# Dataset em Julia

February 12, 2016

# 1 Trabalho de Implementação

## 1.1 INF2912 - Otimização Combinatória

### 1.1.1 Prof. Marcus Vinicius Soledade Poggi de Aragão

### 1.1.2 2015-2

### 1.1.3 Ciro Cavani

**BigData / Globo.com** Algoritmos de clusterização.

## 1.2 Conteúdo

Esse notebook tem as seguintes seções:

1. Generator

   Algoritmo para gerar dataset baseado no código Python feito pelo Poggi.

   Na descrição do trabalho está definido como o dataset é formado. Cada grupo tem um conjunto de features próprias com uma probabilidade de ativação maior do que as features livres.

2. Visualização

   Formas de apresentar o dataset na forma de gráfico bidimensional.

   Foram testadas três algoritmos: norma das partes superior e inferior do vetor de features (recomendado em aula); norma das features do grupo contra as features livres, e; Principal Component Analysis (PCA) para redução de dimensões.

   Os dois primeiros não apresentam muita diferenciação entre os pontos dos grupos. O PCA funciona bem (boa separação) com 3 ou 4 grupos, mas fica com sobreposição para 5+.

3. Avaliação

   Métricas para avaliação de algoritmos de clusterização.

   É implementado um algoritmo de clusterização aleatório ponderado. A partir desse algoritmo, é calculada a matriz de confusão, Accuracy, Precision, Recall e etc.

4. Exportação

   Geração de datasets a serem usados para o desenvolvimento dos algoritmos desse trabalho.

## 1.3 1. Generator

Problema:
Propor um classificador que identifique o grupo de cada objeto.
Dados:

- $g$: número de grupos diferentes

- $n$: número de objetos (não necessariamente diferentes)
- $n_{min}$: número mínimo de objetos em um grupo
- $n_{max}$: número máximo de objetos em um grupo

Para cada Objeto:

- $c$: número de características binárias
- $c_y$: número de características de um determinado grupo
- $c_n$: número de características dos demais grupos ($c_n = c_y(g-1)$)
- $p$: probabilidade de ativação das características de um grupo ($p > 0.5$)
- $1-p$: probabilidade de ativação das características dos demais grupos
- $p' = 0.5$: probabilidade de ativação das características que não são de qualquer grupo
- (as características de cada grupo não tem interseção)

```
In [1]: "gera a distribuição de objetos para os grupos"
        function group_size(g, n, n_min, n_max)
            num_g = Array(Int, g)
            sum = 0
            for i=1:g
                num_g[i] = rand(n_min:n_max)
                sum += num_g[i]
            end
            correct = n / sum
            sum = 0
            for i=1:g
                num_g[i] = round(Int, num_g[i] * correct)
                sum += num_g[i]
            end
            if sum < n
                num_g[g] += 1
            end
            num_g
        end
```

```
Out[1]: group_size (generic function with 1 method)
```

```
In [2]: let n = 20,
            n_min = 2,
            n_max = 5,
            g = 5

            group_size(g, n, n_min, n_max)
        end
```

```
Out[2]: 5-element Array{Int64,1}:
         6
         3
         4
         3
         4
```

```
In [3]: "máscara de características para cada grupo sem interseção"
        function group_mask(g, c, c_y)
            char_g = fill(-1, c)
            index = 1
            for i=1:g, j=1:c_y
```

```
                char_g[index] = i
                index += 1
            end
            char_g
        end
```

```
Out[3]: group_mask (generic function with 1 method)
```

```
In [4]: let g = 5,
            c = 16,
            c_y = 3

            group_mask(g, c, c_y)
        end
```

```
Out[4]: 16-element Array{Int64,1}:
          1
          1
          1
          2
          2
          2
          3
          3
          3
          4
          4
          4
          5
          5
          5
         -1
```

```
In [5]: """gera objetos para grupos seguindo a distribuição num_g,
        a máscara char_g e a probabilidade p de ativação"""
        function generate_data(num_g, char_g, p)
            data = Array(Tuple{Array{Int,1},Int}, 0)
            for i=1:length(num_g),j=1:num_g[i]
                vect = zeros(Int, length(char_g))
                for k=1:length(vect)
                    if char_g[k] == i
                        vect[k] = rand() < p ? 1 : 0
                    elseif char_g[k] != -1
                        vect[k] = rand() < 1 - p ? 1 : 0
                    else
                        vect[k] = rand() < 0.5 ? 1 : 0
                    end
                end
                push!(data, (vect, i))
            end
            data
        end
```

```
Out[5]: generate_data (generic function with 1 method)
```

```
In [6]: "gerador de instâncias para o problema de clusterização"
        function instance_generator(n, c, c_y, p, g, n_min, n_max)
```

```
            if c < g * c_y
                error("c_y too big")
            end

            num_g = group_size(g, n, n_min, n_max)
            char_g = group_mask(g, c, c_y)
            data = generate_data(num_g, char_g, p)
            data
        end
```

Out[6]:
```
instance_generator (generic function with 1 method)
```

In [7]:
```
let n = 20,
    n_min = 2,
    n_max = 5,
    g = 5,
    c = 16,
    c_y = 3,
    p = 0.8

    instance_generator(n, c, c_y, p, g, n_min, n_max)
end
```

Out[7]:
```
20-element Array{Tuple{Array{Int64,1},Int64},1}:
 ([1,1,0,0,0,0,0,0,0,0,0,0,0,1,0,1],1)
 ([0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,1],1)
 ([1,1,1,0,0,1,0,1,0,0,1,0,0,0,0,0],1)
 ([1,1,1,0,0,0,0,1,0,0,1,0,0,1,0,0],1)
 ([0,0,0,1,1,1,1,0,0,0,0,0,0,1,0,0],2)
 ([1,0,1,1,1,1,0,1,1,0,0,0,0,0,0,1],2)
 ([0,1,0,0,0,0,1,1,1,0,0,1,0,0,0,1],3)
 ([0,1,0,0,0,1,1,1,1,0,0,0,0,0,0,0],3)
 ([0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,1],3)
 ([1,0,0,1,0,0,1,0,1,0,1,0,0,0,0,0],3)
 ([0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0],3)
 ([0,0,0,0,1,0,0,0,0,1,1,1,0,0,1,1],4)
 ([0,0,0,0,0,0,0,0,0,1,0,1,0,0,1,1],4)
 ([0,0,0,0,0,0,0,0,1,1,1,1,0,0,1,0],4)
 ([0,0,0,1,0,0,0,0,0,0,1,1,1,0,0,0],4)
 ([0,0,0,1,0,0,1,0,0,0,0,0,0,1,1,1],5)
 ([1,0,0,0,1,0,0,0,0,0,0,0,1,1,1,0],5)
 ([1,0,0,0,0,1,0,0,0,0,1,0,1,1,1,1],5)
 ([0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,0],5)
 ([0,0,0,0,0,1,0,0,0,0,0,0,1,1,0,0],5)
```

In [8]:
```
type Dataset
    groups::Int
    features::Int
    slot::Int
    activation_p::Float64
    size::Int
    size_min::Int
    size_max::Int
    data::Array{Tuple{Array{Int,1}, Int}, 1}
    Dataset(; groups=3, size=10000, size_min=0, size_max=0, features=200, slot=40, activation_p=
        if size < 10
            error("minimum 10")
        end
        if groups > size
            error("too many groups")
        end
        if features < groups * slot
            error("slot too big")
        end

        if size_max == 0
            size_max = 2 * round(Int, size / groups)
        end
        if size_min == 0
            size_min = round(Int, size_max / 10)
        end
        if size_max * groups < size
            error("size_max too tight")
        end

        data = instance_generator(size, features, slot, activation_p, groups, size_min, size_ma
        shuffle!(data)

        new(groups, features, slot, activation_p, size, size_min, size_max, data)
    end
end

data(ds, k) = filter(t -> t[2] == k, ds.data)
count(ds, k) = length(data(ds, k))

"Sumário do Dataset"
function summary(io::IO, ds::Dataset)
    println(io, "Number of Groups: ", ds.groups)
    println(io, "Number of Features: ", ds.features)
    println(io, "Number of Features (group): ", ds.slot)
    println(io, "Probability of Activation: ", ds.activation_p)
    println(io, "Number of Objects (total): ", ds.size)
    println(io, "Number of Objects per Group (min): ", ds.size_min)
    println(io, "Number of Objects per Group (max): ", ds.size_max)

    for k=1:ds.groups
        println(io, "Number of Objects in ", k, ": ", count(ds, k))
    end
end

"Sumário do Dataset"
summary(ds::Dataset) = summary(STDOUT, ds)

let _dataset = Dataset()
    summary(_dataset)
    sleep(0.2)
end
```

```
Number of Groups: 3
Number of Features: 200
Number of Features (group): 40
Probability of Activation: 0.8
Number of Objects (total): 10000
Number of Objects per Group (min): 667
Number of Objects per Group (max): 6666
Number of Objects in 1: 2134
Number of Objects in 2: 3698
Number of Objects in 3: 4168
```

## 1.4 2. Visualization

### 1.4.1 Gadfly

http://gadflyjl.org/

Gadfly is a system for plotting and visualization based largely on Hadley Wickhams's ggplot2 for R, and Leland Wilkinson's book The Grammar of Graphics.

```
In [9]: if Pkg.installed("Gadfly") === nothing
            println("Installing Gadfly...")
            Pkg.add("Gadfly")
            Pkg.add("Cairo")
        end
```

```
In [10]: using Gadfly
         set_default_plot_size(24cm, 12cm)
```

```
In [11]: dataset = Dataset()
```

```
Out[11]: Dataset(3,200,40,0.8,10000,667,6666,[([1,1,1,1,1,1,1,0,1,1  ...  1,0,0,1,1,1,1,1,0,0],1),([0,1
```

```
In [12]: function halfmask(n)
             mask = zeros(n)
             middle = round(Int, n / 2)
             mask[1:middle] = 1
             mask
         end

         halfmask(10)
```

```
Out[12]: 10-element Array{Float64,1}:
          1.0
          1.0
          1.0
          1.0
          1.0
          0.0
          0.0
          0.0
          0.0
          0.0
```

```
In [13]: reversemask(mask) = ones(mask) - mask

         let mask = halfmask(10)
             reversemask(mask)
         end
```

```
Out[13]: 10-element Array{Float64,1}:
          0.0
          0.0
          0.0
          0.0
          0.0
          1.0
          1.0
          1.0
          1.0
          1.0
```

```
In [14]: function halfmasks(n)
             x = halfmask(n)
             y = reversemask(x)
             (x, y)
         end

         let a = rand(10),
             masks = halfmasks(10)
             (masks[1] .* a, masks[2] .* a)
         end
```

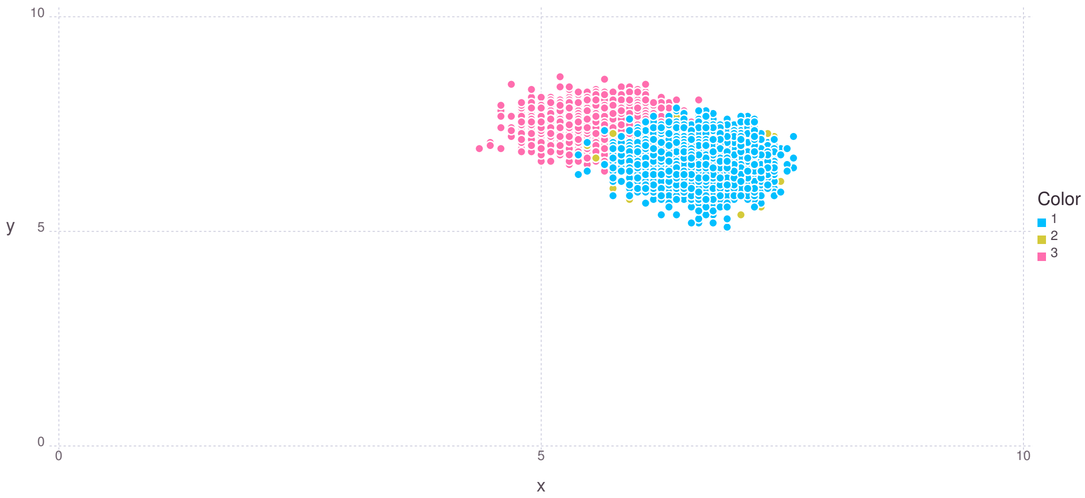
```
Out[14]: ([0.38942646429232486,0.8126841704018781,0.14198783899704548,0.6565786132434721,0.1748763340633
```

```
In [15]: function reduce2d(data, masks)
             x = map(t -> norm(masks[1] .* t[1]), data)
             y = map(t -> norm(masks[2] .* t[1]), data)
             k = map(t -> string(t[2]), data)
             x, y, k
         end

         function plothalf(dataset)
             masks = halfmasks(dataset.features)

             g = Array(Layer, 0)

             for k=1:dataset.groups
                 kdata = data(dataset, k)
                 x, y, color = reduce2d(kdata, masks)
                 push!(g, layer(x=x, y=y, color=color, Geom.point)...)
             end

             plot(g, Scale.x_continuous(minvalue=0, maxvalue=10), Scale.y_continuous(minvalue=0, maxvalu
         end
```

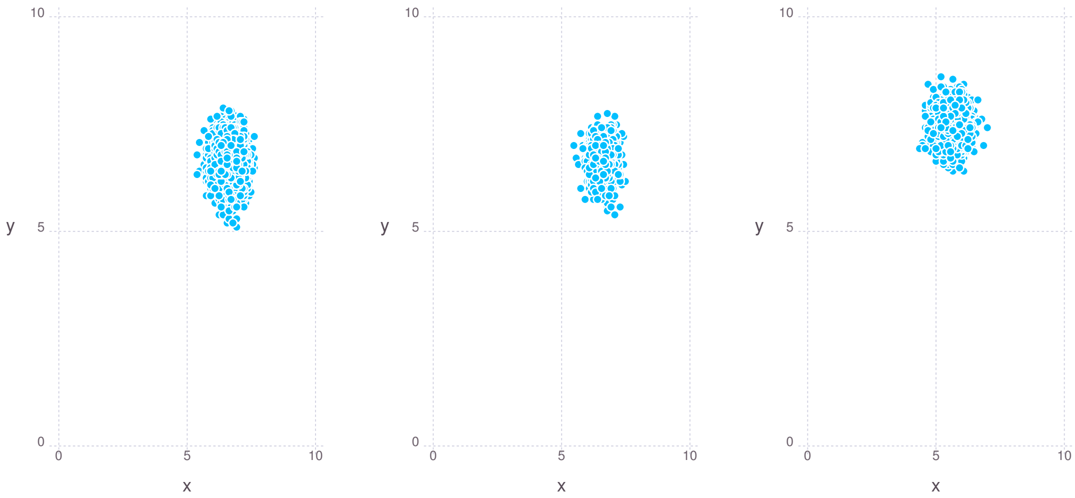
```
Out[15]: plothalf (generic function with 1 method)
```

```
In [16]: plothalf(dataset)
```

```
Out[16]:
```



```
In [17]: function plothalf_multi(dataset)
             masks = halfmasks(dataset.features)

             g = Array(Plot, 0)

             for k=1:dataset.groups
                 kdata = data(dataset, k)
                 x, y, _ = reduce2d(kdata, masks)
                 p = plot(x=x, y=y, Scale.x_continuous(minvalue=0, maxvalue=10), Scale.y_continuous(min
                 push!(g, p)

             end

             hstack(g...)
         end

Out[17]: plothalf_multi (generic function with 1 method)

In [18]: plothalf_multi(dataset)

Out[18]:
```

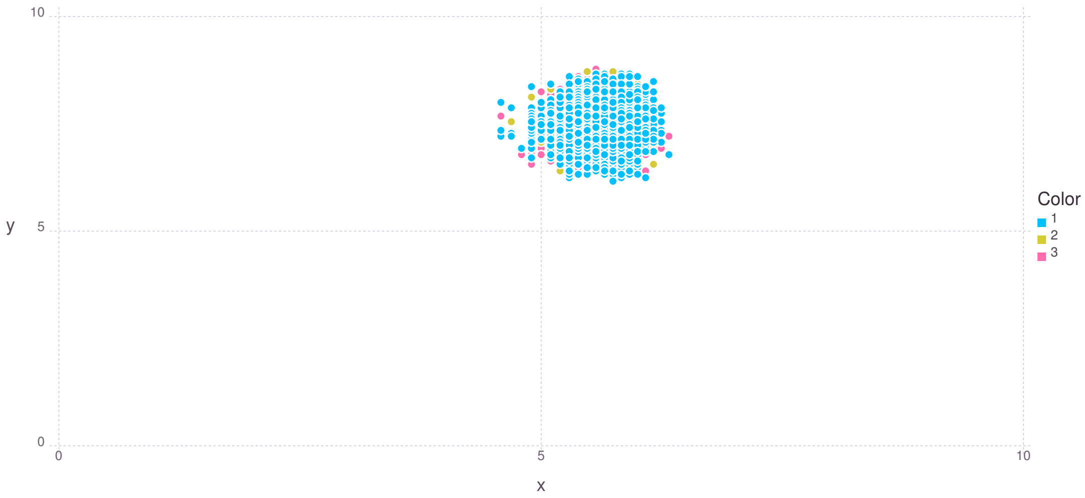


```
In [19]: function featuremask(features, slot, k)
             first = (k - 1) * slot + 1
             last = k * slot
             mask = zeros(features)
             mask[first:last] = 1
             mask
         end

         featuremask(10, 3, 1)

Out[19]: 10-element Array{Float64,1}:
          1.0
          1.0
          1.0
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0

In [20]: function featuremasks(features, slot, k)
             kmask = featuremask(features, slot, k)
             rmask = reversemask(kmask)
             (kmask, rmask)
         end

         let a = rand(10),
             masks = featuremasks(10, 3, 2)
             (masks[1] .* a, masks[2] .* a)
         end

Out[20]: ([0.0,0.0,0.0,0.27244866995210204,0.25232901776981276,0.8907894318757914,0.0,0.0,0.0,0.0],[0.7
In [21]: function plotslot(dataset)
             g = Array(Layer, 0)

             for k=1:dataset.groups
                 masks = featuremasks(dataset.features, dataset.slot, k)
                 kdata = data(dataset, k)
                 x, y, color = reduce2d(kdata, masks)
                 push!(g, layer(x=x, y=y, color=color, Geom.point)...)
             end

             plot(g, Scale.x_continuous(minvalue=0, maxvalue=10), Scale.y_continuous(minvalue=0, maxvalu
         end
```

```
Out[21]: plotslot (generic function with 1 method)
```

```
In [22]: plotslot(dataset)
```

```
Out[22]:
```

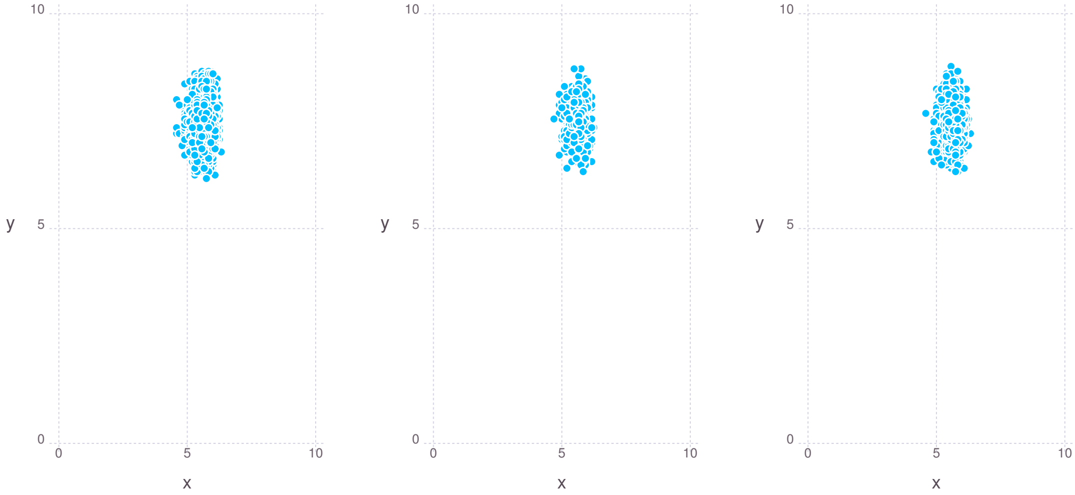


```
In [23]: function plotslot_multi(dataset)
             g = Array(Plot, 0)

             for k=1:dataset.groups
                 masks = featuremasks(dataset.features, dataset.slot, k)
                 kdata = data(dataset, k)
                 x, y, _ = reduce2d(kdata, masks)
                 p = plot(x=x, y=y, Scale.x_continuous(minvalue=0, maxvalue=10), Scale.y_continuous(min
                 push!(g, p)
             end

             hstack(g...)
         end
```

```
Out[23]: plotslot_multi (generic function with 1 method)
```

```
In [24]: plotslot_multi(dataset)
```

```
Out[24]:
```



### 1.4.2 MultivariateStats Package

https://github.com/JuliaStats/MultivariateStats.jl
http://multivariatestatsjl.readthedocs.org/en/latest/index.html
A Julia package for multivariate statistics and data analysis (e.g. dimension reduction)

**Principal Component Analysis (PCA)** http://multivariatestatsjl.readthedocs.org/en/latest/pca.html

```
In [25]: if Pkg.installed("MultivariateStats") === nothing
             println("Installing MultivariateStats...")
             Pkg.add("MultivariateStats")
             Pkg.checkout("MultivariateStats")
         end
```

```
In [26]: vector_matrix(data) = float(hcat(map(first, data)...))

         vector_matrix([([1,2], 1), ([3,4], 2), ([5,6], 3)])
```

```
Out[26]: 2x3 Array{Float64,2}:
          1.0  3.0  5.0
          2.0  4.0  6.0
```

```
In [27]: using MultivariateStats
```

```
In [28]: let
             train = vector_matrix(dataset.data)
             fit(PCA, train; maxoutdim=2)
         end
```

```
Out[28]: PCA(indim = 200, outdim = 2, principalratio = 0.15771)
```

```
In [29]: let
             train = vector_matrix(dataset.data)
             model = fit(PCA, train; maxoutdim=2)

             sample = data(dataset, 1)
             transform(model, vector_matrix(sample))
         end
```

```
Out[29]: 2x6749 Array{Float64,2}:
          -1.96902   -1.50634    -1.2831    ...  -1.62922   -2.32985   -1.89555
          -0.719978  -0.0836953  -0.717166     -0.126463  -0.254988  -0.335478
```

```
In [30]: let train = vector_matrix(dataset.data),
             model = fit(PCA, train; maxoutdim=2)

             sample = data(dataset, 1)
             points = transform(model, vector_matrix(sample))
             vec(points[1,:])
         end
```

```
Out[30]: 6749-element Array{Float64,1}:
          -1.96902
          -1.50634
          -1.2831
          -1.89444
          -1.96776
          -1.20955
          -1.19118
          -1.46827
          -2.08277
          -0.997054
          -1.65846
          -1.26287
          -1.3974
           ⋮
          -1.43349
          -1.26582
          -1.94611
          -1.63743
          -1.59876
          -1.62264
          -1.79522
          -1.25306
          -1.04557
          -1.62922
          -2.32985
          -1.89555
```

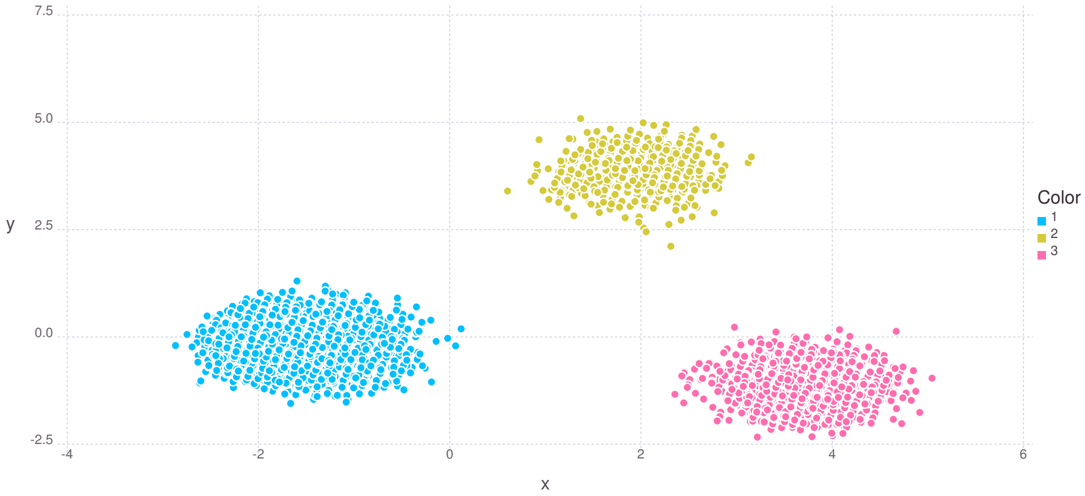
```
In [31]: function plotpca(dataset)
             train = vector_matrix(dataset.data)
             model = fit(PCA, train; maxoutdim=2)

             g = Array(Layer, 0)

             for k=1:dataset.groups
```

```
                kdata = data(dataset, k)
                kpoints = transform(model, vector_matrix(kdata))
                x = vec(kpoints[1,:])
                y = vec(kpoints[2,:])
                color = fill(string(k), size(kpoints, 2))
                push!(g, layer(x=x, y=y, color=color, Geom.point)...)
            end

            plot(g)
        end
```

```
Out[31]: plotpca (generic function with 1 method)
```

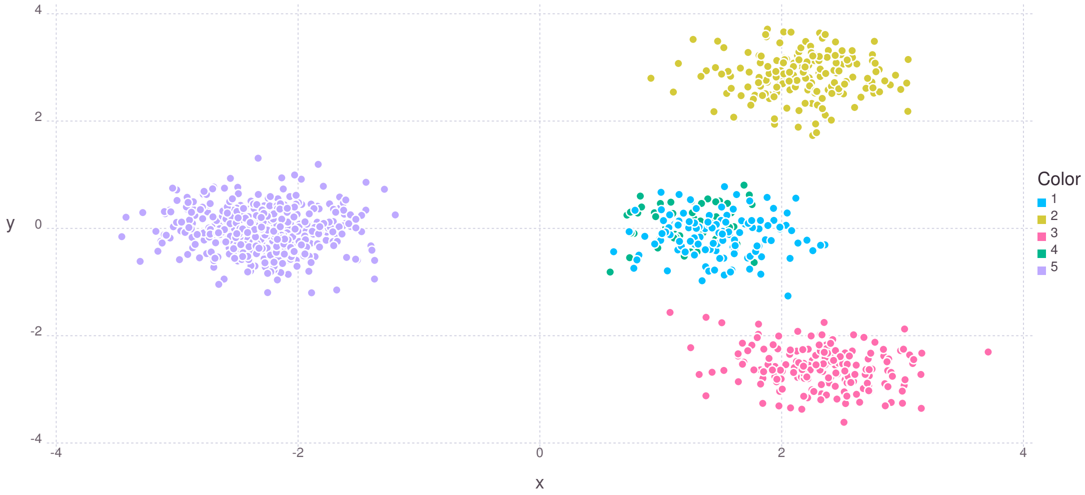
```
In [32]: plotpca(dataset)
```

```
Out[32]:
```



```
In [33]: let _dataset = Dataset(groups=5, size=1000, features=200, slot=40)
             plotpca(_dataset)
         end
```

```
Out[33]:
```



## 1.5 3. Evaluation

```
In [34]: function distribution(dataset)
             groups = Array(Float64, dataset.groups)
             size = 0
             for k=1:dataset.groups
                 size += count(dataset, k)
                 groups[k] = size
             end
             groups /= size
             groups
         end

         distribution(dataset)
```

```
Out[34]: 3-element Array{Float64,1}:
          0.6749
          0.7773
          1.0
```

```
In [35]: function choosek(distribution)
             r = rand()
             for k=1:length(distribution)
                 if r <= distribution[k]
                     return k
                 end
             end
             return 0
         end

         let
             d = [0.3, 0.5, 1.0]
             k = zeros(d)
```

```
            n = 100000
            for _=1:n
                i = choosek(d)
                k[i] += 1
            end
            k / n
        end
```

```
Out[35]: 3-element Array{Float64,1}:
          0.29975
          0.2025
          0.49775
```

```
In [36]: function random_clustering(dataset)
             cdf = distribution(dataset)
             clusters = Array(Int, length(dataset.data))
             for i=1:length(clusters)
                 clusters[i] = choosek(cdf)
             end
             clusters
         end

         random_clustering(dataset)
```

```
Out[36]: 10000-element Array{Int64,1}:
          1
          1
          3
          1
          1
          1
          3
          1
          2
          3
          3
          1
          3
          ⋮
          2
          1
          2
          1
          3
          3
          1
          3
          3
          2
          1
          1
```

### 1.5.1 Confusion Matrix

https://en.wikipedia.org/wiki/Confusion_matrix

```
In [37]: function confusion_matrix(dataset, prediction)
             matrix = zeros(Int, dataset.groups, dataset.groups)
             for p=1:length(prediction)
                 i = dataset.data[p][2]
                 j = prediction[p]
                 matrix[i,j] += 1
             end
             matrix
         end

         let
             prediction = random_clustering(dataset)
             confusion_matrix(dataset, prediction)
         end
```

```
Out[37]: 3x3 Array{Int64,2}:
          4511  680  1558
           694  101   229
          1470  249   508
```

```
In [38]: confusion_matrix(dataset, map(t -> t[2], dataset.data))
```

```
Out[38]: 3x3 Array{Int64,2}:
          6749     0     0
             0  1024     0
             0     0  2227
```

```
In [39]: prediction = random_clustering(dataset)
         matrix = confusion_matrix(dataset, prediction)
         println(matrix, "\n")
         sleep(0.2)
```

```
[4539 703 1507
 688 109 227
 1499 233 495]
```

```
In [40]: let
             n = sum(matrix)
             println("Amostra: ", n)

             trace = diag(matrix)
             println("Traço:\n", trace)

             x = sum(trace)

             println("Acertos: ", x)

             o = n - x

             println("Erros: ", o)

             acc = x / n

             println("Accuracy: ", round(100 * acc, 2), "%")
```

```
        k = 3
        kn = sum(matrix[k,:])
        println(k, " - Objetos: ", kn)

        ktp = matrix[k,k]
        ktpp = ktp / kn

        println(k, " - Acerto Positivo: ", ktp, ", ", round(100 * ktpp, 2), "%")

        kfn = kn - ktp
        kfnp = kfn / o

        println(k, " - Falso Negativo: ", kfn, ", ", round(100 * kfnp, 2), "% (total de erros)")

        kfp = sum(matrix[:,k]) - ktp
        kfpp = kfp / o

        println(k, " - Falso Positivo: ", kfp, ", ", round(100 * kfpp, 2), "% (total de erros)")

        ktn = n - kfn - kfp - ktp
        ktnp = ktn / (n - kn)

        println(k, " - Acerto Negativo: ", ktn, ", ", round(100 * ktnp, 2), "%")

        kacc = (ktp + ktn) / n

        println(k, " - Accuracy: ", round(100 * kacc, 2), "%")

        kprecision = ktp / (ktp + kfp)

        println(k, " - Precision: ", round(100 * kprecision, 2), "%")

        krecall = ktp / (ktp + kfn)

        println(k, " - Recall: ", round(100 * krecall, 2), "%")

        kfscore = 2 * kprecision * krecall / (kprecision + krecall)

        println(k, " - F1-score: ", round(kfscore, 2))

        sleep(0.2)
    end
```

```
Amostra: 10000
Traço:
[4539,109,495]
Acertos: 5143
Erros: 4857
Accuracy: 51.43%
3 - Objetos: 2227
3 - Acerto Positivo: 495, 22.23%
3 - Falso Negativo: 1732, 35.66% (total de erros)
3 - Falso Positivo: 1734, 35.7% (total de erros)
3 - Acerto Negativo: 6039, 77.69%
```

```
3 - Accuracy: 65.34%
3 - Precision: 22.21%
3 - Recall: 22.23%
3 - F1-score: 0.22
```

```
In [41]: immutable SampleEvaluation
             size::Int
             correct::Int
             mistakes::Int
             accuracy::Float64
         end

         immutable ClusterEvaluation
             cluster::Int
             size::Int

             truePositive::Int
             truePositiveShare::Float64
             trueNegative::Int
             trueNegativeShare::Float64

             falseNegative::Int
             falseNegativeShare::Float64
             falsePositive::Int
             falsePositiveShare::Float64

             precision::Float64
             recall::Float64
             fscore::Float64
             accuracy::Float64
         end

         immutable Evaluation
             matrix::Array{Int, 2}
             sample::SampleEvaluation
             clusters::Array{ClusterEvaluation, 1}
         end
```

```
In [42]: function SampleEvaluation(matrix)
             size = sum(matrix)
             correct = sum(diag(matrix))
             mistakes = size - correct
             accuracy = correct / size

             SampleEvaluation(size, correct, mistakes, accuracy)
         end

         function ClusterEvaluation(matrix, s, k)
             kn = sum(matrix[k,:])

             ktp = matrix[k,k]
             ktpp = ktp / kn

             kfn = kn - ktp
             kfnp = kfn / s.mistakes
```

```
    kfp = sum(matrix[:,k]) - ktp
    kfpp = kfp / s.mistakes

    ktn = s.size - kfn - kfp - ktp
    ktnp = ktn / (s.size - kn)

    kacc = (ktp + ktn) / s.size
    kprecision = ktp / (ktp + kfp)
    krecall = ktp / (ktp + kfn)
    kfscore = 2 * kprecision * krecall / (kprecision + krecall)

    ClusterEvaluation(k, kn, ktp, ktpp, ktn, ktnp, kfn, kfnp, kfp, kfpp, kprecision, krecall,
end

function Evaluation(dataset, prediction)
    matrix = confusion_matrix(dataset, prediction)
    s = SampleEvaluation(matrix)
    c = map(k -> ClusterEvaluation(matrix, s, k), 1:dataset.groups)
    Evaluation(matrix, s, c)
end

function Base.show(io::IO, s::SampleEvaluation)
    println(io, "Tamanho: ", s.size)
    println(io, "Acertos: ", s.correct)
    println(io, "Erros: ", s.mistakes)
    println(io, "Accuracy: ", round(100 * s.accuracy, 2), "%")
end

function Base.show(io::IO, c::ClusterEvaluation)
    println(io, "Cluster ", c.cluster)
    println(io)
    println(io, "Tamanho: ", c.size)
    println(io, "Accuracy: ", round(100 * c.accuracy, 2), "%")
    println(io, "Precision: ", round(100 * c.precision, 2), "%")
    println(io, "Recall: ", round(100 * c.recall, 2), "%")
    println(io, "F-score: ", round(c.fscore , 2))
    println(io)
    println(io, "Acerto positivo: ", c.truePositive, " (", round(100 * c.truePositiveShare, 2)
    println(io, "Acerto negativo: ", c.trueNegative, " (", round(100 * c.trueNegativeShare, 2)
    println(io, "Falso negativo: ", c.falseNegative, " (", round(100 * c.falseNegativeShare, 2
    println(io, "Falso positivo: ", c.falsePositive, " (", round(100 * c.falsePositiveShare, 2
end

function Base.show(io::IO, r::Evaluation)
    println(io, r.sample)
    for k in r.clusters
        println(io, k)
    end
end

function evaluation_summary(io::IO, dataset, prediction; verbose=false)
    r = Evaluation(dataset, prediction)
    verbose && println(io, "Matriz de Confusão:\n\n", r.matrix, "\n")
```

```
        print(io, r)
    end

    evaluation_summary(dataset, prediction; verbose=false) = evaluation_summary(STDOUT, dataset, pr

    let
        prediction = random_clustering(dataset)
        evaluation_summary(dataset, prediction, verbose=true)
        sleep(0.2)
    end
```

```
Matriz de Confusão:

[4535 760 1454
 681 109 234
 1497 239 491]

Tamanho: 10000
Acertos: 5135
Erros: 4865
Accuracy: 51.35%

Cluster 1

Tamanho: 6749
Accuracy: 56.08%
Precision: 67.56%
Recall: 67.2%
F-score: 0.67

Acerto positivo: 4535 (67.2%)
Acerto negativo: 1073 (33.01%)
Falso negativo: 2214 (45.51%)
Falso positivo: 2178 (44.77%)

Cluster 2

Tamanho: 1024
Accuracy: 80.86%
Precision: 9.84%
Recall: 10.64%
F-score: 0.1

Acerto positivo: 109 (10.64%)
Acerto negativo: 7977 (88.87%)
Falso negativo: 915 (18.81%)
Falso positivo: 999 (20.53%)

Cluster 3

Tamanho: 2227
Accuracy: 65.76%
Precision: 22.53%
Recall: 22.05%
```

```
F-score: 0.22

Acerto positivo: 491 (22.05%)
Acerto negativo: 6085 (78.28%)
Falso negativo: 1736 (35.68%)
Falso positivo: 1688 (34.7%)
```

```
In [43]: evaluation_summary(dataset, map(t -> t[2], dataset.data))
         sleep(0.2)
```

```
Tamanho: 10000
Acertos: 10000
Erros: 0
Accuracy: 100.0%

Cluster 1

Tamanho: 6749
Accuracy: 100.0%
Precision: 100.0%
Recall: 100.0%
F-score: 1.0

Acerto positivo: 6749 (100.0%)
Acerto negativo: 3251 (100.0%)
Falso negativo: 0 (NaN%)
Falso positivo: 0 (NaN%)

Cluster 2

Tamanho: 1024
Accuracy: 100.0%
Precision: 100.0%
Recall: 100.0%
F-score: 1.0

Acerto positivo: 1024 (100.0%)
Acerto negativo: 8976 (100.0%)
Falso negativo: 0 (NaN%)
Falso positivo: 0 (NaN%)

Cluster 3

Tamanho: 2227
Accuracy: 100.0%
Precision: 100.0%
Recall: 100.0%
F-score: 1.0

Acerto positivo: 2227 (100.0%)
Acerto negativo: 7773 (100.0%)
Falso negativo: 0 (NaN%)
Falso positivo: 0 (NaN%)
```

```
In [44]: let
             n = 100
             k = 3
             c = 16
             c_y = 3

             tiny = Dataset(size=n, groups=k, features=c, slot=c_y)
             summary(tiny)

             assignments = map(t -> rand() <= 0.7 ? k - t[2] + 1 : rand(1:k), tiny.data)

             centermap = zeros(Int, k)
             groups = map(v -> v[2], tiny.data)
             for i=1:k
                 g_index = findin(groups, i)
                 centers = map(i -> assignments[i], g_index)
                 counts = hist(centers, 0:k)[2]
                 center_key = indmax(counts)
                 if centermap[center_key] != 0
                     error("Center already mapped: $(center_key) -> $(centermap[center_key]), now $i?")
                 end
                 centermap[center_key] = i
             end
             println(collect(enumerate(centermap)))
             sleep(0.2)
         end
```

```
Number of Groups: 3
Number of Features: 16
Number of Features (group): 3
Probability of Activation: 0.8
Number of Objects (total): 100
Number of Objects per Group (min): 7
Number of Objects per Group (max): 66
Number of Objects in 1: 25
Number of Objects in 2: 42
Number of Objects in 3: 33
[(1,3),(2,2),(3,1)]
```

```
In [45]: function mapping(dataset, assignments, k)
             centermap = zeros(Int, k)
             groups = map(v -> v[2], dataset.data)
             for i=1:dataset.groups
                 g_index = findin(groups, i)
                 centers = map(i -> assignments[i], g_index)
                 counts = hist(centers, 0:k)[2]
                 center_key = indmax(counts)
                 if centermap[center_key] != 0
                     error("Center already mapped: $(center_key) -> $(centermap[center_key]), now $i?")
                 end
                 centermap[center_key] = i
             end
             centermap
         end

         let
             assignments = map(t -> rand() <= 0.7 ? dataset.groups - t[2] + 1 : rand(1:dataset.groups),
             centermap = mapping(dataset, assignments, dataset.groups)
             collect(enumerate(centermap))
         end
```

```
Out[45]: 3-element Array{Tuple{Int64,Int64},1}:
          (1,3)
          (2,2)
          (3,1)
```

## 1.6 4. Export / Load

### 1.6.1 JLD

https://github.com/JuliaLang/JLD.jl

Saving and loading julia variables while preserving native types

```
In [46]: if Pkg.installed("JLD") === nothing
             println("Installing JLD...")
             Pkg.add("JLD")
         end
```

```
In [47]: using JLD
```

```
In [48]: save("dataset.jld", "large", dataset)
```

```
In [49]: stat("dataset.jld")
```

```
Out[49]: StatStruct(mode=100644, size=23790496)
```

```
In [50]: let ds = load("dataset.jld", "large")
             summary(ds)
             sleep(0.2)
         end
```

```
Number of Groups: 3
Number of Features: 200
Number of Features (group): 40
Probability of Activation: 0.8
Number of Objects (total): 10000
Number of Objects per Group (min): 667
Number of Objects per Group (max): 6666
Number of Objects in 1: 6749
Number of Objects in 2: 1024
Number of Objects in 3: 2227
```

```
In [51]: rm("dataset.jld")
```

```
In [52]: function export_dataset(name, dataset)
             path = "../dataset/" * name
             isdir(path) && rm(path, recursive=true)
             mkdir(path)
             open(path * "/summary.txt", "w") do f
                 summary(f, dataset)
             end
             open(path * "/baseline.txt", "w") do f
                 prediction = random_clustering(dataset)
                 evaluation_summary(f, dataset, prediction)
```

```
            end
            save(path * "/dataset.jld", "dataset", dataset)
            draw(PNG(path * "/plothalf.png", 24cm, 16cm), plothalf(dataset))
            draw(PNG(path * "/plothalf_multi.png", 24cm, 16cm), plothalf_multi(dataset))
            draw(PNG(path * "/plotslot.png", 24cm, 16cm), plotslot(dataset))
            draw(PNG(path * "/plotslot_multi.png", 24cm, 16cm), plotslot_multi(dataset))
            draw(PNG(path * "/plotpca.png", 24cm, 16cm), plotpca(dataset))
        end

        export_dataset("test", dataset)
        readdir("../dataset/test")

Out[52]: 8-element Array{ByteString,1}:
          "baseline.txt"
          "dataset.jld"
          "plothalf_multi.png"
          "plothalf.png"
          "plotpca.png"
          "plotslot_multi.png"
          "plotslot.png"
          "summary.txt"

In [53]: function load_dataset(name)
             path = "../dataset/" * name
             load(path * "/dataset.jld", "dataset")
         end

         load_dataset("test")

Out[53]: Dataset(3,200,40,0.8,10000,667,6666,[([1,1,1,1,1,1,1,0,1,1  ...  1,0,0,1,1,1,1,1,0,0],1),([0,1

In [54]: rm("../dataset/test", recursive=true)

In [55]: function create_large_dataset()
             dataset = Dataset(groups=3, size=1000, features=200, slot=40)
             export_dataset("large", dataset)
         end

         create_large_dataset()

         readdir("../dataset/large")

Out[55]: 8-element Array{ByteString,1}:
          "baseline.txt"
          "dataset.jld"
          "plothalf_multi.png"
          "plothalf.png"
          "plotpca.png"
          "plotslot_multi.png"
          "plotslot.png"
          "summary.txt"

In [56]: function create_small_dataset()
             dataset = Dataset(groups=3, size=100, features=200, slot=40)
             export_dataset("small", dataset)
         end
         create_small_dataset()

         readdir("../dataset/small")
```

```
Out[56]: 8-element Array{ByteString,1}:
          "baseline.txt"
          "dataset.jld"
          "plothalf_multi.png"
          "plothalf.png"
          "plotpca.png"
          "plotslot_multi.png"
          "plotslot.png"
          "summary.txt"
```