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] += n - sum
    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}:
 4
 5
 5
 4
 2
```

In [3]:
```
let n = 1000000,
    n_min = Int(n/2) - Int(n/10),
    n_max = Int(n/2) + Int(n/10),
    g = 5
```

```
        sizes = group_size(g, n, n_min, n_max)
        _n = sum(sizes)
        println(sizes)
        println(_n)
        sleep(0.2)
    end
```

```
[170319,197110,177801,246076,208694]
1000000
```

In [4]:
```
"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[4]: group_mask (generic function with 1 method)

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

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

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

In [6]:
```
"""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[6]: generate_data (generic function with 1 method)

In [7]:
```
"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[7]: instance_generator (generic function with 1 method)

In [8]:
```
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[8]:
```
20-element Array{Tuple{Array{Int64,1},Int64},1}:
 ([1,1,1,0,0,0,0,1,0,1,0,1,1,0,1,0],1)
 ([1,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0],1)
 ([1,1,0,0,0,1,1,0,1,0,0,0,0,1,1,1],1)
 ([1,1,1,0,0,1,0,0,0,1,0,0,0,0,1,0],1)
 ([0,1,1,0,0,1,0,1,0,1,0,0,0,0,0,0],1)
 ([1,1,0,0,1,0,0,0,0,0,0,1,0,0,0,1],1)
 ([1,0,0,1,1,1,0,0,0,0,0,0,0,0,1,1],2)
 ([0,1,0,1,0,1,0,0,0,0,0,1,0,1,0,1],2)
 ([0,0,0,0,1,1,0,0,1,0,0,0,0,0,0,0],2)
 ([0,0,0,1,1,0,0,0,1,0,0,0,1,0,0,1],2)
 ([1,0,1,1,1,1,1,0,1,0,0,0,1,1,0,1],2)
 ([1,0,0,0,0,0,1,1,1,0,0,1,0,0,0,0],3)
 ([0,1,0,0,0,0,1,1,1,0,0,0,0,0,1,0],3)
 ([1,0,0,0,0,0,1,0,0,1,0,1,1,0,0,1],4)
```

```
([1,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0],4)
([0,0,0,0,0,1,0,0,0,1,0,1,1,0,1,1],4)
([0,0,1,0,0,0,1,0,1,1,1,0,0,0,0,0],4)
([0,0,0,0,1,0,0,0,0,1,0,0,1,0,1,0],5)
([0,0,0,0,0,0,0,0,0,0,1,0,0,1,1,0],5)
([0,0,0,1,0,0,0,1,1,0,0,0,1,1,1,1],5)
```

In [9]:
```
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 = round(Int, 1.2 * size / groups)
        end
        if size_min == 0
            size_min = round(Int, size_max / 2)
        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): 2000
Number of Objects per Group (max): 4000
Number of Objects in 1: 2973
Number of Objects in 2: 3891
Number of Objects in 3: 3136
```

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

```
In [11]: using Gadfly
         set_default_plot_size(24cm, 12cm)
```

```
In [12]: dataset = Dataset()
```

```
Out[12]: Dataset(3,200,40,0.8,10000,2000,4000,[([1,1,1,1,0,0,1,1,0,1  ...  0,1,0,1,1,0,1,1,1,1],1),([1,0
```

```
In [13]: function halfmask(n)
             mask = zeros(n)
             middle = round(Int, n / 2)
             mask[1:middle] = 1
             mask
         end

         halfmask(10)
```

Out[13]: 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 [14]:
```
reversemask(mask) = ones(mask) - mask

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

Out[14]: 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 [15]:
```
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[15]: ([0.8505375194282974,0.7710145615337027,0.7041930845395834,0.3159312943879906,0.69363618507309

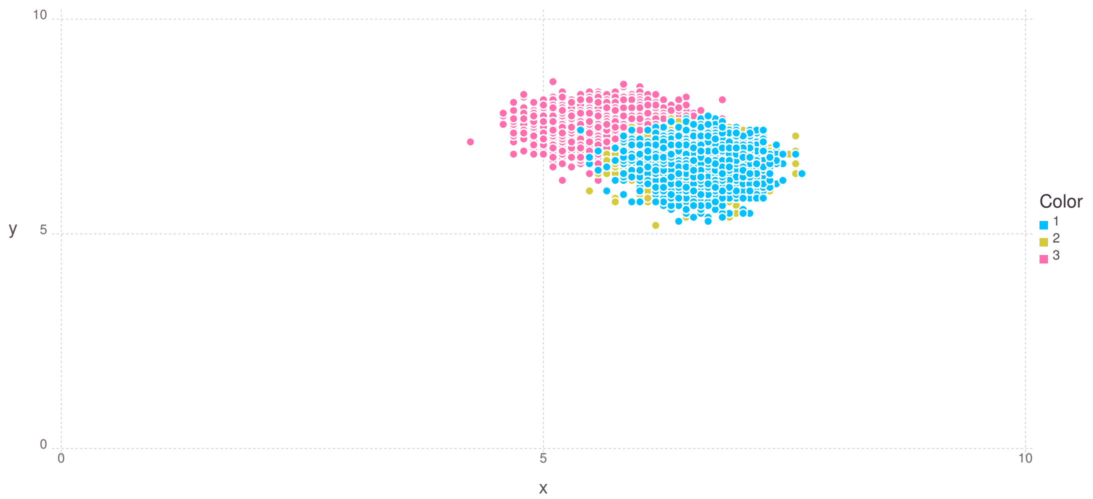
In [16]:
```
function reduce2d(data, masks)
    x = map(t -> norm(masks[1] .* t[1]), data)
    y = map(t -> norm(masks[2] .* t[1]), data)
    k = map(t -> string(t[2]), data)
    x, y, k
end

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[16]: plothalf (generic function with 1 method)

In [17]:
```
plothalf(dataset)
```

Out[17]:

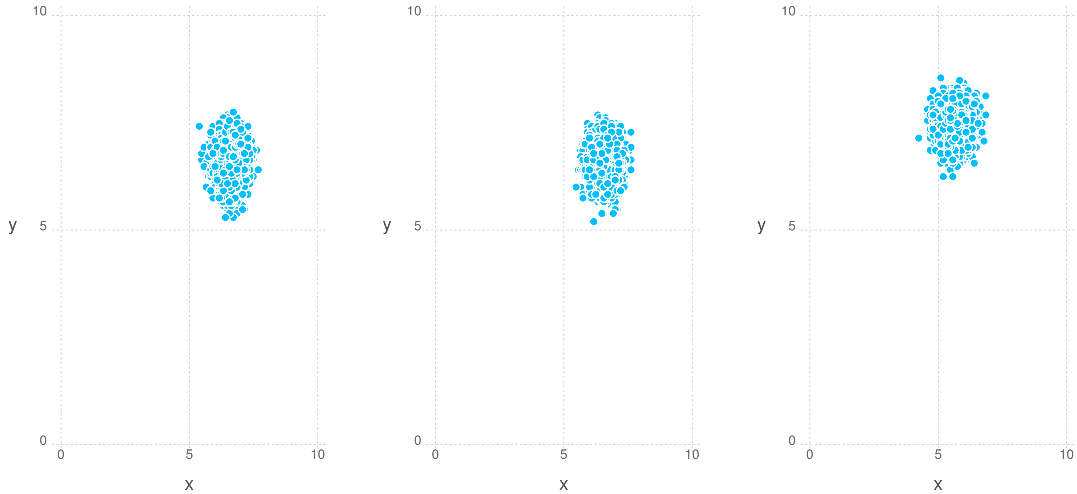


In [18]:
```
function plothalf_multi(dataset)
    masks = halfmasks(dataset.features)

    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[18]: plothalf_multi (generic function with 1 method)

In [19]:
```
plothalf_multi(dataset)
```

Out[19]:



In [20]:
```
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[20]:
```
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 [21]:
```
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
```

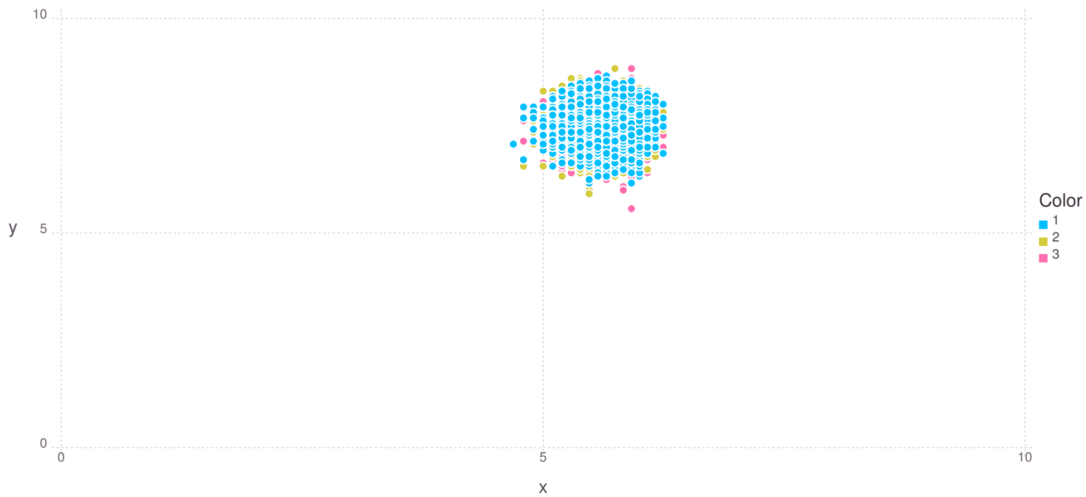
Out[21]: ([0.0,0.0,0.0,0.7800623905853532,0.5056243806634053,0.8839236043120529,0.0,0.0,0.0,0.0],[0.164

```
In [22]: 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[22]: plotslot (generic function with 1 method)

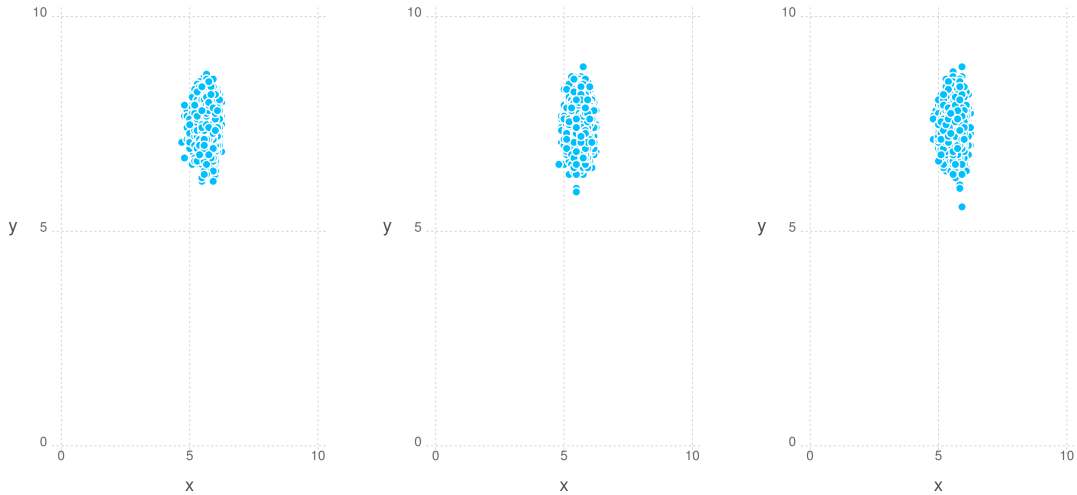
```
In [23]: plotslot(dataset)
```

Out[23]:




```
In [24]: 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[24]: plotslot_multi (generic function with 1 method)

In [25]: plotslot_multi(dataset)

Out[25]:
```



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

In [27]: vector_matrix(data) = float(hcat(map(first, data)...))

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

Out[27]: 2x3 Array{Float64,2}:
          1.0  3.0  5.0
          2.0  4.0  6.0

In [28]: using MultivariateStats

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

```
Out[29]: PCA(indim = 200, outdim = 2, principalratio = 0.20118)
```

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

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

```
Out[30]: 2x3636 Array{Float64,2}:
          -2.46235   -1.676    -2.36586   -2.92933  ...  -2.21661  -2.76574  -3.00483
          -0.653523  -1.85303  -0.700858  -1.06335      -1.55338  -1.62989  -1.92368
```

```
In [31]: 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[31]: 3636-element Array{Float64,1}:
          -2.46235
          -1.676
          -2.36586
          -2.92933
          -2.64967
          -2.44774
          -2.68232
          -2.88877
          -2.56573
          -2.46719
          -2.3454
          -2.46315
          -3.38957
            ⋮
          -2.79646
          -2.96348
          -2.67705
          -2.52688
          -2.66194
          -3.14876
          -2.99832
          -2.87497
          -2.78294
          -2.21661
          -2.76574
          -3.00483
```

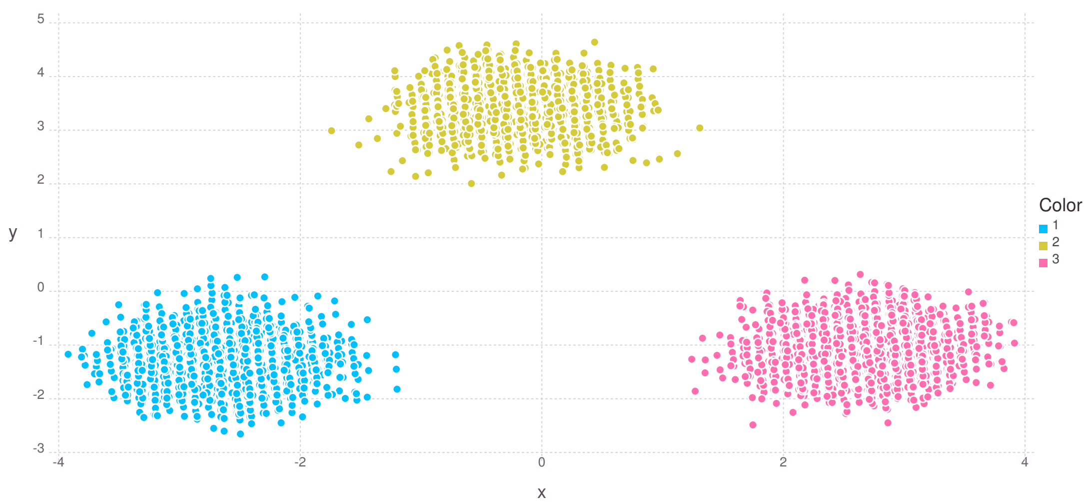
```
In [32]: function plotpca(dataset)
             train = vector_matrix(dataset.data)
             model = fit(PCA, train; maxoutdim=2)

             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[32]: plotpca (generic function with 1 method)

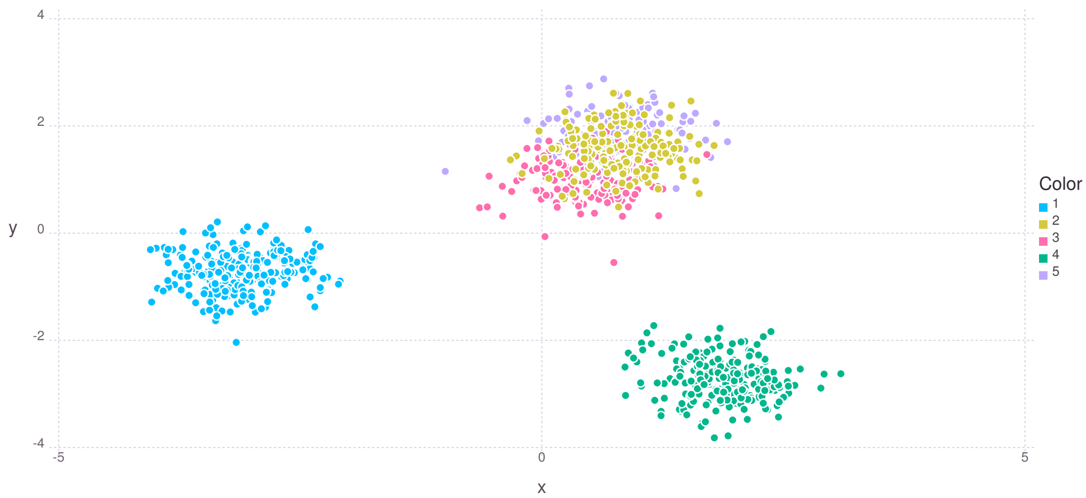
In [33]:
```
plotpca(dataset)
```

Out[33]:



In [34]:
```
let _dataset = Dataset(groups=5, size=1000, features=200, slot=40)
    plotpca(_dataset)
end
```

Out[34]:



1.5 3. Evaluation

In [35]:
```
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[35]:
```
3-element Array{Float64,1}:
 0.3636
 0.6211
 1.0
```

In [36]:
```
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[36]:
```
3-element Array{Float64,1}:
 0.30013
 0.19782
 0.50205
```

In [37]:
```
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[37]:
```
10000-element Array{Int64,1}:
 1
 2
 3
 3
 3
 2
 1
 1
 2
 1
 1
 3
 1
 ⋮
 3
 1
 2
 3
 1
 3
 2
 2
 2
 1
 1
 2
```

1.5.1 Confusion Matrix

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

In [38]:
```
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[38]:
```
3x3 Array{Int64,2}:
 1313  934  1389
  969  628   978
 1339  987  1463
```

In [39]:
```
confusion_matrix(dataset, map(t -> t[2], dataset.data))
```

Out[39]:
```
3x3 Array{Int64,2}:
 3636     0     0
    0  2575     0
    0     0  3789
```

In [40]:
```
prediction = random_clustering(dataset)
matrix = confusion_matrix(dataset, prediction)
println(matrix, "\n")
sleep(0.2)
```

```
[1305 916 1415
 942 636 997
 1397 989 1403]
```

In [41]:
```
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:
[1305,636,1403]
Acertos: 3344
Erros: 6656
Accuracy: 33.44%
3 - Objetos: 3789
3 - Acerto Positivo: 1403, 37.03%
3 - Falso Negativo: 2386, 35.85% (total de erros)
3 - Falso Positivo: 2412, 36.24% (total de erros)
3 - Acerto Negativo: 3799, 61.17%
```

```
3 - Accuracy: 52.02%
3 - Precision: 36.78%
3 - Recall: 37.03%
3 - F1-score: 0.37
```

In [42]:
```
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 [43]:
```
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, k
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")
```

```julia
        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:

[1330 972 1334
 928 655 992
 1365 915 1509]

Tamanho: 10000
Acertos: 3494
Erros: 6506
Accuracy: 34.94%

Cluster 1

Tamanho: 3636
Accuracy: 54.01%
Precision: 36.71%
Recall: 36.58%
F-score: 0.37

Acerto positivo: 1330 (36.58%)
Acerto negativo: 4071 (63.97%)
Falso negativo: 2306 (35.44%)
Falso positivo: 2293 (35.24%)

Cluster 2

Tamanho: 2575
Accuracy: 61.93%
Precision: 25.77%
Recall: 25.44%
F-score: 0.26

Acerto positivo: 655 (25.44%)
Acerto negativo: 5538 (74.59%)
Falso negativo: 1920 (29.51%)
Falso positivo: 1887 (29.0%)

Cluster 3

Tamanho: 3789
Accuracy: 53.94%
Precision: 39.35%
Recall: 39.83%
```

```
F-score: 0.4

Acerto positivo: 1509 (39.83%)
Acerto negativo: 3885 (62.55%)
Falso negativo: 2280 (35.04%)
Falso positivo: 2326 (35.75%)
```

```
In [44]: evaluation_summary(dataset, map(t -> t[2], dataset.data))
         sleep(0.2)
```

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

Cluster 1

Tamanho: 3636
Accuracy: 100.0%
Precision: 100.0%
Recall: 100.0%
F-score: 1.0

Acerto positivo: 3636 (100.0%)
Acerto negativo: 6364 (100.0%)
Falso negativo: 0 (NaN%)
Falso positivo: 0 (NaN%)

Cluster 2

Tamanho: 2575
Accuracy: 100.0%
Precision: 100.0%
Recall: 100.0%
F-score: 1.0

Acerto positivo: 2575 (100.0%)
Acerto negativo: 7425 (100.0%)
Falso negativo: 0 (NaN%)
Falso positivo: 0 (NaN%)

Cluster 3

Tamanho: 3789
Accuracy: 100.0%
Precision: 100.0%
Recall: 100.0%
F-score: 1.0

Acerto positivo: 3789 (100.0%)
Acerto negativo: 6211 (100.0%)
Falso negativo: 0 (NaN%)
Falso positivo: 0 (NaN%)
```

```
In [45]: 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): 20
Number of Objects per Group (max): 40
Number of Objects in 1: 32
Number of Objects in 2: 35
Number of Objects in 3: 33
[(1,3),(2,2),(3,1)]
```

```
In [46]: 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[46]: 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 [47]: if Pkg.installed("JLD") === nothing
             println("Installing JLD...")
             Pkg.add("JLD")
         end
```

```
In [48]: using JLD
```

```
In [49]: save("dataset.jld", "large", dataset)
```

```
In [50]: stat("dataset.jld")
```

```
Out[50]: StatStruct(mode=100644, size=23790496)
```

```
In [51]: 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): 2000
Number of Objects per Group (max): 4000
Number of Objects in 1: 3636
Number of Objects in 2: 2575
Number of Objects in 3: 3789
```

```
In [52]: rm("dataset.jld")
```

```
In [53]: 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[53]:
```
8-element Array{ByteString,1}:
 "baseline.txt"
 "dataset.jld"
 "plothalf_multi.png"
 "plothalf.png"
 "plotpca.png"
 "plotslot_multi.png"
 "plotslot.png"
 "summary.txt"
```

In [54]:
```
function load_dataset(name)
    path = "../dataset/" * name
    load(path * "/dataset.jld", "dataset")
end

load_dataset("test")
```

Out[54]:
```
Dataset(3,200,40,0.8,10000,2000,4000,[([1,1,1,1,0,0,1,1,0,1  ...  0,1,0,1,1,0,1,1,1,1],1),([1,(
```

In [55]:
```
rm("../dataset/test", recursive=true)
```