Lab 08: -Means Clustering

2023-11-17

Table of contents

# 1. K-Means Clustering

-Means Clustering is a widely-used unsupervised machine learning algorithm, ideal for partitioning datasets into distinct, non-overlapping groups or ‘clusters’. We’ve seen it in the context of regional frequency analysis.

## 1.1 Algorithm

* **Inputs:** (number of clusters), $\vb{x} = \{x\_1, x\_2, x\_n\}$ (data points)
* **Outputs:** $\vb{c} = \{\mu\_1, \mu\_2, \ldots, \mu\_n\}$ (cluster assignments), $\vb{\mu} = \{\mu\_1, \mu\_2, \ldots, \mu\_k \}$ (cluster centroids)
* **Steps:**
  1. Randomly initialize cluster centers: $\vb{\mu} = \mu\_1^{(0)}, \mu\_2^{(0)}, \ldots, \mu\_k^{(0)} \in \mathbb{R}^d$
  2. Iterate until convergence:
     1. Assign each observation to the closest (in Euclidean distance) mean:
     2. Recompute each as the mean of all points assigned to it
     3. Terminate when the total change of the cluster centroids satisfies

# 2. Instructions

using CSV  
using DataFrames  
using Plots  
using StatsBase: mean, std

We will work in an external script. Open the file kmeans.jl and edit the functions provided. It’s a Julia file, so you can run line by line and work in the REPL.

To make the functions created available to you here, run the following command:

include("kmeans.jl")

check\_convergence (generic function with 1 method)

## 2.1 Initialize Centroids

First, edit the init\_centroids function. It takes in a matrix indexed by observations and features, and returns a matrix with rows (one for each centroid) and columns (one for each feature) where is the number of features of . The code provided initializes each centroid to a random value.

You can change this to whatever you like – be sure to explain your reasoning. One common approach is to choose random observations from the dataset as your initial centroids. Be sure to make sure that your centroids are distinct!

## 2.2 Euclidean Distance

In order to assign observations to clusters, we need to be able to compute the distance between between an observation and a centroid. We will use the Euclidean distance, which is defined above. This function should take in two generic vectors and return a scalar.

## 2.3 Assign Clusters

There is just one line of code to edit here.

The argmin function may be your friend.

## 2.4 Update Centroids

As you loop through the algorithm, you will need to update the centroids. This function takes in the data matrix , the cluster assignments $\vb{c}$, which is a vector of integers, and the number of clusters . It returns a matrix with rows (one for each centroid) and columns (one for each feature) where is the number of features of .

## 2.5 -means algorithm

This function is provided for you. You do not need to edit it. You simply need to define all the functions it calls.

function kmeans(X::AbstractMatrix, k::Int; τ=1e-5, maxiter=500)  
 n, d = size(X) # get the number of observations and features  
  
 # initialize the cluster centroids (μ)  
 μ = init\_centroids(X, k)  
 μ\_history = [μ]  
  
 is\_converged = false # initialize the flag  
 j = 1 # initialize the counter  
  
 # go through the loop until convergence is reached  
 while !is\_converged  
 cluster\_assignments = assign\_clusters(X, μ)  
 cluster\_centroids = update\_centroids(X, cluster\_assignments, k) # update the centroids  
  
 # add the current centroids to the history  
 push!(μ\_history, cluster\_centroids)  
  
 # check for convergence  
 is\_converged = check\_convergence(μ\_history, τ) # check for convergence  
  
 # if convergence seems unlikely, stop  
 if j > maxiter  
 @warn "Failed to converge after $j iterations"  
 return cluster\_assignments, μ\_history  
 end  
  
 # increase the counter  
 j += 1  
 end  
  
 cluster\_assignments = assign\_clusters(X, μ)  
  
 return cluster\_assignments, μ\_history  
end

# 3. Analysis

Our input data for this clustering analysis will be stations from the GHCND dataset (original [here](https://www.ncei.noaa.gov/data/global-historical-climatology-network-daily/doc/ghcnd-stations.txt)). We will subset only stations in Texas, and we will cluster on their longitude and latitude.

# Define a function to parse each line  
function parse\_line(line)  
 station = strip(line[1:11]) # Station ID  
 latitude = parse(Float64, strip(line[13:20])) # Latitude  
 longitude = parse(Float64, strip(line[22:30])) # Longitude  
 elevation = parse(Float64, strip(line[32:37])) # Elevation  
 state = strip(line[39:40]) # State Abbreviation (if present)  
 name = strip(line[41:end]) # Station Name  
 return (station, latitude, longitude, elevation, state, name)  
end  
  
# Read the file and process each line  
function read\_file(filename)  
 data = []  
 open(filename) do file  
 for line in eachline(file)  
 push!(data, parse\_line(line))  
 end  
 end  
 return DataFrame(data, [:Station, :Latitude, :Longitude, :Elevation, :State, :Name])  
end  
  
# Usage  
filename = "data/ghcnd\_stations.txt"  
stations = read\_file(filename)  
stations = stations[stations[!, "State"].=="TX", :]  
  
describe(stations)

Now we can run the clustering analysis we’ve implemented

X = Matrix(stations[!, [:Latitude, :Longitude]])  
K = 10 # choose your own!  
cluster\_assignments, μ\_history = kmeans(X, K)

┌ Warning: Failed to converge after 501 iterations  
└ @ Main In[11]:24

([0, 0, 0, 0, 0, 0, 0, 0, 0, 0 … 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [[0.934342471425092 0.14160577001956887; 0.7219082433767496 0.8934504887115557; … ; 0.03164814689852424 0.1369962762160316; 0.6294768169729952 0.30403271730844517], [0.32775696566866186 0.804376327750315; 0.4420750425053589 0.08094632500533483; … ; 0.6405194357768995 0.9634733456966537; 0.2781712309315567 0.4541642316978818], [0.3353524566199191 0.10018527167824798; 0.33026010857214216 0.7284712401451534; … ; 0.5879030209561606 0.22874838957784782; 0.9058451682511648 0.229368114912335], [0.7674397775945951 0.24996793444692644; 0.17551121763172262 0.9056507703857801; … ; 0.18865357056493082 0.15281171359017287; 0.05391899638333275 0.2778113289907177], [0.18907805412022194 0.8848683416908238; 0.6054135996723505 0.5356912810180448; … ; 0.4296216122791555 0.5457037959249753; 0.13063304669647124 0.5303106915392972], [0.8825002671117446 0.2122839902720779; 0.9079159618275982 0.9684369053379268; … ; 0.730914055769938 0.039170052998776406; 0.06883106119780258 0.486219450158461], [0.5509300908771835 0.34327699953110646; 0.817330445014761 0.32245128966083914; … ; 0.22149944066723803 0.0739092631935333; 0.42741998536875925 0.8698070249620892], [0.5816123033735165 0.3295676605088358; 0.2546288442057908 0.2413679689111392; … ; 0.005791367991982099 0.6625288623937104; 0.23607790148316055 0.9025979497982316], [0.632102853567038 0.5627470562710053; 0.7090492653718089 0.11272550146530558; … ; 0.17625812227923088 0.9444749283668273; 0.016850769144145628 0.4169977015467732], [0.8317047458176728 0.2704368181887803; 0.37976269440538946 0.13738056362324758; … ; 0.32858115875434857 0.31881863267981136; 0.680432769807729 0.0007583129566101077] … [0.8585762112171537 0.3222077847729763; 0.2840694628786955 0.3964288065236784; … ; 0.8631330584926993 0.845382602553864; 0.751409663901479 0.60412073812939], [0.21813796881621106 0.8151086327634028; 0.5169028020196139 0.8227152499276821; … ; 0.42674842760266285 0.1961517083550539; 0.8296104538329608 0.032669496408495036], [0.2912852392312042 0.9591304844026259; 0.601193206397815 0.6089713924974097; … ; 0.7495479239950663 0.615335792485034; 0.6020064861885728 0.9212082079991049], [0.09313614256629932 0.085171224044722; 0.7145875299816014 0.3577910429007203; … ; 0.14916894520178126 0.9318843016653026; 0.5989300274282887 0.732318873688263], [0.5554013798018533 0.6588724066912339; 0.6304802529963696 0.28867881839825293; … ; 0.26679634253598283 0.6631243100119916; 0.9665857174466926 0.6728358169657743], [0.9699491122771021 0.5137977596826913; 0.9364785985472995 0.22957258788641188; … ; 0.6075857210271004 0.6847125757641577; 0.5096608840796759 0.4639840976421138], [0.9865196167838161 0.7704094907736481; 0.24258707013058145 0.6762380086928029; … ; 0.29267025134280034 0.806330819019089; 0.2950509182839688 0.37391084079220793], [0.08230409966253971 0.7432265680745705; 0.723410061300014 0.8683462038904612; … ; 0.9853619558633807 0.3132865337340647; 0.35046825720015384 0.9841692477124848], [0.12977079463968566 0.12870831049821807; 0.5862367700390877 0.8034790262231477; … ; 0.9639197652105037 0.5622082812672097; 0.32859255643395846 0.746807735507028], [0.10707208851775452 0.5408844560117687; 0.27325902127169965 0.5486157818614747; … ; 0.3651230243957734 0.794899322796348; 0.4453750503978855 0.7505970539124455]])

# 4. Analysis

Once your code appears to be working:

1. Plot your cluster assignments on a map. Do they look logical? What does / does not make sense to you?
2. Check for consistency by re-running your code and comparing the plots. Do they look the same? Why or why not?
3. Plot the (two-dimensional) cluster centroids as a function of the number of iterations.
4. Try different values of . How does the clustering change? What is the best value of in your opinion? How could you determine this?

If you have extra time, alter other parts of the code (e.g., use elevation to cluster, or use a different distance metric). How does this change your results?