1 System Requirements & Algorithm

Management Problem

With the development of COVID-19 vaccines came the need to quickly develop a supply chain that is efficient to rapidly distribute it on a large scale. Distribution centres (DCs) are at the core of building supply chains as they are contact points to both incoming deliveries from manufacturers and outgoing deliveries to hospitals. As such, DCs need to be located so they can be used to deliver vaccines to the largest area possible in the shortest time. As such, DCs should be placed to optimise for distance to hospitals and coverage.

We find optimal locations for DCs using the weighted k-means algorithm to cluster hospitals according to their location and number of cases of COVID-19. We will look specifically at the mainland US.

The Algorithm

The K-means clustering algorithm is a method of unsupervised learning for classification that partitions data into "K" number of clusters.

Steps of the algorithm:

1. Initialise the position for k centroids
2. Assign each point to the nearest centroid
3. Recalculate position of centroids
4. Repeat steps 2 and 3 until the centroids no longer move.

As hospitals in areas experiencing more COVID-19 cases should get more vaccine deliveries, they should be closer to DCs.

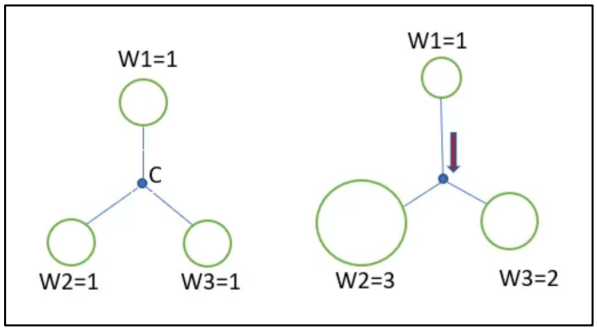

Figure 1: Weighted centroids

As such, we should use a weighted approach in calculating centroids to account for the higher preference for some hospitals (see Figure 1). This means that optimal DC locations would be closer to hospitals with more cases.

Requirements

Functional Requirements:

Inputs include

- Chosen number of clusters k,
- Latitude and longitude of each hospital and
- Weights associated which in this case, we take as the number of cases of COVID-19 that experienced in the county that each hospital is located in

Outputs include

- Latitude and longitude of the optimal DC locations given k clusters, and
- List of hospitals under each DC's coverage

Non-function Requirements:

The software had to be able to take a large amount of data (up to 10,000 rows) in order to assign all hospitals in the mainland US to DCs. It has to also be able to cluster data up to a large k and produce results within one hour.

How the Algorithm Solves the Problem

The algorithm should give clusters of hospitals that minimise average distance to each hospital and maximise coverage for each DC while also giving hospitals with more cases a greater preference. The centroids of each cluster would give the optimal location

The algorithm can help planners consider where to place DCs. Outputs from the algorithm can be interpreted to give insights into several factors, such as:

- Average distance from hospitals to each DC
- Number of hospitals under each DC
- The relative size of each DC (from sum of weights)

By using the algorithm, they can find the optimal location of k DCs and consider whether the factors above fit with their requirements.

2 Explaining how the code works

All code is written from scratch with the exception of the k++ initialisation method (sources are in code appendix).

Hierarchy of Functions

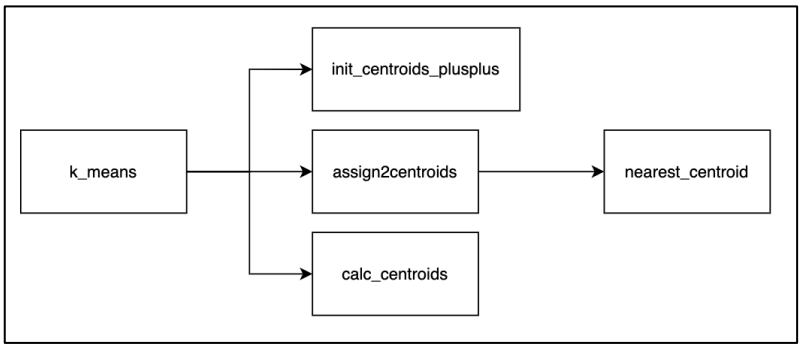

Figure 2: Hierarchy of Functions

The final function *k_means* is decomposed into 3 main functions and 1 sub-function (see Figure 2).

```
init_centroids_plusplus(k: int, df: pd.DataFrame)
```

This function returns a dictionary of initial coordinates of centroids, using the k++ method. This method distributes initial centroids more evenly in order to converge more quickly while also reducing the effect initialisation on the final clustering to give more consistent results.

```
nearest_centroid(d_loc: list , centroids: dict)
```

nearest_centroid iterates over the entire centroids dictionary to calculate the distance between each centroid and the data point. The calculated distance and corresponding centroid are stored in the *dist2nearest* and *nearest* holder variables. Once all centroids have been iterated over, *nearest* is returned, giving the nearest centroid.

```
assign2centroids(df: pd.DataFrame, centroids: dict)
```

assign2centroids iterates over the DataFrame to set a new value for the "centroid" column of each row to be the nearest centroid (obtained from *nearest_centroid* function). It returns a DataFrame with all points reassigned to the nearest centroid.

```
calc_centroids(k: int, df: pd.DataFrame)
```

calc_centroids iterates over centroids and subsets the DataFrame by column “centroid” to give *sub_df* of points assigned to each centroid. The “lat”, “long” and “weights” column are then used to calculate geometric means for latitude and longitude. These values are stored in a dictionary *cens.* With all centroids calculated, this dictionary is returned.

Final K-means Function

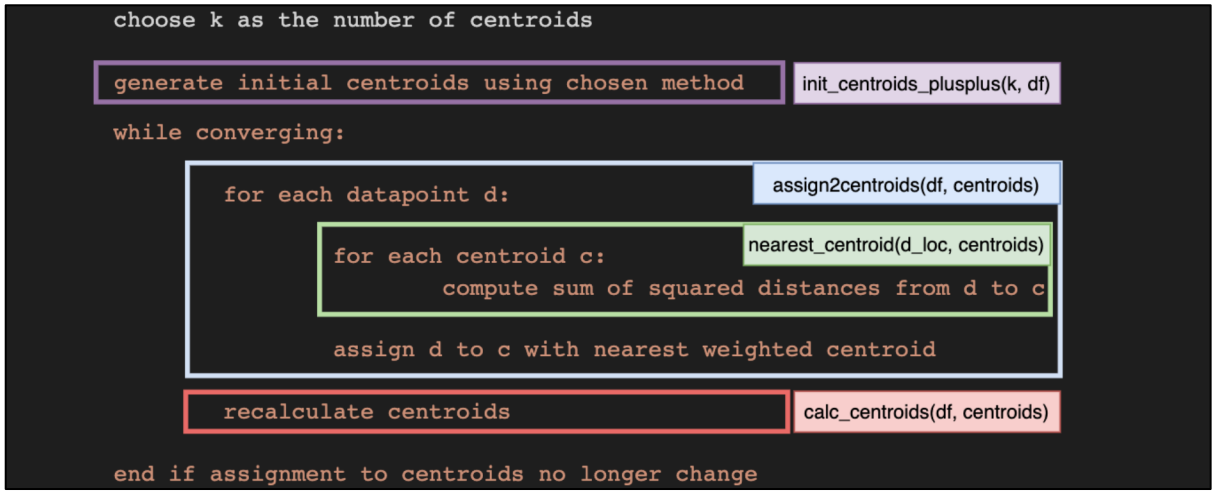

Figure 3: Core functions according to Pseudo Code

The final function is implemented according to Pseudo code using the core functions (see Figure 3). The centroids are first initialised and data points are assigned to nearest centroids. In a for loop that sets maximum iterations to 150, data points are assigned to nearest centroids and centroids are recalculated. If recalculated centroids are the same as in the previous iteration, we break from the loop and return the position of centroids and DataFrame which contains assignments to centroids.

Flow of Code

The main code is divided into sections:

1. Installing Modules & Importing Libraries
   All relevant libraries are installed and functions are imported.

2. Loading Dataset
   The function for data preparation *get_data* is defined. It includes loading the data from a .csv file, dropping rows containing invalid data and data that is out of scope of the mainland US and renaming.

3. Visualisation Functions

   This section visualises dataset and contains functions to visualise data points (*viz_scatter*) and centroids (*viz_centroids*) or both (*viz*).

4. Core Functions

   The functions above are defined in this section.

5. Final K-means

   Core functions are pieced together according to Pseudo code.

6. Implementation

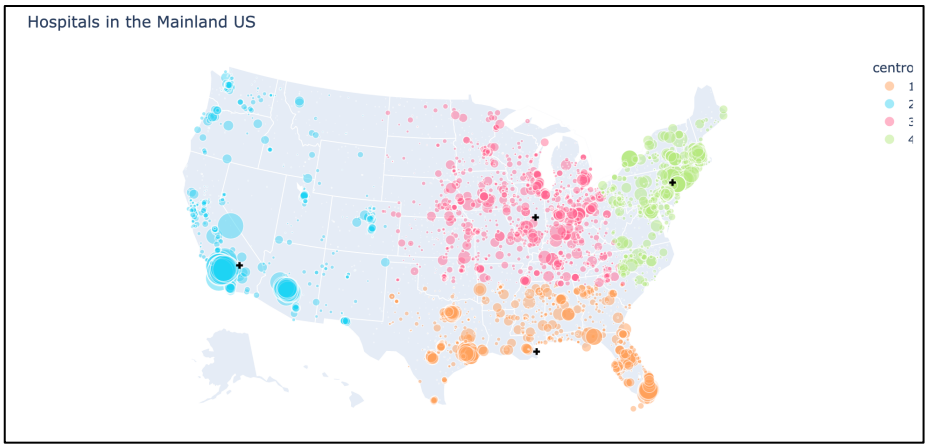

Figure 4: Clustering with k=4

K_means is implemented with k=4 as an example.

7. Interpreting Results

```
Optimal DC Locations
{1: 'Saint Bernard Parish, Louisiana, United States',
 2: 'Powerline Road, San Bernardino County, California, United States',
 3: 'North 1100 East Road, Piatt County, Illinois, 61913, United States',
 4: '3, Tara Drive, Troy Hills, Parsippany-Troy Hills, Morris County, New Jersey, 07054, United States'}
```

```
Average distance from DC to Hospital for each DC
{1: 648.9165990281155,
 2: 757.2386814886006,
 3: 579.107051589054,
 4: 338.80072069129676}
```

Figure 5: Interpretations

Outputs from implementation above are interpreted, giving insights such as address of the optimal locations and average hospital-DC distances.

Generalisation of the Code

The code is generalised such that it can be used for distribution centre planning for other types of supply chain and also any other types of general location clustering and can be used for both weighted and non-weighted approaches (by setting weight=1).

Alternative applications in DC planning:

- Finding optimal locations for last-mile hubs in one-day delivery
- Finding optimal location for delivery drone launch points

Examples of other general location clustering:

- Finding centres of population

Data Representation

Data is represented in tabular form in one single Pandas DataFrame containing a hospital's information, location and centroid it has been assigned to. This provides for ease of iterating and reassigning values. Additionally, Pandas DataFrames efficiently handles large amounts of data while it minimises the number of lines of code necessary to do basic operations such as sorting, condition-based filtering and aggregating. It is also easily compatible with visualisation tools such as Plotly.

Location of centroids are represented as a Python dictionary. This is for concisely accessing centroid locations and ease of iterating over the centroids.

Noteworthy Coding Functions

List and dictionary comprehensions were used extensively throughout the code where possible in order to remove unnecessary lines of code. The function get_data() is also often used as argument for the *k means* function instead of passing the DataFrame directly especially when a smaller sample of the dataset is being used.

3 Complexity Analysis

The size of input dataset n to the k-means algorithm can vary drastically in size depending on the scale and scope. For example, clustering individual customers instead of hospitals can increase size of data drastically. The scope of geographical area being clustered could also significantly increase the number of points to be clustered and hence data size. Other inputs such as number of dimensions d and clusters k also can also vary according to requirements of the problem. For example, if it is required that points have to be close to their centroids and only few points can be assigned to one centroid, k may be fixed to be large.

The complexity of the K-means algorithm is:

$$O(n * k * i * d)$$

n: number of data points
k: number of desired clusters
i: number of iterations
d: dimensions

It is observed that the i is directly proportional n and as k and d can be fixed, the effective time complexity becomes $O(n^2)$ (Pakhira, 2014). As such, the algorithm is considered computationally difficult (NP-hard).

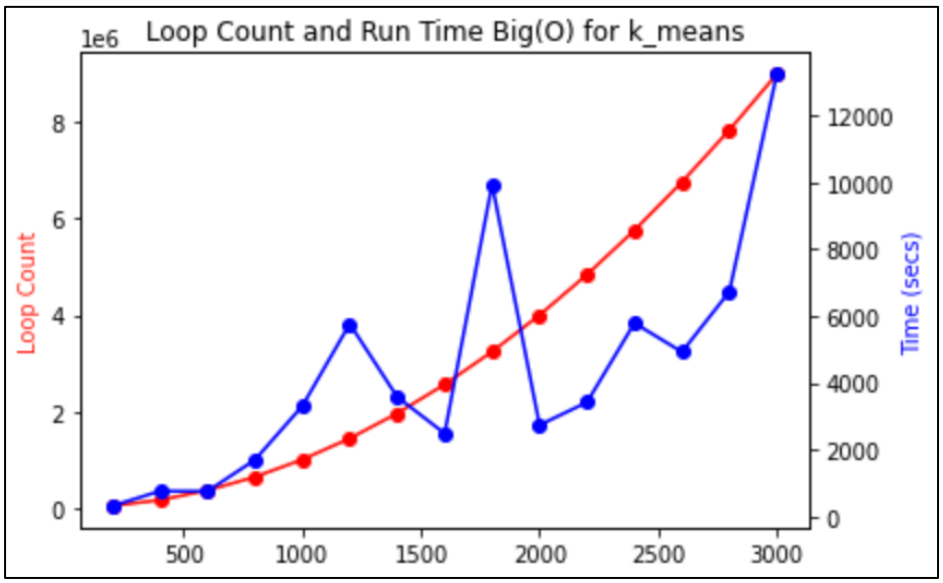

Figure 6: Big(O) plot

The runtime plot of data sizes from 300 to 3000 shows a shape similar to the theoretical loop count plot that assumes $O(n^2)$. Some data sizes resulted in very long run time while the majority follows a positively increasing trend. The outliers could be a result of initialisation.

With a large k and large dataset, it can take significantly longer to converge.

4 Data used to test the code

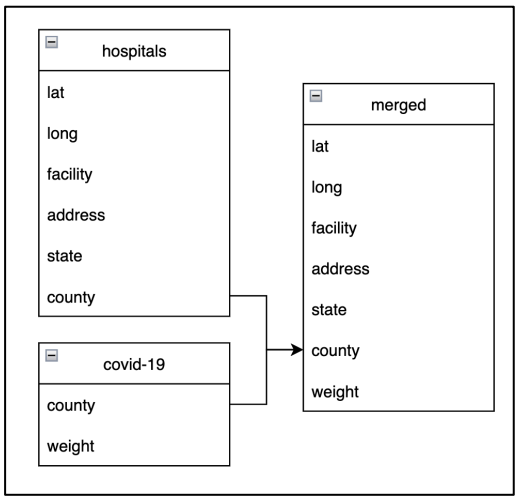

Figure 4: ER Diagram

The final data used is from merging two tables (see Figure 4). The first contains the location and information of US hospitals[1]. The second contains the number of active cases of COVID-19 in US counties in September 2020[2]. Both datasets were obtained from Kaggle. Real data was used in order to more closely represent the real-world situation.

This data could be used to test the algorithm with varying number of clusters k. An example in the code used k=4. This gave optimal location of DCs and hospital allocations.

DataFrames of size 300 to 3000 were created by randomly sampling from this dataset in order to analyse the complexity. Random sampling mitigates the effects of centroid initialisation on the number of iterations, which directly affects complexity. This was incorporated into the *get_data* function.

[1] https://www.kaggle.com/datasets/carlosaguayo/usa-hospitals

[2] https://www.kaggle.com/datasets/sudalairajkumar/covid19-in-usa

5 Conclusions

Following Good Design Principles

The code follows good design principles by decomposing into smaller problems and solving them with smaller functions. Sections were designed to be modular in order to reuse functions and reduce the number of lines of code and repetition. For example, visualisation functions were designed such that they could visualise centroids and points together and separately. The function *calc_centroids* could also be used to both initialise and calculate centroids.

Additionally, the code was designed to be customisable as functions have optional parameters. For example, the initialisation method can be chosen in *k_means* as well as viewing options for the visualisation.

Alternative Approaches

DBSCAN would also alternatively be appropriate for this problem as it is density-based. This would allow hospitals that are near each other to be place the same cluster, possibly enabling more efficient delivery routing. However, it would not assume spherical clusters and would not optimise hospital-DC distance.

Hierarchical clustering would provide a more structure approach as points are clustered into a hierarchy. This allows us to choose the number of clusters more easily based on requirements such as the different sizes of the DCs. However, it is not suitable for large datasets as it has a large time complexity.

Possible Improvements

The code could be improved to by including methods to find optimal k such as the Silhouette method. This would eliminate the need to try different cases of k.

How the code can be generalised

The code can be modified in order to address any general k-means clustering. Haversine distance can be replaced with Euclidean distance to cluster non-locational data. It could also be modified to cluster a dynamic number of attributes by adding another for loop to perform the steps on each attribute instead of hard coding it for only latitude and longitude. As such, it would be able to solve a clustering problem for more than two-dimensional data.

Limitations in test data and Implications for findings

The test data considers all hospitals as the same size when in fact they are not. As such it does not consider the capabilities of storing and administering vaccines. This means optimal locations may not necessarily lead to an efficient supply chain.

The test data also uses active cases as weights for clustering. Hence, optimal locations may only be relevant to only the current situation. Since DCs are planned for long-term use, other factors should be as weights such as predicted annual number of vaccine doses required for each location.

References

M. K. Pakhira, "A Linear Time-Complexity k-Means Algorithm Using Cluster Shifting," 2014 International Conference on Computational Intelligence and Communication Networks, Bhopal, India, 2014, pp. 1047-1051, doi: 10.1109/CICN.2014.220.