data1010-hw11

November 28, 2019

0.1 Homework 11

DATA 1010

[55]:
```
using Pkg; Pkg.activate(".");
using Plots, Clustering, Distributions, Statistics, LaTeXStrings
using DataStructures: DefaultDict
gr(fontfamily = "Palatino", legend = false);
```

```
Activating environment at `~/hw11/Project.toml`
```

```
        ArgumentError: Package LaTeXStrings not found in current path:
    - Run `import Pkg; Pkg.add("LaTeXStrings")` to install the LaTeXStrings␣
↪package.
```

```
        Stacktrace:

         [1] require(::Module, ::Symbol) at ./loading.jl:876

         [2] top-level scope at In[55]:2
```

0.2 Problem 1

In this problem, we will carry out the plan we detailed in class on November 15 for sampling from a particular probability measure on the set of possible tweets. See Problem 3 from the class notebook for details.

Step 1: In lieu of scraping tweets, we'll use Moby Dick to estimate English language letter-pair frequencies. Read in the document as a string, and perform the following cleaning steps:

(a) Use `lowercase` to lowercase every letter.

(b) Use `replace` to replace every `\n` with a space.

(c) Use `replace` to replace every character that is not a lowercase letter or space with the empty string. You'll want to use regular expressions for this; if you aren't familiar,

check out regexr.com for some helpful information and tools. The `replace` function accepts a string and a replacement *pair*, as in: `replace(string_to_replace_in, regex => replacement_string)`. A regex literal is a string preceded with `r`, as in `r"[a-z]"`.

(d) Use `replace` to replace every instance where multiple spaces appear consecutively with a single space. Once again, you'll want to use regular expressions.

[2]:
```
doc = read("mobydick.txt", String);
```

[3]:
```
# housekeeping / misc


testString = "RegExr      was created by     gskinner.com, and is proudly␣
 ↪hosted by Media Temple.\nEdit the Expression & Text to see matches. Roll␣
 ↪over matches or the expression for details. PCRE & JavaScript flavors of␣
 ↪RegEx are supported."
```

[3]:
```
"RegExr      was created by     gskinner.com, and is proudly hosted by Media
Temple.\nEdit the Expression & Text to see matches. Roll over matches or the
expression for details. PCRE & JavaScript flavors of RegEx are supported."
```

[4]:
```
docA = lowercase(doc)              #a
docB = replace(docA, "\n" => " "); #b
```

```
WARNING: Some output was deleted.
```

[5]:
```
# replace every character that is not a lowercase letter or space with the␣
 ↪empty string
docC = replace(docB, r"[^a-z\ ]" => ""); #c

#replace(testString, r"[^a-z\ ]" => "")
```

```
WARNING: Some output was deleted.
```

[6]:
```
docD = replace(docC, r"[ ]{2,}" => " ");  #d
```

```
WARNING: Some output was deleted.
```

Note: if you feel that you're not well positioned to do this part of the problem in a reasonable amount of time, you can submit your name here and get a working piece of code for this step.

Step 2: Build a dictionary storing the number of occurrences of each pair of consecutive letters. Use a `DefaultDict` in case there are letter pairs which never occur.

[7]:
```
letPairs = DefaultDict(0)

for i in 1:length(docD)
    letPairs[(docD[i],docD[i+1])] += 1
end
```

```
WARNING: Some output was deleted.
```

Step 3: Convert this count map into a proportion map by dividing each dictionary value by the number of letter pairs in the cleaned document.

[48]:
```
numberPairs = sum(values(letPairs))
letProp = DefaultDict(0)
for (key, value) in letPairs
    letProp[key] = value/numberPairs
end
letProp;
```

[48]:
```
DefaultDict{Any,Any,Int64} with 582 entries:
  ('a', 'f') => 0.000540917
  ('k', 'a') => 3.69197e-5
  ('p', 'c') => 4.29299e-6
  ('s', 'k') => 0.000424148
  ('s', ' ') => 0.0231564
  ('s', 'u') => 0.00163906
  ('a', 'm') => 0.00158154
  ('l', 'm') => 0.000331419
  ('e', 'p') => 0.000847437
  ('a', 'b') => 0.00170088
  ('b', 'v') => 1.03032e-5
  ('y', 'e') => 0.00141068
  ('x', 'o') => 6.86879e-6
  ('h', 'a') => 0.009946
  ('b', 'b') => 0.000362328
  ('i', 't') => 0.00675459
  ('u', 's') => 0.00267282
  (' ', 'n') => 0.00414274
  ('e', 's') => 0.00742172
  ('a', 'd') => 0.00272176
  ('z', 'z') => 3.77783e-5
  ('y', 'd') => 1.54548e-5
  ('s', 'y') => 0.000151113
  ('e', 'e') => 0.0040045
  ('e', 'w') => 0.000772738
            =>
```

Step 4: Write a function which carries out `n_iterations` Metropolis-Hastings updates beginning with the tweet `aaaaa...a` (that's 280 `a`s).

Notes: represent the tweet as a character array rather than a string (like `tweet = fill('a', 280)`), so you can mutate one element at a time without having to copy the string. You can `join` the array to return an ordinary string at the end of the function body. Also, watch out for positions 1 and 280; they require separate handling.

[95]:
```
function metro(n_iterations, doc, pdict)
    alphabet = "abcdefghijklmnopqrstuvwxyz "

    # 1st iteration:
    tweet = fill('a', 280)



    function update(n, tweet)
        if n == 0
            return join(tweet, "") #not sure this will be ok; CHK
        end

        k = rand(1:280) # Choose position k uniformly at random
        k_old = tweet[k]
        k_new = alphabet[rand(1:27)] # Choose new letter/space randomly

        function checkGo(f_old, f_new)
             = f_new/f_old
            #= I was turned onto this elegant way of checking the condition␣
↪by Yi; another way
            would have been to sample from a Bern r.v..=#
            if rand() <
                tweet[k] = k_new
            end
        end


        # Main logic
        if k == 1
            f_old = pdict[k_old, tweet[k+1]]
            f_new = pdict[k_new, tweet[k+1]]
            checkGo(f_old, f_new)
        elseif k == 280
            f_old = pdict[tweet[k-1], k_old]
            f_new = pdict[tweet[k-1], k_new]
            checkGo(f_old, f_new)
        else
            f_old = pdict[(tweet[k-1]), k_old] * pdict[k_old, tweet[k+1]]
            f_new = pdict[(tweet[k-1]), k_new] * pdict[k_new, tweet[k+1]]
            checkGo(f_old, f_new)
        end
        update(n-1, tweet)
    end

    # Call the update function
    update(n_iterations, tweet)
```

```
end
```

[95]:
```
metro (generic function with 1 method)
```

[96]:
```
testS = metro(1000, docD, letProp)
```

[96]:
```
"haray wataca taliafahavaif gasor wiatimavubooinorowar igeaayomiloarawale
acoulens pag esaromegsa malyietatalf ki chavakiman recr psitosaeigiscamabeahr
aclerikimanoalwahe stfer ceeala n apapuesjucandllladara ana tabeavaay hewe
aitroat eval gea aren pnaun t at ttr voketasaviarlauta"
```

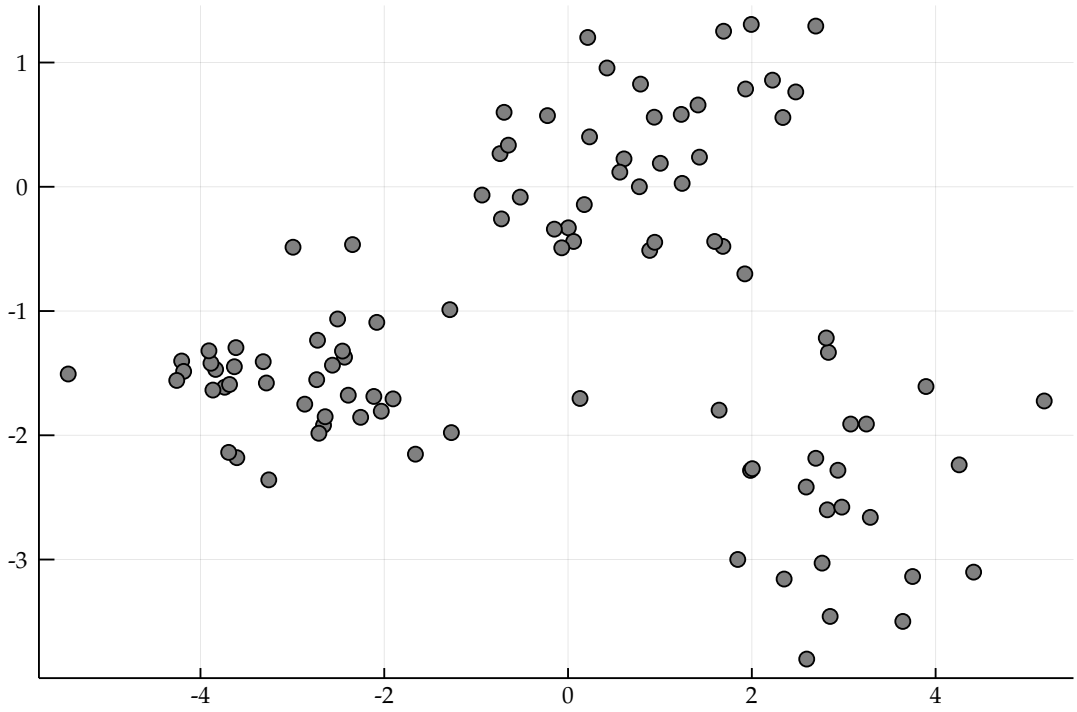
0.3 Problem 2

A **clustering** algorithm takes a collection of points and an integer k as input and returns a partition of those points into k groups.

(a) Visually group the points in the scatter plot below into three clusters. (Nothing to write here; just do it in your head before scrolling down.)

[4]:
```
include("clustering-points.jl")
```

[4]:

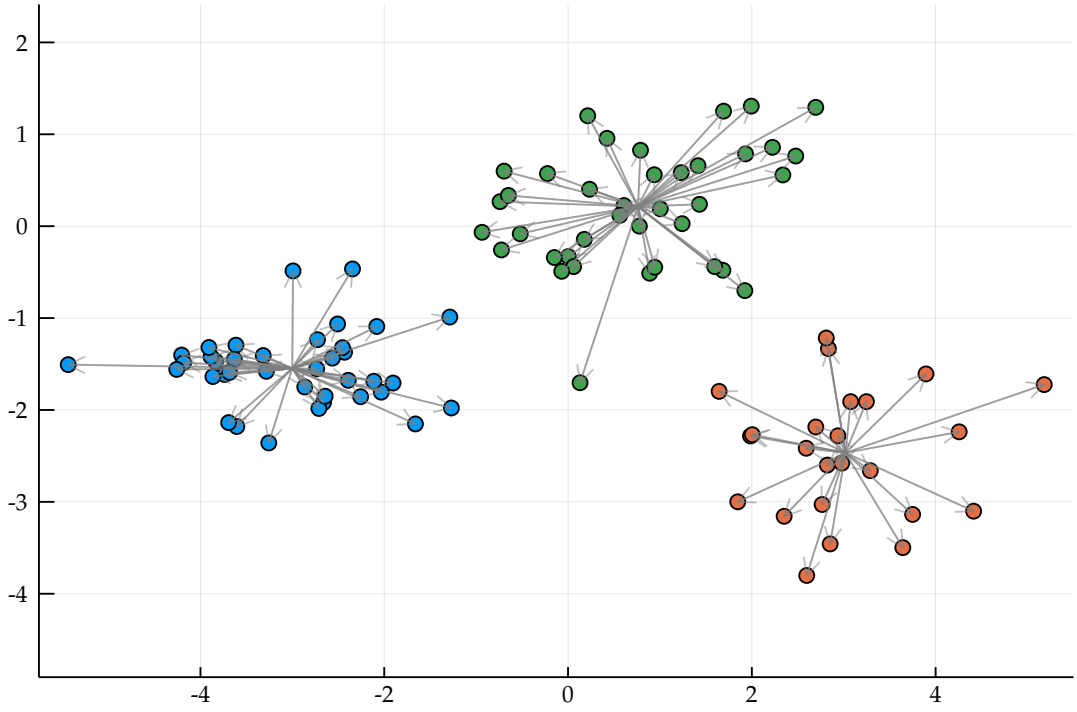


One of the most fundamental clustering algorithms is called ***k*-means**, and it returns the grouping which minimizes the sum of squared distances from each point

to its group's centroid:

[5]:
```
clusters = kmeans(hcat(X...), 3)
centers = Tuple.(eachcol(getproperty(clusters, :centers)))
function showclusters(X, centers, groups; kw...)
    scatter(Tuple.(X), group = groups, ratio = 1; kw...)
    for (point, group) in zip(X, groups)
        plot!([centers[group], Tuple(point)], lineopacity = 0.5,
              arrow = arrow(), color = :gray)
    end
    current()
end
showclusters(X, centers, getproperty(clusters, :assignments))
```

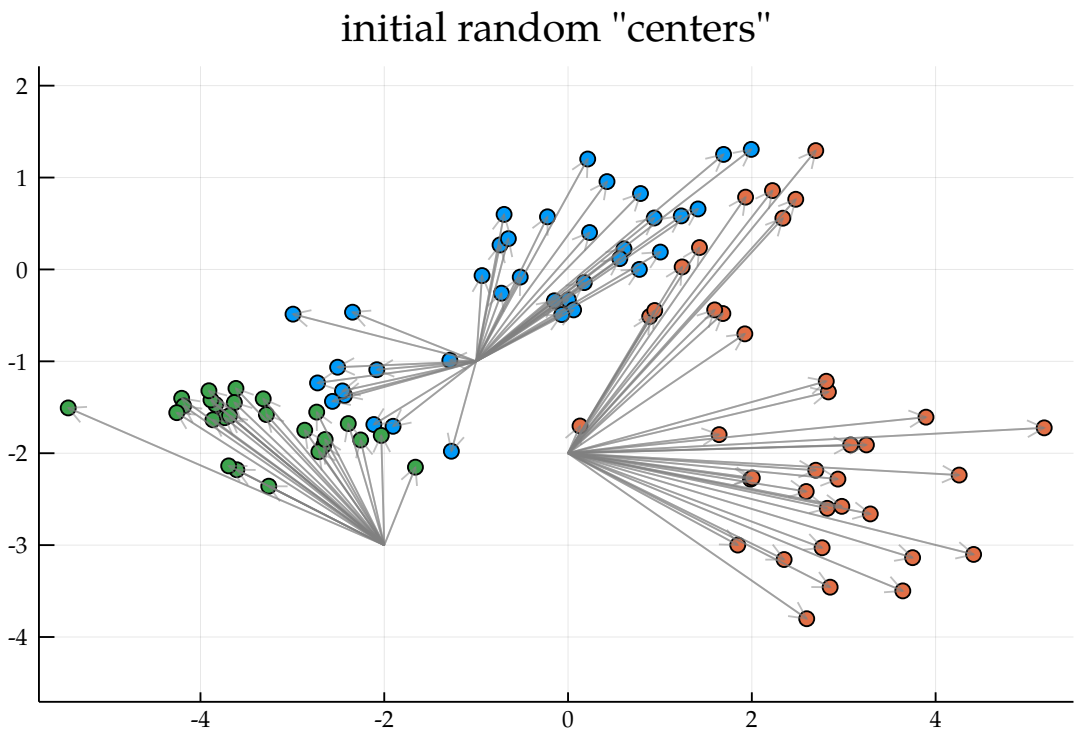
[5]:



The standard k-means algorithm proceeds by choosing k initial centers randomly, identifying for each point which center it's closest to, and updating the centers to be the centroids of the points assigned thereto.

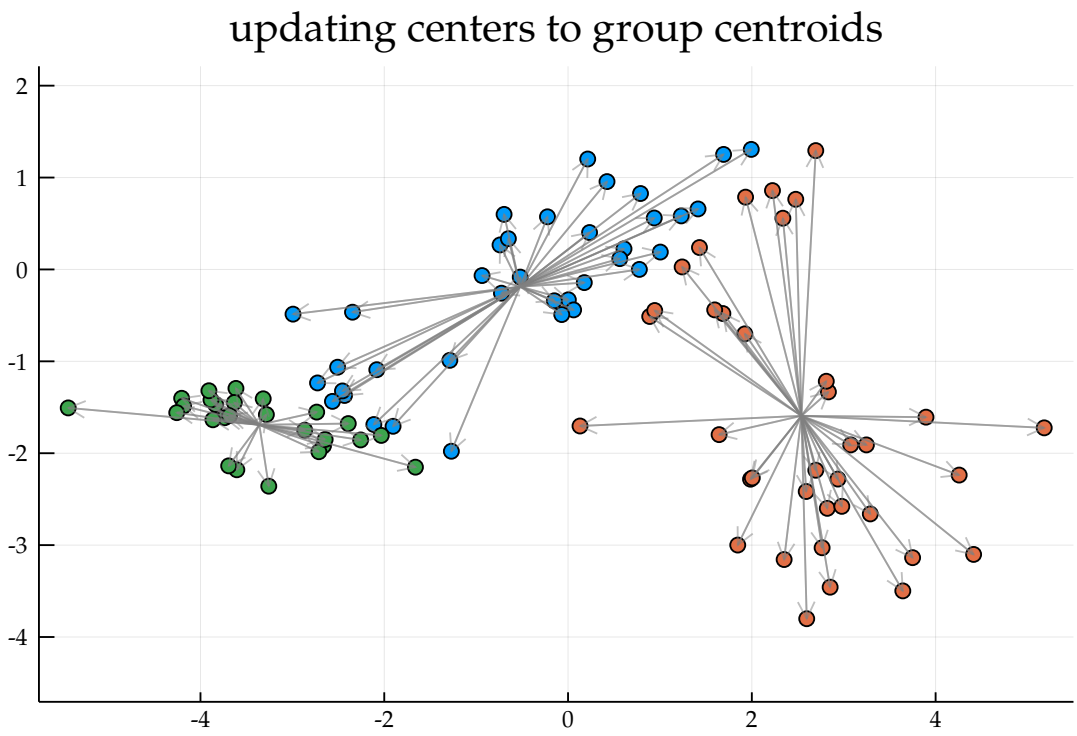
[8]:
```
centers = [[-1, -1], [0, -2], [-2, -3]]
groups = [argmin([norm(x-c) for c in centers]) for x in X]
showclusters(X, Tuple.(centers), groups; title = "initial random \"centers\"")
```

[8]:



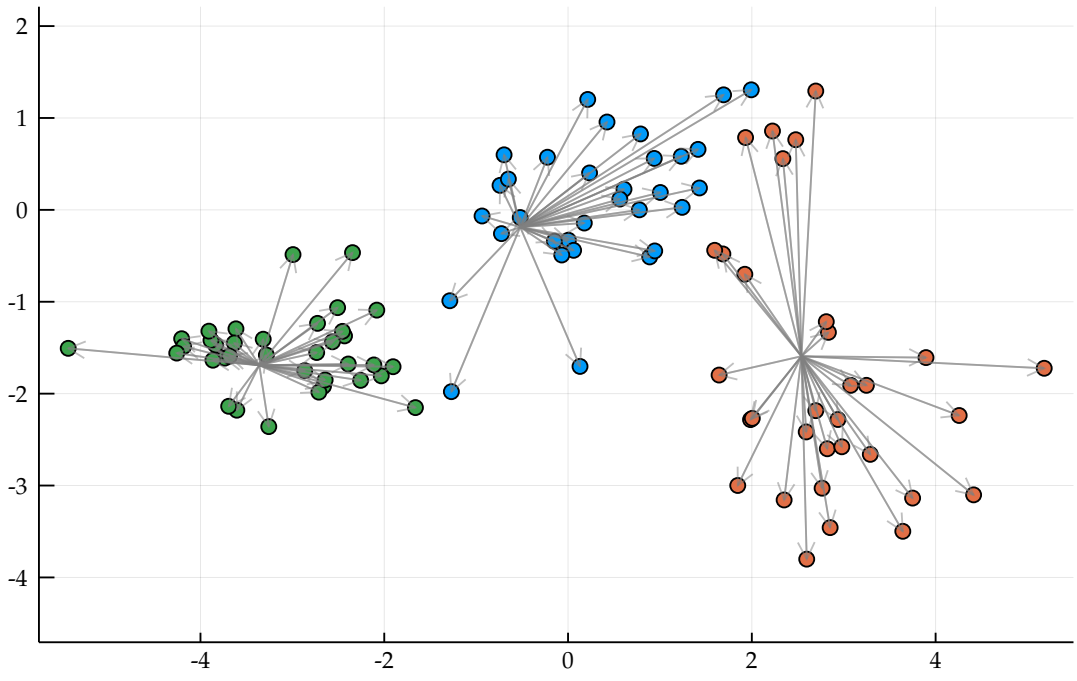
[9]:
```
centers = [mean(X[groups .== c]) for c in 1:3]
showclusters(X, Tuple.(centers), groups, title = "updating centers to group␣
↪centroids")
```

[9]:



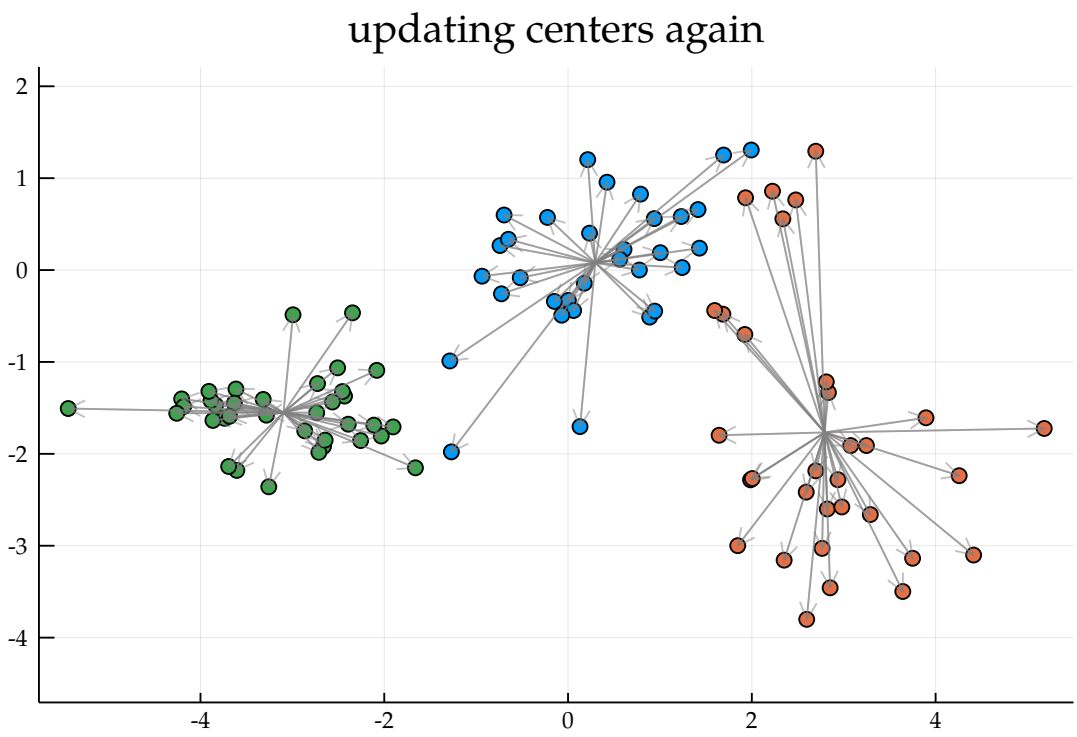
[10]:
```
groups = [argmin([norm(x-c) for c in centers]) for x in X]
showclusters(X, Tuple.(centers), groups, title = "re-grouping the points")
```

[10]:



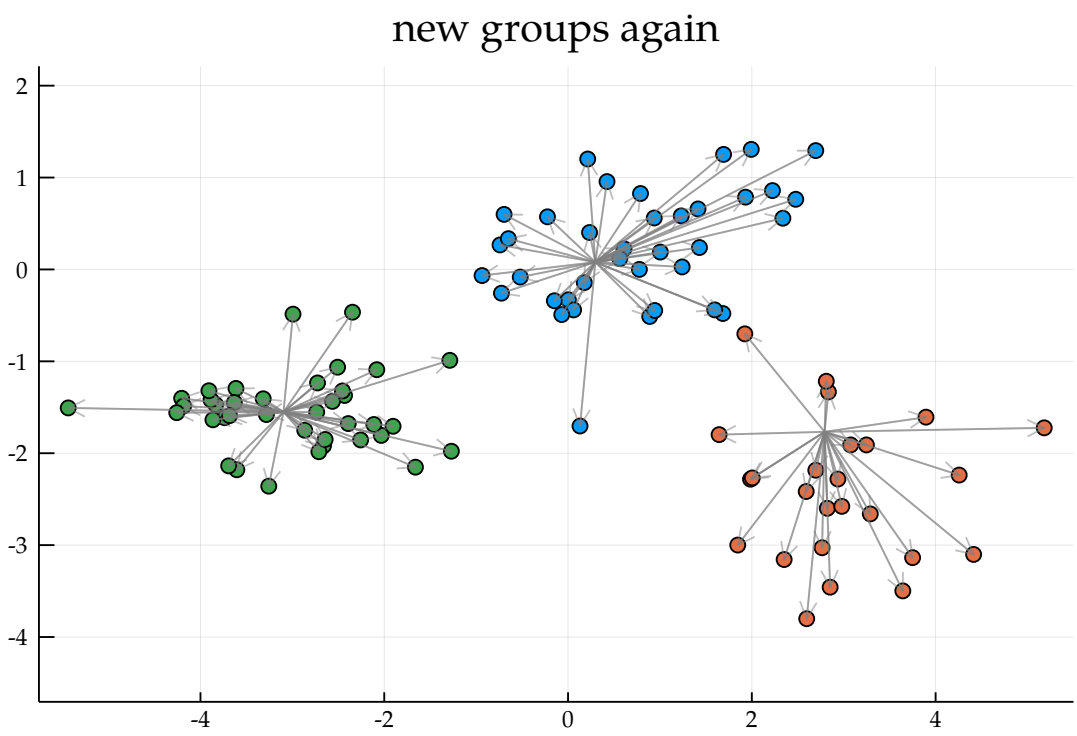
[11]:
```
centers = [mean(X[groups .== c]) for c in 1:3]
showclusters(X, Tuple.(centers), groups, title = "updating centers again")
```

[11]:



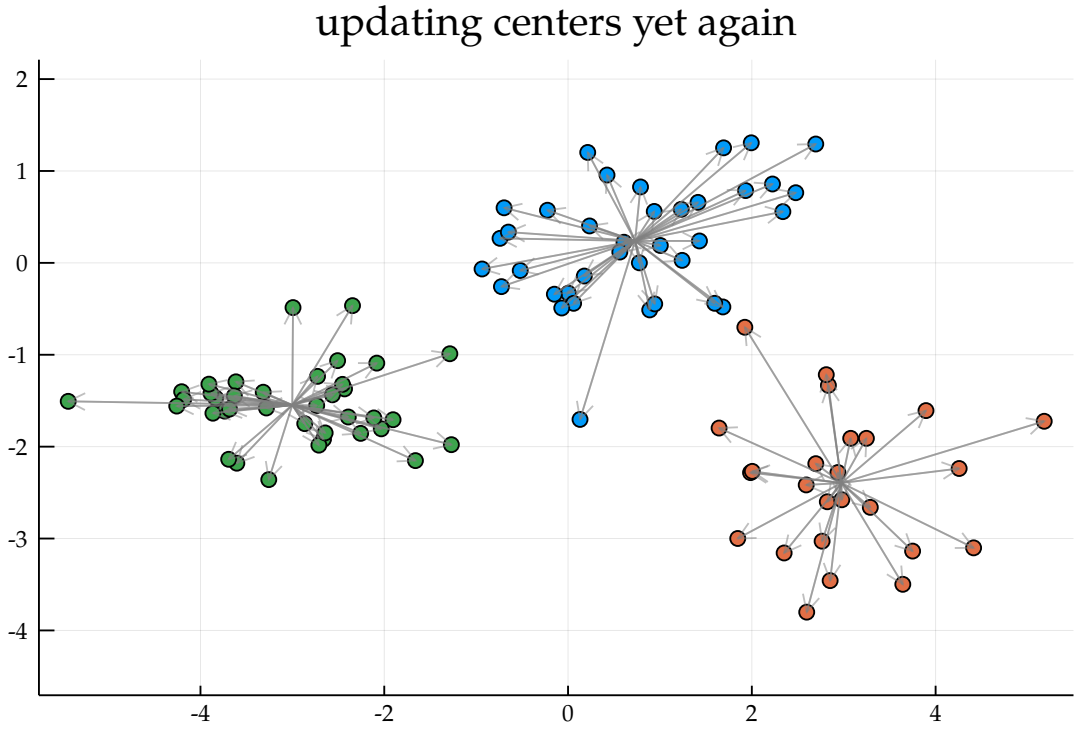
[12]:
```
groups = [argmin([norm(x-c) for c in centers]) for x in X]
showclusters(X, Tuple.(centers), groups, title = "new groups again")
```

[12]:



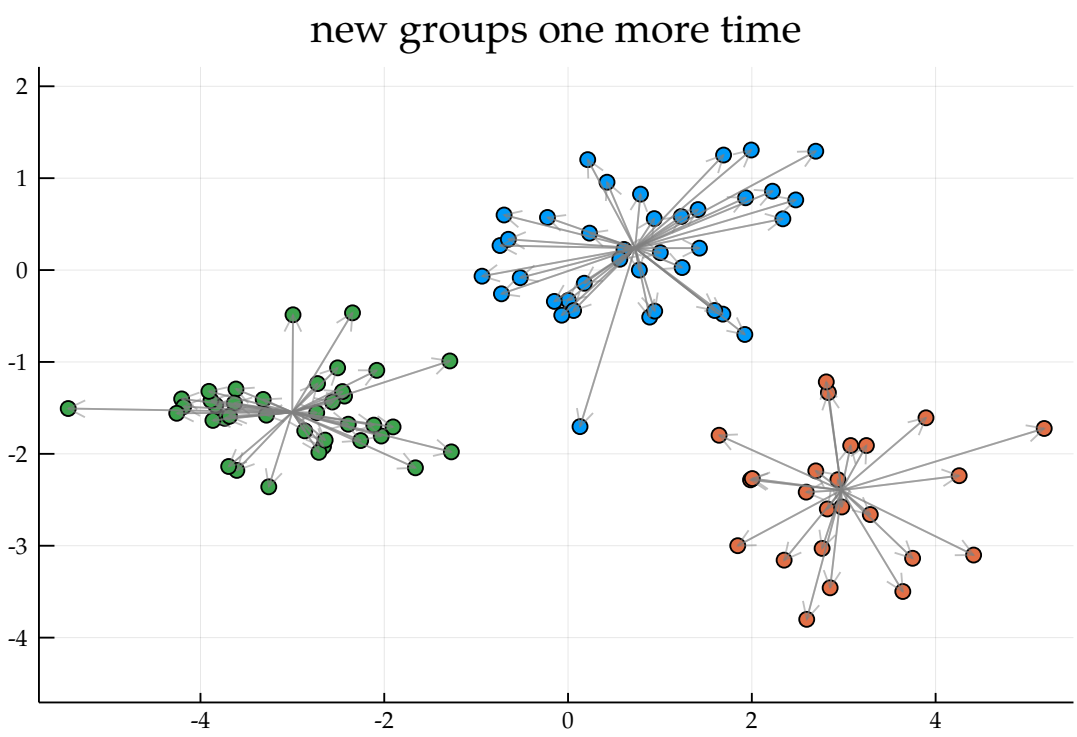
[13]:
```
centers = [mean(X[groups .== c]) for c in 1:3]
showclusters(X, Tuple.(centers), groups, title = "updating centers yet again")
```

[13]:



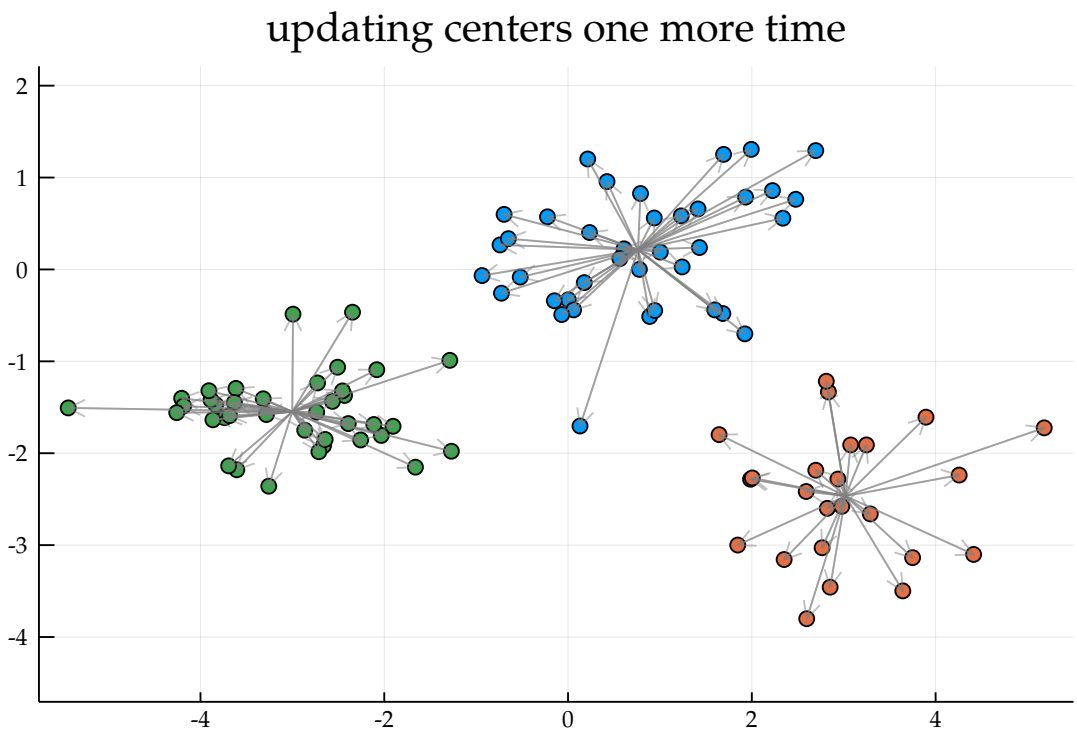
[14]:
```
groups = [argmin([norm(x-c) for c in centers]) for x in X]
showclusters(X, Tuple.(centers), groups, title = "new groups one more time")
```

[14]:



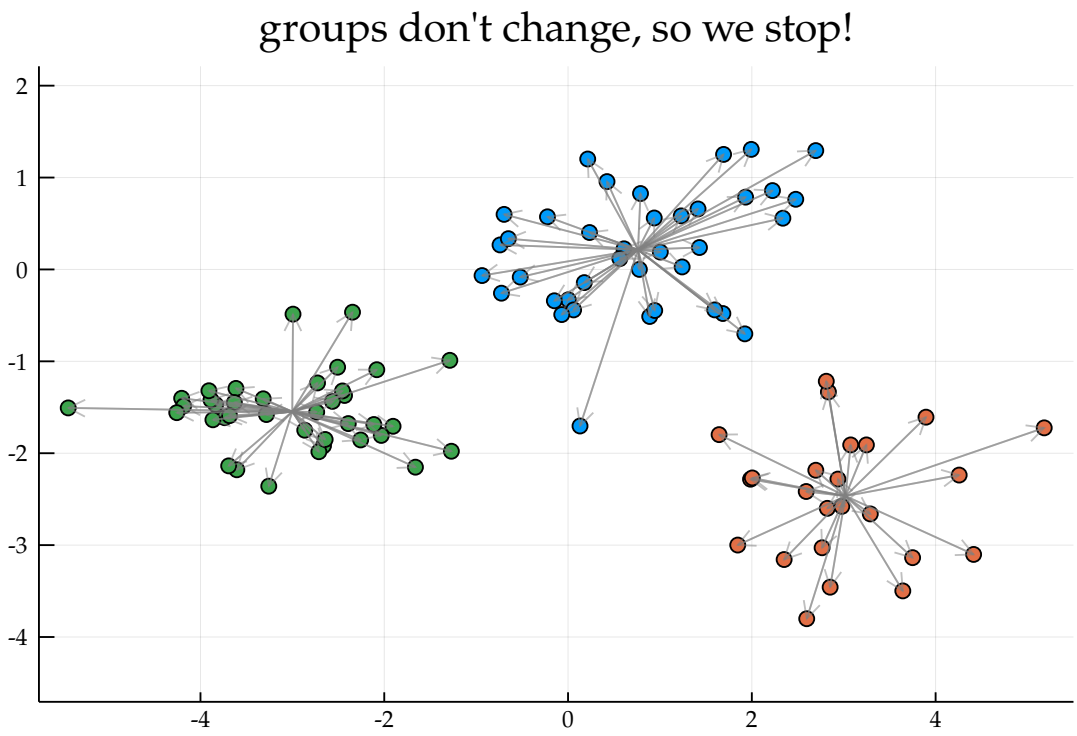
[15]:
```
centers = [mean(X[groups .== c]) for c in 1:3]
showclusters(X, Tuple.(centers), groups, title = "updating centers one more␣
↪time")
```

[15]:



[16]:
```
groups = [argmin([norm(x-c) for c in centers]) for x in X]
showclusters(X, Tuple.(centers), groups, title = "groups don't change, so we␣
↪stop!")
```

[16]:



(b) Is k-means sensitive to the scaling of the features? In other words, if we change units for x_1, does that potentially change the resulting clusters?

Proposed answer: k-means is not sensitive to the feature scaling. The centroids which minimize the sum of squared distances to the points in their grouping will still minimize the sum of squared distances even after feature scaling.

In class we described the Gaussian mixture model (GMM) sort of like a clustering algorithm, because almost all of the points could be described as ``clearly purple'' or ``clearly green.'' (Note that the means of the Gaussians in the GMM play a similar role to the k-means centroids.)

We can make the GMM into an actual clustering algorithm by associating each point with the value of the hidden variable Z with the largest conditional probability given the point's observed X value. In other words, in the two-cluster case, we check for each point $\mathbf{x}_i$ whether its corresponding π_i value is less than 1/2.

Let's make a connection between the GMM (as a clustering algorithm) and k-means:

(b) Suppose that rather than estimating the covariance matrices at each step in the EM algorithm for the Gaussian mixture model, we treat them as constant matrices of the form ϵI (where I is the identity matrix and ϵ is some fixed, small constant). Show that with this adjustment, the EM algorithm for the Gaussian mixture model with very small ϵ is approximately the same as the k-means clustering algorithm.

Proposed answer. With this adjustment, the variances of the MVNs will be

very small. So, intuitively, the data points will partition more clearly into clusters, like the k means algorithm. I'm not sure how to show this more rigorously.

[0]: