DBSCAN and GMMs with tidyclust in R

Brendan Callender

Advisor: Dr. Bodwin

Committee Members: Dr. Lund, and Dr. Dekhtyar

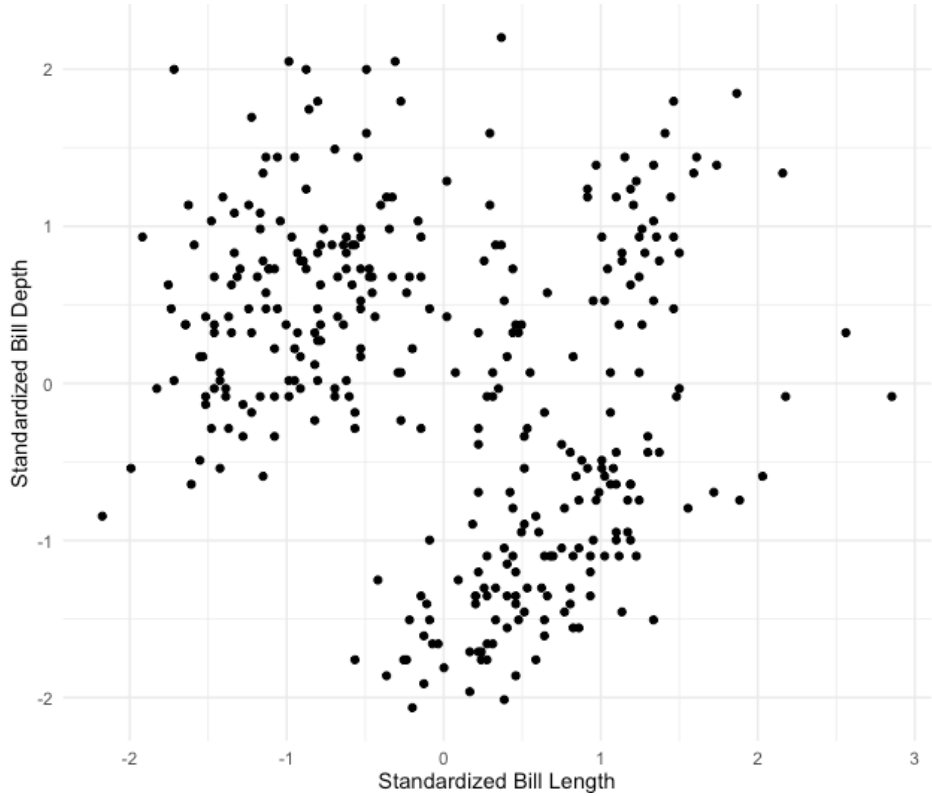
Agenda

1. What is **tidyclust?**

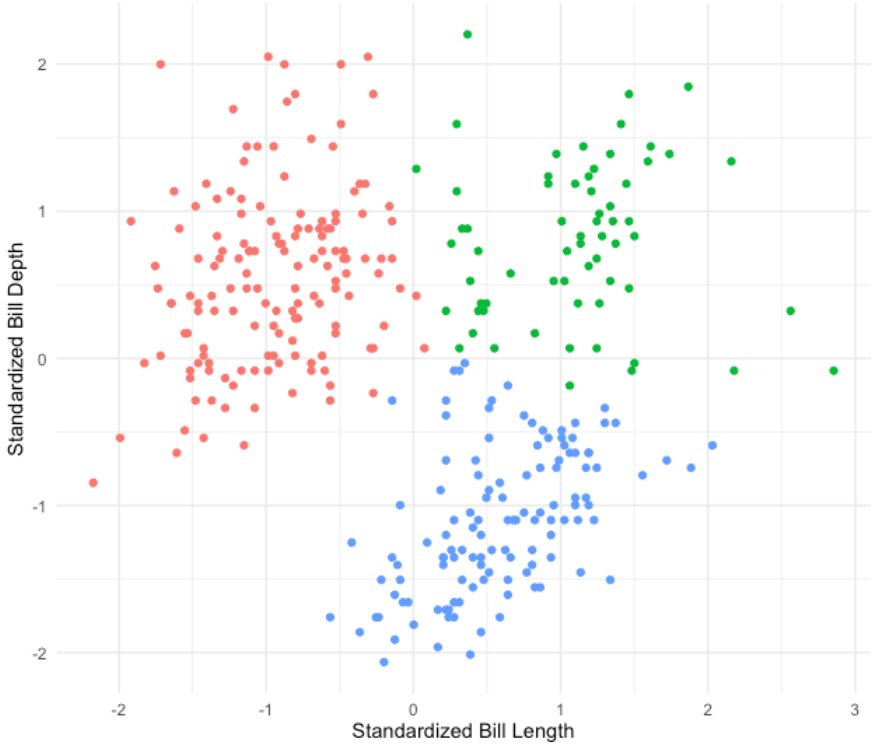
2. What I added to **tidyclust**
   - Density-based clustering with **DBSCAN**
   - Model-based clustering with **GMMs**

Clustering

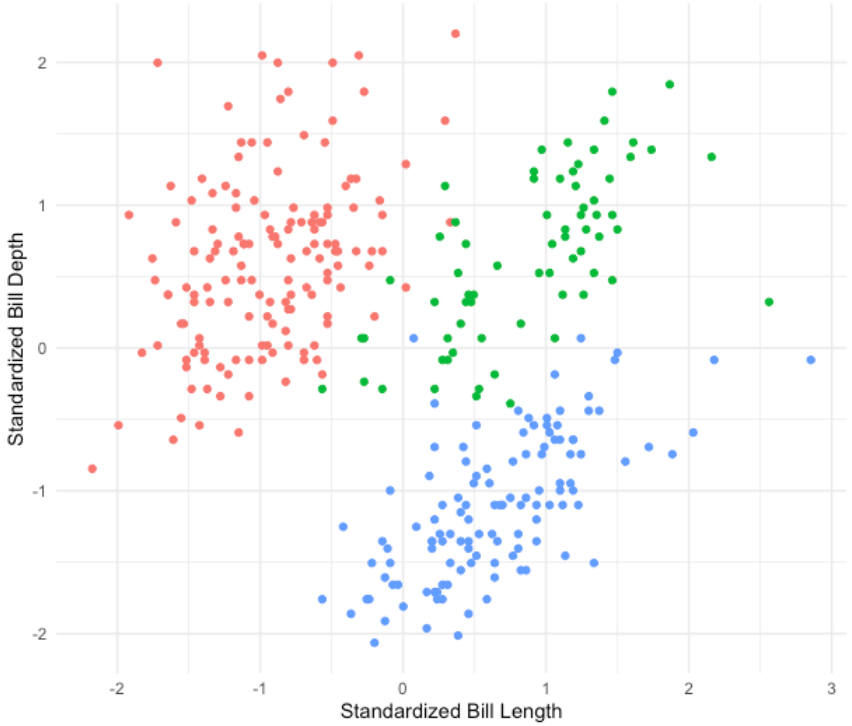
- Unsupervised learning method
- Used to find **groupings** in data based on internal patterns



Example Clustering Result

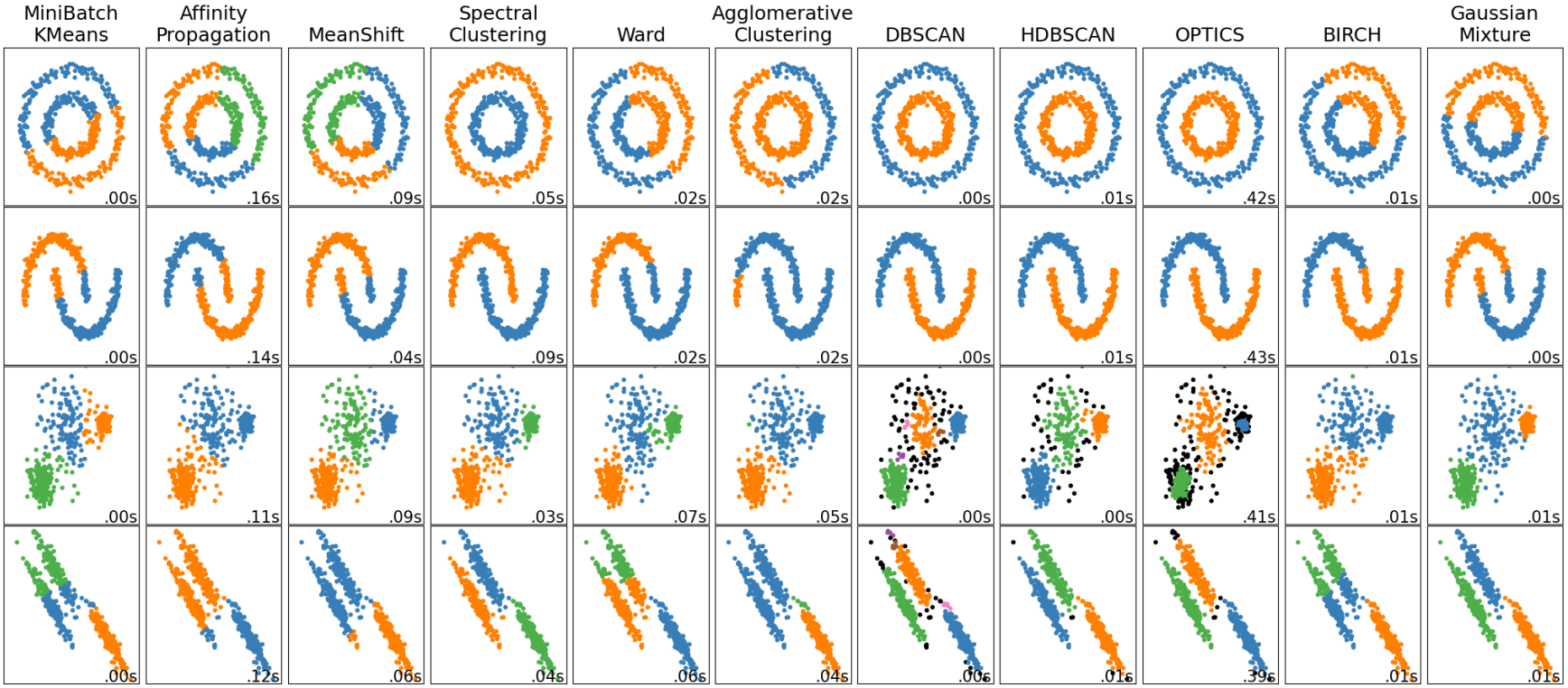


Actual Penguins Species



There are many different clustering methods!

Each has its own **strengths** and **weaknesses**



These methods are scattered across many R packages!

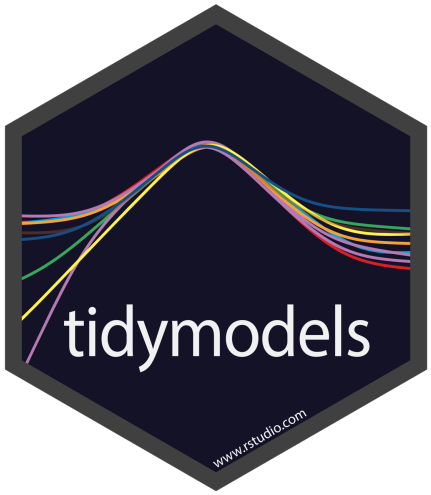
R has **many ways** to do the **same thing**

- Different packages pick what works best for them
- Ex. Making a simple scatterplot

```
1 plot(y ~ x, data) # formulas
2
3 plot(data$x, data$y) # dollar signs
4
5 data %>% # tidy
5   ggplot() +
6   geom_point(aes(x = x, y = y))
```

The tidyclust package

- Unifies clustering methods in R under a common interface
  - All methods use the **same syntax**!
- Like **tidymodels** but for unsupervised learning!
  - Follows conventions set in the **tidyverse**



A quick tutorial for **tidyclust**

```
1 library(tidyclust)
2 library(tidymodels)
```

Using k-means clustering as an example

Creating a clustering specification

```
k_means_spec <- k_means(
  mode = "partition",
  engine = "dbscan",
  num_clusters = 3
)
```

Where…

- `mode` controls the behavior of the specification
- `engine` controls which underlying package implementation to use
- `num_clusters`…

Creating a recipe

```
k_means_recipe <- recipe( ~ predictor1 + predictor2 + …)|>
 step_naomit(all_predictors()) |>
 step_normalize(all_numeric_predictors())
```

The recipe can be used to…

1. Select which columns to use for the model using formula notation
2. Preform preprocessing steps using `step_*()` functions
    - Example. `step_pca()` and so much more!

Creating a workflow

```
k_means_wflow <- workflow()|>
 add_model(k_means_spec)|>
 add_recipe(k_means_recipe)
```

The workflow is used to combine a model and a recipe together

- Allows for easy mixing and matching of different model specifications and setups

Fitting the model

You can fit directly using the **clustering specification**

```
k_means_fit <- k_means_spec |>
  fit(~ predictor1 + predictor2 + ..., data)
```

Or use a **workflow**

```
k_means_fit <- k_means_wflow |>
  fit(data)
```

- For workflows, we just need to provide the data since the recipe controls which columns to use

What else?

Extract key features of the model fit

```
1 k_means_fit |> extract_fit_summary()
```

Use the model to predict on new data

```
1 predict(k_means_fit, new_data)
```

Extract underlying engine object

```
1 k_means_fit |> extract_fit_engine()
```

Model Argument Tuning

```
data_cvs <- vfold_cv(data, v = 5)

k_means_tune <- k_means(num_clusters = tune())

k_means_grid <- grid_regular(
    num_clusters(c(3,7)),
    levels = 5
)

k_means_tune_res <- tune_cluster(
    k_means_tune,
    resamples = data_cvs,
    grid = k_means_grid
)
```

This process is the **same** across all clustering specifications!

What I added to **tidyclust**!

- Density-based clustering with DBSCAN
- Model-based clustering with Gaussian Mixture Models

What this took…

1. Researching each method
2. Finding a current implementation in R
3. Writing a lot of code

Density-based Clustering with **DBSCAN**

DBSCAN

Density-based Spatial Clustering of Applications with Noise

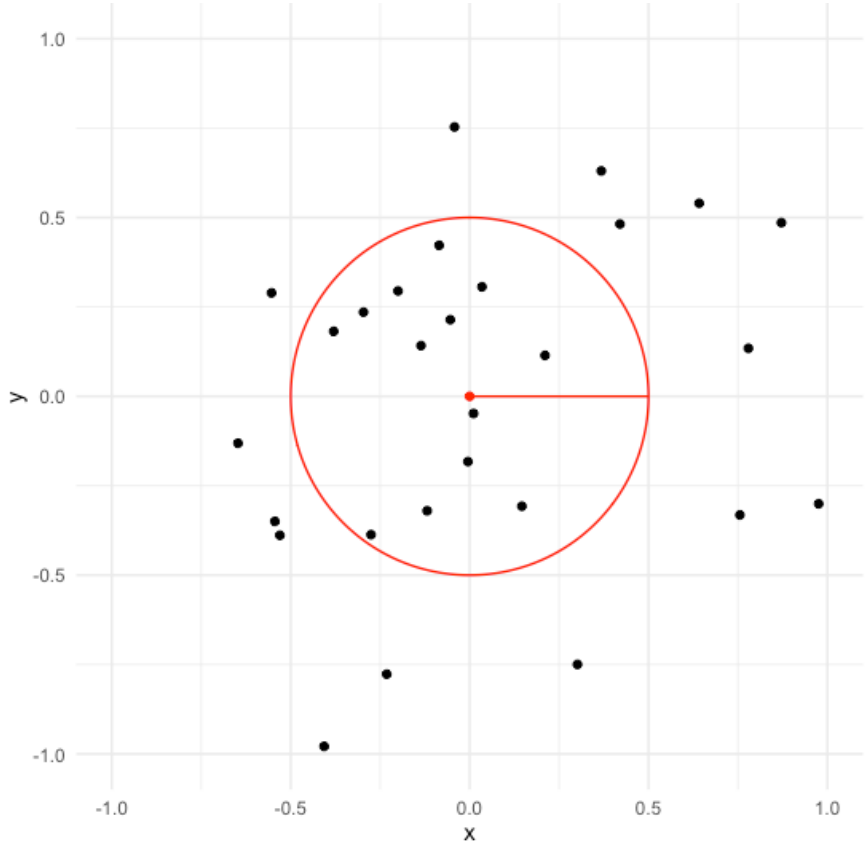
Arguments:

- ϵ (epsilon)
  - Controls the radius of the region used to…
    1. Compute **density estimates**
    2. Determine **connected points**
- MinPts
  - Used as a **density threshold** to identify clusters

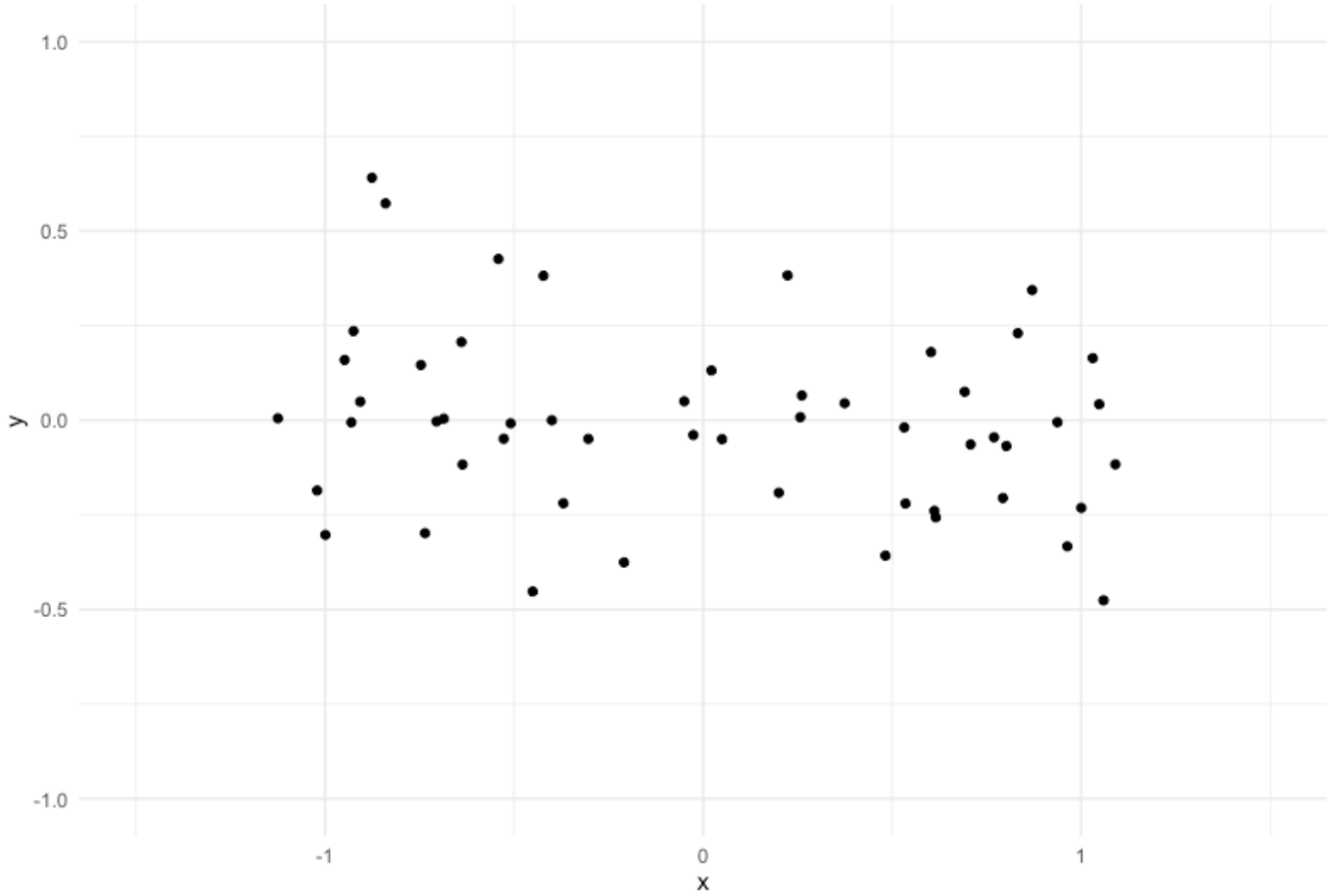
Important DBSCAN Definitions

1. The **ϵ-neighborhood** of a point is the set of all points that are within ϵ distance from point

2. A **core point** is a point that contains at least MinPts number of points within its ϵ-neighborhood



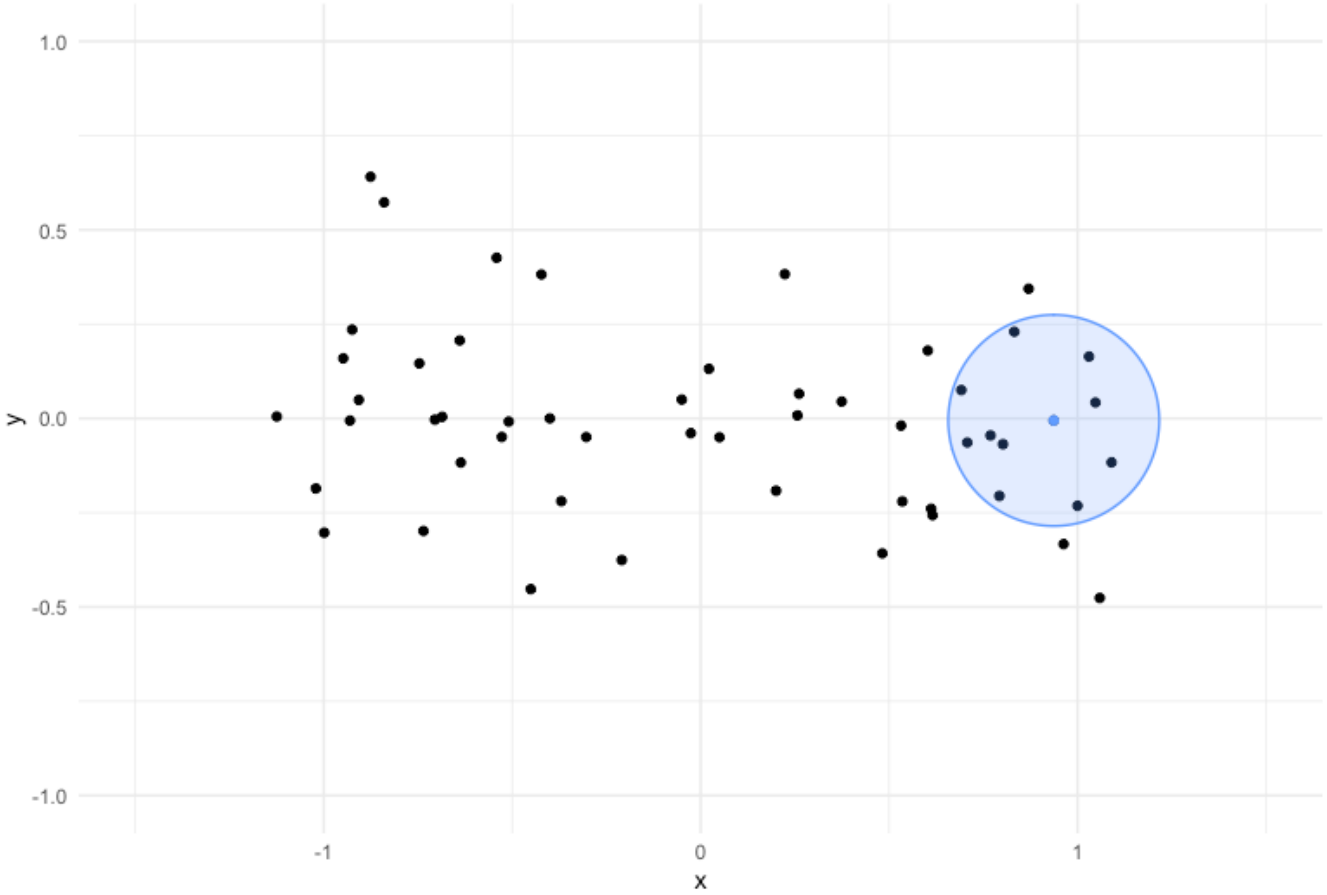
The DBSCAN Algorithm



$\epsilon = 0.28$

MinPts = 7

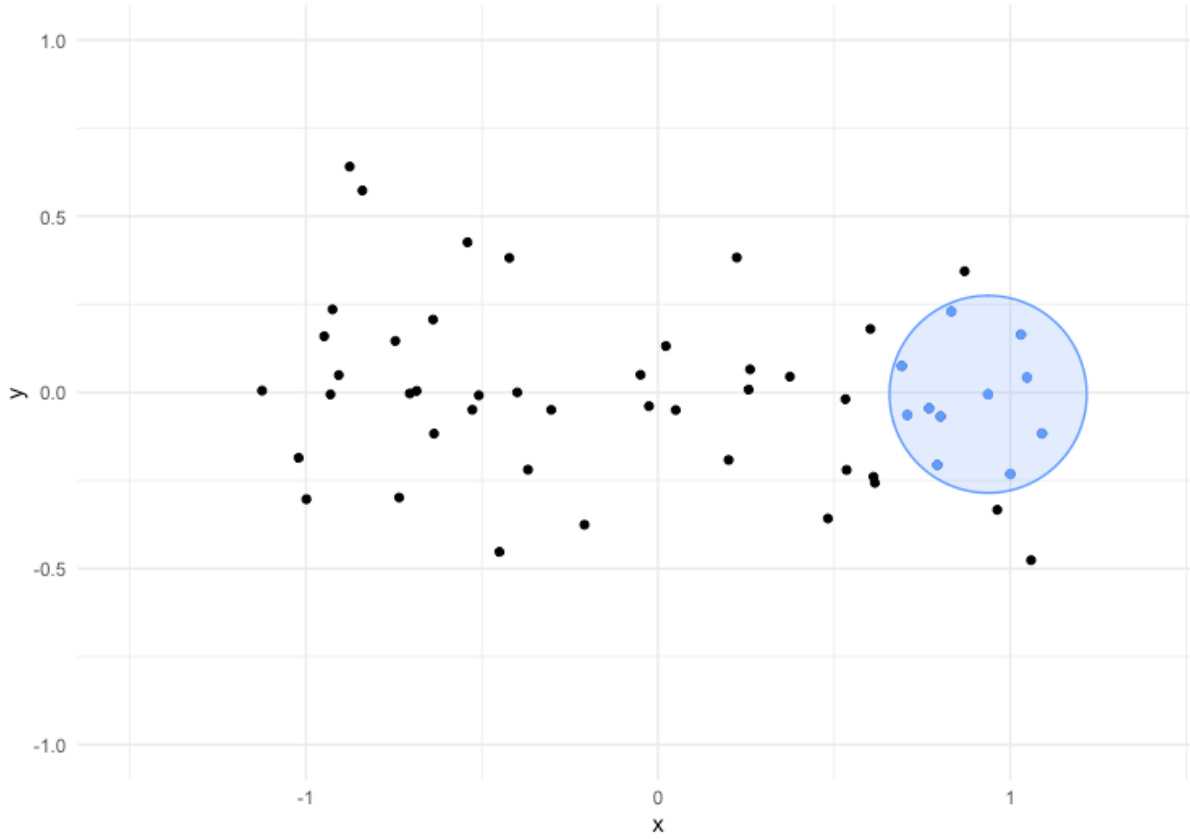
Core Point Discovery



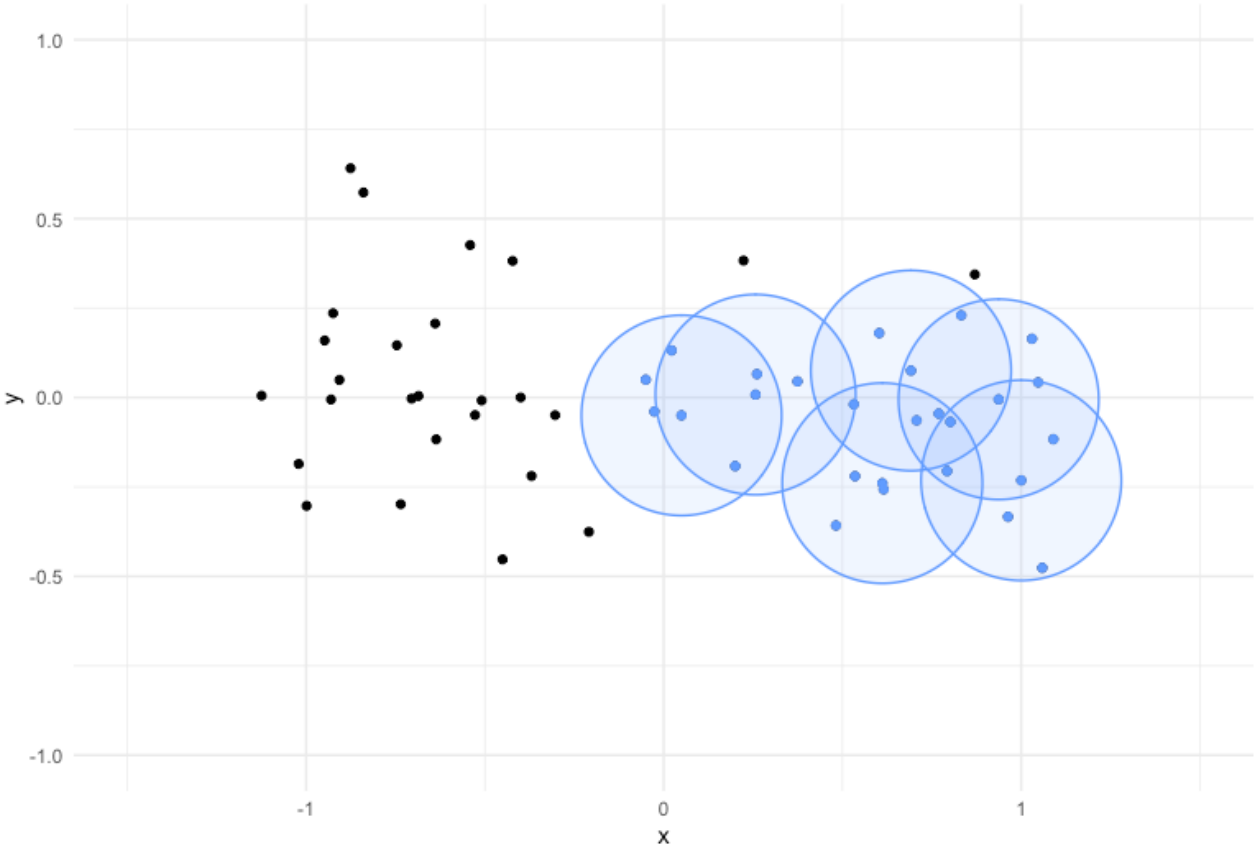
The fitting process begins by **scanning** through the dataset for **core points**

Cluster Formation

Once a core point is found a new **cluster is formed** and all points within the ϵ-neighborhood of the point are added



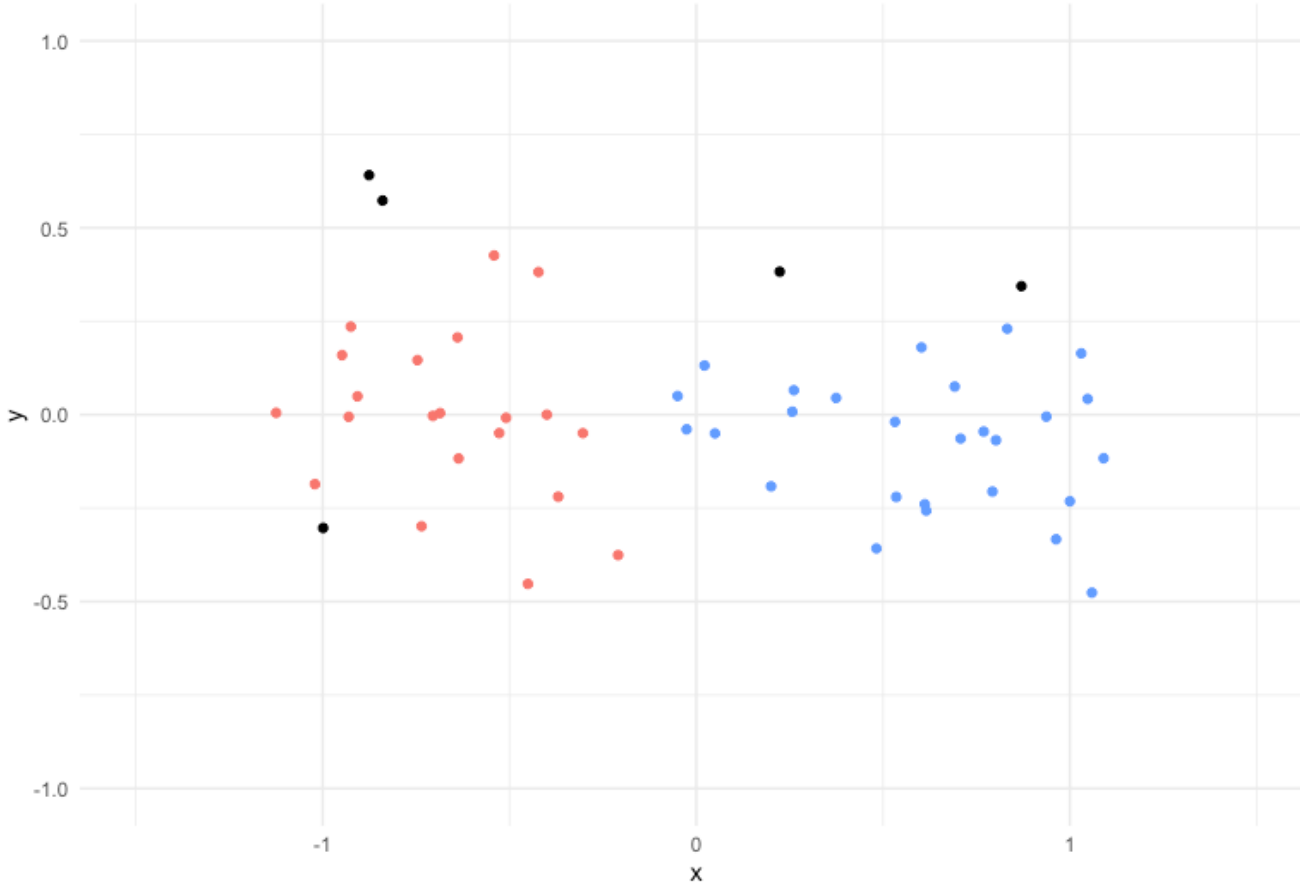
Recursive Clustering Building



If a **core point** is added to the cluster, the **cluster is expanded** to all points the ϵ-neighborhood of the core point.

Final Cluster Assignments

This process is repeated until **all core points** have been **found**



DBSCAN with the **dbscan** package

```
1  library(dbscan)
2  dbscan(x, eps, minPts = 5, weights = NULL, borderPoints = TRUE, ...)
```

Where…

- `x` is the data to perform DBSCAN on
- `eps` is the radius of the ϵ-neighborhood used to identify core points
- `minPts` is the density threshold used to identify core points

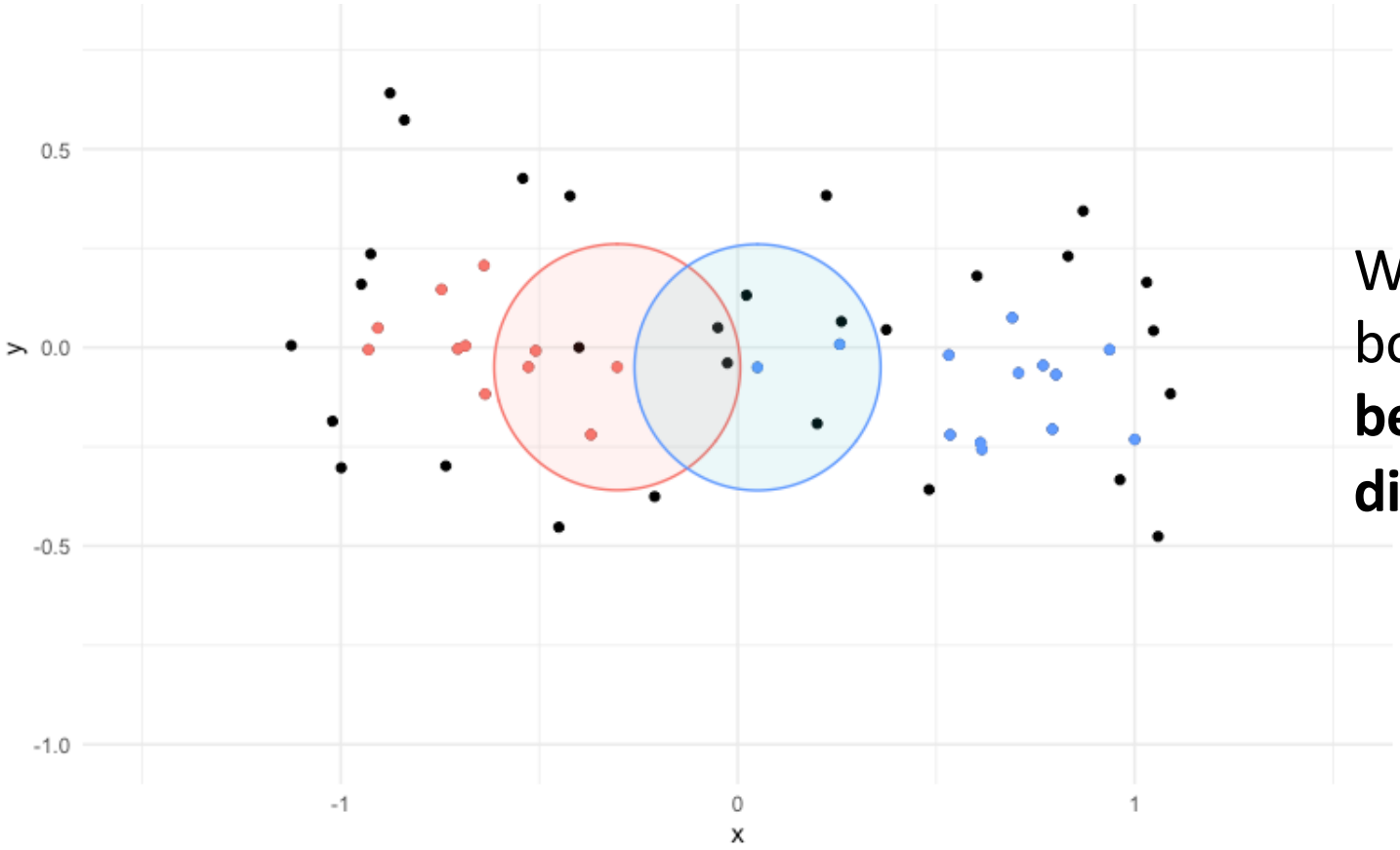
DBSCAN in **tidyclust**

```
db_clust_spec <- db_clust(
  mode = "partition",
  engine = "dbscan",
  radius = NULL,
  min_points = NULL
)
```

Where…

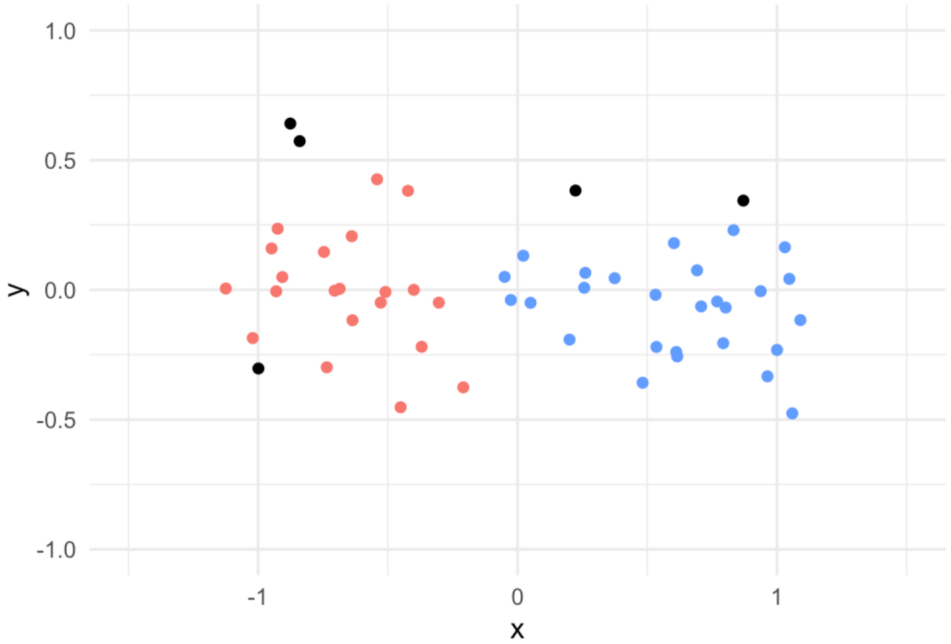
- `radius` is the radius of the ϵ-neighborhood used to identify core points
- `min_points` is the density threshold used to identify core points

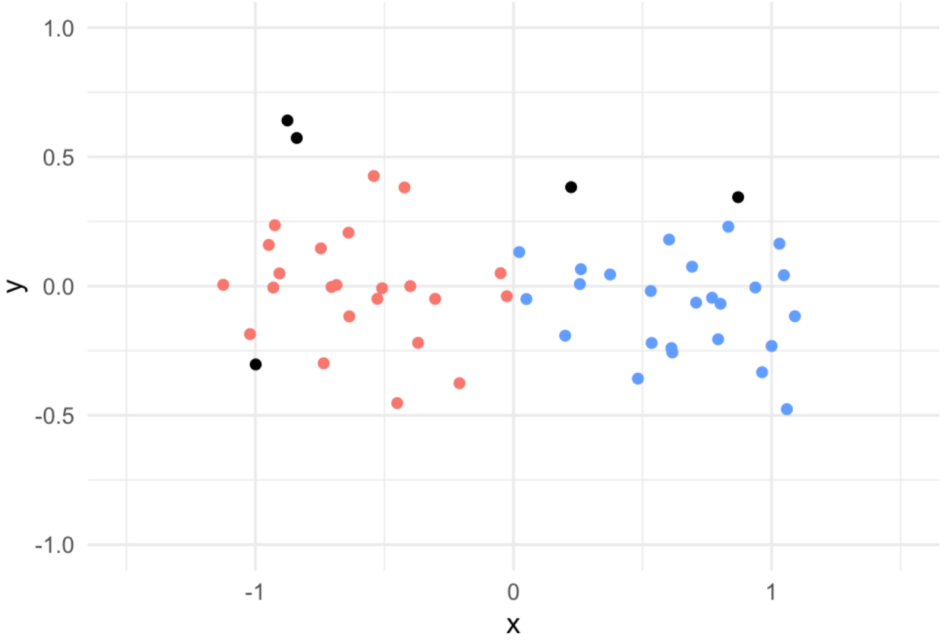
How `db_clust()` fits differently than `dbscan()`



What happens when a border point lies **between** core points in **different clusters**?

`dbscan()` results can differ depending on the order the data is processed

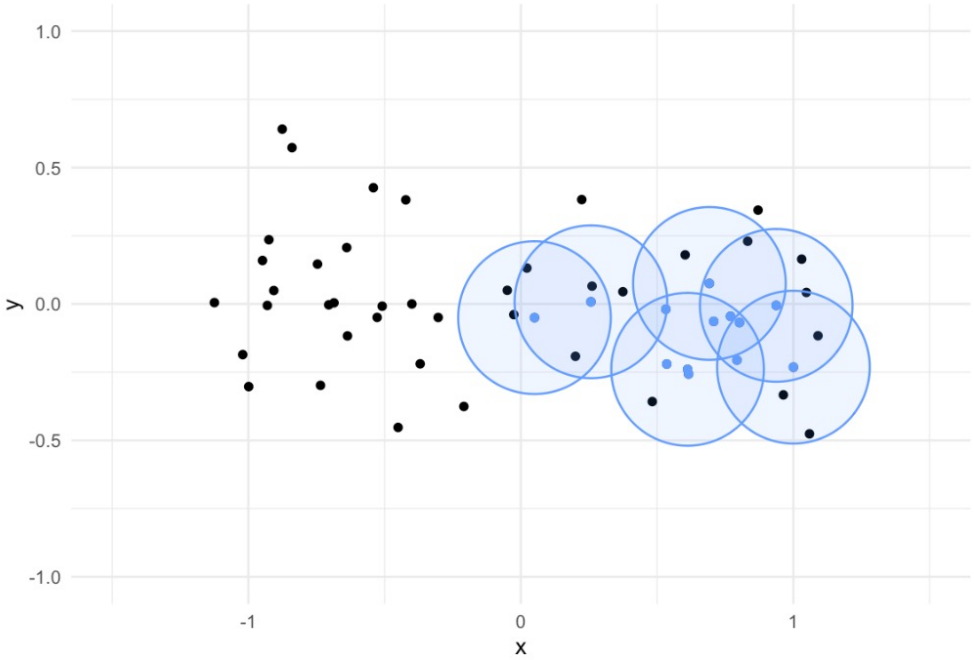


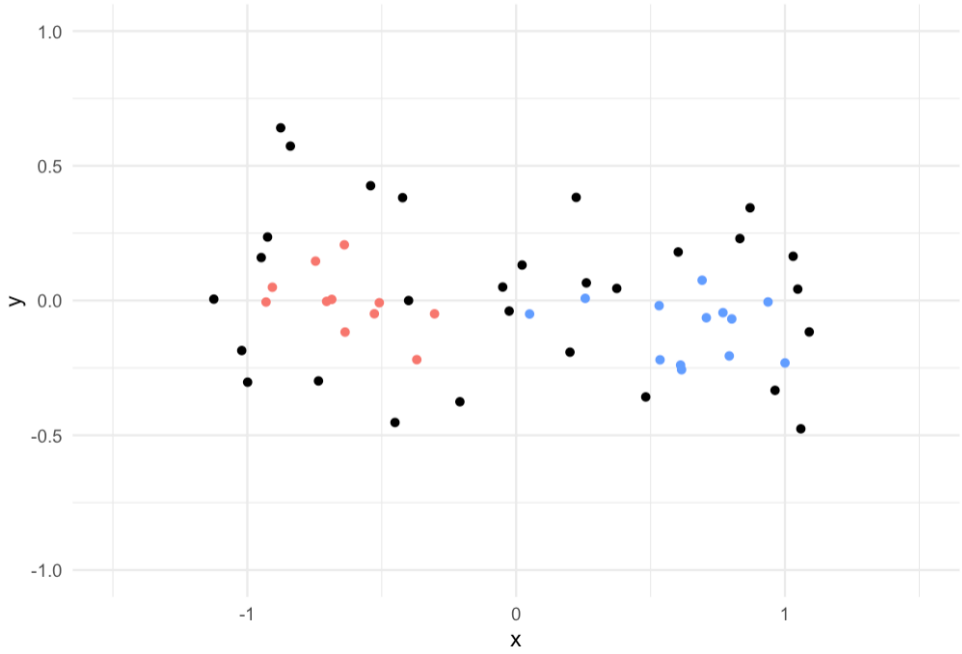


`db_clust()` Fitting Process

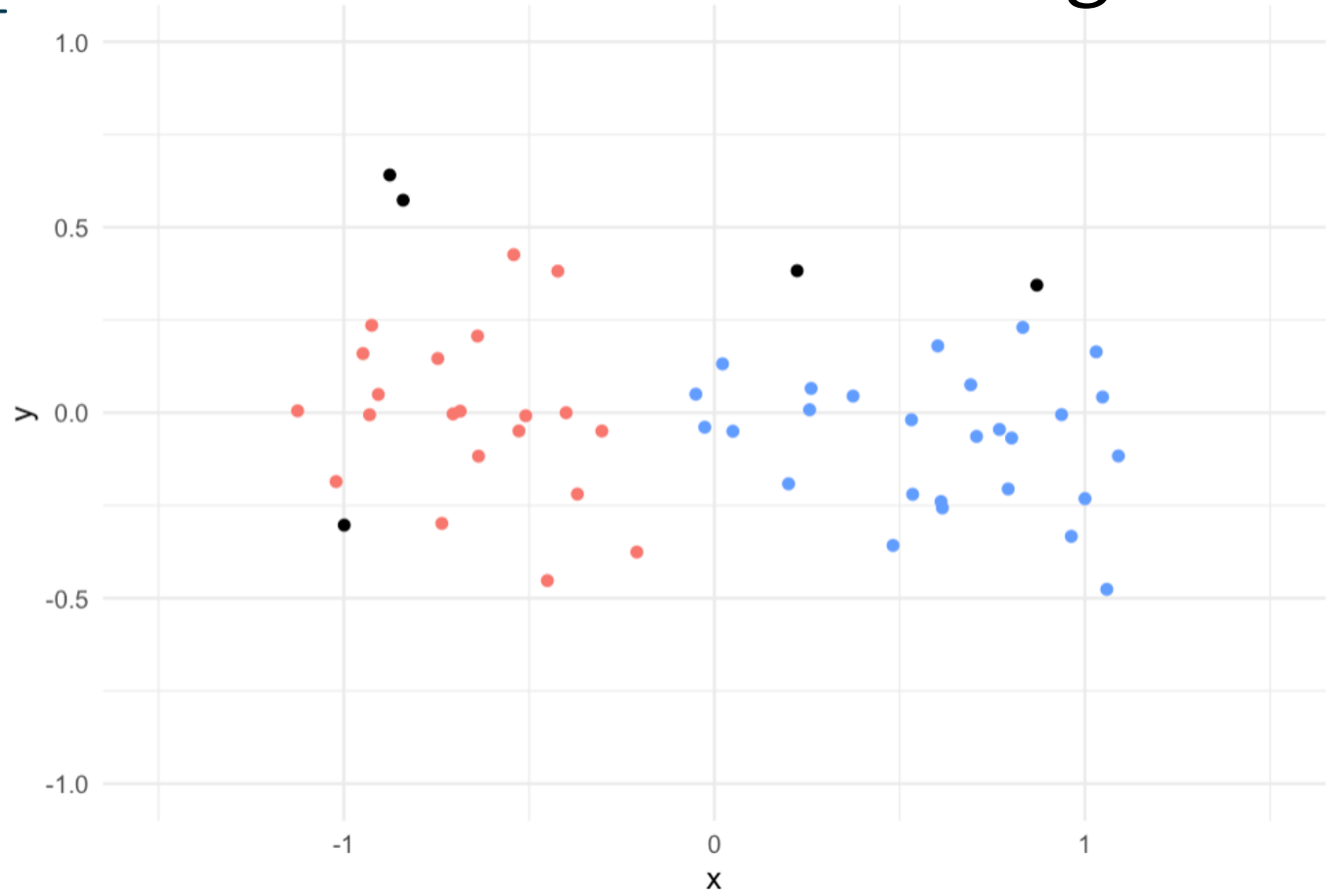
1. Core point discovery
2. Cluster Formation
3. Recursive Cluster building
   - Only expand clusters to other **core points**
   - **Wait** to assign **border points** until all core points have been found
4. Assign clusters to border points based on **nearest core point**

`db_clust()` Recursive Cluster Building

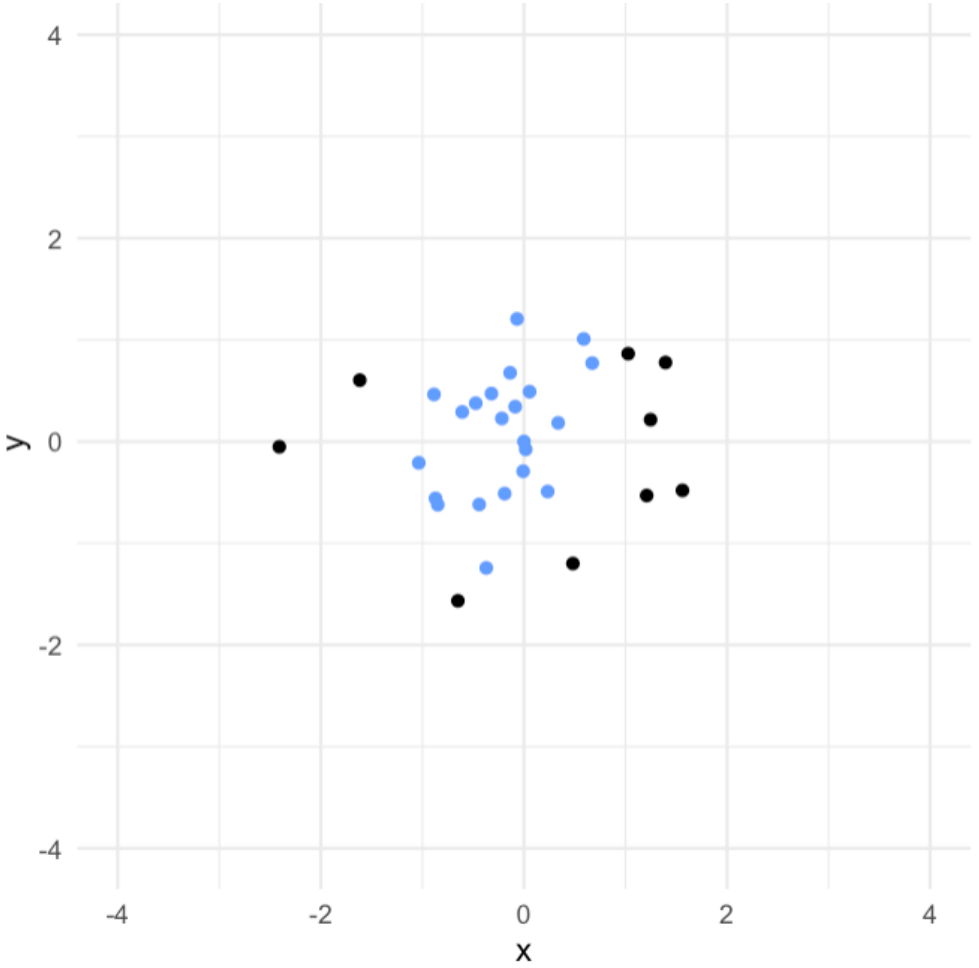




`db_clust()` Final Cluster Assignments



How `db_clust()` predicts differently than `dbscan()`



How `db_clust()` predicts differently than `dbscan()`

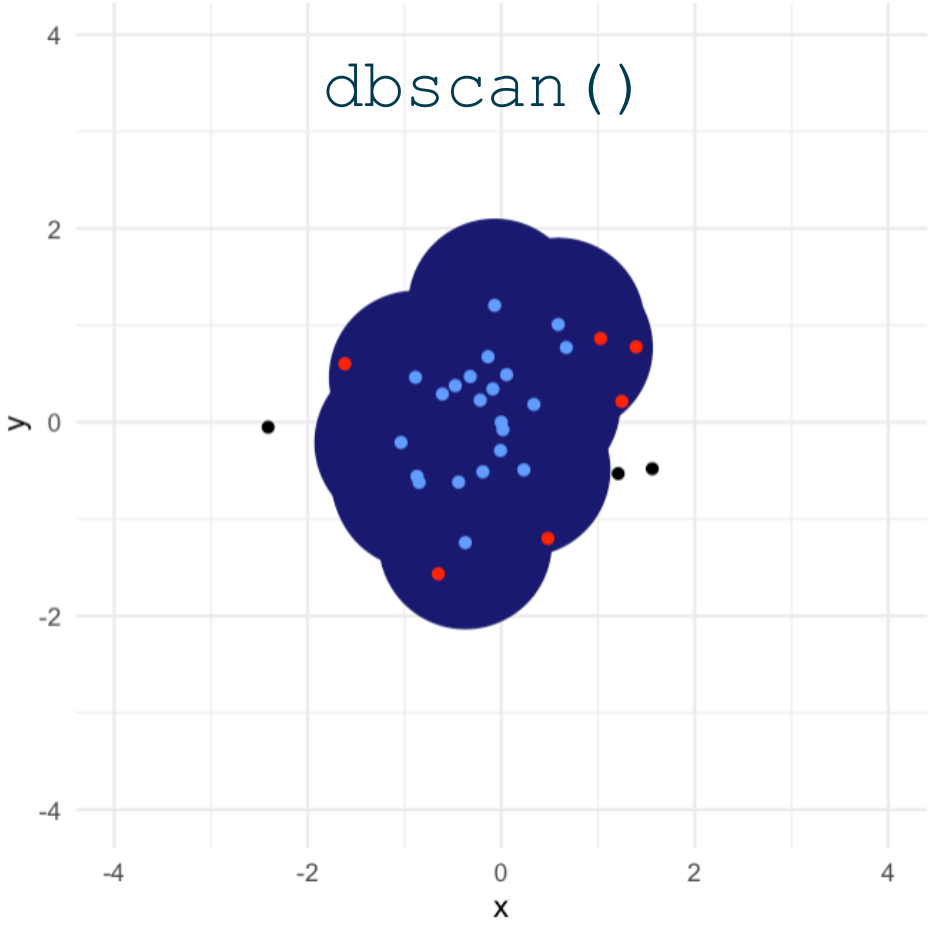
```
1    predict(dbscan_fit, data, newdata)
```

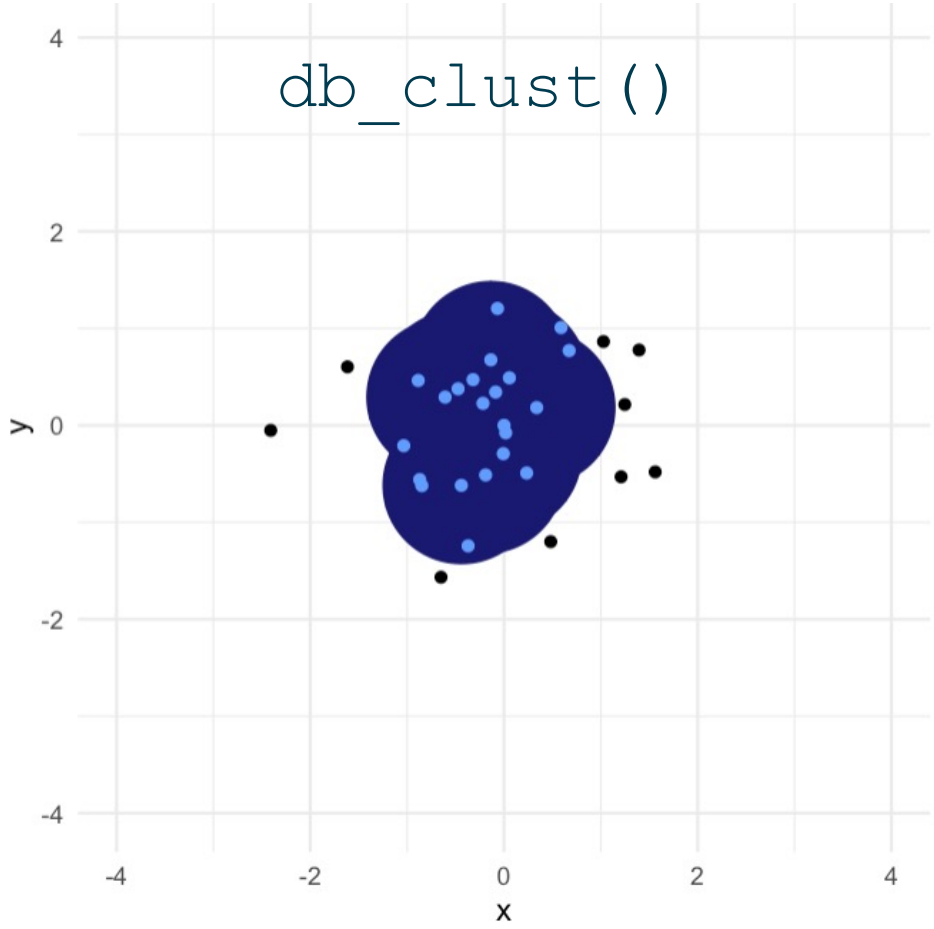
For `dbscan()`, a new observation will be predicted to a cluster if it lies within the **ϵ-neighborhood** of a **any point** in a cluster

```
1    predict(db_clust_fit, new_data)
```

For `db_clust()`, a new observation will be predicted to a cluster if it lies **within** the **ϵ-neighborhood** of a **core point**

Prediction Comparison





Model-based Clustering with GMMs

Gaussian Mixture Models (GMMs)

- Assumes the data is composed of **clusters** which are each **generated** from separate **multivariate Gaussian distributions**

$$f(x) = \sum_{g=1}^{G} p_g \Phi\left(x|\mu_g, \Sigma_g\right)$$

Where…

- p_g is the weight for the gth Guassian component
- μ_g is the mean vector for the gth Gaussian component
- Σ_g is the variance covariance matrix for the gth Gaussian component

Gaussian Mixture Models (cont.)

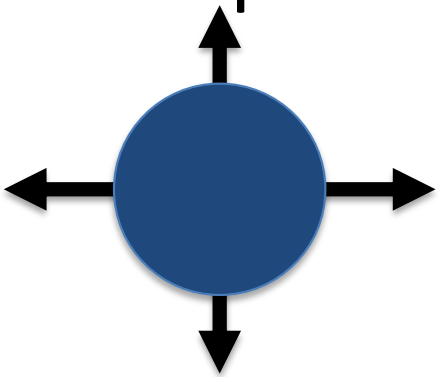
- Model-based methods can provide soft clustering labels
  - The estimated pdfs can be used to estimate the **probability** an observation belongs to each cluster

$$\hat{\gamma}_{ic} = \frac{\hat{p}_g f\left(x_i | \hat{\theta}_c\right)}{\sum_{j=1}^{C} \hat{p}_j f\left(x_i | \hat{\theta}_j\right)}$$

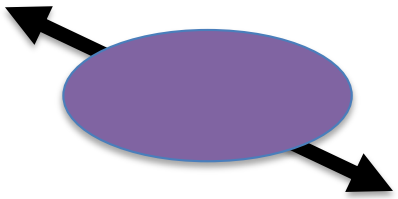
Importance of Variance-Covariance Matrices

For Gaussian distributions, $\boldsymbol{\Sigma}$ controls the distribution…

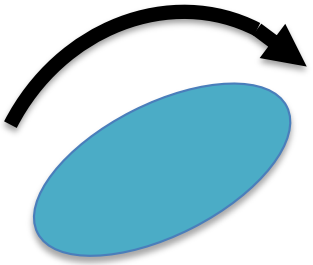
Shape



Size

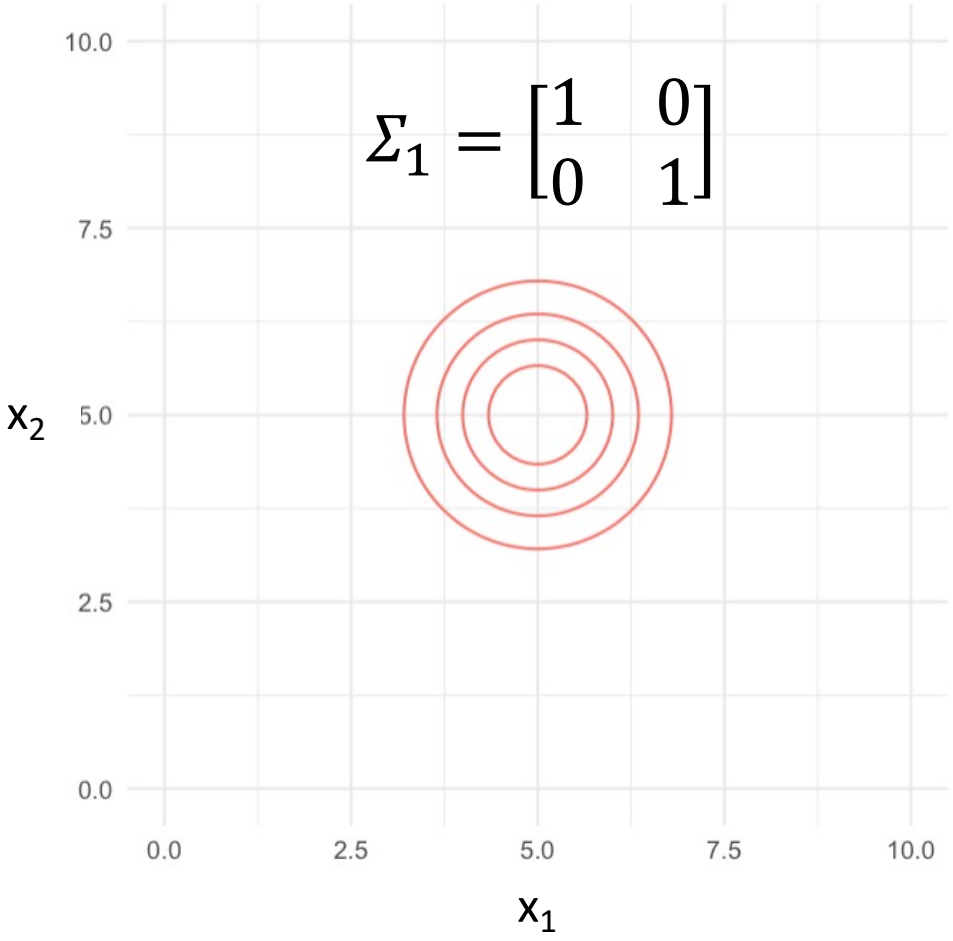


Orientation

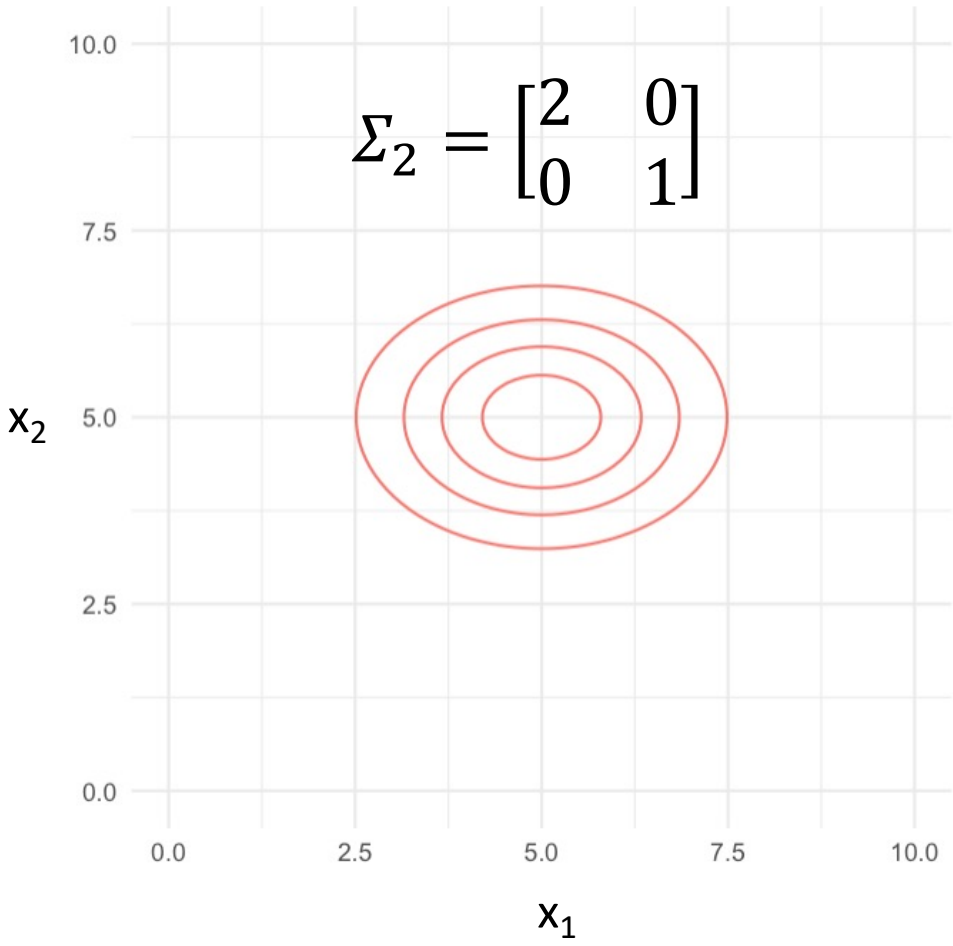


Shape

$$\Sigma_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

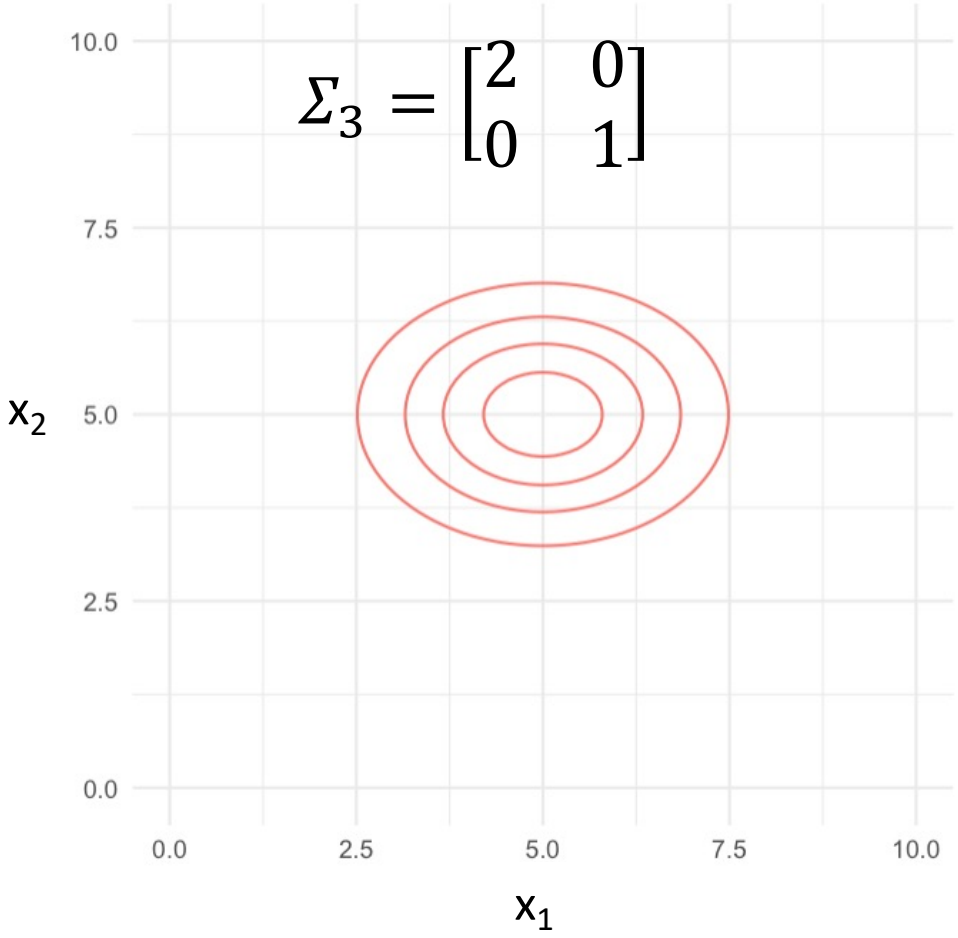


$$\Sigma_2 = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

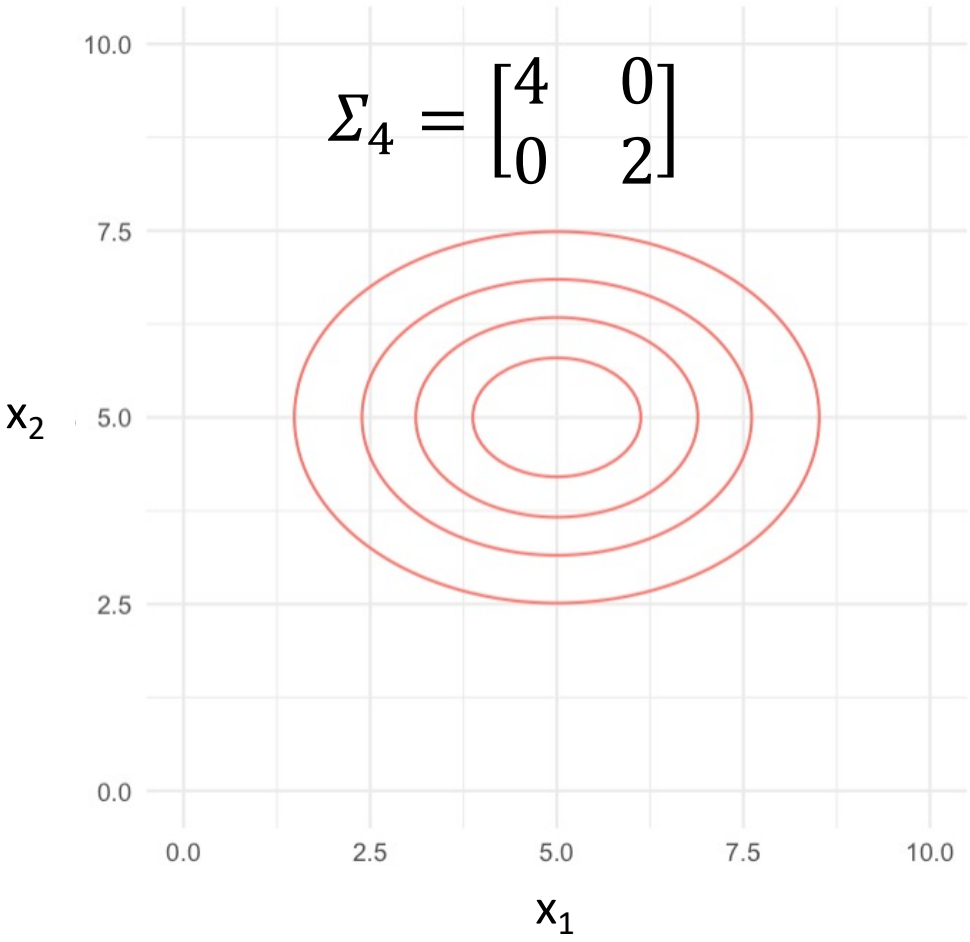


Size

$$\Sigma_3 = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

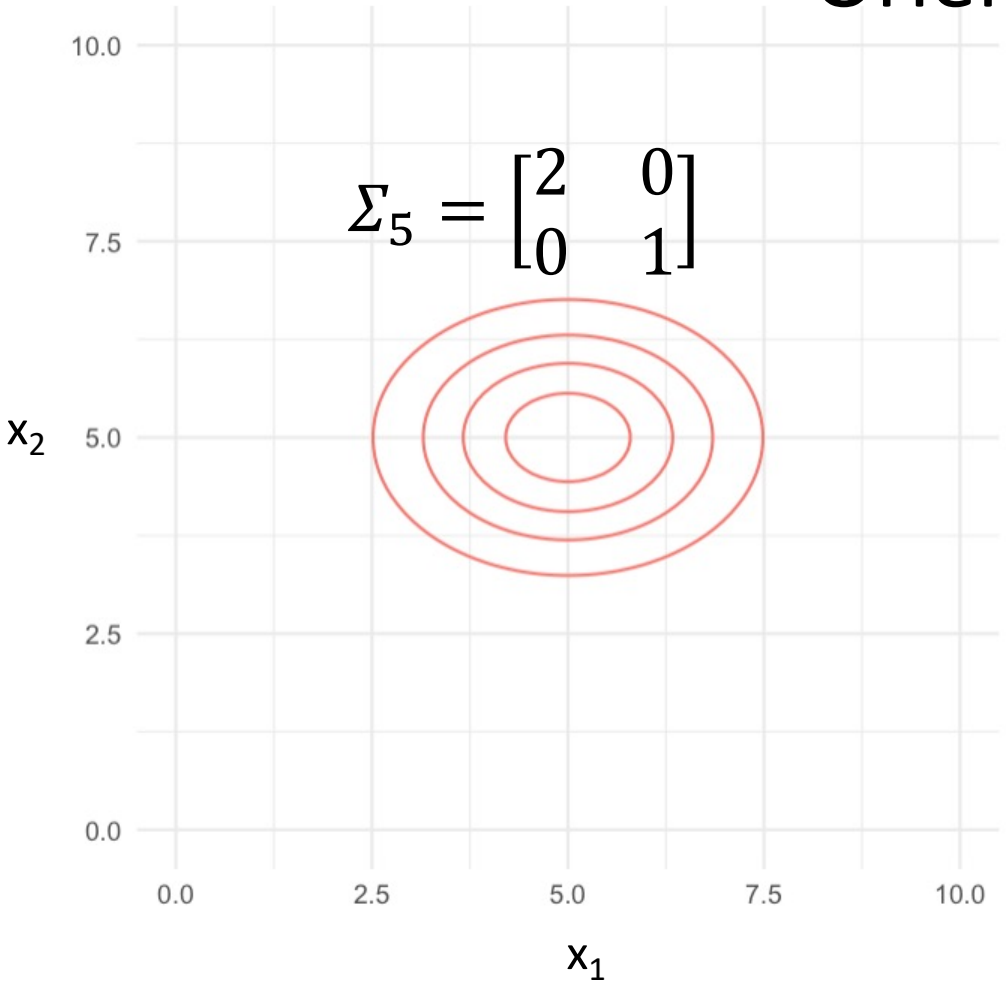


$$\Sigma_4 = \begin{bmatrix} 4 & 0 \\ 0 & 2 \end{bmatrix}$$



Orientation

$$\Sigma_5 = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$



$$\Sigma_6 = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$$

GMM Model Specifications

- **14** different possible **model specifications**
  - Circular clusters or ellipsoidal?
  - Zero or non-zero covariances?
  - Should clusters have the same shape/size/orientation?

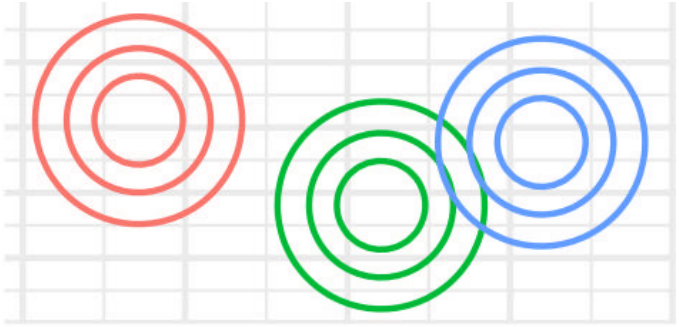
- Commonly referenced with 3-character **model names**

  Examples:

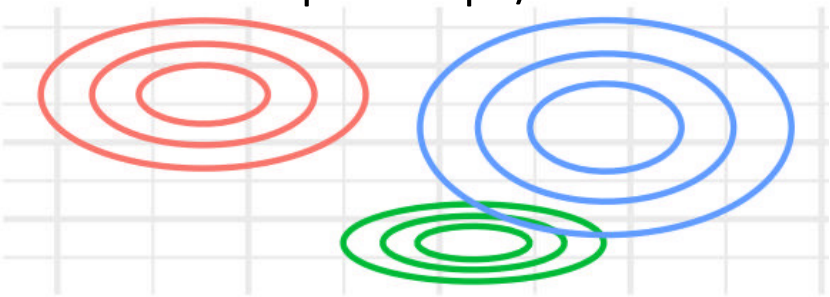
  EII, VII, EEI, EVI, EVE, EVV, VVV

GMM Model Specifications (cont.)

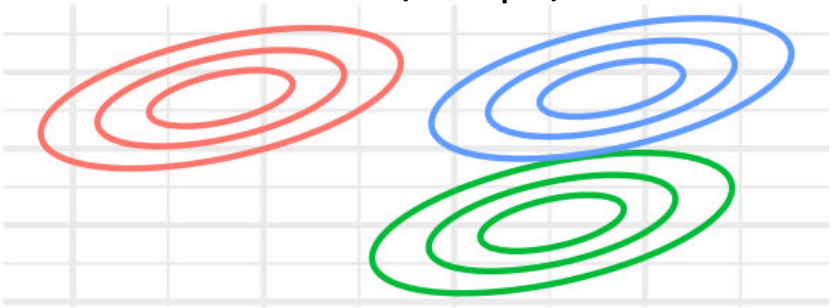
EII – Circular, Equal Size



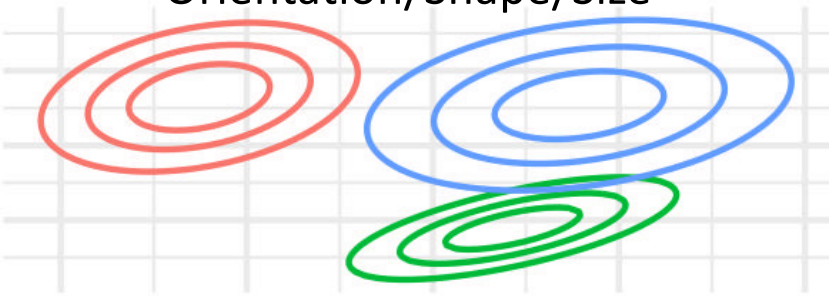
EEI – Ellipse, Zero Covariance, Unequal Shape/Size



EEE – Ellipse, Equal Orientation/Shape/Size



VVV – Ellipse, Unequal Orientation/Shape/Size



GMMs with the **mclust** package

```
1 Mclust(data, G = 1:9, modelNames = c("EII", "VII", ..., "VVV"), ...)
```

Where…

- `data` is the data to fit a GMM to
- `G` is number of Gaussians to fit
- `modelNames` is a vector containing the model names to fit

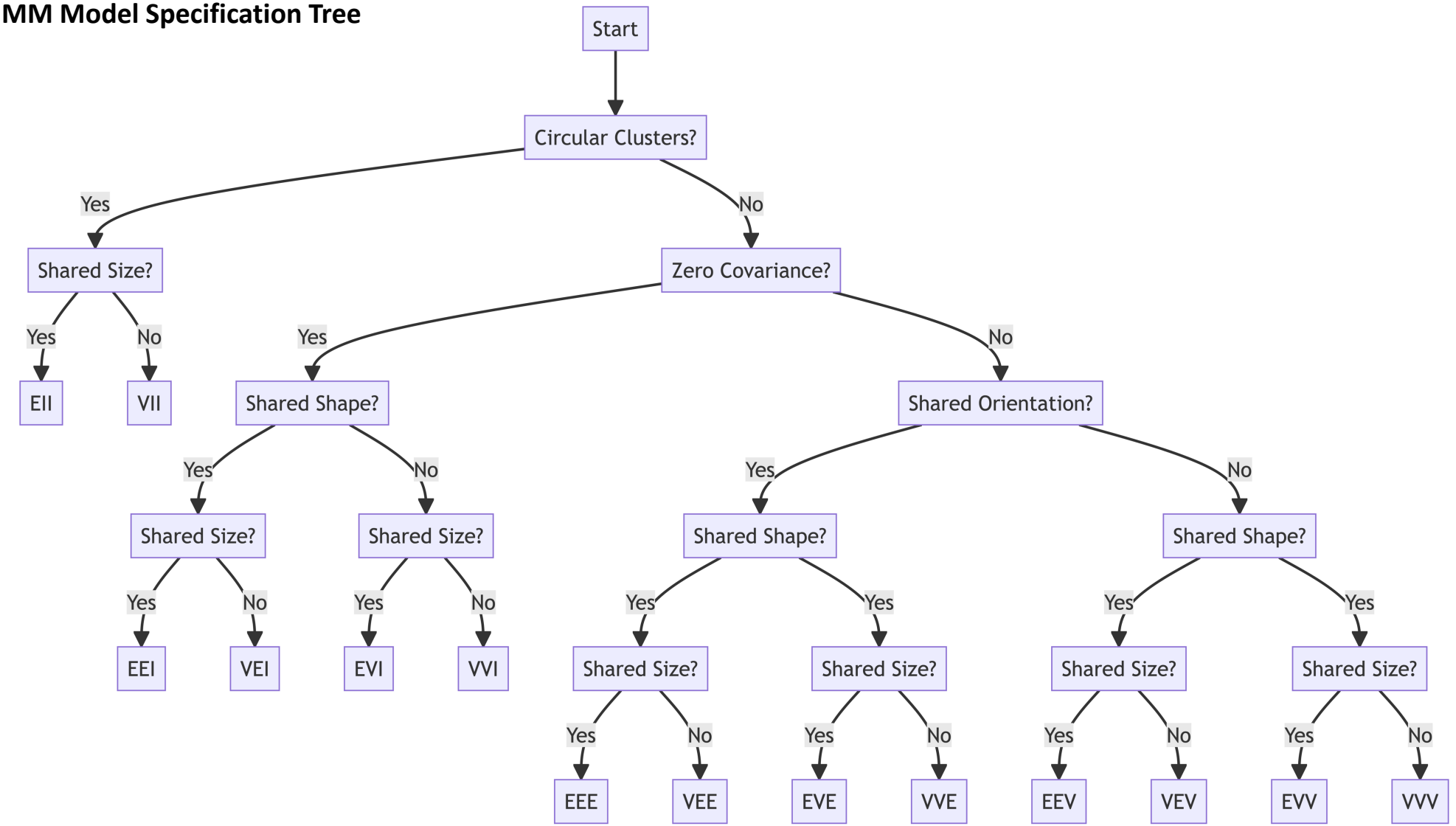
`Mclust` will fit all combinations of `G` and `modelNames` and return best result based on BIC

Applying tidy principles to GMM Model Names

- Make model **arguments** more **self-documenting**
  - Argument names guide user when selecting values

- Separation of **fitting** and **tuning**
  - Fit a single model with `fit()`
  - Use tuning when comparing multiple specifications

GMM Model Specification Tree



GMMs via **tidyclust**

```
gm_clust_spec <- gm_clust(num_clusters,
                          circular = TRUE,
                          zero_covariance = TRUE,
                          shared_orientation = TRUE,
                          shared_size = TRUE) %>%
  set_engine("mclust") %>%
  set_mode("parition")
```

Where…

- `circular` controls whether fitted clusters will be circular or ellipsoidal
- `zero_covariance` controls whether clusters will have zero or non-zero covariances
- `shared_{}` controls whether clusters will have a shared shape, size, and orientation

`gm_clust()` Arguments

All arguments default to TRUE

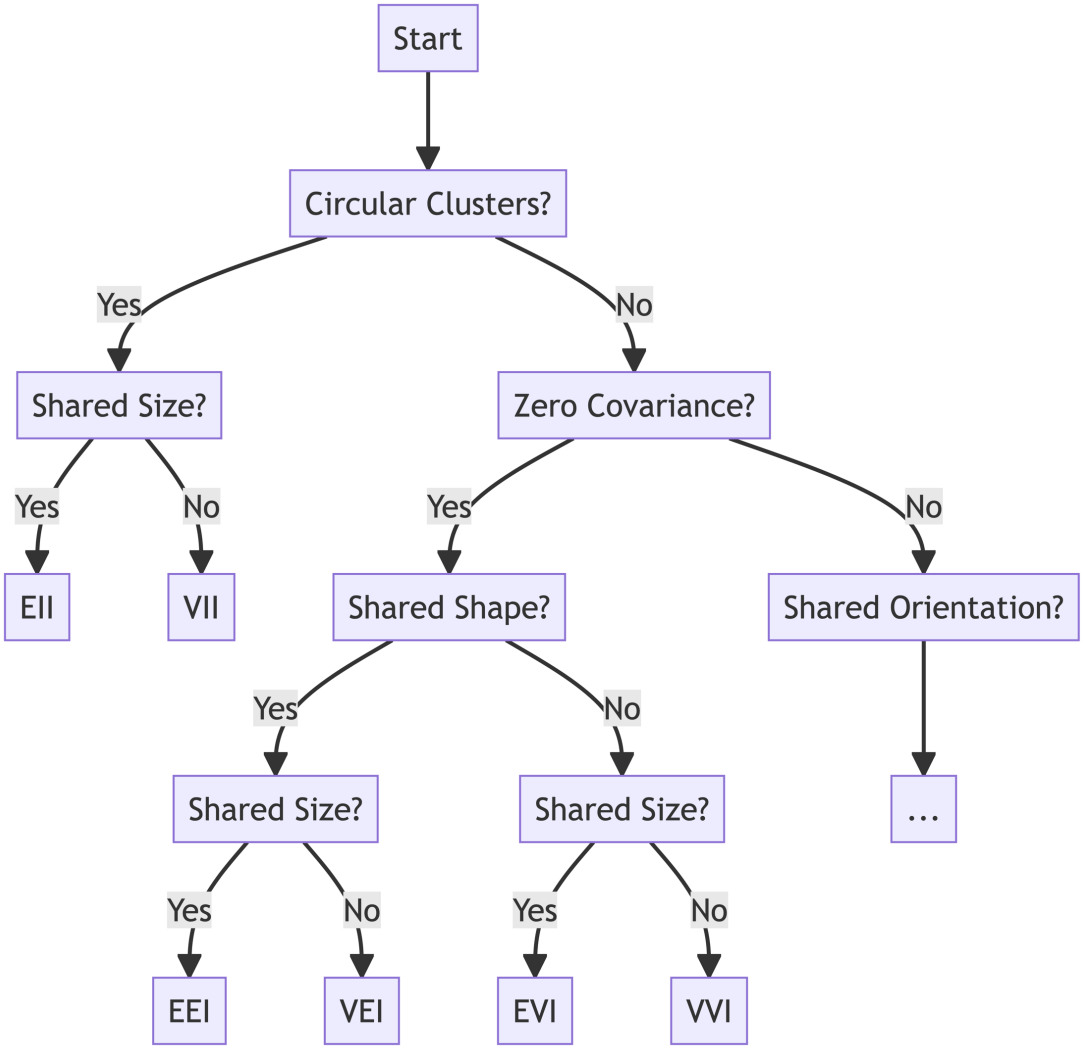
- Arguments are named such that **TRUE** means **constraining** the model to be more **simple**
- Reduces the number of parameters than need to be estimated
- Not all datasets will be able to estimate the parameters required for the most complex model

`gm_clust()` Arguments (cont.)

- 5 TRUE/FALSE arguments
  - $2^5 = 32$ argument combinations but 14 model specifications?

- Ex. Circular clusters
  - Automatically have same shape and zero covariances!

GMM Model Specification Tree



Fitting with `gm_clust()`

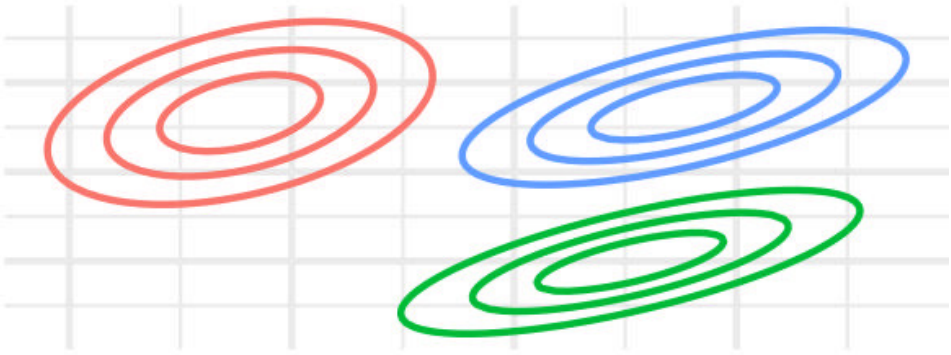
```
gm_clust_fit <- gm_clust_spec %>%
  fit(~ predictor1 + predictor2 + ..., data)
```

```
gm_recipe <- recipe(. ~ predictor1 + predictor2 + ..., data) %>%
  step_naomit(...)

gm_workflow <- workflow() %>%
  add_model(gm_clust_spec) %>%
  add_recipe(gm_recipe)

gm_clust_fit <- gm_workflow %>%
  fit()
```

Predicting with `gm_clust()`

```
predict(gm_clust_fit, new_data)
```

New observations will be **predicted** to belong to the cluster in which they have the **largest probability** of belonging to



Thank you!

Especially to Dr. Bodwin!!!