- Installed scikit-learn, matploit, seaborn, numpy and pandas

- Started off by fetching data (latitude, longitude) using scikit-learn from from [The California housing dataset — Scikit-learn course (inria.github.io)]

Project Overview:

This project tackles the problem of finding an optimal gas pipeline route to serve a set of houses. The code addresses two key objectives:

1. **Efficiency:** Minimizing the total distance between all houses and the pipeline (implemented in the `efficient` function).
2. **Fairness:** Minimizing the maximum distance any single house has to the pipeline (implemented in the logic for finding `taskTwoanswer`).

Challenges Encountered

Data Representation and Handling Vertical Lines:

- The code uses slope-intercept form (y = mx + b) to represent lines. This simplifies calculations but excludes vertical lines (where x is constant). Vertical pipelines might be valid options in some scenarios, and the current approach cannot handle them directly.

Challenges in Finding Efficient Lines:

2. **Local Minima vs. Global Minima:**
   - The current approach calculates the total distance for each potential pipeline and selects the one with the minimum value. This identifies a line with a local minimum total distance, but it might not be the absolute best solution (global minimum) that minimizes the total distance for all houses. As the number of houses and potential pipelines increases, finding the global minimum becomes more complex.
3. **Computational Complexity:**
   - Evaluating every possible pipeline to find the most efficient one can become computationally expensive, especially for a large number of houses.

Challenges in Finding Fair Lines:

1. **Computational Cost:**
   - The code finds the fair line by calculating the maximum distance to each house for every line, then selecting the line with the minimum maximum distance. While effective, this approach can be computationally expensive for a large number of

houses and pipelines. Calculating the maximum distance for every house-pipeline combination can become time-consuming.

KEY DECISIONS

1. **Line Representation:**
    - The code prioritizes simplicity by using slope-intercept form for line representation. This allows for efficient distance calculations but doesn't handle vertical lines natively.
2. **Finding Efficient Line (in `efficient` function):**

    One key decision made in designing such an algorithm is to focus on finding a local minimum, rather than the absolute minimum. A local minimum means that the cost is lower than any of its neighboring lines. This is a good approximation to the true minimum, and it can be computed much more efficiently.

3. **Finding Fair Line (logic for `taskTwoanswer`):**
    - Employed a basic method to identify the fair line. It calculates the maximum distance to each house for every line (`distance_list`) and then selects the line with the minimum maximum distance (`min_distance`).

Solutions Implemented

Objective 1: For Efficiency of a line:
A straightforward approach is used to find a line with a local minimum total distance. It calculates the total distance for each potential pipeline (by iterating through `line_list`) and selects the one with the smallest total distance (`min_distance`).

- This part of the code finds a line that minimizes the total distance between all the points and the line.
    - It creates an empty list `line_list` to store the lines.
    - It iterates through all pairs of points in the `point_list`.
        - It calculates the slope (`m`) and y-intercept (`c`) of the line that goes through the two points using the following formula: `m = (y2 - y1) / (x2 - x1)`
          `c = y1 - mx1` (where x1, y1 are coordinates of the first point and x2, y2 are coordinates of the second point).
        - It excludes lines with a vertical slope (infinite slope) by checking if the difference between the x-coordinates of the two points is zero.
        - If the slope is not vertical, it appends a list containing the slope (`m`) and y-intercept (`c`) to the `line_list`.
- The code then defines a function called `efficient` that takes the `distance_list` as input.

    - This list contains the total distances between each line in *line_list* and all the points in *point_list.*
    - The function finds the line with the minimum total distance by finding the index of the minimum value in *distance_list.*
    - It then returns the line from *line_list* corresponding to that index.
- Finally, the code calls the *efficient* function with the *distance_list* and prints the most efficient line (the line with the minimum total distance).

Objective 2: Minimizing Maximum Distance (Fair Line)

Objective 2 focuses on finding a line that minimizes the **maximum distance** between any house (point) and the line. This is also referred to as finding a **fair line**.

    - The code employs a basic method to identify the fair line. It calculates the maximum distance to each house for every line (*distance_list*) and then selects the line with the minimum maximum distance (*min_distance*)..

Here's how the code addresses Objective 2:

1. Iterate Through Lines**:** It loops through each line (*i*) in the *line_list*.
2. Calculate Distances to All Points**:** For each line *i*, it creates a temporary *distance_list* to store the distances between all points (*j*) in the *point_list* and the current line i. It calculates these distances using the same *distance* function as in Objective 1.
3. Find Maximum Distance**:** It finds the maximum distance within the *distance_list* for the current line *i*. This represents the farthest any house is from that particular line.
4. Store Maximum Distances**:** It appends the maximum distance for the current line *i* to a new list *distance_list2.*
5. Find Line with Max-min Distance**:** Finally, it finds the minimum value in *distance_list2*. This represents the line with the **smallest maximum distance** among all lines, which translates to the fairest line where no house is excessively far away.
6. Print Fair Line**:** It identifies the index of the minimum value in *distance_list2* and uses that index to retrieve the corresponding line from *line_list*. This line represents the solution for Objective 2 (the fairest line).

Objective 3: Finding K Most Efficient Lines

Objective 3 aims to identify multiple efficient lines, not just a single one.

1. Define `multiple_efficient` Function: It defines a function named `multiple_efficient` that takes an integer `k` as input. This `k` represents the number of most efficient lines to find.
2. Iterative Process: Inside the function, it loops `k` times.

- In each iteration, it calls the `efficient` function (from Objective 1) with the current `distance_list`. The `efficient` function finds the line with the minimum total distance among all remaining lines.
- The identified line is then appended to the `taksThreeAnswer` list.
- Importantly, the function also removes the identified line (the most efficient one in that iteration) from the `distance_list`. This ensures it doesn't get picked again in subsequent iterations.

3. Return Top K Lines: After `k` iterations, the `taksThreeAnswer` list will contain the `k` most efficient lines found based on the total distance criteria. The code assigns this list to `ans3`