Genetic Algorithms for Cluster Analysis

A Comparison with K-Means & Gaussian Mixture Models

Delaney Scheiern

MATH482B, Colgate University

December 11, 2020

1. Introduction

From trivial tasks like playing chess, to lifesaving ones like identifying cancerous tissues, machine learning is advancing and challenging society. But why have we begun to rely so heavily on what computers tell us to do? Machine learning can produce results from data in one hour that would take humans years to find, if ever. Machine learning has been used in nearly every field, from sports to health services. Specifically, cluster analysis has branched out as one of the most popular forms of machine learning. With clustering, we can predict how our observations fit into a specific number of groups that are as similar as possible [4]. This method has been applied in a variety of areas, ranging from predicting someone's asthma type based on different factors [9], to predicting which soccer position someone will excel at based on their skills [12].

Project Description

This project introduces Genetic Algorithms for Cluster Analysis, aided by a discussion of more simplistic clustering techniques such as k-Means and Gaussian Mixture Model clustering. The project focuses on a density-based initialization of chromosomes as the genetic clustering technique. These methods will be demonstrated in an application to environmental conditions in the United States for clusters created from health insurance coverage statistics.

Below, important definitions are introduced, including the clustering algorithms that are the focus of this paper – K-Means, Gaussian Mixture Models, and Genetic Algorithms.

1.1. Machine Learning

Machine Learning is the science of programming computers to learn from data of observations and experiences [8]. The training data for machine learning models, which is the data that helps the program learn, is made up of observations called training instances or samples [8]. For each model there is a performance measure, which helps the programmer know when the model is performing well or as expected. With this, the programmer can make changes to various parameters in the model, or acquire more training data if needed. Machine Learning helps discover patterns that are not easily found in data.

1.1.1. Clustering

Cluster analysis helps arrange observations into different groups, without the user needing to tell the program which observations belong to which group. These clustering algorithms perform two distinct functions. The first part calculates the similarity between observations based on the specified parameters [9]. The second part groups observations into clusters in order to have strong similarity within clusters and weak connection between different clusters [9].

1.2. k-Means

The k-Means algorithm is a clustering algorithm and considered hard classification. In k-Means clustering, the number of clusters is decided preemptively based on one of many possible algorithms. K-Means is a distance-based method, so observations are divided into k clusters based on the Euclidean distance of the observation to the mean of each cluster [2]. The Euclidean distance is defined as $d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$ [4]. The mean value is consistently updated until the algorithm produces the finalized clusters [2]. There are two main disadvantages of k-Means. Firstly, k-Means can only find circular, spherical, or hyper-spherical clusters, depending on the dimension of the data. Secondly, since this is hard classification, observations are assigned to groups without indicating to us the actual probability that they belong to those clusters.

1.3. Gaussian Mixture Models

Gaussian Mixture Models (GMMs) can be leveraged in cluster analysis to overcome issues of distance-based clustering methods, specifically k-Means. To cluster with GMMs, we need to know which observations belong to which Gaussian distributions, as well as the specific parameters for each Gaussian distribution. Once we obtain these distribution details, each distribution can be considered a cluster. GMM clustering is soft classification, so we are given the probability of being in each cluster for each observation rather than just

assigning a cluster. GMMs are not restricted to spherical clusters, so they can more accurately define clusters that are in close proximity and oddly shaped. The necessary values for clustering can be found with the Expectation-Maximization algorithm, described in Section 2.4.3.

1.4. Genetic Algorithms for Clustering

Genetic Algorithms are extremely efficient at solving optimization problems. They use Charles Darwin's concept of natural selection and survival of the fittest. We consider the first group of proposed solutions as the first generation. Then, based on the user's objective or objectives, a new generation is created through mating and mutation. Below are additional details about the steps for a generic Genetic Algorithm. Depending on the method, the steps within the selection process may be rearranged to accomplish different goals.

1.4.1. Chromosome Representation

The term chromosomes is used to describe the potential solutions of the optimization problem. Although they encode bits of data like human chromosomes, they are distinct in every other way. Each generation is made up of many chromosomes that will be manipulated until the best solutions are found. Chromosomes consist of genes, and they oftentimes do not need to be the same length depending on what they represent.

1.4.2. Selection

Selection is the process of choosing the next generation of chromosomes at each step. It consists of crossover, mutation, fitness computations, and elitism.

Crossover and Mutation Crossover represents the mating process. Pairs of parent chromosomes are crossed over according to the chosen crossover procedure. There may be many crossover points between pairs, and crossover may occur with a specific probability rather than for every pair. Crossover creates a more diverse generation, where positive aspects of the parent generation are carried to the child generation. Similarly, mutation prevents the algorithm from getting stuck at local optima by introducing new genes into the gene pool. Mutation can occur with a specific probability or adaptive probability, although very few genes are mutated in each generation.

Fitness Computation A fitness computation is necessary to find the "fittest" for the evolutionary concept of survival of the fittest. The fitness computation is the user's objective, and it is calculated for each chromosome in a given generation. There may be multiple objectives for one optimization problem. The next generation

is often picked from a ranking of chromosomes based on the fitness computation, sometimes before and sometimes after crossover and mutation.

Elitism Elitism is the process of carrying over the best chromosomes or genes into the next generation. Since typical implementations of crossover and mutation may transform some of the best solutions into poor solutions, transferring the best fitting solutions into the next generation without alteration will guarantee that high quality genes are still represented in the new generation.

1.5. Cluster Evaluation

When we begin using very high dimensional data, it becomes difficult to evaluate our cluster fit visually. In order to evaluate how accurate our clusters are, there are multiple statistics we can calculate. There are both external and internal indices. An external index measures the agreement between our clustering results and prior knowledge we have about the clustering structure of our data. If we do not have prior information, we must use an internal index, which evaluates the clusters based on quantities and features inherent in the data set [2]. Most of the internal methods aim to evaluate the compactness – how closely related the objects within a cluster are – and separation – how distinct a cluster is from other clusters – of a solution.

2. Methods

2.1. Data

The data to be clustered was collected from the American Community Survey (ACS) from the Census Bureau [13]. The ACS collects vital information on the people of the United States each year that helps guide policies and plan for the future. The data was collected through the Python package `census`, which is a wrapper for the Census API. It consisted of Health Insurance Coverage statistics by type for each county (or equivalent) in the United States in 2019. Data was also collected for various environmental air pollution values from the CDC National Environmental Public Health Tracking Network [7]. The health insurance data will be used to create clusters, while the environmental data will be used to analyze the resulting clusters. All the collected variables are displayed in Table 1 with descriptions. After removing observations with invalid Median Income data points and including only counties with all data points available, there were 2768 United States counties represented.

county_FIPS	Unique ID code for county
Population	Total population of county
Median_Income	Median income of county
Per_VA	Percentage of population with VA Health Care only
Per_Emp	Percentage of population with employer-based health insurance only
Per_Medicare	Percentage of population with Medicare coverage only
Per_Caid	Percentage of population with Medicaid/means-tested public coverage only
Per_Private	Percentage of population with direct-purchase (private) health insurance only
Per_Two	Percentage of population with two or more types of health insurance coverage
Per_None	Percentage of population with no health insurance coverage
pm	Average concentration of PM 2.5 ($\frac{\mu g}{m^3}$) in 2016
benz	Average air concentration of Benzene ($\frac{\mu g}{m^3}$) in 2011
ace	Average air concentration of Acetaldehyde ($\frac{\mu g}{m^3}$) in 2011
form	Average air concentration of Formaldehyde ($\frac{\mu g}{m^3}$) in 2011

Table 1: Data variables to be used for cluster analysis

2.2. Clustering Tendency

Clustering tendency is an important quality of data when performing cluster analysis. Not all data will group together into clusters, and it may be too uniformly distributed to make significant groups to analyze [1]. Clusters can be analyzed visually in very low dimensions, but in higher dimensions this is not an option. One common way to measure clustering tendency in this case is the Hopkins' Statistic. The Hopkins' statistic compares the distances between points sampled from our data and their nearest neighbors with the distances between uniformly sampled points and their nearest neighbors [1]. The Hopkins' value can be caluclated as in [10]:

$$H = \frac{\sum U_i}{\sum U_i + \sum W_i} \tag{1}$$

where U is the distance from a real point to its nearest neighbor and W is the distance from a randomly chosen point within the data space to the nearest real data point. A small percentage between 5% and 10% of the real data points are sampled for this calculation, which encourages the nearest-neighbor distances to be independent and therefore approximate a Beta distribution [10].

If there is not clustering tendency, then the distances for the sampled data and randomly generated data would be similar on average [1], resulting in a value near 0.5. Highly clustered data will have a Hopkins' value

near one. In some literature and coding implementations, the Hopkins' Statistic is represented as the inverse. For this project, the Hopkins' Statistic calculation was implemented using the `clustertend` Python package, which indeed implements the inverse Hopkins' Statistic. Thus, we are aiming for a value near zero to indicate high clustering tendency.

2.3. K-Means

2.3.1. Traditional Algorithm

The k-Means algorithm returns a set of cluster centers, as well as a label for the cluster that each observation is assigned to. It is a greedy algorithm that is guaranteed to converge to a local minimum [2]. It aims to choose k centers, C, to minimize the chosen objective function, which is Sum of Squared Errors (SSE),

$$SSE(C) = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - c_k\| \tag{2}$$

where c_k is the centroid of cluster C_k, and x_i is a data observation assigned to cluster C_k [2]. The Euclidean distance is used in the traditional k-Means algorithm, although other proximity measures such as the Manhattan distance and Cosine similarity can be used when finding the closest cluster centers [2]. Algorithm 1 demonstrates the k-Means process, as described in [3].

Algorithm 1: k-Means

Arbitrarily choose a set of k initial centers $C = \{c_1, c_2, ...c_k\}$;
while C *continues to change* **do**
  foreach $i \in \{1, 2, ...k\}$ **do**
    set the cluster C_i to be the set of points in X that are closer to c_i than they are to c_j for all $j \neq i$;
  end
  foreach $i \in \{1, 2, ...k\}$ **do**
    set c_i to be the center of mass of all points in $C_i : c_i = \frac{1}{C_i} \sum_{x \in C_i} x$;
  end
end

2.3.2. k-Means++ Seeding

The k-Means algorithm is highly suseptible to the initial cluster centers that are selected. Thus, an improved method for initialization was proposed in [3], and has become a standard for the k-Means algorithm. The k-Means++ center selection algorithm still chooses initial centers at random from the data observations, but

weighs the data points according to their Euclidean distance squared from the closest center already chosen. Let $D(x)$ denote the shortest Euclidean distance from a data point x to the closest center already chosen. Then, the k-Means++ algorithm can be described by Algorithm 2.

Algorithm 2: k-Means++

Take one center c_1, chosen uniformly at random from X;

while $length(C) < k$ **do**

  Take a new center c_i, choosing $x \in X$ with probability $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$;

end

Proceed with standard k-Means algorithm;

2.3.3. Implementation

The k-Means algorithm with k-Means++ seeding was implemented using the `KMeans` module from the `sklearn.cluster` Python package. The algorithm is implemented for a range of k values to evaluate changes in evaluation indices.

2.4. Gaussian Mixture Model

2.4.1. Gaussian Distribution

The Gaussian Distribution is also referred to as the normal distribution. It has many computational qualities that make it effective for use in machine learning. The density of a univariate random variable, x, in a Gaussian distribution is given by

$$p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{3}$$

where σ^2 is the variance of the distribution and μ is the mean of the distribution [4]. This paper will deal more with multivariate Gaussian distributions, which can be written as

$$p(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-\frac{D}{2}} |\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}-\boldsymbol{\mu})\right) \tag{4}$$

where D is the dimension of $\boldsymbol{x}$, $\boldsymbol{x} \in \mathbb{R}^D$, $\boldsymbol{\mu}$ is a mean vector, and $\boldsymbol{\Sigma}$ is a covariance matrix [4]. The covariance measures how the mean values of two variables move together. If they increase and decrease together, they have a positive covariance; if they increase when the other decreases, they have a negative covariance. The covariance matrix contains all the pairwise covariance calculations. We can also represent the normal distribution as $p(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ [4]. Often we use the Gaussian distribution by applying transformations to the mean and covariance of the random variable [4].

2.4.2. Mixture Models

Mixture models allow us to more accurately express data than a simple distribution [4]. They can describe datasets that have multiple clusters of data [4]. We will focus on Gaussian Mixture Models (GMMs), which are combinations of Gaussian distributions. A GMM with K Gaussian distributions can mathematically be defined as

$$p(\boldsymbol{x}|\boldsymbol{\theta}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

$$0 \leq \pi_k \leq 1, \qquad \sum_{k=1}^{K} \pi_k = 1 \tag{5}$$

where $\boldsymbol{\theta} = \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k : k = 1, ..., K$ is the collection of model parameters, π_k are the mixture weights, and other variables are as defined above [4].

2.4.3. Expectation-Maximization (EM) Algorithm

The Expectation-Maximization (EM) algorithm is an iterative method for learning the parameters in mixture models [4]. The algorithm consists of two steps – the *E-step* and *M-step* – described as in [4]

- *E-step:* Evaluate the responsibilities, r_{nk}, using the current parameters
- *M-step:* Re-estimate the parameters $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k$ using the updated responsibilities

This process increases likelihood – the probability that we would observe the value, given our distribution parameters – each step. Machine learning oftentimes uses the log-likelihood instead of the likelihood due to computational feasibility of optimizing sums rather than products. The EM algorithm converges to the minimum of the negative log-likelihood, which is the optimum [4].

2.4.4. Implementation

The GMM was implemented using the `GaussianMixture` module from the `sklearn.mixture` Python package. The algorithm is implemented for a range of k values to evaluate changes in evaluation indices.

2.5. Genetic Algorithm for Clustering

This section describes the specific steps chosen for this paper's Genetic Algorithm.

2.5.1. Chromosome Representation

With clustering, there are two main choices in representations of clusters in chromosomes. Both choices allow us to represent solutions with various amounts of clusters, which overcomes a downfall of the other two methods that require a priori knowledge of the number of clusters. The first option is to consider each gene within the chromosome as an observation. Then, the value put into that gene slot is the number of the cluster to which that observation belongs. The second way, and the way that is implemented in this algorithm, encodes the cluster centers in the chromosomes. Since the dimension of the data is known, the number of clusters in our solution will be the total length divided by the data's dimension. Each cluster center is grouped together, and they are aligned as shown in Figure 1.

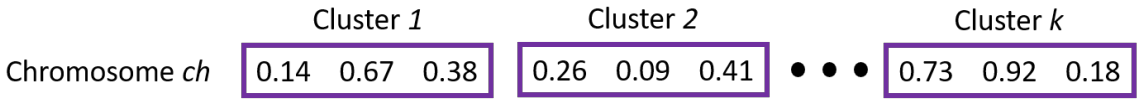

Figure 1: Chromosome representation for proposed Genetic Algorithm

2.5.2. Initializing Clusters

The proposed algorithm uses a density-based approach to initializing chromosomes and clusters, which is better at finding abnormally shaped clusters than traditional methods [15]. First, the normalized dataset V is found with the equation $V = \frac{V - V_{min}}{V_{max} - V_{min}}$ for each column or variable in the dataset [15]. Then the algorithm finds the pairwise differences between all the points in the normalized data vector V. The algorithm creates a list r that is the same length as the number of chromosomes for the implementation. The list r represents a list of radii linearly spaced between two values a and b. This algorithm used $a = 0.04$ and $b = 0.15$, although these values will need to be adjusted for each new dataset. Like in the SeedClust algorithm presented in [15], for each radius r_x, the algorithm counts the number of points within the radius for each point in the dataset. Observations with more points in their proximity are assumed to be good cluster centers since they are most likely within a cluster. Then, all points with the maximum number of points within the radius, and all points with one fewer than that maximum, are added as centers for chromosome x. The threshold for the number of points within the radius decreases until at least five cluster centers are assigned to the chromosome. After repeating this process for each radius, there are x chromosomes with at least five cluster centers each. This process is reiterated in Algorithm 3.

Once the chromosomes are created, all data observations need to be assigned to a cluster prior to fitness

calcualtions. This algorithm guarantees that each cluster center is assigned at least three points so that we do not prematurely converge to a small amount of clusters. Within each chromosome, each cluster center is first assigned the ten closest data points to itself, even if these points may be closer to a different center. Then, the remaining observations are assigned to the cluster closest to them by Euclidean distance. Algorithm 3 also describes this process.

Algorithm 3: Chromosome Initialization and Assignment

V = normalized dataset;
pairs = pairwise distances of observations in V;
foreach $x \in range(popsize)$ **do**
    density = the count of points within $r[x]$ distance of each point;
    maxdens = the highest count of points within the radius for all observations;
    Add the points with *maxdens* or $maxdens - 1$ as centers in chromosome x;
    while *chromosome x has fewer than 5 centers* **do**
        Add points with a lower density threshold (ex: $maxdens - 2$, $maxdens - 3$, ...);
    end
    Add finished chromosome x to overall set of chromosomes;
end
foreach $x \in range(popsize)$ **do**
    C_x = all cluster centers of chromosome x;
    Find pairwise distances between V and C_x;
    foreach *center* cen_i *in* C_x **do**
        assign the 3 closest data observations to cen_i;
    end
    foreach *observation not yet assigned* **do**
        assign observation to closest cluster center;
    end
end

2.5.3. Fitness Computation

This Genetic Algorithm uses Sum Squared Errors (SSE) from Equation 2 as its objective function. A small value from SSE represents observations that are in close proximity to their respective cluster centers. A value of zero would represent each point being assigned to its own individual cluster. Thus, using this objective

function would encourage our algorithm to create many cluster centers. Since we are searching for interpretable clusters, the algorithm adds a penalty for a large number of clusters. Since the Genetic Algorithm works to maximize the fitness objective, we define the fitness function as

$$f = \frac{1}{SSE + \left(\frac{\text{number of clusters in chromosome}}{2}\right)^2} \tag{6}$$

This value is calculated for each proposed chromosome.

2.5.4. Selection

The selection process for this algorithm consists of the crossover and mutation procedures. Since these procedures keep a constant number of chromosomes, there is not a thorough selection process. Rather than selection being performed prior to crossover and mutation, the selection framework helps both of these processes throughout.

Crossover Crossover switches information from two parent chromosomes to produce two new children chromosomes. Since there can be a different number of cluster centers in each chromosome, the crossover process will change the length of the chromosomes for the next generation, also called the 'offspring.' This process allows the algorithm to explore a different number of clusters, which is a different subset of the solution space. This is important since we are not specifying the optimal number of clusters and the algorithm must make this discovery on its own. This algorithm uses single point crossover, and the crossover point is placed between cluster centers so that they are not separated. The crossover index for each chromosome is randomly chosen such that there will never be a single cluster center remaining after the crossover occurs. For a given pair of chromosomes, this algorithm will detach and swap the sections of the chromosomes after the calculated crossover index. A diagram of this process is displayed in Figure 2.

This algorithm uses an adaptive crossover probability when determining which pairs of chromosomes will undergo the crossover procedure. Given the fitnesses of each chromosome from the previous step, the algorithm chooses pairs of chromosomes based on the roulette wheel technique. Roulette wheel selection simulates a 'spinning wheel' where each slot's size is proportional to the corresponding chromosome's fitness [5]. Then a random value is uniformly selected, and wherever it lands on the wheel is the chromosome that is selected. Pairs continue to be selected in this way until the original size of the generation is reached. For each pair, the probability of crossover is calculated as in [15]:

$$P_{cross} = \begin{cases} \frac{f_{max} - f'}{f_{max} - f_{avg}} & \text{if} f' > f_{avg} \\ 1 & \text{if} f' \leq f_{avg} \end{cases} \tag{7}$$

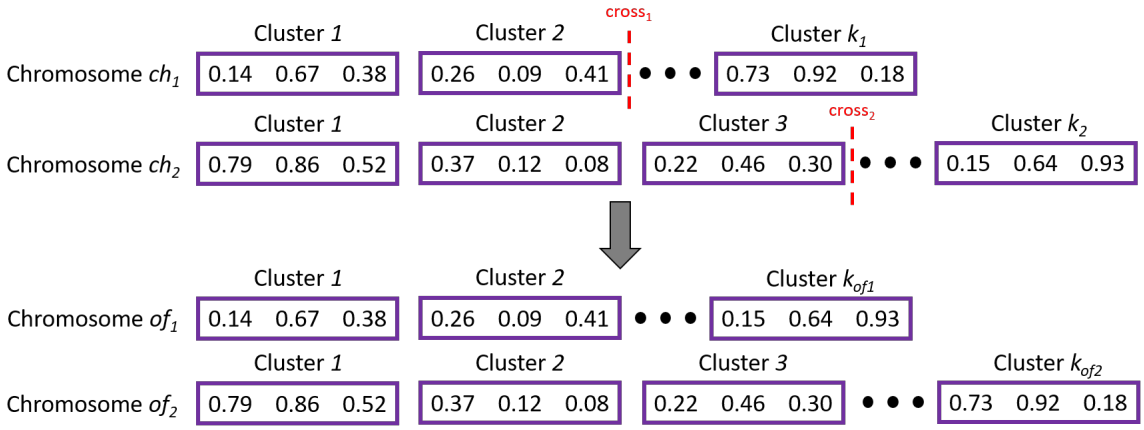

Figure 2: Crossover procedure for proposed Genetic Algorithm

where f_{min}, f_{max}, and f_{avg} are the minimum, maximum, and average fitness values of the current population, respectively. Then, the algorithm chooses a uniformly random value between zero and one. If the probability of crossover is greater than this randomly generated value, then the crossover procedure will proceed. Otherwise, the two chromosomes are directly added to the offspring.

Mutation Mutation adds variability to the population, which helps the algorithm explore more of the solution space. Mutation should be relatively rare so that the good solutions are not ruined by mutating them into suboptimal solutions. The mutation procedure begins by calculating the probability of mutation for each chromosome, via the equation presented in [15]:

$$P_{mutation} = \begin{cases} 0.5 \times \frac{f_{max} - f_i}{f_{max} - f_{avg}} & \text{if} f > f_{avg} \\ 0.5 & \text{if} f \leq f_{avg} \end{cases} \tag{8}$$

Then, for each chromosome, a random value δ is uniformly chosen between zero and one. If this value is below the mutation probability for that chromosome, then one of that chromosome's cluster centers is uniformly randomly chosen for mutation. The algorithm adds a scalar equivalent to δ multiplied by the difference between the maximum and minimum values in the chosen cluster. This part of the mutation procedure is demonstrated in Figure 3. If the mutation probability is below δ, then that chromosome is copied over into the next generation.

Elitism This Genetic Algorithm does not include a formalized elitism process. Instead, the crossover and mutation algorithms with adaptive probabilities help to gain the same effect as elitism. Since the crossover process is more likely to choose chromosomes with high fitness, and chromosomes can be chosen multiple

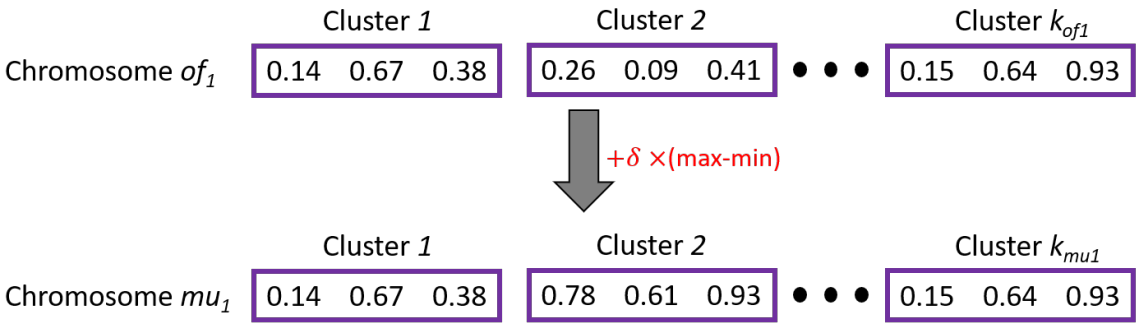

Figure 3: Mutation procedure for proposed Genetic Algorithm

times for crossover, the higher quality cluster centers are more likely to appear in each new generation. Also, if the fitness of a chromosome is above average, then it is less likely to be selected for mutation. Thus the quality genes will be carried into the next generation in this way as well.

2.5.5. Termination

This algorithm executes for a fixed number of generations, which can be specified by the user. A higher number of generations implies a more accurate clustering, since more of the solution space has been explored. Once the termination generation is acquired, the algorithm returns the last generation for future analysis. Either the chromosome with the best fitness value or the chromosome within a certain range of k with the best fitness value should be chosen as the best chromosome for the solution to the clustering problem.

Algorithm 4: Selection: Crossover and Mutation

G = number of generations;

foreach $g \in 1, ..., G$ **do**

  Calculate the fitness of each chromosome;

  Calculate roulette wheel probabilities for each chromosome;

  foreach *x in range(popsize/2)* **do**

    c_1, c_2 = randomly choose two chromosomes based on roulette wheel;

    P_{cross} = crossover probability for c_1 and c_2;

    check = random value between 0 and 1;

    if $check \leq P_{cross}$ **then**

      Find random crossover points for both chromosomes;

      Swap ends of chromosomes;

      Add new chromosomes to POP_c;

    else

      Add original chromosomes to POP_c;

    end

  end

  Redo assignment process for POP_c;

  Calculate fitnesses for all chromosomes in POP_c;

  Calculate the mutation probabilites, $P_{mutation}$ for each chromosome;

  foreach $x \in range(popsize)$ **do**

    check2 = random value between 0 and 1;

    if $check \leq P_{mutation}$ **then**

      Select random cluster within chromosome x;

      Add $check * (maxofcluster - minofcluster)$ to selected cluster;

      Add mutated chromosome to POP_m;

    else

      Add unchanged chromosome to POP_m;

    end

  end

  Recalculate assignments for POP_m;

end

2.6. Cluster Evaluation

Since the cluster structure of this paper's project data is unknown, we will focus on different types of internal indices rather than external indices. I will compare the results of K-Means, GMMs, and GAs using three cluster evaluation methods: Sum of Squared Errors, Davies-Bouldin Index, and Silouette Index. In addition, the results from the GA will be used as initial clusters to k-Means in place of the k-Means++ seeding method, as done in [15].

Davies-Bouldin Index Like most cluster evaluation methods, the Davies-Bouldin Index tries to minimize the scatter within clusters and maximize the distance between clusters. The process for calculating the Davies-Bouldin Index is described in [11]. First, calculate the scatter for each cluster, which is defined as

$$S_i = \left(\frac{1}{T_i} \sum_{j=1}^{T_i} |X_j - A_i|^p \right)^{\frac{1}{p}} \tag{9}$$

where X_j is an observation in a certain cluster, A_i is the centroid of that cluster, and T_i is the number of observations in that cluster. The value p depends on the distance measure being used in the clustering method it is evaluating. Then, the separation between clusters i and j is calculated with

$$M_{i,j} = ||A_i - A_j||_p = \left(\sum_{k=1}^{n} |a_{k,i} - a_{k,j}|^p \right)^{\frac{1}{p}} \tag{10}$$

where $a_{k,i}$ and $a_{k,j}$ are the components of each centroid, since the centroids are n-dimensional. Then for clusters i and j, we calculate

$$R_{i,j} = \frac{S_i + S_j}{M_{i,j}} \tag{11}$$

which, when maximized, satisfies certain properties that make the scatter small and separation large. We identify that largest value of $R_{i,j}$ and assign it to D_i. Finally, we calculate the Davies-Bouldin Index, which is the average of each D_i:

$$DB = \frac{1}{N} \sum_{i=1}^{N} D_i \tag{12}$$

where N is the number of clusters. A Davies-Bouldin Index near zero represents well-defined clusters.

Silhouette Index Similarly to the Davies-Bouldin Index, the Silhouette Index is based on the pairwise difference of between- and within-cluster distances [2]. The procedure for finding the Silhouette Index is described in [14]. To find the Silhouette Index, first we calculate the cohesion, $a(x)$, and the separation, $b(x)$. The cohesion is the average distance of an observation to all other observations in the same cluster. The

separation is the average distance of an observation from all the observations in the other clusters. Then we calculate the silhouette, which is

$$s(x) = \frac{b(x) - a(x)}{max\{a(x), b(x)\}} \tag{13}$$

Each silhouette will be in the range $[-1, 1]$ where -1 is a poor fit and 1 is a good fit. Finally we calculate the Silhouette Coefficient, which is

$$SC = \frac{1}{N} \sum_{i=1}^{N} s(x) \tag{14}$$

A larger average Silhouette Index indicates a better overall quality of the clustering. To simplify the selection process, the Silhouette Index will be negated so that both objectives will be minimized.

2.7. Data Analysis of Clusters

Once the clustering process is complete, the environmental variables described in Section 2.1 will be used to analyze the different qualities of the clusters. A one-way analysis of variance (ANOVA) will be conducted to test whether the means of the different envrionmental factors are significantly different. If so, this indicates that an underlying pattern was found to group the counties together. Although these variables are not completely Gaussian, the one-way ANOVA is robust to the normality assumption [6]. The ANOVA will be conducted individually for each of the four pollution variables. Let α_i be the difference in means between two clusters for a specific pollutant. Then null hypothesis is $H_0 : \alpha_i = 0 \forall i$, or that all pairs of differences in means are zero. The alternative hypothesis is H_a : at least one α_i is non zero [6].

3. Results

3.1. Clustering Tendency

The Hopkins' statistic for the health insurance data, with a sample size of 250, was 0.1263. As this is relatively close to zero, this data has significant clustering tendency that can be analyzed.

3.2. K-Means and Gaussian Mixture Model Results

The K-Means and Gaussian Mixture Models were conducted on the normalized data for a large range of number of clusters, from two to 50. These results are summarized in Figure 4, Figure 5, and Figure 6. The sum of squared error for k-Means is shown in Figure 4, and it is steadily decreasing as the number of clusters increases. The 'elbow,' or where the decrease seems to slow the most, is around ten clusters. Often the 'elbow' is chosen as the appropriate number of clusters for the data.

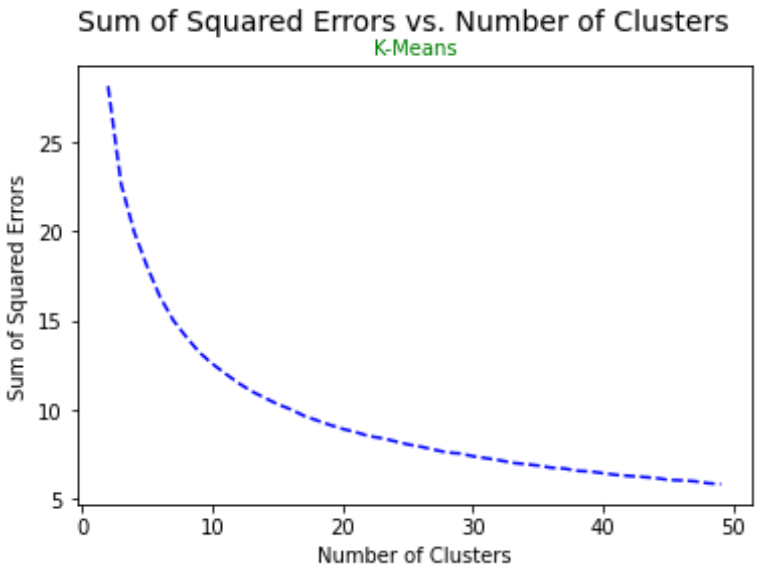

Figure 4: Sum of Squared Errors for K-Means with different numbers of clusters. The number of clusters ranges from 2 to 50.

In Figure 5, the Davies-Bouldin Index remains fairly consistent with some oscillations for both k-Means and GMMs as the number of clusters increases. A lower Davies-Bouldin Index indicates more compact clusters. The Gaussian Mixture Model has a consistently worse Davies-Bouldin Index than the k-Means model. For k-Means, there is a small peak near $k = 5$, which suggests this is a suboptimal number of clusters for this data. The GMM hovers around an index of 2.7, while the k-Means model holds steady near 1.3. Both of these trends suggest that a k value near ten would be a wise number of clusters to choose for these models, since it creates well-defined clusters without creating uninterpretable clusters.

Similar trends are found in Figure 6. The Silhouette Index is plotted against the number of clusters, and a maximal Silhouette Index is optimal. For small values of k, the Silhouette Index decreases as the number of clusters increases. Then, the index for both the K-Means and Gaussian Mixture Models slows its decrease and begins to level out near 10 clusters. Thus, a smaller number of clusters is preferable when considering this index for this data. The Silhouette Index levels out near 0.2 for k-Means and -0.05 for the GMM. Like for the Davies-Bouldin Index, the Silhouette Index for Gaussian Mixture Models is consistently worse than that for k-Means.

By evaluating the sum of squared errors, Davies-Bouldin Index, and Silhouette Index, it appears that the ideal number of clusters for this health insurance data is near ten.

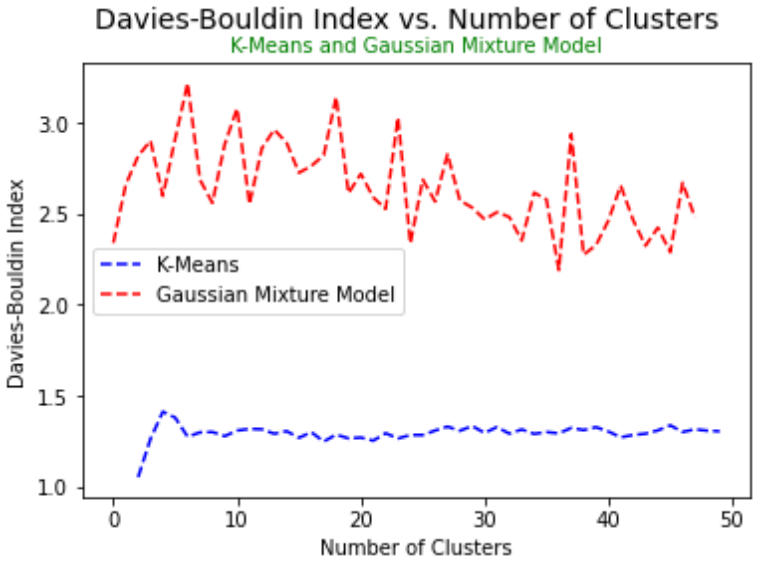

Figure 5: Davies-Bouldin Index for K-Means and Gaussian Mixture Models with different numbers of clusters. The number of clusters ranges from 2 to 50.

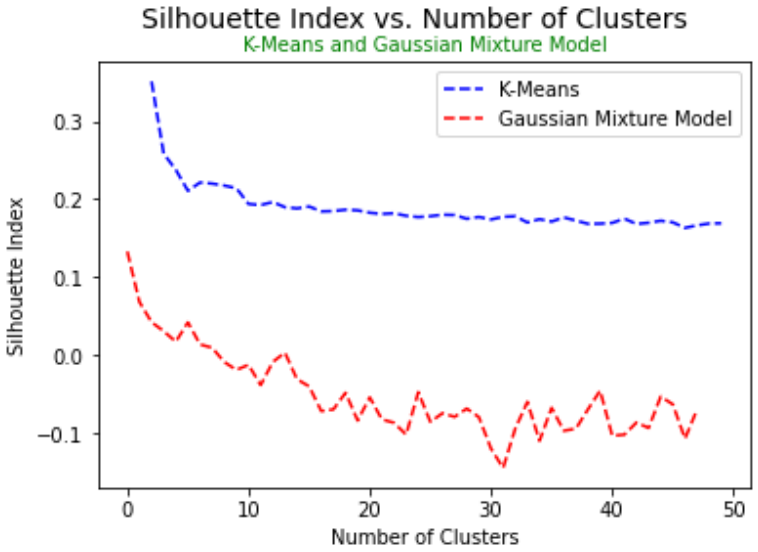

Figure 6: Silhouette Index for K-Means and Gaussian Mixture Models with different numbers of clusters. The number of clusters ranges from 2 to 50.

3.3. Genetic Algorithm Results

The first generation for the Genetic Algorithm consisted of 70 chromosomes. Figure 7 shows the Davies-Bouldin Index and Silhouette Index for the chromosomes in the first, middle, and final generations. The chromosomes for the first generation are shown in a group in the bottom left of the graph, which is actually where the objective functions are the best. This indicates that our chromosome initialization process is very precise. The chromosomes in the middle generation are spread throughout the plot, indicating that the algorithm was thoroughly exploring the solution space, albeit coming across many suboptimal solutions. The chromosomes for the final generation are mixed between the beginning and middle generations, indicating that the algorithm was beginning to converge towards a more optimal solution after exploring the solution space. With the final generation of the GA, the Davies-Bouldin Index and Silhouette Index were slightly worse than the indices from k-Means.

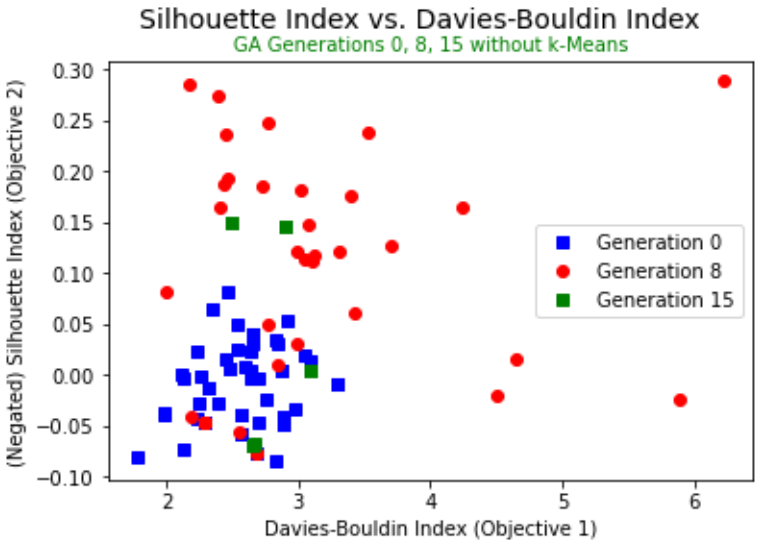

Figure 7: Change in Davies-Bouldin and Silhouette Index values for Genetic Algorithm for generation 0, 8, and 15.

The results from the GA were also used as initial clusters for the k-Means algorithm, as opposed to using the k-Means++ seeding method. These results are plotted in Figure 8. The Silhouette Index for k-Means with GA inputs is slightly better than the index for the k-Means++ inputs. We can also compare the sum of squared errors for these generations after processing through the k-Means model. Figure 9 shows overlapping histograms of sum of squared errors (SSE) after generations zero, eight, and 15 were used as initial centers for the k-Means models. Generation zero had the highest SSE, generation eight had the lowest SSE, and generation 15 converged on an SSE between generations zero and eight. It is likely that generation eight

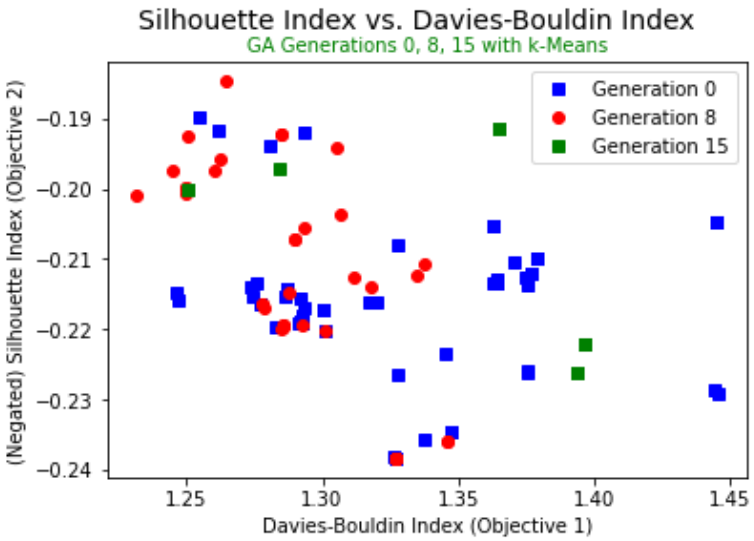

Figure 8: Change in Davies-Bouldin and Silhouette Index for k-Means using Genetic Algorithm chromosomes as initial centers.

consists of chromosomes with more clusters, which lowers the SSE, yet added a penalty to our fitness value.

3.4. Data Analysis of Clusters

A violin plot for each cluster for each of the environmental variables is displayed in Figure 10. There are noticable differences in the means across groups. Four one-way ANOVAs were conducted for each of the environmental factors. Each of the tests had a p-value $< 2e-16$. Thus, we reject the null hypothesis for each of these tests and determine that there is a statistically significant difference in means across clusters.

4. Discussion

The application of these algorithms to the health insurance data, paired with the analysis of the environmental qualities of the clusters, indicates that counties can be divided into coverage groups that have environmental health implications. A further development of this application could analyze which qualities in the clusters were the strongest grouping factors. It could also make connections between environmental hazards based on health insurance access.

The implemented Genetic Algorithm has many parameter values that can be adjusted to produce better values for our cluster evaluation indices. For example, the initial clustering densities, number of points assigned to each cluster, and number of chromosomes in each generation all have large impacts on the performance of the algorithm and can be adjusted. However, this algorithm also requires an immense amount of computing

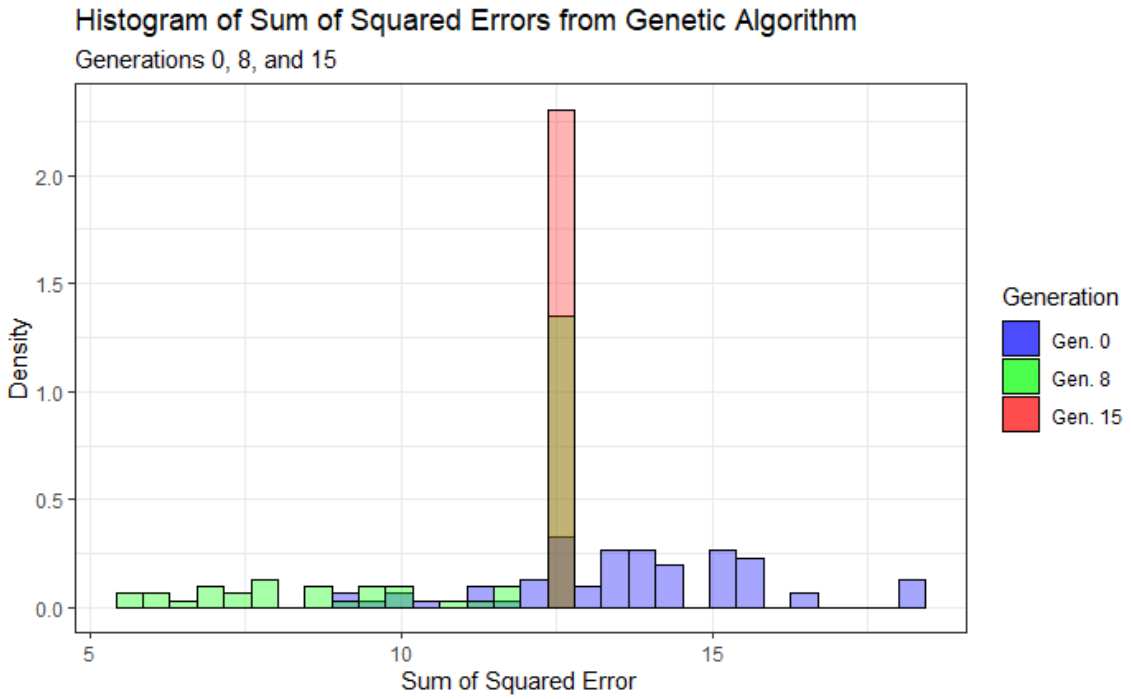

Figure 9: Histograms of sum of squared errors after generations 0, 8, and 15 were used as initial centers for the k-Means models.

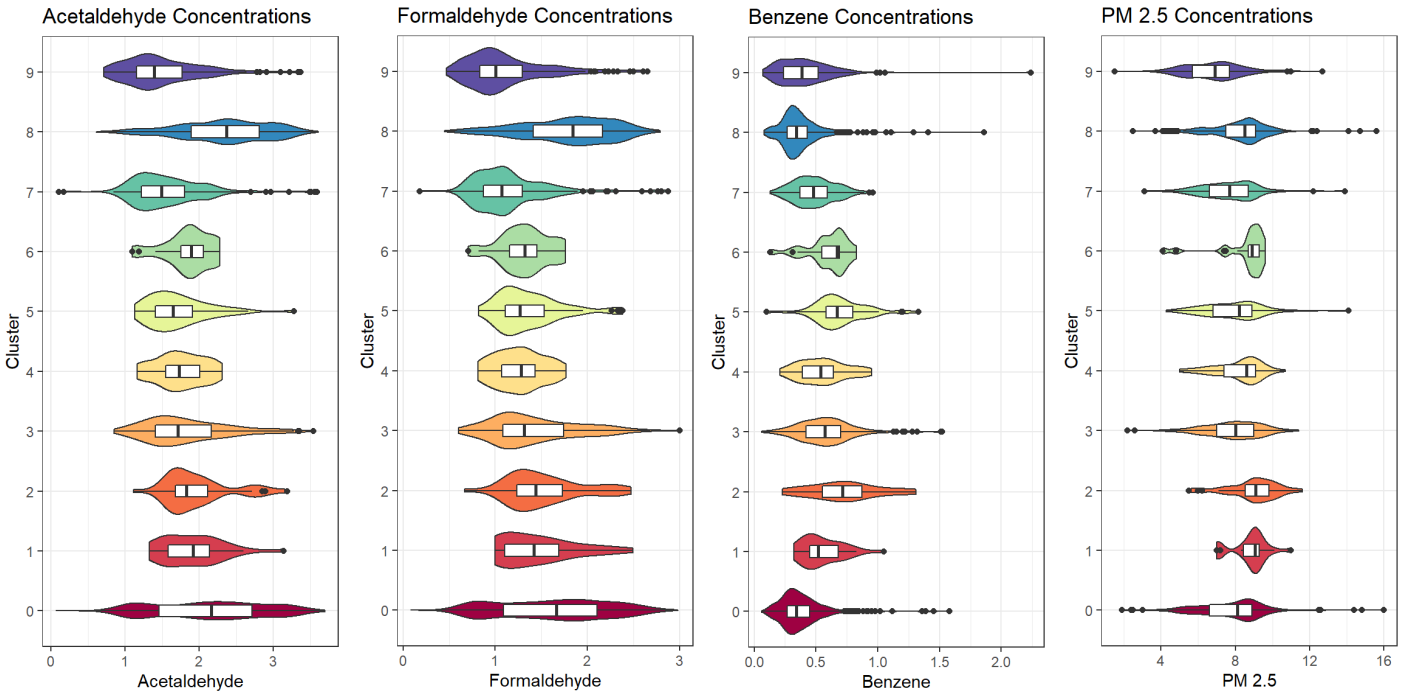

Figure 10: Violin plots for each cluster for each environmental variable.

power and time. This algorithm uses parameter values based on research and experimentation to estimate which values may produce the desired results. It likely would be greatly improved by doing an iterative search on each parameter to see which values perform the best.

Specifically, the penalty added to the fitness function has a large influence on the number of clusters in each generation. With the number of clusters being squared, rather than multiplied by a scalar, there is a very strong push towards the minimum number of clusters. This decision is justified by the point that we want only a few clusters for post-analysis purposes, rather than trying to find a true underlying structure with an optimal number of clusters. Also, our analysis of the clustering indices for k-Means and GMMs indicated that an optimal value of clusters will be near ten. Limiting the number of clusters is forcing the algorithm to find connections and patterns between points that it would otherwise not find if it was able to create as many clusters as it wanted.

Additionally, since this data does not have a true, defined cluster structure, any inferences from these clusters must be made with caution. There is no way to truly evaluate the quality of these clusters, because in theory they do not exist. However, we are still able to learn from them. By seeing which observations were grouped together, we can learn about new connections that were not clear before.

As previously stated, there are thousands of possible variations on Genetic Algorithms that have already been discovered, with many more techniques developing. Many of those methods are multiobjective, so they have more than one fitness function. They attempt to optimize both functions while exploring a diverse range of the solution space, then choose the importance of the fitness functions depending on the application problem once the final generation of chromosomes is acquired. Multiobjective GAs have an optimal Pareto front, which is when all objectives are optimized to the point where improving one would hinder another one. The chromosomes on the Pareto front are called non-dominated solutions. During the selection process, the GA must order the chromosomes, yet there is not a clear ordering with multiple objectives. Thus, Pareto fronts are identified and removed from the dataset until the goal population size is reached. A Pareto front consists of all non-dominated solutions. A non-dominated solution is a solution that does not have another solution that is better in both objectives. There are five Pareto fronts visualized in Figure 11. Each solution in each Pareto front is a non-dominated solution when disregarding solutions from previous Pareto fronts. It is important that the fitness objectives vary enough so that optimizing one does not inherently optimize the other. For example, the Davies-Bouldin Index and Silhouette Index both try to minimize the intra cluster distance and maximize the inter cluster distance. When these objectives are used, the Genetic Algorithm is more likely to converge prematurely to a suboptimal solution, as displayed in Figure 12.

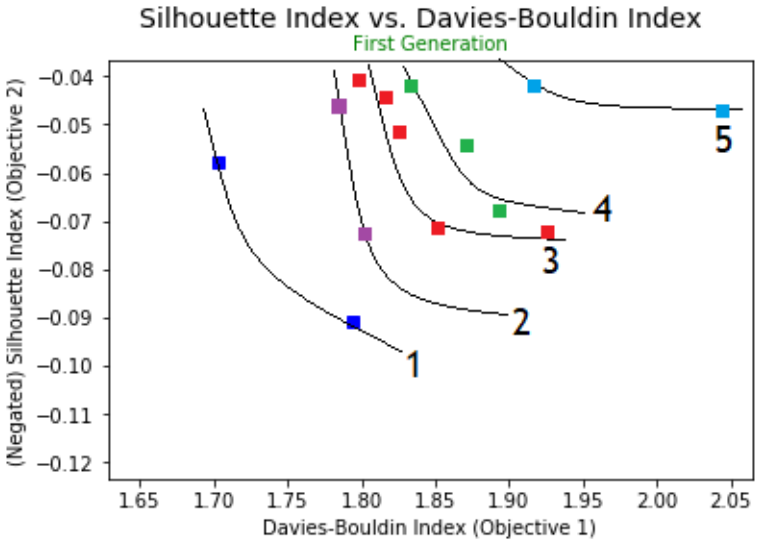

Figure 11: Silhouette Index versus Davies-Bouldin Index for fifteen points sampled from the first generation of data. Displays the five Pareto fronts in this sample.

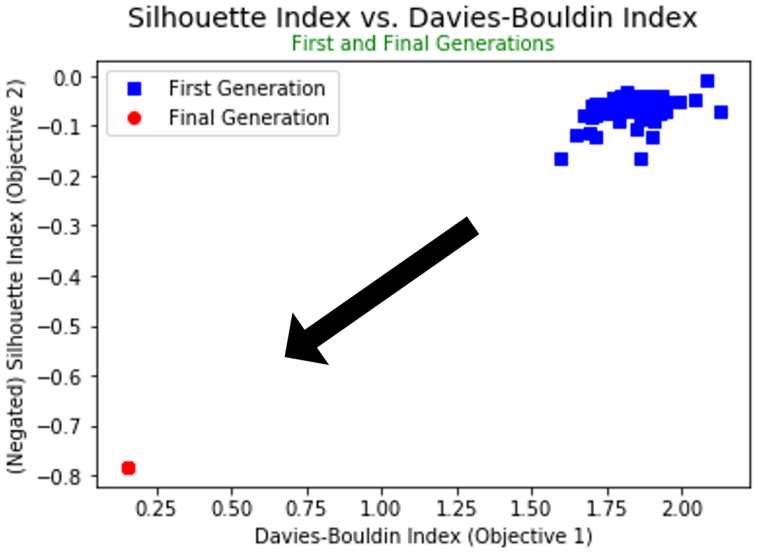

Figure 12: Multiobjective Genetic Algorithm with Davies-Bouldin Index and Silhouette Index as fitness functions, converging to suboptimal solution.

5. Conclusion

Genetic Algorithms are highly maleable and can be applied to a vast variety of applications. Each application will warrant its own detailed alterations of each step in the algorithm to produce the optimal solution. In the case of health insurance coverage, the algorithm was able to create clusters of counties based on particular similarities, and these clusters had distinct environment pollutant values.

Appendices

A. Code - K-Means

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.cluster import KMeans

data = pd.read_csv((r'data\hc_environ_stats.csv'),index_col=0)
data = data.drop(["county_FIPS","Median_Income","Population","pm","benz","form","ace"], axis=1)
data = data.reset_index(drop=True, level=0)

k_data = data.copy()
dbi = []
sil = []
inertia = []
for i in range(2,50):
    kmeans = KMeans(n_clusters = i)
    kmeans.fit(k_data)

    pred = kmeans.labels_

   inertia.append(kmeans.inertia_)
    dbi.append(metrics.davies_bouldin_score(k_data, pred))
    sil.append(metrics.silhouette_score(k_data, pred, metric='euclidean'))
```

B. Code - Gaussian Mixture Models

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.mixture import GaussianMixture

data = pd.read_csv((r'data\hc_environ_stats.csv'),index_col=0)
data = data.drop(["county_FIPS","Median_Income","Population","pm","benz","form","ace"], axis=1)
data = data.reset_index(drop=True, level=0)

gmm_data = data.copy()
dbi = []
sil = []
for i in range(2,50):
    gmm = GaussianMixture(n_components = i)
    kmeans.fit(k_data)

    pred = gmm.predict(gmm_data)

    dbi.append(metrics.davies_bouldin_score(gmm_data, pred))
    sil.append(metrics.silhouette_score(gmm_data, pred, metric='euclidean'))
```

C. Code - Genetic Algorithm

```
import pandas as pd
from sklearn import metrics
import numpy as np
import random



def setup(pop_size, r):
    X = pd.read_csv((r'data\hc_environ_stats.csv'),index_col=0)
    X = X.drop(["county_FIPS","Median_Income","Population","pm","benz","form","ace"], axis=1)
    X = X.reset_index(drop=True, level=0)

    [n,m] = X.shape
    V = X.apply(lambda v: (v-v.min())/(v.max()-v.min()), axis=0)
    return [X, V, n, m, r]

def init_chroms(pop_size, r, V):
    pairs = metrics.pairwise_distances(V)
    pairs = pd.DataFrame(pairs)
    all_chroms = pd.DataFrame()
    for x in range(pop_size):
        r_x = r[x]
        dens = pairs[pairs <= r_x].count()
        max_value = dens.max()
        chrom = pairs.loc[(dens <= max_value) & (dens > max_value-1),:]
        gap = 2
        while chrom.shape[0] < 5:
            chrom = pairs.loc[(dens <= max_value) & (dens > max_value-gap),:]
            gap = gap + 1
        these_chroms = V.loc[V.index & chrom.index]
        these_chroms = these_chroms.assign(Chrom_ID=x)
        these_chroms = these_chroms.assign(Center_ID=range(chrom.shape[0]))
        all_chroms = pd.concat([all_chroms,these_chroms])
    all_chroms = all_chroms.reset_index(drop=True, level=0)
    return all_chroms
```

```
def assignments(pop_size, V, chrom_df):
    all_assignments = pd.DataFrame()
    for ch in range(pop_size):
        row_assign = np.full(V.shape[0], np.nan)
        sub = chrom_df.loc[chrom_df['Chrom_ID'] == ch]
        dist = pd.DataFrame(metrics.pairwise_distances(V, sub.iloc[:,0:-2]))
        for center in range(dist.shape[1]):
            smalls = np.array(dist.nsmallest(3, center, keep='all').index.tolist())
            row_assign[smalls] = center
        for obs in range(len(row_assign)):
            if not np.isnan(row_assign[obs]):
                continue
            closest_center = dist.iloc[obs,:].nsmallest(1).index[0]
            row_assign[obs] = closest_center
        all_assignments = all_assignments.append(pd.Series(row_assign), ignore_index=True)
    return all_assignments

def fitness(pop_size, chrom_df, all_assignments, V):
    fits = np.full(pop_size, np.nan)
    for ch in range(pop_size):
        sub = chrom_df.loc[chrom_df['Chrom_ID'] == ch]
        temp_sum = 0
        for center in range(sub.shape[0]):
            this_cen = sub.loc[sub['Center_ID'] == center].iloc[:,0:-2]
            indices = V.index[(all_assignments.iloc[:,:] == center).loc[ch,:] == True].tolist()
            inClust = V.iloc[list(indices),:]
            diff = inClust.subtract(np.tile(this_cen,(inClust.shape[0],1)), axis=1)
            temp_fit = np.einsum('ij,ij->i', diff, diff)
            temp_sum += temp_fit.sum()
        temp_sum += (sub.shape[0]/2)**2 # penalty for too many clusters
        fits[ch] = 1/temp_sum
    return fits

def selection(fits, pop_size, chrom_df, V):
    pop_c = crossover(fits, pop_size, chrom_df)
```

```
    pop_c = pop_c.reset_index(drop=True, level=0)
    assigns = assignments(pop_size, V, pop_c)
    fits = fitness(pop_size, pop_c, assigns, V)
    pop_m = mutation(fits, pop_size, pop_c)
    return pop_m

def crossover(fits, pop_size, chrom_df):
    rwprobs = [fits[x]/fits.sum() for x in range(len(fits))]
    pop_cross = pd.DataFrame()
    curr_id = 0
    for ch in range(int(pop_size/2)):
        [cr1, cr2] = random.choices(range(pop_size), weights = rwprobs, k = 2)
        fprime = max([fits[cr1],fits[cr2]])
        p_cross = (fits.max() - fprime)/(fits.max()-fits.mean())
        prob_check = random.random()
        centers1 = chrom_df.loc[chrom_df['Chrom_ID'] == cr1]
        centers2 = chrom_df.loc[chrom_df['Chrom_ID'] == cr2]
        if prob_check <= p_cross:
            if centers1.shape[0] == 2 & centers2.shape[0] == 2:
                mid1 = 1
                mid2 = 1
            elif centers1.shape[0] == 2:
                mid1 = 1
                mid2 = random.randint(1,centers2.shape[0]-2)
            elif centers2.shape[0] == 2:
                mid2 = 1
                mid1 = random.randint(1,centers1.shape[0]-2)
            else:
                mid1 = random.randint(1,centers1.shape[0]-2)
                mid2 = random.randint(1,centers2.shape[0]-2)

            off1start = centers1.loc[(centers1['Center_ID'] >= 0) | (centers1['Center_ID'] <
                mid1),:]
            off1end = centers2.loc[centers2['Center_ID'] >= mid2,:]
            off1 = off1start.append(off1end)
            off1 = off1.assign(Chrom_ID=curr_id)
```

```
            off1 = off1.assign(Center_ID=range(off1.shape[0]))
            curr_id += 1

            off2start = centers2.loc[(centers2['Center_ID'] >= 0) | (centers2['Center_ID'] <
                mid2),:]
            off2end = centers1.loc[centers1['Center_ID'] >= mid1,:]
            off2 = off2start.append(off2end)
            off2 = off2.assign(Chrom_ID=curr_id)
            off2 = off2.assign(Center_ID=range(off2.shape[0]))
            curr_id += 1

            pop_cross = pop_cross.append(off1)
            pop_cross = pop_cross.append(off2)
        else:
            centers1 = centers1.assign(Chrom_ID=curr_id)
            centers1 = centers1.assign(Center_ID=range(centers1.shape[0]))
            curr_id += 1
            centers2 = centers2.assign(Chrom_ID=curr_id)
            centers2 = centers2.assign(Center_ID=range(centers2.shape[0]))
            curr_id += 1
            pop_cross = pop_cross.append(centers1)
            pop_cross = pop_cross.append(centers2)
    return pop_cross.sort_values(['Chrom_ID','Center_ID'])

def mutation(fits, pop_size, chrom_df):
    p_mut = np.array([0.5*(fits.max()-fits[x])/(fits.max()-fits.mean()) for x in range(len(fits))])
    p_mut[p_mut > 1] = 0.5
    pop_mut = pd.DataFrame()
    for ch in range(pop_size):
        centers = chrom_df.loc[chrom_df['Chrom_ID'] == ch,:]
        ran = random.random()
        if ran <= p_mut[ch]:
            i = random.randint(0,centers.shape[0]-1)
            [mx, mn] = [centers.loc[centers['Center_ID'] == i].iloc[:,0:-2].max(axis=1),
                centers.loc[centers['Center_ID'] == i].iloc[:,0:-2].min(axis=1)]
            temp_row = centers.loc[centers['Center_ID'] == i,:]
```

```
            temp_vals = temp_row.iloc[:,0:-2]
            row = temp_vals.copy()
            row += ran*(mx.iat[0]-mn.iat[0])
            row = row.assign(Chrom_ID=ch)
            row = row.assign(Center_ID=i)
            ind = centers.index[centers['Center_ID'] == i].tolist()
            centers = centers.copy().drop(index=ind)
            centers = centers.append(row)

            centers = centers.sort_values('Center_ID')
            pop_mut = pop_mut.append(centers)
        else:
            pop_mut = pop_mut.append(centers)
    return pop_mut


def main():
    pop_size = 70
    r = list(np.linspace(start=0.04, stop=0.15, num=pop_size))
    [X, V, n, m, r] = setup(pop_size, r)
    all_chroms = init_chroms(pop_size, r, V)
    G = 30 # number generations
    for g in range(G):
        assigns = assignments(pop_size, V, all_chroms)
        assigns.to_csv(r'data\assignsgen{}.csv'.format(g))
        fits = fitness(pop_size, all_chroms, assigns, V)
        pd.Series(fits).to_csv(r'data\fitsgen{}.csv'.format(g))
        pop_m = selection(fits, pop_size, all_chroms, V)
        all_chroms = pop_m.reset_index(drop=True, level=0)
        all_chroms.to_csv(r'data\chromsgen{}.csv'.format(g))
    final_assigns = assignments(pop_size, V, all_chroms)
    final_fits = fitness(pop_size, all_chroms, final_assigns, V)
    pd.Series(final_fits).to_csv(r'data\fits_finalgen.csv')
    final_assigns.to_csv(r'data\assigns_finalgen.csv')
```

```
if __name__ == "__main__":
    main()
```

References

[1] Andreas Adolfsson, Margareta Ackerman, and Naomi Brownstein. To cluster, or not to cluster: An analysis of clusterability methods. *Pattern Recognition*, 88:13–26, apr 2019.

[2] Charu C. Aggarwal and Chandan K. Reddy, editors. *Data Clustering : Algorithms and Applications*. CRC Press LLC, 2013.

[3] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical Report 2006-13, Stanford InfoLab, June 2006.

[4] Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020.

[5] D. Doval, S. Mancoridis, and B.S. Mitchell. Automatic clustering of software systems using a genetic algorithm.

[6] Julian J. Faraway. *Practical Regression and Anova using R*. July 2002.

[7] U.S. Center for Disease Control and Prevention. National environmental public health tracking network. Retrieved through online Database.

[8] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. O'Reilly Media, Inc., California, USA, 2017.

[9] Pranab Haldar, Ian Pavord, Dominick Shaw, Michael Berry, Mike Thomas, Christopher Brightling, Andrew Wardlaw, and Ruth Green. Cluster analysis and clinical asthma phenotypes. *American journal of respiratory and critical care medicine*, 178:218–24, 06 2008.

[10] Richard G. Lawson and Peter C. Jurs. New index for clustering tendency and its application to chemical problems. *J. Chem. Inf. Comput. Sci.*, 30:36–41, August 1990.

[11] Slobodan Petrović. A comparison between the silhouette index and the davies-bouldin index in labelling ids clusters.

[12] César Soto-Valero. A gaussian mixture clustering model for characterizing football players using the ea sports' fifa video game system. *RICYDE. Revista internacional de ciencias del deporte*, 13:244–259, 07 2017.

[13] U.S. Census Bureau. 2019 american community survey 1-year data. Retrieved through Census API.

[14] Xu Wang and Yusheng Xu. An improved index for clustering validation based on silhouette index and calinski-harabasz index. *IOP Conference Series: Materials Science and Engineering*, 569:052024, aug 2019.

[15] Xiangbing Zhou, Fang Miao, and Hongjiang Ma. Genetic algorithm with an improved initial population technique for automatic clustering of low-dimensional data. *Information*, April 2018.