# Title: DBSCAN

Author: KARIUKI REUBEN\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Run this code before you start

library(tidyverse)

#install.packages('dbscan')  
library(dbscan)

library(factoextra)

options(tibble.print\_max = Inf)  
options(tibble.width = Inf)

We will now us the DBSCAN clustering algorithm to address our business problem.

TECA has a problem. They have a customer loyalty program, but it is not as effective as it could be. TECA would like to do two things to improve their ROI on their customer loyalty program. First, they would like to increase the number of their loyalty customers. Over the last three years, the amount and percentage of transactions from loyalty customers has steadily increased, but overall, those percentages are still lower never reaching higher than 30%. TECA management knows there is more they can do. Second, TECA would like to increase the yield and profitability of their loyalty customers. They get the sense that some customers are spending good money on high margin products, but others seem to just dash in and out of the store, purchasing gas and cigarettes on the weekend and nothing more. From prior research, TECA learned that they do not have a good model for which transactions are going to come from loyalty customers the accuracy of their model is only about 60%. So, all in all, they realize that loyalty customers are good, but some are probably better than others. What TECA would like to do is find a way to figure out what types of customers they have. Can they separate their customers into subgroups or clusters? If so, they can market to their loyalty customers in different ways and promote those products and habits that will be most beneficial for TECA.

The first thing we need to do to help TECA is to bring in the dataset we are going to work with. This is TECA data that has been transformed to be used in our analysis. Let’s load it and examine it a bit. We first notice that the dataset has a little over 200,000 rows and 12 features. This data is aggregated at the level of an individual loyalty card holder. Thus, each row is the sum of all purchases for that person for a three-year period. So, what features are included? As you can see, each feature is a sum of the items purchase in a particular area of the store. I describe each column below: \* area.fresh: products in this area include, for example, pizza, hot sandwiches, salads, and roller grill items; \* area.snacks: products in this area include, for example, candy, gum, chips, and salty snacks; and \* area.cooler: products in this area include, for example, energy drinks, canned soda, and juice; \* area.grocery: products in this area include, for example, milk, eggs, and cheese; area. Nongrocery: products in this area include, for example, clothing, magazines, medicine, and newspapers; area. Alcohol: products in this area include, for example, wine and beer; area. Tobacco: products in this area include, for example, cigarettes and chewing tobacco. area. Fuel: products in this area include, for example, gas; area. Dispensed: products in this area include, for example, cold and hot dispensed (fountain) drinks; area. Lottery: products in this area include, for example, lottery tickets; area.miscellaneous: products in this area include, for example, store services and coupons; refill.yes indicates whether or not the purchase was a refill of fountain soda or coffee.

# Explore the data

clustering\_input2 <-readRDS("C:/Users/User/OneDrive/Documents/projects/DBSCAN.rds")  
str(clustering\_input2)

## tibble [272,417 × 12] (S3: tbl\_df/tbl/data.frame)  
## $ area.fresh : int [1:272417] 0 0 1 0 0 2 0 1 0 0 ...  
## $ area.snacks : int [1:272417] 0 0 0 1 0 0 0 0 1 0 ...  
## $ area.cooler : int [1:272417] 0 0 0 1 0 0 0 0 0 0 ...  
## $ area.grocery : int [1:272417] 0 0 0 0 0 0 0 0 0 0 ...  
## $ area.nongrocery : int [1:272417] 0 0 0 0 0 0 0 0 0 0 ...  
## $ area.alcohol : int [1:272417] 0 0 0 0 0 0 0 0 0 0 ...  
## $ area.tobacco : int [1:272417] 0 0 0 0 0 0 0 0 0 0 ...  
## $ area.fuel : int [1:272417] 1 0 0 0 0 0 0 0 0 0 ...  
## $ area.dispensed : int [1:272417] 0 0 0 0 0 0 1 0 0 1 ...  
## $ area.lottery : int [1:272417] 0 1 0 0 4 0 0 0 0 0 ...  
## $ area.miscellaneous: int [1:272417] 0 0 0 0 0 0 0 0 0 0 ...  
## $ refill.yes : int [1:272417] 0 0 0 0 0 0 1 0 0 1 ...

slice\_sample(clustering\_input2, n=10)

## # A tibble: 10 × 12  
## area.fresh area.snacks area.cooler area.grocery area.nongrocery area.alcohol  
## <int> <int> <int> <int> <int> <int>  
## 1 0 1 0 0 0 0  
## 2 1 0 1 1 0 0  
## 3 1 1 0 0 0 0  
## 4 1 0 0 0 0 0  
## 5 0 0 0 0 0 0  
## 6 0 0 0 0 0 0  
## 7 0 0 1 0 0 0  
## 8 1 1 0 0 0 0  
## 9 3 1 6 0 0 0  
## 10 0 0 1 0 0 0  
## area.tobacco area.fuel area.dispensed area.lottery area.miscellaneous  
## <int> <int> <int> <int> <int>  
## 1 0 0 0 0 0  
## 2 0 0 0 0 0  
## 3 0 0 2 0 0  
## 4 0 1 1 0 0  
## 5 0 1 2 0 0  
## 6 0 1 0 0 0  
## 7 0 0 0 0 0  
## 8 1 0 2 1 0  
## 9 0 2 0 0 0  
## 10 0 0 0 0 0  
## refill.yes  
## <int>  
## 1 0  
## 2 0  
## 3 2  
## 4 0  
## 5 2  
## 6 0  
## 7 0  
## 8 1  
## 9 0  
## 10 0

Of course, since there is no right answer here, no target value to predict, there is no need to split the data into testing and training data sets (though we could use hold out data to see what the clusters look like on new data). Rather, all of the data will be used to form clusters. Our first task then, is to standardize our data. Recall, that DBSCAN uses a distance function to determine whether a point is within another point’s epsilon neighborhood. Thus, we need all of the variables to be on similar scales. This function takes the existing dataframe and converts all of the variables using z-score standardization. That is, the mean of each feature is subtracted from each individual feature and then divided by that featureâs standard deviation. This transformation rescales the feature such it has a mean of zero and a standard deviation of one. Thus, it is measured in how many standard deviations it falls above or below its mean. The as.data.frame() function is used here to keep clustering\_input2 as a dataframe, instead of a matrix.

# Z-score standardization

clustering\_input2 <- as.data.frame(scale(clustering\_input2))

Before we run the model, let’s trim down our data. We have a lot of data, here, and it will take a long time for most normal computers to run the model. Additionally, we don’t need to examine every data point for all three years to get a sense of what the clusters are. Rather, we can take a sample and use that. The first line below again sets the starting point of the random number generator in the sample function. That way if we run this again we can get similar results. You can put any number into the set.seed() function, but you want to keep it the same every time. If you try to replicate my results, you will want to use the number I put here. (Of course, replicating results is not always possible due to differences in R versions and package versions, etc.). The second line creates a new dataframe. It has three parts. The first part, nrow(clustering\_input2), lists the total number of rows in clustering\_inputs2. The second part, `sample(nrow(clustering\_input2), 15000) randomly selects 15,000 numbers from 0 to nrow(clustering\_input2). Finally, this range of numbers is passed to clustering\_input2[ , ]. The square brackets index the old dataframe, clustering\_input2. They tell the new dataframe to take the old dataframe and keep only the 15,000 selected rows. This is the part before the comma. The blank after the comma tells the new dataframe to take all of the columns.

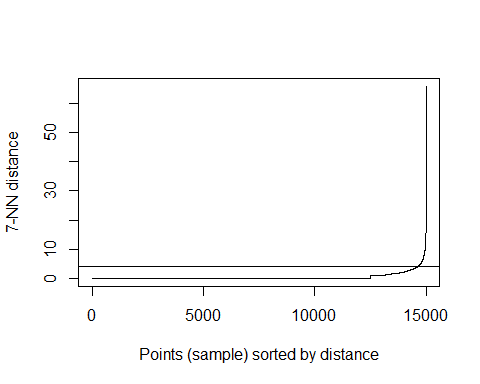
# Trim the data to a more manageable size

set.seed(777)  
clustering\_input2\_s <- clustering\_input2[sample(nrow(clustering\_input2), 15000), ]

We need to run the model next, but the algorithm requires two things from us–the amount for epsilon and the minimum points that each epsilon neighborhood should include. What should we put for these values? Unfortunately, there is not absolutely correct answer. Let’s start with minimum points. The default here for the algorithm is 5 points. Let’s start with that and go a little bit higher. Let’s pick minPts = 7. I don’t have a really good reason to do that other than through trial and error. Once we have the minimum points it is still really unclear what epsilon should be. That is problematic because this choice really affects the model. So, let’s use one of the many different methods available for helping us with this. Let’s use the minimum points to graph a k-nearest neighbors distance plot. This plots the k-nearest neighbor distances in a matrix of points. It measures the distance to the k nearest neighbors for each plot. K in this case is the minimum points, 7 for us. Then, it sorts those by distance, lowest to highest. This is then the x-axis. On the y-axis, the graph plots the distance to the nearest 7 neighbors for each point. The plot can be used to help find a suitable value for the eps neighborhood for DBSCAN by looking for the knee or elbow or bend in the plot.

As you can see here, our data points are quite close together until the end of the sort, where there are clearly some outliers. If we look for the bend in this plot and make that epsilon, it looks like we should choose about 4. I have drawn a line on the graph to show that. Thus, let’s use 4 for epsilon

dbscan::kNNdistplot(clustering\_input2\_s, k = 7)  
abline(h = 4)



Next, let’s run the model. The first line below sets the starting point of the random number generator in the dbscan function. This needs to be done every time a function using it is run.

To implement DBSCAN we use the dbscan() function that comes with the dbscan package. The main inputs to the function are, of course, first, the data. Second, we enter the number for epsilon, eps, that we just determined. Next, we pick the minimum points, minPts, which we decided on before at 7. Even with smaller data this may take a while to run. Try to be patient.

# Run the model

set.seed(777)  
clusters\_db <- dbscan(clustering\_input2\_s, eps = 4, minPts = 7)

So, how many clusters did we get? We can check that by running the table() function on the cluster vector located in the clusters\_db object. Examining this we see that we have 4 clusters, but that most of our data is in cluster 1. We also have some data points that are given a cluster 0. Can you guess what that means? Cluster 0 is used for outliers that were not included in any cluster. There were 263 of these.

# Count clusters

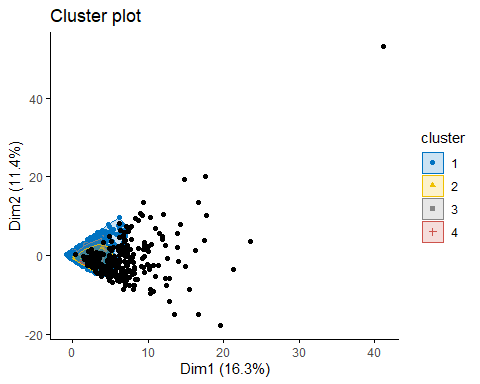
table(clusters\_db$cluster)

##   
## 0 1 2 3 4   
## 263 14120 411 196 10

Finally, let’s try to visualize the multi-dimensional clustering space in two dimensions. This is done with the factoextra package. It uses something called principal component analysis to collapse our 12-dimensional space into two dimensions. The result highlights that our clusters, at least in two dimensions are not easy to separate and are quite dense. It further illustrates that there are some significant outliers in our data. Thus, with this data, it makes a lot of sense to use DBSCAN versus another clustering method like k-means that does ignore outliers.

# Visualize the clusters

fviz\_cluster(clusters\_db, clustering\_input2\_s, geom = "point", show.clust.cent = FALSE, palette = "jco", ggtheme = theme\_classic())



Finally, let’s take the vector of clusters, cluster, from the clusters object, clusters\_db, put it back into our data, and look at what the clusters mean. To do this, we first, make cluster a new column in our dataset. Next, let’s group by cluster and count how many observations fall into each cluster with n(). This should equal the table() output above. In addition, let’s take the average of all of our other features using across( , mean) to get a sense of that the average person in the cluster is like.

What do we get? Let’s first remember that we are looking at standardized numbers in this table which have a mean of 0. Thus, negative numbers mean that the average for that cluster for that feature is below the overall average for that cluster. Let’s briefly look at cluster 0, since these are the outliers. They are high in all categories. This means that our outliers are high values, which is not surprising. \* Cluster 1 is by far our biggest cluster. Looking at the averages for each of the features we see that each one is negative. That means that the average person in this cluster purchases fewer than average of each item. This is discouraging news for TECA and tells them that most of their loyalty customers are in a group that does not visit the stores very often. \* Cluster 2 has its highest average for non-grocery items. They also purchase fresh items, snacks, cooler, and grocery items at above average rates. This is good and TECA could work to increase this clusters purchase of these items. \* Cluster 3 buys mostly miscellaneous items, as well as lottery tickets and some soda. This is not a helpful group for TECA since many of these items have low profit margin. Keeping these customers coming and buying is critical, as is converting them to some of the higher-margin products. \* Finally, cluster 4 is a tiny cluster that is probably too small to be concerned with. They do buy some high margin items, however, and TECA could seek to grow this cluster.

# Examine clusters

clustering\_input2\_s$cluster <- clusters\_db$cluster  
clustering\_look <- clustering\_input2\_s %>% group\_by(cluster) %>% summarize(n=n(), across('area.fresh':'refill.yes', mean))  
clustering\_look

## # A tibble: 5 × 14  
## cluster n area.fresh area.snacks area.cooler area.grocery area.nongrocery  
## <int> <int> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 0 263 2.27 2.04 2.35 1.93 2.42   
## 2 1 14120 -0.0280 -0.0354 -0.0369 -0.0194 -0.166  
## 3 2 411 0.0343 0.00287 0.0454 0.0366 4.23   
## 4 3 196 -0.190 -0.0607 -0.185 -0.0640 -0.166  
## 5 4 10 0.393 -0.0927 0.534 0.120 8.62   
## area.alcohol area.tobacco area.fuel area.dispensed area.lottery  
## <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 2.02 2.31 1.07 2.35 1.71   
## 2 -0.0350 -0.0469 -0.0309 -0.0446 -0.0381  
## 3 -0.0959 0.0934 -0.162 -0.100 -0.0422  
## 4 -0.0760 -0.0643 -0.190 0.00782 0.202   
## 5 0.197 -0.300 -0.188 -0.234 0.0723  
## area.miscellaneous refill.yes  
## <dbl> <dbl>  
## 1 2.11 2.08   
## 2 -0.126 -0.0353  
## 3 -0.126 -0.0621  
## 4 6.48 -0.113   
## 5 -0.126 0.117