Crime Mapping and Navigation –

This is a capstone project by Omkar Mishra, which maps all the areas of Delhi with their appropriate crime scores so as to deploy a system of navigation that tries to avoid these places while going from point A to point B.

The code for this project can be found at - [omi265/capstone (github.com)]

Data Collection –

The data for this project was collected by scraping through various governmental and non-governmental websites along with data libraries like Indiastat.

Cleaning –

The data was cleaned and put into a csv file with the key being the area name. In total it contains data of about **166** areas in Delhi and their respective crime scores.

Clustering –

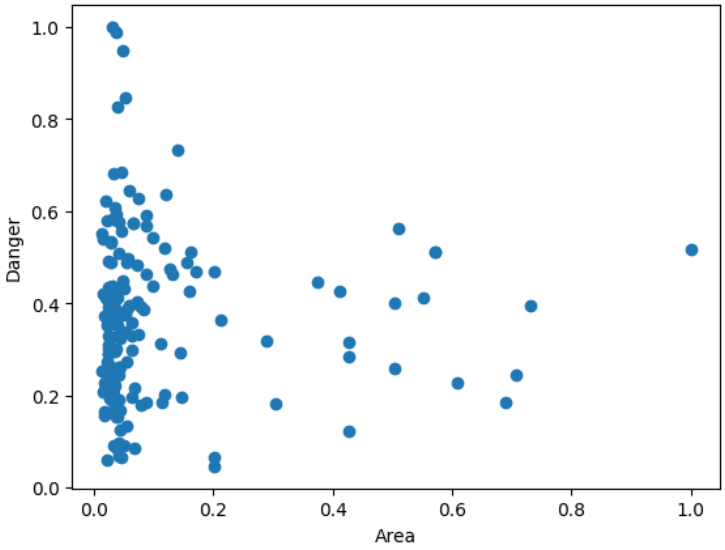
The final data sheet had 10 variables accounting for the name of the area, its latitudes and longitudes, the area covered by that specific location, as well as the number of incidents that have taken place in each category. Each crime was assigned a dangerousness value which was accordingly **–**

(5: gangrape, 5: murder, 5: rape, 4: robbery, 3: assault murder, 2: sexual harassment, 1: theft).

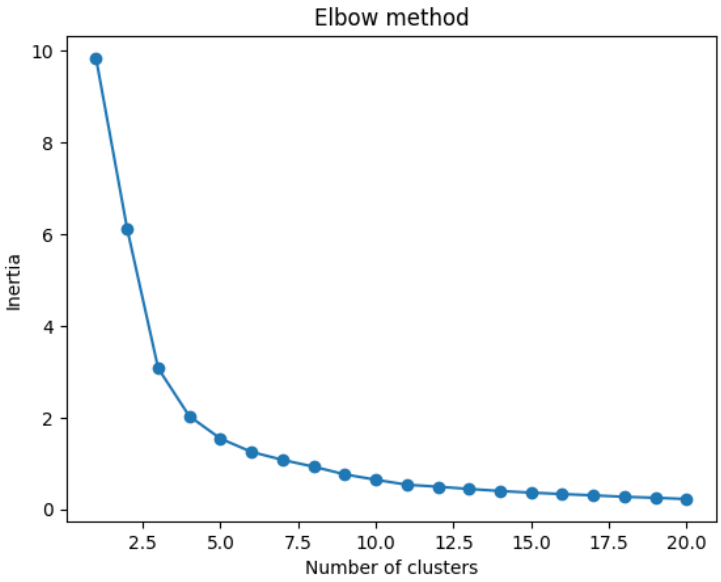
These values were multiplied with their respective incidents to give us a combined danger score. This dangerousness score and the area the crimes were committed in together were used to run a clustering algorithm on all the locations and identifying locations with the maximum density of crime (as these locations will be the most useful to avoid)

area_nam	murder	rape	gangrape	robbery	theft	assualt mu	sexual har	danger	area	Danger/ar	lat	long
CHITRANJ	2	6	1	35	442	19	7	686	2.65933	257.9598	28.53632	77.2492

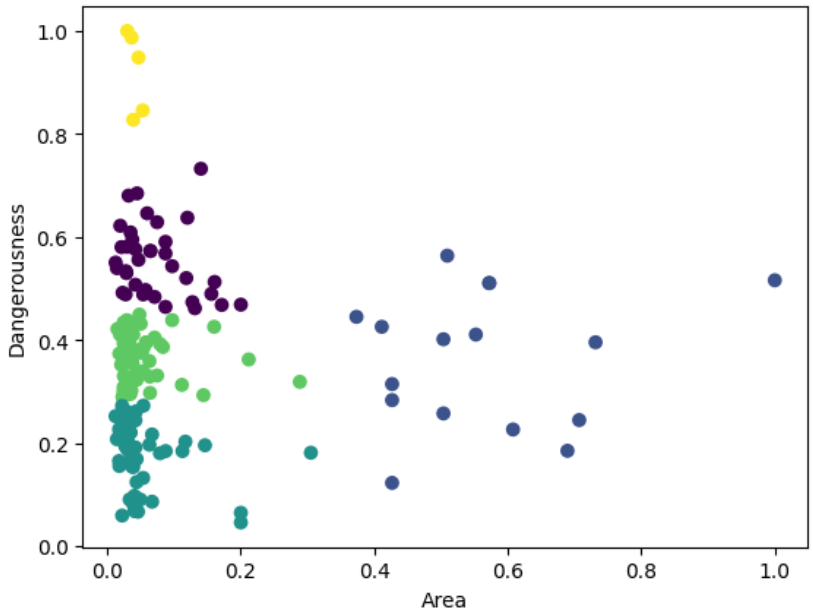
The locations when put on a graph that plots the points with the x axis representing Area and the Y axis representing the Danger scores. We then went ahead and normalized the values so that the maximum of each reads as 1.0.



A graph was created for calculating the most efficient cluster value using the elbow method, which was chosen to be 5 –



A K-Means clustering algorithm that classified the 166 locations into 5 different clusters was run giving us this graph –



The level of dangerousness for each cluster was identified using this graph –

Dark blue – Level 1

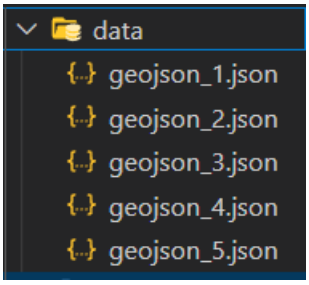
Dark Green – Level 2

Light green – Level 3

Purple – Level 4

Yellow – Level 5

The values for all of these locations were then separately put into a cleaner file which produced a geojson file for each of these levels of dangerousness and giving us 5 data files –



Each containing data for their respective level areas.

Navigation –

Various Open-Source libraries were tried to make the avoidance mechanism like Open Street Directions, Leaflet Routing machine, Open-Source Routing Machine and others. All of them provided some form of avoidance function but none of which could be utilized as they couldn't take an array of obstacles and then route around them, but were either for a single polygon or for avoiding specific type of obstacles like toll roads and bridges. We ended up using the Map Box library for creating the map, getting geographic locations as well as directions. The Turf and Polyline libraries were used along with Map Box to provide us with specific geometries of each route and the obstacles. The project was created as an HTML page which was supported by a JavaScript file doing most of the rendering and calculations.

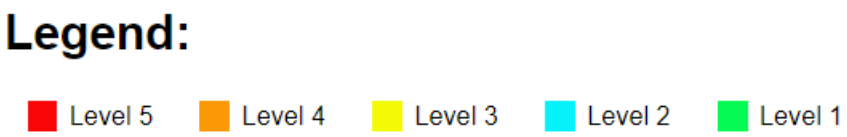
- The first step was to render a map of and center it to Delhi, this was done using the Map Box Library.

```
unknown, 2 days ago | 1 author (unknown)
mapboxgl.accessToken =
    "pk.eyJ1Ijoib21rYXIyNjUiLCJhIjoiY2xlcW5lbm5hMG56ZDN1bzE1YWN0MzNkcSJ9.cCFrcF40FFfue9TNpCYj6w";
const map = new mapboxgl.Map({
    container: "map",
    style: "mapbox://styles/mapbox/navigation-night-v1",
    center: [77.219656, 28.632161],
    zoom: 10,
});
```

- The next step was to input the geojson files and use them to create the obstacles on the map to be avoided. Each of these obstacles was created according to the dangerousness of the locations, i.e. the more densely populated areas by crime were given the largest radius to avoid.

```
const obstacle_5 = turf.buffer(clear_5, 0.5, { units: "kilometers" });
const obstacle_4 = turf.buffer(clear_4, 0.4, { units: "kilometers" });
const obstacle_3 = turf.buffer(clear_3, 0.3, { units: "kilometers" });
const obstacle_2 = turf.buffer(clear_2, 0.2, { units: "kilometers" });
const obstacle_1 = turf.buffer(clear_1, 0.1, { units: "kilometers" });
```

- These obstacles were then rendered on the map by adding a layer for each different obstacle level. These obstacles were then given a color based on their level of dangerousness accordingly.



```
map.addLayer({
  id: "clear_5",
  type: "fill",
  source: {
    type: "geojson",
    data: obstacle_5,
  },
  layout: {},
  paint: {
    "fill-color": "#fa0505",
    "fill-opacity": 0.8,
  },
});
```

- Every time the points A and B are selected on the map, either by clicking at two different points or typing the locations out, a route is calculated. If the number of tries for that route has not exceeded the maximum number of tries, the function continues. Otherwise it adds a No route found in the max attempts card to the application which is displayed to the user.

```
directions.on("route", (event) => {
  map.setLayoutProperty("theRoute", "visibility", "none");
  map.setLayoutProperty("theBox", "visibility", "none");

  if (counter >= maxAttempts) {
    noRoutes(reports);
    counter = 0;
    dataCounter = 0;
  } else {
```

- On a route being found within the maximum number of attempts, the algorithm then checks if the geometry of the route clashes with that of the obstacles. These set of obstacles are chosen based on the attempt number the algorithm is it. For the first 20 tries, the algorithm tries to avoid all possible obstacles and sets the value of clear as true. If not then in the next 20 tries, it removes the areas with the lowest level of dangerousness.

```
switch (dataCounter) {
  case 0:
    hit_obstacle = "all dangerous areas";
    if (turf.booleanDisjoint(obstacle_5, routeLine)) {
      if (turf.booleanDisjoint(obstacle_4, routeLine)) {
        if (turf.booleanDisjoint(obstacle_3, routeLine)) {
          if (turf.booleanDisjoint(obstacle_2, routeLine)) {
            clear = turf.booleanDisjoint(obstacle_1, routeLine);
          }
        }
      }
    }
    console.log(clear);
    break;
  case 1:
    hit_obstacle = "all dangerous areas upto level 2";
    if (turf.booleanDisjoint(obstacle_5, routeLine)) {
      if (turf.booleanDisjoint(obstacle_4, routeLine)) {
        if (turf.booleanDisjoint(obstacle_3, routeLine)) {
          clear = turf.booleanDisjoint(obstacle_2, routeLine);
        }
      }
    }
    console.log(clear);
    break;
  case 2:
    hit_obstacle = "all dangerous areas upto level 3";
    if (turf.booleanDisjoint(obstacle_5, routeLine)) {
      if (turf.booleanDisjoint(obstacle_4, routeLine)) {
        clear = turf.booleanDisjoint(obstacle_3, routeLine);
      }
    }
    console.log(clear);
    break;
  case 3:
    hit_obstacle = "all dangerous areas upto level 4";

    if (turf.booleanDisjoint(obstacle_5, routeLine)) {
      clear = turf.booleanDisjoint(obstacle_4, routeLine);
    }
    console.log(clear);
    break;
  case 4:
    hit_obstacle = "only the most dangerous areas";
    clear = turf.booleanDisjoint(obstacle_5, routeLine);
    console.log(clear);

    break;
}
```

- If a route is found, i.e. the Boolean value of clear is true, then the user is displayed with the Route Found card alerting them of the level of areas that the route avoids, while also letting them see the actual route on the screen along with the turn by turn directions. All of the counters are again set to 0 for the next navigation task.

```
if (clear === true) {
  collision = "the safest possible route !";
  detail = `takes ${(route.duration / 60).toFixed(
    0
  )} minutes and avoids ${hit_obstacle}`;
  emoji = "✔️";
  map.setPaintProperty("theRoute", "line-color", "#74c476");
  map.setLayoutProperty("theBox", "visibility", "none");
  counter = 0;
  dataCounter = 0;
  console.log(hit_obstacle);
} else {
```

- If no route is found in that particular try, i.e. the geometry of the route collides with the geometry of the obstacles, then the counters are incremented and a polygon is calculated around the route which steadily increases based on the attempt number the machine is at. This yellow box is also displayed to the user. The user also gets notified that the route is bad as it hits obstacles and the attempt number the machine is on by sending another card which also tells the user the amount of time this route might have taken.

```
} else {
  counter = counter + 1;
  dataCounter = Math.floor(counter / 20);
  console.log(dataCounter);
  polygon = turf.transformScale(polygon, counter * 0.025);
  bbox = turf.bbox(polygon);
  collision = "is bad.";
  detail = `takes ${(route.duration / 60).toFixed(
    0
  )} minutes and hits some possibly dangerous areas`;
  emoji = "⚠️";
  map.setPaintProperty("theRoute", "line-color", "#de2d26");
```

- A random waypoint is selected inside the yellow box which is relatively close to the original route and this waypoint is then used to create another set of directions.

```
} else {
  counter = counter + 1;
  dataCounter = Math.floor(counter / 20);
  console.log(dataCounter);
  polygon = turf.transformScale(polygon, counter * 0.025);
  bbox = turf.bbox(polygon);
  collision = "is bad.";
  detail = `takes ${(route.duration / 60).toFixed(
    0
  )} minutes and hits some possibly dangerous areas`;
  emoji = "⚠";
  map.setPaintProperty("theRoute", "line-color", "#de2d26");
```

- This Process continues, till a clear route has been found or the algorithm reaches its maximum number of attempts.

This method was chosen due to the sheer number of obstacles that needed avoidance during the process of navigation. Other methods such as perimeter avoidance were tried, which would have resulted in a very weird route as it would have gone along the perimeter of every obstacle that the route had found as well as took a long time for route calculation. The number of obstacles made it clear that the best way was to find random waypoints around the obstacles and the route to then reroute using that same waypoint. This also meant that the user could visually see what was happening as the machine continued with its attempts while also giving them statistics about the routes that they are avoiding.

The number of tries was to be kept not so low as the machine cannot find a route but not so high so as to take a substantially long time to come up with a route. The chosen number came out to be 100, which gives us a result within a maximum of 10 seconds. The division for removal of obstacles based on the attempt number was simple, we just divided the tries into equal parts, i.e. 20 for each level of obstacles.

This gave us a result of a comparatively faster calculation while also not leaving room for trial and error for the algorithm.