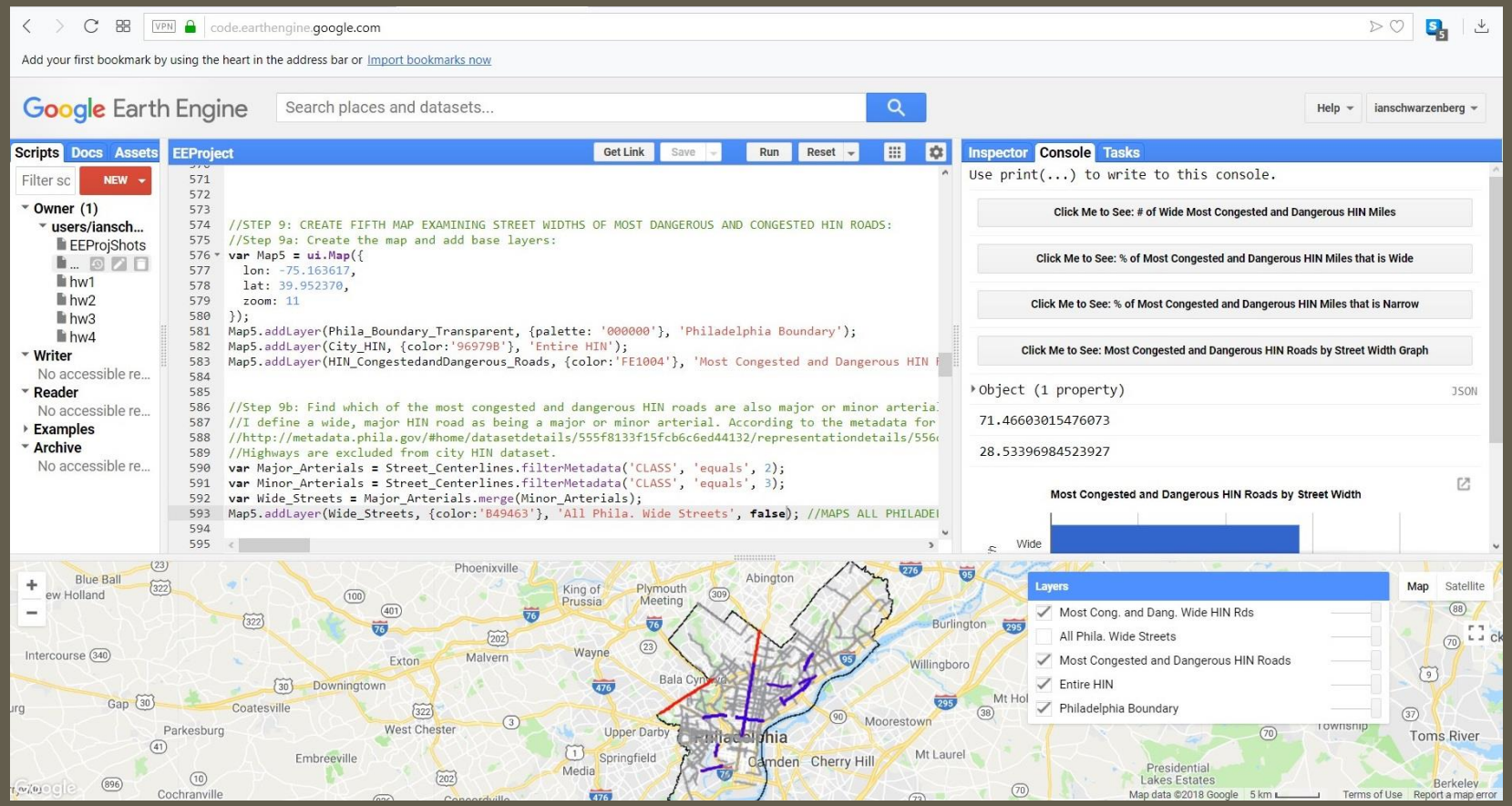
```
var HIN_CongestedandDangerous_WideRoads_Chart_Data = {
  cols: [{id: 'Width',  label: 'Road Width', type: 'string'},
         {id: 'Pct', label: '% of Road Mileage', type: 'number'}],
  rows: [{c: [{v: 'Wide'}, {v: 71.46603015476073}]},
         {c: [{v: 'Narrow'}, {v: 28.5339698452}]}],
};

var options = {
  title: 'Most Congested and Dangerous HIN Roads by Street Width',
  vAxis: {title: 'Road Width'},
  legend: {position: 'none'},
  hAxis: {title: '% of Road Mileage', ticks: [{v: 0}, {v: 25}, {v: 50}, {v: 75}, {v: 100}]} //X axis scale only went up to 80, so this makes it go up to 100. Found out how to do from https://developers.google.com/earth-
engine/charts_feature_by
};

var HIN_CongestedandDangerous_WideRoads_Chart = new ui.Chart(HIN_CongestedandDangerous_WideRoads_Chart_Data, 'BarChart', options);
//print(HIN_CongestedandDangerous_WideRoads_Chart);

var Map5Button4 = ui.Button({
  label: 'Click Me to See: Most Congested and Dangerous HIN Roads by Street Width Graph',
  onClick: function() {
    print(HIN_CongestedandDangerous_WideRoads_Chart);
  }
});
```



Expanded Street Width Graph in Separate Tab

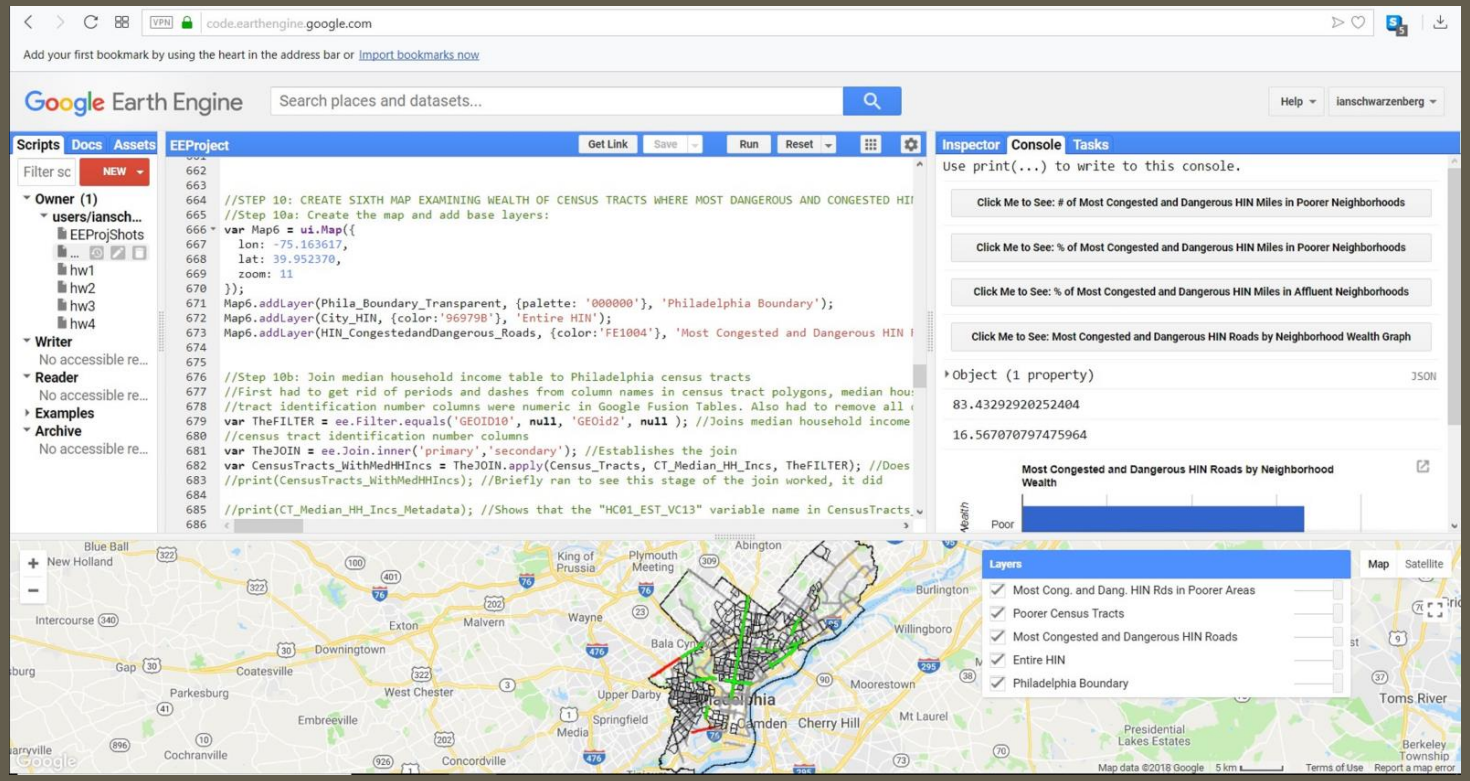
FINAL MAP

- Once those four statistics and graph buttons were created, the final map was made. The purpose of this map is to show the user which of the most congested and dangerous HIN roads are also considered wide. The statistics and graph show that a clear majority of the most congested and dangerous HIN roads are also wide. This is not surprising considering how wide roads have room to hold more cars, which increases congestion. It is also not surprising considering how more cars being near can increase collisions amongst themselves. It is also not surprising since pedestrians would have harder times crossing wider streets in time, increasing their chances of being hit by passing vehicles.



Map 5 Final

Map 6/Choice f



Map 6 Overview

SETUP

- At this point in the analysis process, I now have: 1) a variable of the most congested HIN roads, 2) a variable of the most congested *and* dangerous HIN roads, and 3) basic statistics and feature collections detailing 1) and 2) as well as those streets' widths displayed through buttons. The goal of this sixth map is to additionally examine characteristics of the most congested *and* dangerous HIN streets to see why they might be this way, specifically examining the *wealth of the neighborhoods they pass through*. If most of the neighborhoods they pass through end up being defined as "poor", then this could reveal potential reasons as to why these roads are especially dangerous and congested. For example, poor neighborhoods tend to have lesser quality infrastructure than wealthier neighborhoods in American cities, which could lead to more congested and dangerous roads.
- The first step after establishing the map with ui.Map({}) was to load in the Philadelphia census tracts shapefile, an American Community Survey fusion table containing each census tract's median household income, and a metadata fusion table containing the column name descriptions of the fusion table with each census tract's median household income for reference:

```
var Census_Tracts =  ee.FeatureCollection('ft:1gZonlFMK3uPYcuCoN2hWBwn-FuDnF5YoEMBOp3Uv'); // Loads in 2010 city census tract polygons shapefile (source: Philadelphia Department of Planning and
//Development)
var CT_Median_HH_Incs = ee.FeatureCollection('ft:1IvCiogCQVZIve8y99S9N13S-wzKyQARIZoCvAyqY'); // Loads in table of all city census tract's median household incomes (source: US Census Bureau
//American Community Survey 2012-16)
var CT_Median_HH_Incs_Metadata = ee.FeatureCollection('ft:12W3bePDjJLjY1kM49hHiBnvXJ7xtQuoNafpSWoQO'); // Loads in the metadata for the table of all city census tract's median household incomes
//(source: US Census Bureau American Community Survey 2012-16)
```

- The next step was to get rid of periods and dashes from column names in the census tract polygons, median household income and median household income metadata fusion tables. This was done since Earth Engine had trouble reading column names with these characters in them. I also made sure that all relevant fusion tables' census tract identification number columns were being read as numeric in Google Fusion Tables to ease the inner join process. There were also many *observations* in the median household income table that had just dashes, causing problems with Earth Engine reading those columns as numeric. As a result, I had to remove all those dashes and replace them with null values before initiating the inner join process.

INNER JOINING

- The next step was to start the inner join of the census tract median household incomes to the census tract polygons by tract name. The primary features of the inner join were the census tract polygons since these are what I want the median household income information to be added to, and the secondary features were the census tract median household incomes. The fields of each dataset representing tract names were used to link each median household income to its corresponding tract, representing the common identifier between the two datasets:

```
var TheFILTER = ee.Filter.equals('GEOID10', null, 'GEOid2', null ); //Joins median household income data (secondary features) to census tract polygons (primary features) based on their common
//census tract identification number columns
var TheJOIN = ee.Join.inner('primary','secondary'); //Establishes the join
var CensusTracts_WithMedHHIncs = TheJOIN.apply(Census_Tracts, CT_Median_HH_Incs, TheFILTER); //Does the join
```

- The next step was to map that join over the census tracts polygon feature collection. The function that does this starts with creating a new variable to place the columns I want to extract in. It then makes variable versions of temporary copies of the primary features' (the census tract polygons) and the secondary features' (the median household incomes) columns, called 'PRIMARY' and 'SECONDARY' in the function. This is done using ee.Feature.element.get(), since the primary and secondary parts of the join are *elements* as opposed to *features*. The next four lines then extract the columns I want from the primary and secondary features using feature.get(). I extract each census tract's ID number, census tract name and geometry from the primary census tract polygon features, and the median household incomes from the secondary median household income fusion table features. The metadata fusion table helped with identifying the HC01_EST_VC13 column in the secondary features, which represents each tract's median household income. The function then returns the new variable with just these columns using return ee.Feature({}):

```
var CensusTracts_WithMedHHIncs_New = CensusTracts_WithMedHHIncs.map(function(element){
    var PRIMARY       = ee.Feature(element.get('primary'));
    var SECONDARY     = ee.Feature(element.get('secondary'));
    var IDNUM         = PRIMARY.get('GEOID10');
    var CTNAME        = PRIMARY.get('NAMELSAD10');
    var MEDHHINC      = SECONDARY.get('HC01_EST_VC13');
    var geom          = ee.Feature(element.get('primary')).geometry();
    return ee.Feature(geom, {'IDNUM':IDNUM, 'CTNAME':CTNAME, 'MEDHHINC':MEDHHINC});
}); //Final stage of the join which creates the final dataset with the Philadelphia census tracts and their median household incomes
```

POOR TRACTS

- The next step was to find which Philadelphia census tracts were "poor" and which were "affluent". I defined poorer tracts as tracts with median household incomes below the median Philadelphia census tract median household income. Firstly, I found that by using ee.Reducer.median() to find the median of the column in the final join result representing all Philadelphia census tract median household incomes:

```
var median = ee.Reducer.median( ); //Median reducer, will apply to the MEDHHINC values for all Philadelphia census tracts to find median of them
//print(ee.Number(CensusTracts_WithMedHHIncs_New.reduceColumns(median, ['MEDHHINC']))); //Shows median Philadelphia census tract median household income to be $39370.1538461538 44
```

- I then used feature.filterMetadata() to select out Philadelphia census tracts that had *below* the median Philadelphia census tract median household income of $39,370.1538461538 44 in order to extract poor census tracts. Then using the exact same method for creating a transparent Philadelphia boundary for all maps, I made the poor census tracts transparent and added that to this map:

```
var PoorTracts = CensusTracts_WithMedHHIncs_New.filterMetadata('MEDHHINC', 'less_than', 39370.1538461538 44); //Selects out tracts defined as being poorer than rest of Philadelphia
var PoorTracts_Transparent = empty.paint({
  featureCollection: PoorTracts,
  color: 1,
  width: 0.75
}); //Creates transparent version of the poor tracts feature collection, the variable "empty" is taken from step 1d

Map6.addLayer(PoorTracts_Transparent, {palette: '000000'}, 'Poorer Census Tracts'); //MAPS POORER CENSUS TRACTS
```

- Then using the same exact special intersection method I used for other maps, I intersected the most congested and dangerous HIN roads with the poor census tracts to create a variable containing the most congested and dangerous HIN roads that travel through poor census tracts:

```
var HIN_CongestedandDangerous_PoorAreas = intersection_Join.apply(HIN_CongestedandDangerous_Roads, PoorTracts, intersection_Filter); //Intersects most congested and dangerous HIN roads with
//poor tracts to find which most congested and dangerous HIN roads are also in poor areas

Map6.addLayer(HIN_CongestedandDangerous_PoorAreas, {color:'03FF11'}, 'Most Cong. and Dang. HIN Rds in Poorer Areas'); //MAPS THE MOST DANGEROUS CONGESTED HIN ROADS WHICH ARE ALSO IN
POOR AREAS
```

STATISTICS

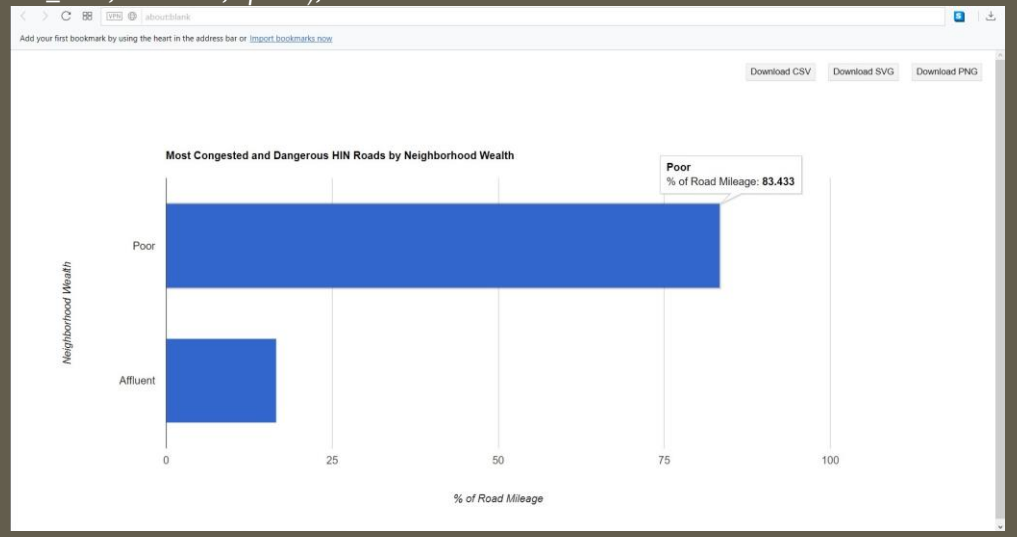
- I then created three buttons detailing statistics about the most congested and dangerous roads' relationship to poorer census tracts: 1) the *number* of miles of the most congested and dangerous road that pass through poorer census tracts, 2) the *percent* of most congested and dangerous road that that passes through poorer census tracts, and 3) the *percent* of most congested and dangerous road that passes through *more affluent* census tracts in order to enable the user to get a more visual sense of the proportion of most congested and dangerous HIN streets that travel through poorer census tracts:

```
var Map6Button1 = ui.Button({
  label: 'Click Me to See: # of Most Congested and Dangerous HIN Miles in Poorer Neighborhoods',
  onClick: function() {
    print(ee.Number(HIN_CongestedandDangerous_PoorAreas.reduceColumns(Adder, ['length'])));
  }
});

var Map6Button2 = ui.Button({
  label: 'Click Me to See: % of Most Congested and Dangerous HIN Miles in Poorer Neighborhoods',
  onClick: function() {
    print(ee.Number((56.443049649939596/67.65080668920355)*100));
  }
});

var Map6Button3 = ui.Button({
  label: 'Click Me to See: % of Most Congested and Dangerous HIN Miles in Affluent Neighborhoods',
  onClick: function() {
    print(ee.Number(100-83.43292920252404));
  }
});
```

GRAPH

- I then finally made the graph and corresponding button to further enable the user to see the proportion of most congested and dangerous streets that travel through poorer census tracts, using the exact same button and graph creation method templates from previous maps:

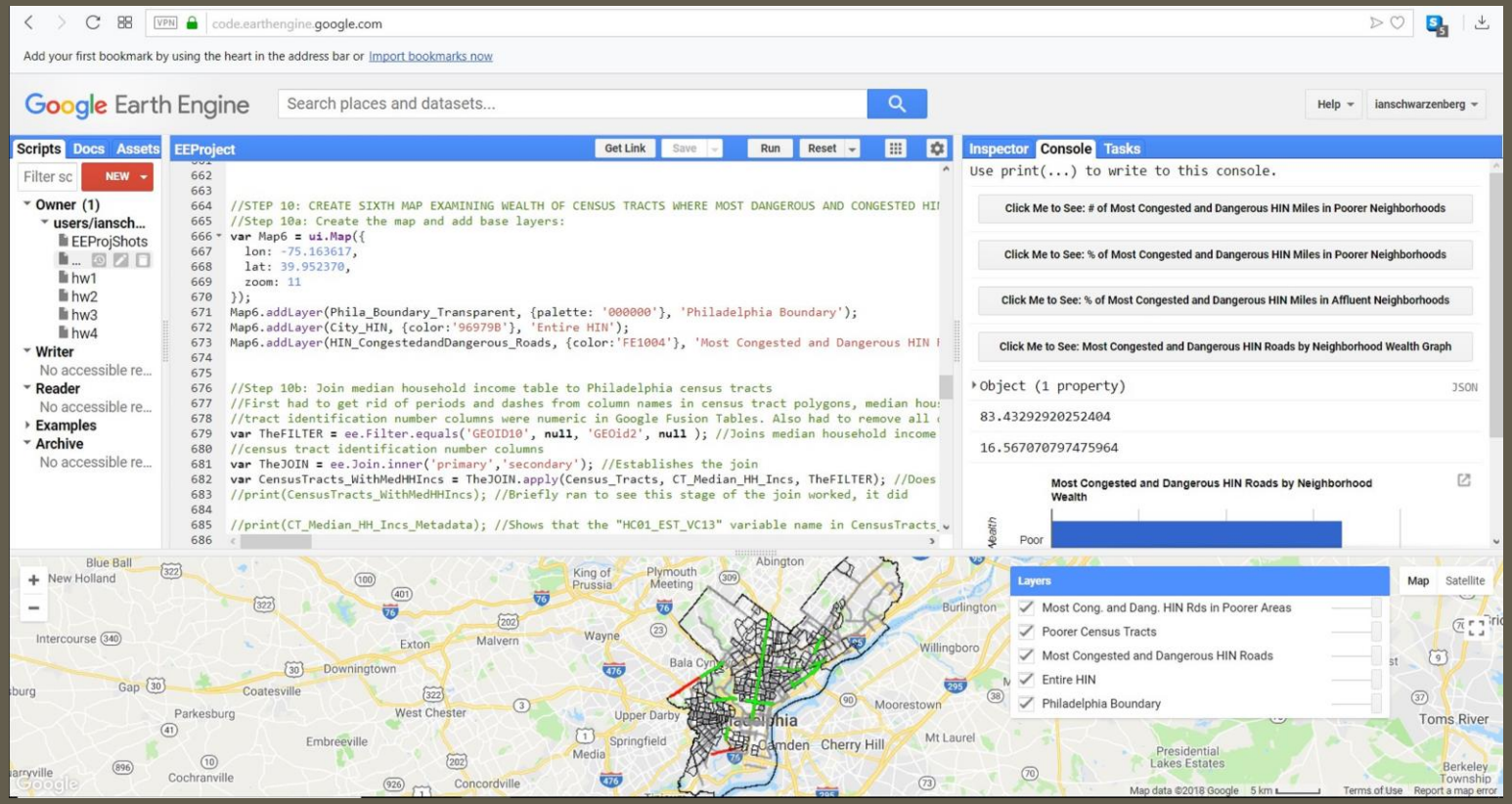
```
var HIN_CongestedandDangerous_PoorAreas_Chart_Data = {
  cols: [{id: 'Wealth',  label: 'Neighborhood Wealth', type: 'string'},
         {id: 'Pct', label: '% of Road Mileage', type: 'number'}],
  rows: [{c: [{v: 'Poor'}, {v: 83.43292920252404}]},
         {c: [{v: 'Affluent'}, {v: 16.5670707975}]}],
};

var options = {
  title: 'Most Congested and Dangerous HIN Roads by Neighborhood Wealth',
  vAxis: {title: 'Neighborhood Wealth'},
  legend: {position: 'none'},
  hAxis: {title: '% of Road Mileage'}
};

var HIN_CongestedandDangerous_PoorAreas_Chart = new ui.Chart(HIN_CongestedandDangerous_PoorAreas_Chart_Data, 'BarChart', options);
//print(HIN_CongestedandDangerous_PoorAreas_Chart);

var Map6Button4 = ui.Button({
  label: 'Click Me to See: Most Congested and Dangerous HIN Roads by Neighborhood Wealth Graph',
  onClick: function() {
    print(HIN_CongestedandDangerous_PoorAreas_Chart);
  }
});
```



Expanded Neighborhood Poverty Graph in Separate Tab

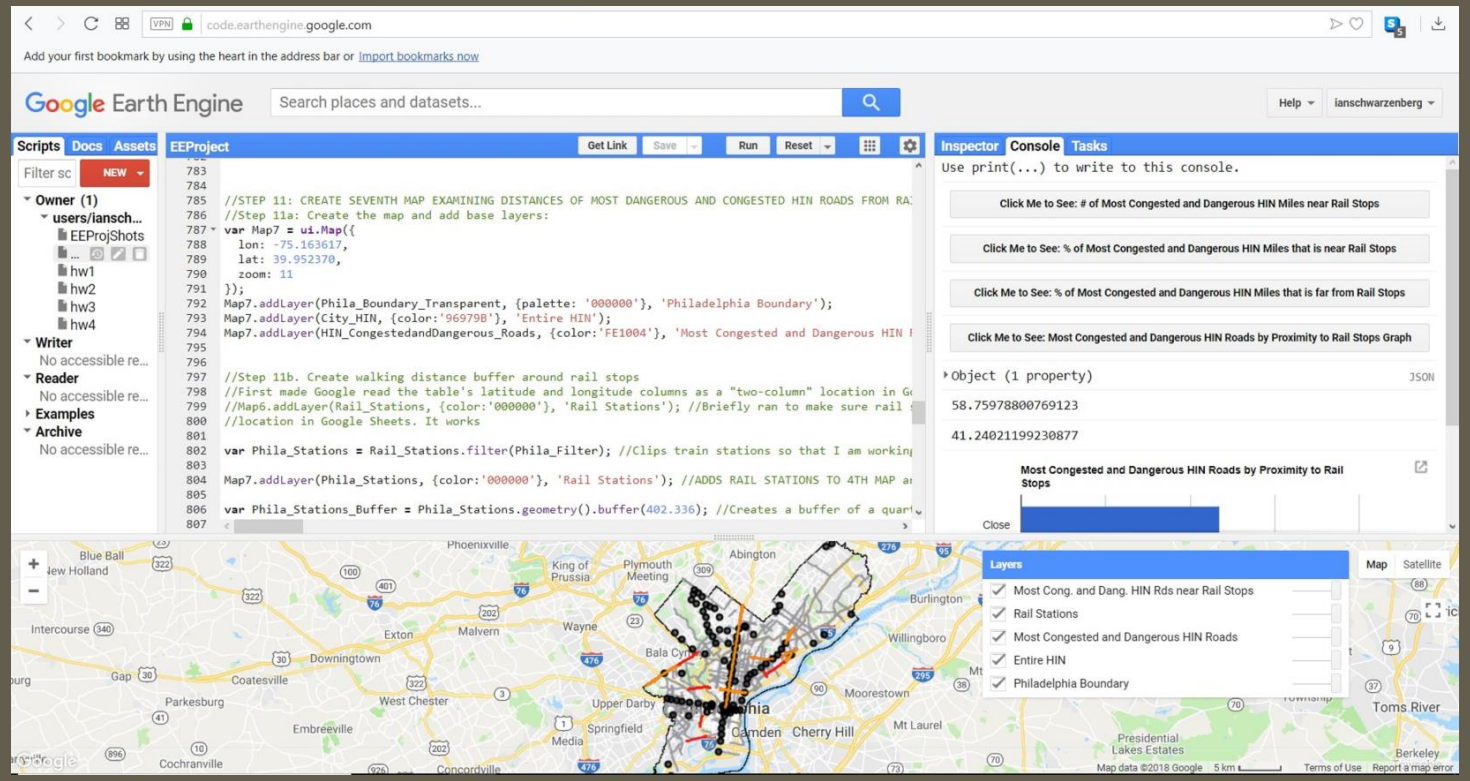
FINAL MAP

- Once those four statistics and graph buttons were created, the final map was made. The purpose of this map is to show the user which of the most congested and dangerous HIN roads travel through poorer areas. The statistics and graph show that a clear majority of the most congested and dangerous HIN roads travel through poorer neighborhoods. This is not surprising considering how poor neighborhoods tend to have lesser quality infrastructure than wealthier neighborhoods in US cities, which could lead to more congested and dangerous roads since the road infrastructure in these areas may not be able to hold more vehicles as safely as wealthier areas. Roads in these areas also may not have had as many pedestrian safety upgrades compared to roads in more affluent areas, making roads in poorer areas more dangerous for pedestrians.



Map 6 Final

Map 7/Choice g



Map 7 Overview

CLIPPING AND BUFFERING

- At this point in the analysis process, I now have: 1) a variable of the most congested HIN roads, 2) a variable of the most congested *and* dangerous HIN roads, and 3) basic statistics and feature collections detailing 1) and 2), those streets' widths, and the wealth of the neighborhoods they pass through displayed through buttons. The goal of this seventh map is to additionally examine characteristics of the most congested *and* dangerous HIN streets to see why they might be this way, specifically examining their proximity to *rail stops*. If most of the most congested and dangerous roads end up being "far" from rail stops, then this could reveal potential reasons as to why these roads are especially dangerous and congested. For example, a road being far from a rail stop could encourage more people living around them to drive, causing congestion. More cars on the roads could also potentially lead to more crashes.
- The first step after establishing the map with ui.Map({}) was to load in the Philadelphia rail stop points shapefile:

```
var Rail_Stations = ee.FeatureCollection('ft:1wWXf-LjX_CY9UQhXpM99zMRt0e4f6ItX1S9yG-Kg'); // Loads in all Philadelphia area rail stations (Sources: Southeastern Pennsylvania Transportation
//Authority and Delaware Valley Regional Planning Commission)
```

- The next step was to clip the rail stations to the Philadelphia boundary in order to only deal with rail stops within the city limits:

```
var Phila_Stations = Rail_Stations.filter(Phila_Filter); //Clips train stations so that I am working with just ones in Philadelphia. The "Phila_Filter" used for the clip comes from step 2a
Map7.addLayer(Phila_Stations, {color:'000000'}, 'Rail Stations'); //ADDS RAIL STATIONS TO 4TH MAP and ran to make sure clip worked, it did
```

- The next step was to buffer the rail stops by 402.336 meters, equivalent to a quarter mile. I wanted to examine if the most congested and dangerous roads passed within walking distance of these rail stops, to see if people living near these roads live within walking distance of these stops. A quarter mile is considered walking distance in the US,[2] leading me to use this as a definition for walking distance:

```
var Phila_Stations_Buffer = Phila_Stations.geometry().buffer(402.336);
```

[2] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3377942/

INTERSECTING AND STATISTICS

- Using my special intersection method from previous maps, I then intersected the most congested and dangerous roads with the rail station buffers to create a variable containing the roads that pass through within walking distance of the rail stops:

```
var HIN_CongestedandDangerous_ByTransit = intersection_Join.apply(HIN_CongestedandDangerous_Roads, Phila_Stations_Buffer, intersection_Filter); //Intersects most congested and dangerous HIN roads
//with rail stop buffers to find which most congested and dangerous HIN roads are also within walking distance of rail stops.

Map7.addLayer(HIN_CongestedandDangerous_ByTransit, {color:'FF9403'}, 'Most Cong. and Dang. HIN Rds near Rail Stops'); //MAPS THE MOST DANGEROUS CONGESTED HIN ROADS WHICH ARE ALSO
NEAR RAIL STOPS
```

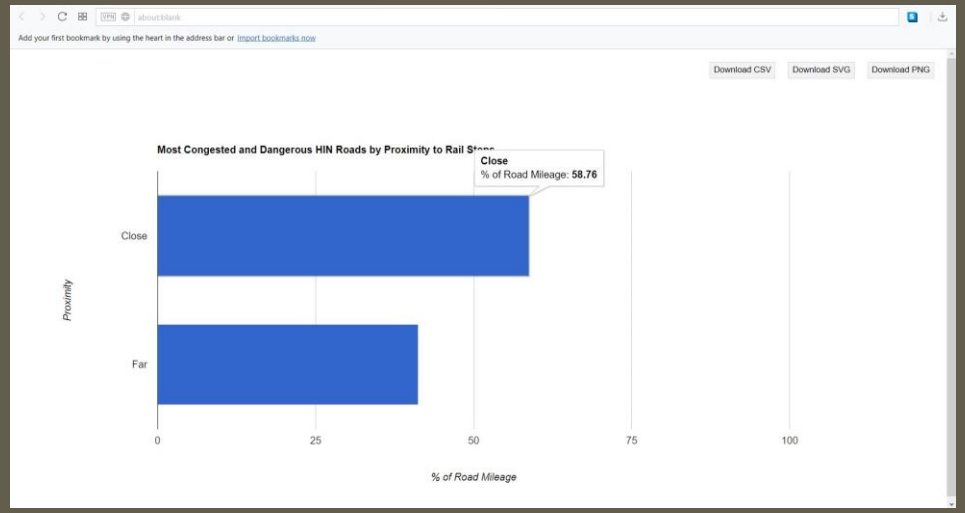
- I then created three buttons detailing statistics about the most congested and dangerous roads' relationship to rail stops: 1) the *number* of miles of the most congested and dangerous whole streets that pass by rail stops, 2 the *percent* of miles of the most congested and dangerous whole streets that pass by rail stops, and 3) the *percent* of most congested and dangerous whole streets that do *not* pass by rail stops in order to enable the user to get a more visual sense of the proportion of most congested and dangerous HIN streets that travel by rail stops:

```
var Map7Button1 = ui.Button({
  label: 'Click Me to See: # of Most Congested and Dangerous HIN Miles near Rail Stops',
  onClick: function() {
    print(ee.Number(HIN_CongestedandDangerous_ByTransit.reduceColumns(Adder, ['length'])));
  }
});

var Map7Button2 = ui.Button({
  label: 'Click Me to See: % of Most Congested and Dangerous HIN Miles that is near Rail Stops',
  onClick: function() {
    print(ee.Number((39.751470596069/67.65080668920355)*100));
  }
});

var Map7Button3 = ui.Button({
  label: 'Click Me to See: % of Most Congested and Dangerous HIN Miles that is far from Rail Stops',
  onClick: function() {
    print(ee.Number(100-58.759788007691 23));
  }});
```

GRAPH

- I then finally made the graph and corresponding button to further enable the user to see the proportion of most congested and dangerous streets that travel by rail stops, using the exact same button and graph creation method templates from previous maps:

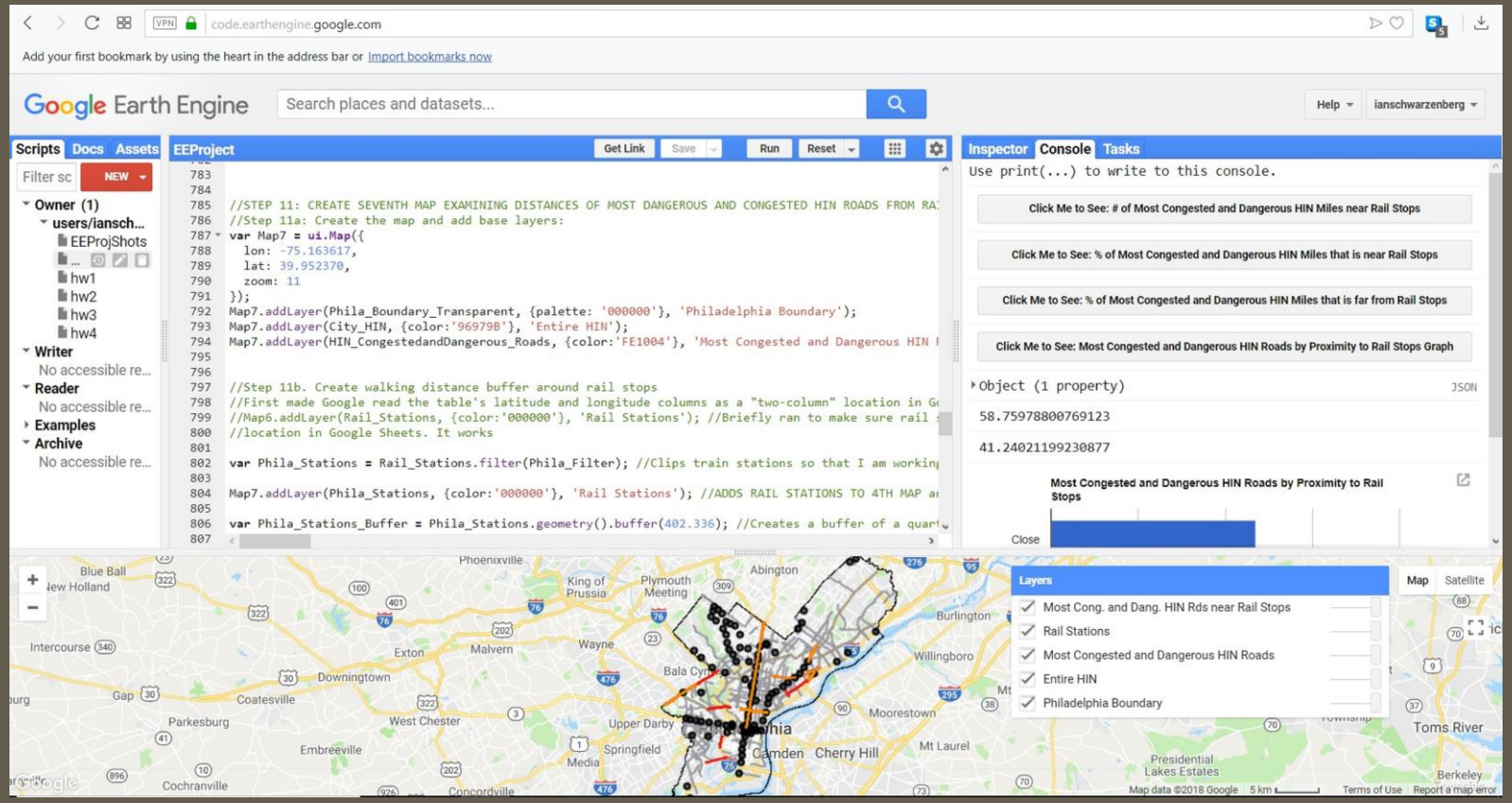
```
var HIN_CongestedandDangerous_ByTransit_Chart_Data = {
  cols: [{id: 'Proximity',  label: 'Proximity to Rail Stops', type: 'string'},
         {id: 'Pct', label: '% of Road Mileage', type: 'number'}],
  rows: [{c: [{v: 'Close'}, {v: 58.759788007691 23}]},
         {c: [{v: 'Far'}, {v: 41.240211 9923}]}],
};

var options = {
  title: 'Most Congested and Dangerous HIN Roads by Proximity to Rail Stops',
  vAxis: {title: 'Proximity'},
  legend: {position: 'none'},
  hAxis: {title: '% of Road Mileage', ticks: [{v: 0}, {v: 25}, {v: 50}, {v: 75}, {v: 100}]}, //X axis scale only went up to 60 originally which was bad since that distorted the difference between the proportions when in reality the difference is smaller
};

var HIN_CongestedandDangerous_ByTransit_Chart = new ui.Chart(HIN_CongestedandDangerous_ByTransit_Chart_Data, 'BarChart', options);
//print(HIN_CongestedandDangerous_ByTransit_Chart);

var Map7Button4 = ui.Button({
  label: 'Click Me to See: Most Congested and Dangerous HIN Roads by Proximity to Rail Stops Graph',
  onClick: function() {
    print(HIN_CongestedandDangerous_ByTransit_Chart);
  }
});
```



Expanded Rail Stop Proximity Graph in Separate Tab

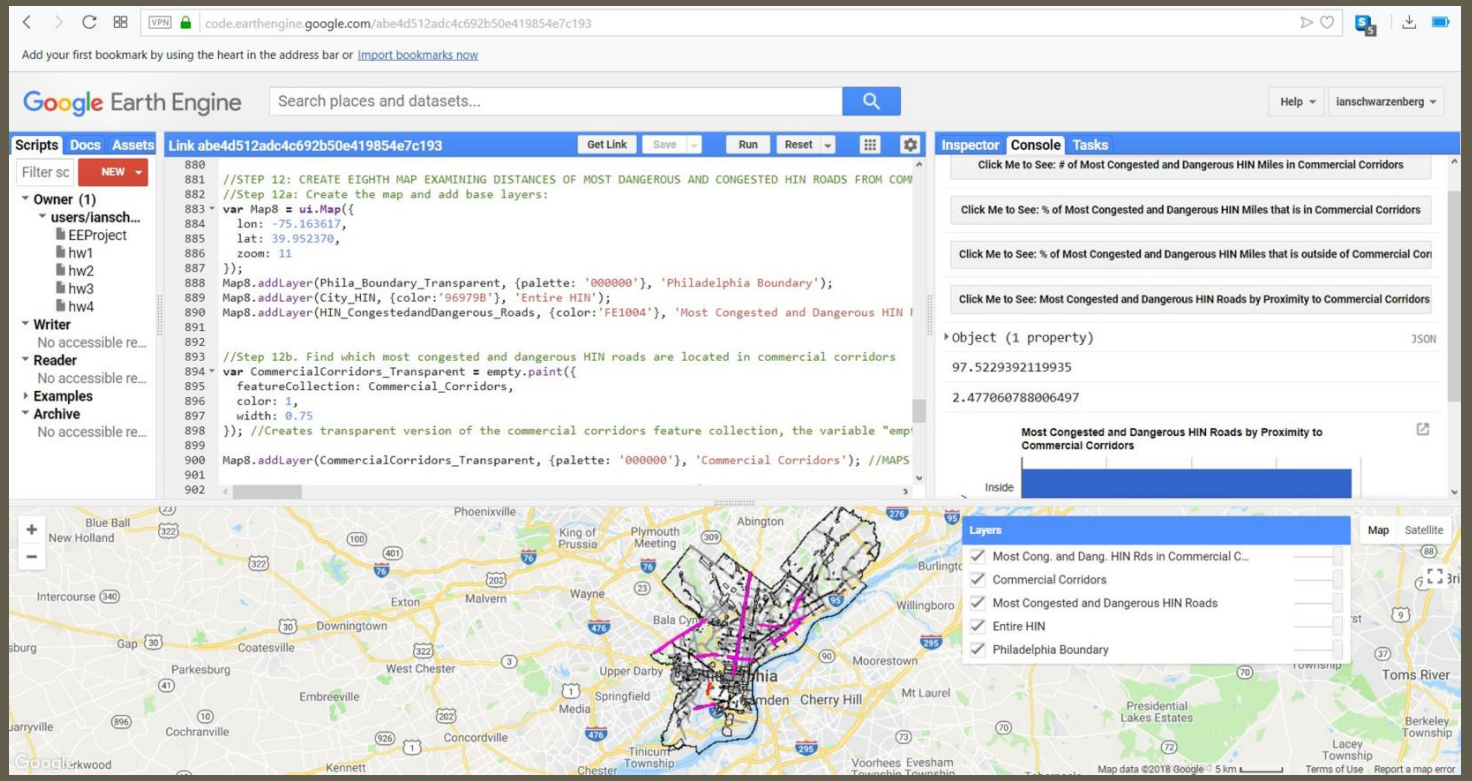
FINAL MAP

- Once those four statistics and graph buttons were created, the final map was made. The purpose of this map is to show the user which of the most congested and dangerous HIN roads travel within walking distance of rail stops. The statistics and graph show that a slight majority of the most congested and dangerous HIN roads travel right by rail stops. This may be the most surprising finding, as roads farther from rail stops would logically be more congested. This is because people who live within walking distance of rail stops would have closer access to transit. Therefore, it is very surprising to find how even a slight majority of the most congested and dangerous HIN roads travel right by rail stops.



Map 7 Final

Map 8/Choice h



Map 8 Overview

INTERSECTING

- At this point in the analysis process, I now have: 1) a variable of the most congested HIN roads, 2) a variable of the most congested *and* dangerous HIN roads, and 3) basic statistics and feature collections detailing 1) and 2), those streets' widths, and the wealth of the neighborhoods they pass through and their proximity to rail stops displayed through buttons. The goal of this eighth *and final* map is to additionally examine characteristics of the most congested *and* dangerous HIN streets to see why they might be this way, specifically examining their proximity to *concentrations of commercial parcels*, represented by *commercial corridors*. If most of the most congested and dangerous roads end up passing through commercial corridors, then this could reveal potential reasons as to why these roads are especially dangerous and congested. For example, a road passing through a commercial corridor could attract more people to drive and walk on it, as the high concentration of commercial parcels serve to draw people to travel to them.
- The first step after establishing the map with ui.Map({}) was to load in the Philadelphia commercial corridors shapefile:

```
var Commercial_Corridors = ee.FeatureCollection('ft:15e5-R5f6ZiE4By_IzfeCyAQTG6VmL6gPyj7R71LQ'); // Loads in all Philadelphia commercial corridors (Source: Philadelphia Department of Planning and
//Development)
```

- Using my special intersection method from previous maps, I then intersected the most congested and dangerous roads with the commercial corridors to create a variable containing the roads that pass through these corridors:

```
var HIN_CongestedandDangerous_ByCommercial = intersection_Join.apply(HIN_CongestedandDangerous_Roads, Commercial_Corridors, intersection_Filter); //Intersects most congested and dangerous
//HIN roads with commercial corridors to find which most congested and dangerous HIN roads are also within commercial corridors
Map8.addLayer(HIN_CongestedandDangerous_ByCommercial, {color:'F403FF'}, 'Most Cong. and Dang. HIN Rds in Commercial Corridors'); //MAPS THE MOST DANGEROUS CONGESTED HIN ROADS WHICH
ARE ALSO IN COMMERCIAL CORRIDORS
```

- Then using the exact same method for creating a transparent Philadelphia boundary for all maps, I made the commercial corridors transparent and added that to this map:

```
var CommercialCorridors_Transparent = empty.paint({
  featureCollection: Commercial_Corridors,
  color: 1,
  width: 0.75
}); //Creates transparent version of the commercial corridors feature collection, the variable "empty" is taken from step 1d

Map8.addLayer(CommercialCorridors_Transparent, {palette: '000000'}, 'Commercial Corridors'); //MAPS COMMERCIAL CORRIDORS
```

STATISTICS

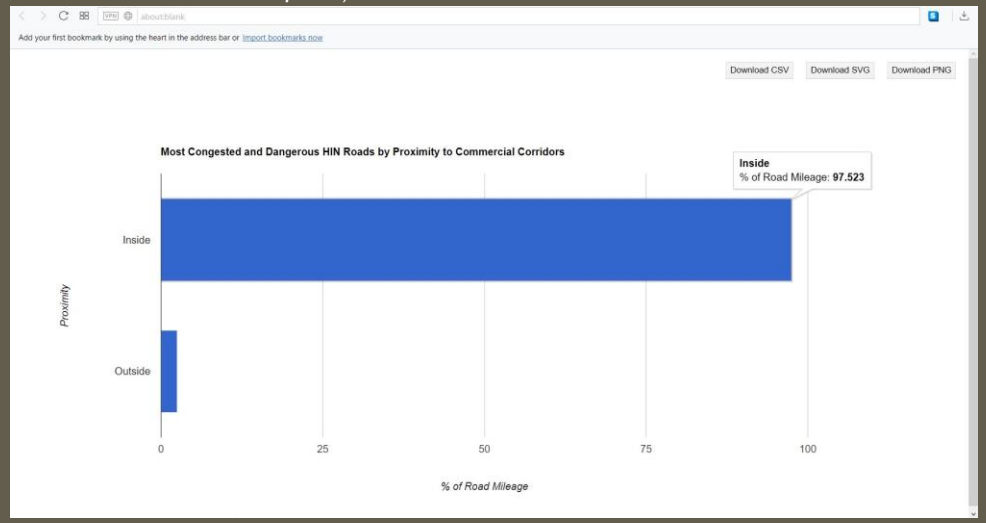
- I then created three buttons detailing statistics about the most congested and dangerous roads' relationship to commercial corridors: 1) the *number* of percent of the most congested and dangerous road that pass through commercial corridors, 2) the *percent* of most congested and dangerous road that that pass through commercial corridors, and 3) the *percent* of most congested and dangerous road that do *not* pass through commercial corridors in order to enable the user to get a more visual sense of the proportion of most congested and dangerous HIN streets that travel through commercial corridors:

```
var Map8Button1 = ui.Button({
  label: 'Click Me to See: # of Most Congested and Dangerous HIN Miles in Commercial Corridors',
  onClick: function() {
    print(ee.Number(HIN_CongestedandDangerous_ByCommercial.reduceColumns(Adder, ['length'])));
  }
});

var Map8Button2 = ui.Button({
  label: 'Click Me to See: % of Most Congested and Dangerous HIN Miles that is in Commercial Corridors',
  onClick: function() {
    print(ee.Number((65.97505508393522/67.65080668920355)*100));
  }
});

var Map8Button3 = ui.Button({
  label: 'Click Me to See: % of Most Congested and Dangerous HIN Miles that is outside of Commercial Corridors',
  onClick: function() {
    print(ee.Number(100-97.52293921 19935));
  }
});
```

GRAPH

- I then finally made the graph and corresponding button to further enable the user to see the proportion of most congested and dangerous streets that travel through commercial corridors, using the exact same button and graph creation method templates from previous maps:

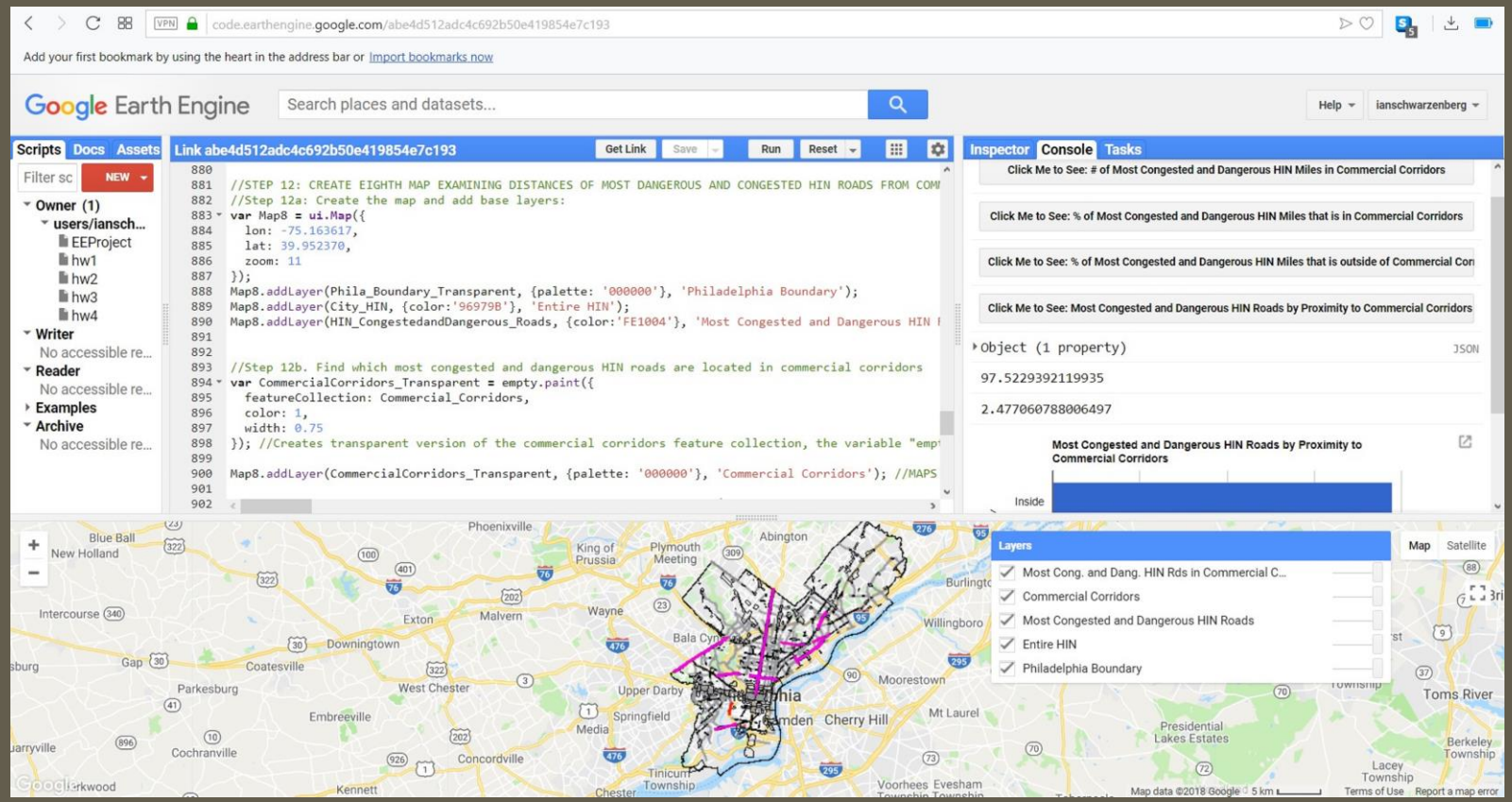
```
var HIN_CongestedandDangerous_ByCommercial_Chart_Data = {
  cols: [{id: 'Proximity',  label: 'Proximity to Commercial Corridors', type: 'string'},
         {id: 'Pct', label: '% of Road Mileage', type: 'number'}],
  rows: [{c: [{v: 'Inside'}, {v: 97.52293921199 35}]},
         {c: [{v: 'Outside'}, {v: 2.47706078801}]}],
};

var options = {
  title: 'Most Congested and Dangerous HIN Roads by Proximity to Commercial Corridors',
  vAxis: {title: 'Proximity'},
  legend: {position: 'none'},
  hAxis: {title: '% of Road Mileage'}
};

var HIN_CongestedandDangerous_ByCommercial_Chart = new ui.Chart(HIN_CongestedandDangerous_ByCommercial_Chart_Data, 'BarChart', options);

var Map8Button4 = ui.Button({
  label: 'Click Me to See: Most Congested and Dangerous HIN Roads by Proximity to Commercial Corridors Graph',
  onClick: function() {
    print(HIN_CongestedandDangerous_ByCommercial_Chart);
  }
});
```



Expanded Commercial Corridor Proximity Graph in Separate Tab

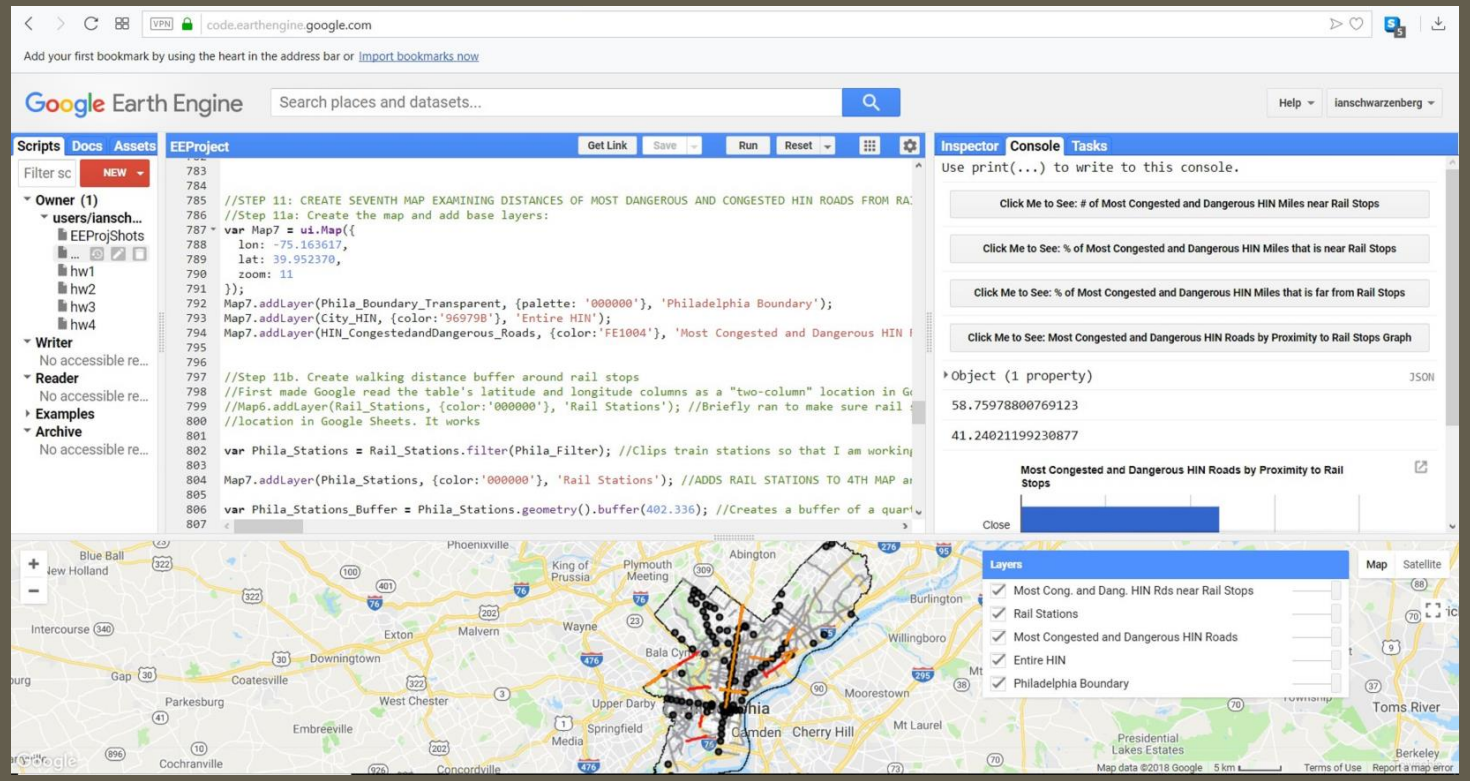
FINAL MAP

- Once those four statistics and graph buttons were created, the final map was made. The purpose of this map is to show the user which of the most congested and dangerous HIN roads travel through commercial corridors. The statistics and graph show that a clear majority of the most congested and dangerous HIN roads travel through commercial corridors. This is not surprising considering how commercial corridors could attract more people to drive and walk on the corridors' streets, as the high concentration of commercial parcels serve as trip generators which draw people to drive and walk to them. The higher number of pedestrians and cars in these corridors could potentially lead to more crashes among those cars due to their closeness, and more pedestrians getting hit by those higher volume of cars.



Map 8 Final

Conclusion



Map 7 Overview – Showing Most Surprising Findings

METHODS

- **In short, I:**
  - **Found which HIN roads are the most congested** by inner joining road congestion data to the city HIN by street name, then mapping functions over that result to create new columns in it averaging congestion data by busiest timeframe, then filtering that result and completing a series of merges to select out the most consistently congested roads.
  - **Found which HIN roads are simultaneously the most congested *and* the most dangerous** by spatially joining KSI crashes to the original HIN in its entirety to get the average HIN KSI crash rate, then filtering out congested HIN roads with KSI crash rates higher than the HIN average to get the HIN roads that are simultaneously the most congested and the most dangerous out of any other HIN road.
  - **Found the most congested *and* dangerous HIN roads' relationship to street width** by intersecting city street centerlines to the most congested and dangerous HIN roads using a special intersection join method to find which of the most congested and dangerous HIN roads are considered wide.
  - **Found the most congested *and* dangerous HIN roads' relationship to neighborhood wealth** by inner joining median household income data to census tract polygons, filtering the census tracts with the joined median household income data to find poor city census tracts, then intersecting the most congested and dangerous HIN roads with the poor census tracts using a special intersection join method to see which most congested and dangerous HIN road segments pass through poor areas

METHODS

- **Found the most congested *and* dangerous HIN roads' spatial relationship to rail stations** by creating a quarter mile walking distance buffer around rail station points, then intersecting the most congested and dangerous HIN roads with those rail station buffers using a special intersection join method to find which of the most congested and dangerous HIN roads pass by the rail stops.
- **Found the most congested *and* dangerous HIN roads' spatial relationship to commercial corridors** by intersecting the most congested and dangerous HIN roads with commercial corridor polygons to find which of those roads pass through commercial corridors.
- **Created an interactive user interface** by creating buttons which reveal statistics and graphs that portray information to the user about conditions on the most congested and dangerous HIN roads, *eliminating* Google Earth Engine's default map and *replacing* it with *eight separate maps* detailing the project, making only *one* map show at a time which the user can determine when entering the application, and creating prompts which show the viewer how to use the application, let the user choose which map(s) to see, and detailing each map's respective data and buttons.

MAPS

- **The application consists of eight maps that replace the default one:**
  - **Map 1/Choice a** simply shows the user the end result of joining congestion data to the HIN.
  - **Map 2/Choice b** simply shows the user the end result of the process of averaging each HIN road's TTI and PTI data by the busiest timeframes.
  - **Map 3/Choice c** simply shows the user the end result of finding: which HIN roads are most congested, the process of finding each HIN road's mileage, and the process of finding each road's KSI crash count and crash rate.
  - **Map 4/Choice d** shows the user which HIN roads are simultaneously the most congested and dangerous out of all other HIN roads, and basic statistics about those roads in a user-friendly manner.
  - **Map 5/Choice e** shows the user how most of the most congested and dangerous HIN roads are considered wide.
  - **Map 6/Choice f** shows the user how most of the most congested and dangerous HIN roads pass through poorer neighborhoods.
  - **Map 7/Choice g** shows the user how most of the most congested and dangerous HIN roads pass by rail stops, which is the most surprising finding.
  - **Map 8/Choice h** shows the user how most of the most congested and dangerous HIN roads pass through commercial corridors.