Dataset em Julia

February 14, 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.

In [1]:
```
using Base.Test
```

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 [2]:
```
"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[2]:
```
group_size (generic function with 1 method)
```

In [3]:
```
let n = 20,
    n_min = 2,
    n_max = 5,
    g = 5

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

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

In [4]:
```julia
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
```

```
[192633,224008,196578,174819,211962]
1000000
```

In [5]:
```julia
"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[5]:
```
group_mask (generic function with 1 method)
```

In [6]:
```julia
let g = 5,
    c = 16,
    c_y = 3

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

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

In [7]:
```julia
"""gera objetos para grupos seguindo a distribuição num_g,
a máscara char_g e a probabilidade p de ativação"""
```

```julia
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[7]: generate_data (generic function with 1 method)

In [8]:
```julia
"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[8]: instance_generator (generic function with 1 method)

In [9]:
```julia
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[9]:
```
20-element Array{Tuple{Array{Int64,1},Int64},1}:
 ([1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1],1)
 ([1,1,0,0,1,0,0,1,0,0,0,0,0,1,1,1],1)
 ([1,1,1,0,0,0,0,1,0,0,0,0,0,0,0,0],1)
 ([1,1,0,1,0,0,0,0,0,0,0,0,0,1,0,0],1)
 ([0,0,0,0,1,0,1,0,0,0,0,0,0,0,1,1],2)
 ([0,0,0,1,1,1,0,1,0,0,0,0,1,1,1,0],2)
 ([0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0],2)
 ([0,0,0,1,1,1,0,0,0,0,1,0,0,0,0,0],2)
 ([0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,1],2)
```

```
([0,0,0,1,0,0,0,1,1,0,0,0,0,0,0,0],3)
([0,0,0,0,0,1,1,1,1,0,0,0,1,0,0,0],3)
([0,0,0,1,0,0,1,1,1,0,0,0,1,0,0,0],3)
([0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,1],3)
([0,1,1,0,0,0,0,0,1,0,0,0,0,0,0,1],3)
([0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1],4)
([0,0,0,0,0,0,0,0,0,1,1,1,0,0,1,1],4)
([0,0,1,1,0,1,0,1,0,1,0,0,0,0,0,0],4)
([0,0,1,0,0,0,0,0,0,1,1,1,0,0,1,0],4)
([0,0,1,0,0,0,1,1,1,0,0,1,1,1,1,0],5)
([1,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1],5)
```

In [10]:
```
immutable Input
    data::Array{Vector{Int}, 1}
    size::Int
    dimension::Int

    Input(data::Array{Vector{Int}, 1}) = begin
        size = length(data)
        if size == 0
            error("empty data")
        end
        dimension = length(data[1])
        if dimension == 0
            error("empty dimension")
        end
        for i in data
            if length(i) != dimension
                error("wrong dimension: expected $dimension, actual $(length(i))")
            end
        end

        new(data, size, dimension)
    end
end

let x = Vector{Int}[[1,2], [3,4], [5,6]]
    @test Input(x).data == x
    @test Input(x).size == 3
    @test Input(x).dimension == 2
end

let x = Vector{Int}[]
    # Empty data
    @test_throws ErrorException Input(x)
end

let x = Vector{Int}[[]]
    # Empty dimension
    @test_throws ErrorException Input(x)
end

let x = Vector{Int}[[1], [2,3]]
    # Wrong dimension
```

```
        @test_throws ErrorException Input(x)
    end
```

Out[10]: ErrorException("wrong dimension: expected 1, actual 2")

In [11]:
```
immutable Dataset
    clusters::Int
    dimension::Int
    slot::Int
    activation_p::Float64
    size::Int
    size_min::Int
    size_max::Int
    input::Input
    target::Vector{Int}

    Dataset(; clusters=3, size=1000, size_min=0, size_max=0, dimension=200, slot=40, activation
        if size < 10
            error("minimum 10")
        end
        if clusters > size
            error("too many clusters")
        end
        if dimension < clusters * slot
            error("slot too big")
        end

        if size_max == 0
            size_max = round(Int, 1.2 * size / clusters)
        end
        if size_min == 0
            size_min = round(Int, size_max / 2)
        end
        if size_max * clusters < size
            error("size_max too tight")
        end

        data = instance_generator(size, dimension, slot, activation_p, clusters, size_min, size
        shuffle!(data)

        input = Input(map(t -> t[1], data))
        target = map(t -> t[2], data)

        new(clusters, dimension, slot, activation_p, size, size_min, size_max, input, target)
    end
end

let ds = Dataset()
    @test ds.clusters > 1
    @test ds.size > 10
    @test ds.size == ds.input.size
    @test ds.dimension == ds.input.dimension
    @test ds.size == length(ds.target)
    @test all(0 .< ds.target .<= ds.clusters)
end
```

```
In [12]: let x = [1, 2, 3, 3, 2]
             findin(x, 3)
         end
```

```
Out[12]: 2-element Array{Int64,1}:
          3
          4
```

```
In [13]: data(ds, k) = map(i -> ds.input.data[i], findin(ds.target, k))
         count(ds, k) = length(data(ds, k))

         "Sumário do Dataset"
         function summary(io::IO, ds::Dataset)
             println(io, "Clusters: ", ds.clusters)
             println(io, "Dimension (features): ", ds.dimension)
             println(io, "Features per Cluster: ", ds.slot)
             println(io, "Probability of Activation: ", ds.activation_p)
             println(io)
             println(io, "Size: ", ds.size)
             println(io, "Min Cluster size: ", ds.size_min)
             println(io, "Max Cluster size: ", ds.size_max)
             for k=1:ds.clusters
                 println(io, "Cluster ", k, " size: ", count(ds, k))
             end
         end

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

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

```
Clusters: 3
Dimension (features): 200
Features per Cluster: 40
Probability of Activation: 0.8

Size: 1000
Min Cluster size: 200
Max Cluster size: 400
Cluster 1 size: 274
Cluster 2 size: 338
Cluster 3 size: 388
```

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 [14]: if Pkg.installed("Gadfly") === nothing
             println("Installing Gadfly...")
```

```
        Pkg.add("Gadfly")
        Pkg.add("Cairo")
    end
```

In [15]:
```
using Gadfly
set_default_plot_size(24cm, 12cm)
```

In [16]:
```
function halfmask(n)
    mask = zeros(n)
    middle = round(Int, n / 2)
    mask[1:middle] = 1
    mask
end

halfmask(10)
```

Out[16]:
```
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 [17]:
```
reversemask(mask) = ones(mask) - mask

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

Out[17]:
```
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 [18]:
```
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
```

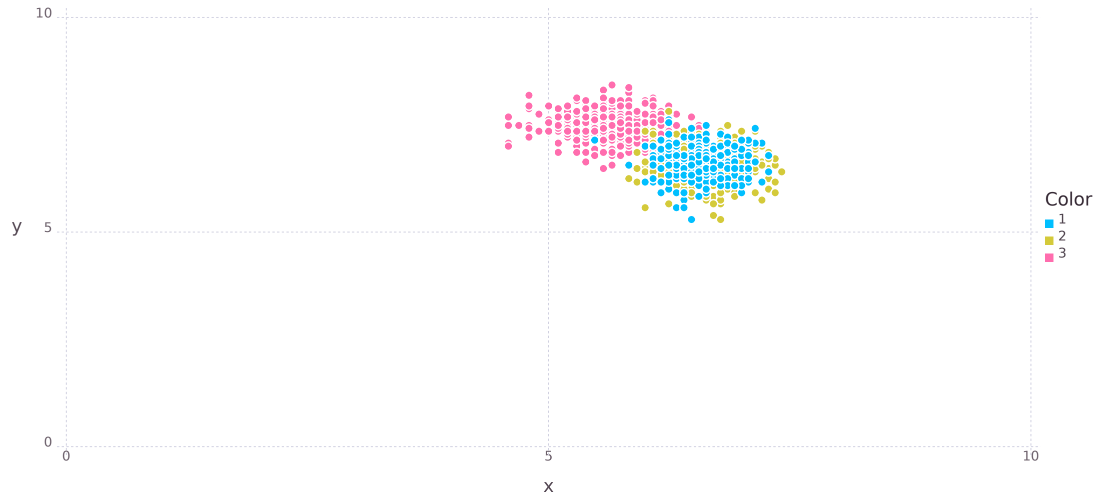
Out[18]: ([0.9084727272248192,0.8534360320567183,0.5803161389304039,0.023718470886070264,0.6193120528540

In [19]:
```
function reduce2d(data, masks)
    x = map(v -> norm(masks[1] .* v), data)
    y = map(v -> norm(masks[2] .* v), data)
    x, y
end

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

    g = Array(Layer, 0)

    for k=1:dataset.clusters
        kdata = data(dataset, k)
        x, y = reduce2d(kdata, masks)
        color = fill(string(k), length(kdata))
        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[19]: plothalf (generic function with 1 method)

In [20]:
```
let
    dataset = Dataset()
    plothalf(dataset)
end
```

Out[20]:

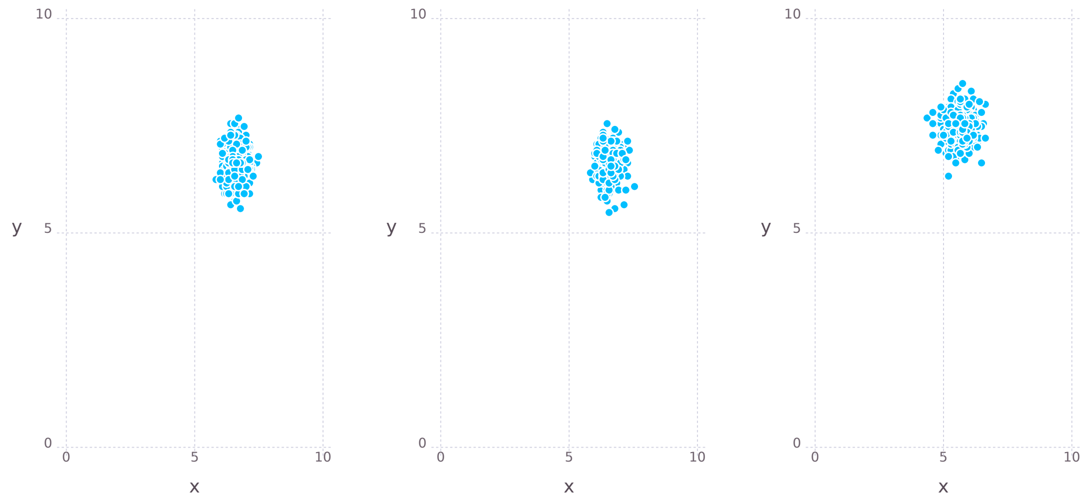


In [21]:
```
function plothalf_multi(dataset)
    masks = halfmasks(dataset.dimension)

    g = Array(Plot, 0)

    for k=1:dataset.clusters
        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[21]: plothalf_multi (generic function with 1 method)

In [22]:
```
let
    dataset = Dataset()
    plothalf_multi(dataset)
end
```

Out[22]:



In [23]:
```
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[23]: 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 [24]:
```
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
```

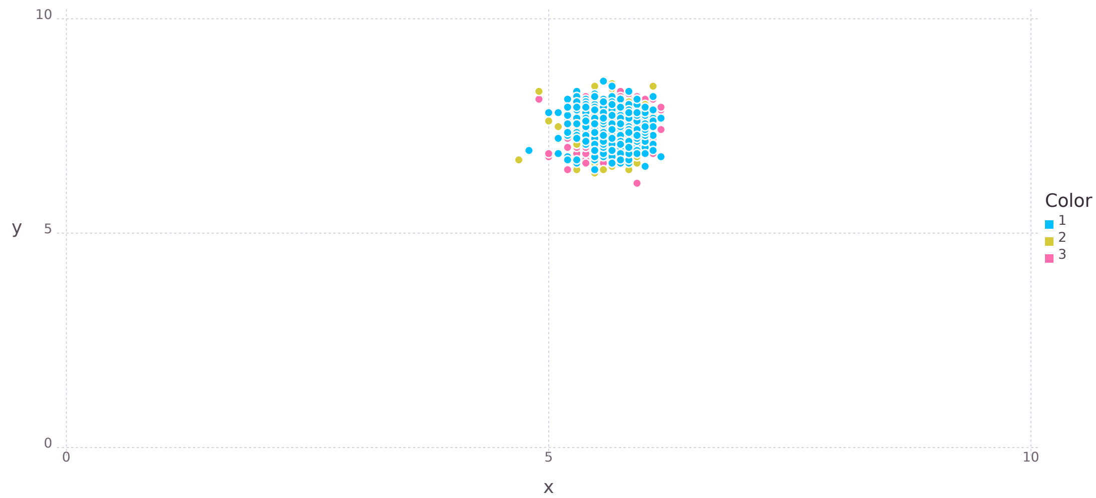
Out[24]: ([0.0,0.0,0.0,0.6051776801438544,0.012517108982922132,0.09929079399969076,0.0,0.0,0.0,0.0],[0.

In [25]:
```
function plotslot(dataset)
    g = Array(Layer, 0)

    for k=1:dataset.clusters
        masks = featuremasks(dataset.dimension, dataset.slot, k)
        kdata = data(dataset, k)
        x, y = reduce2d(kdata, masks)
        color = fill(string(k), length(kdata))
        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
```

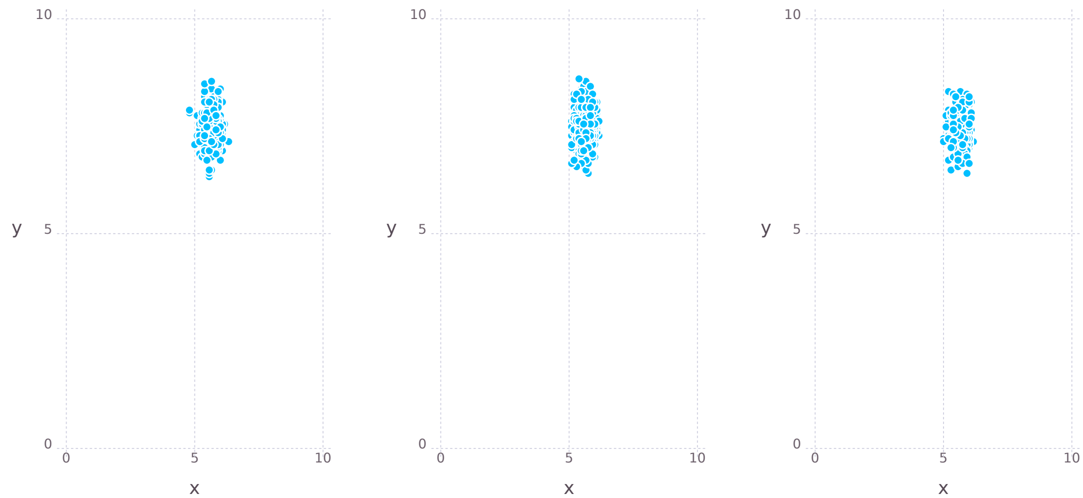
Out[25]: plotslot (generic function with 1 method)

In [26]:
```
let
    dataset = Dataset()
    plotslot(dataset)
end
```

Out[26]:




```
In [27]: function plotslot_multi(dataset)
             g = Array(Plot, 0)

             for k=1:dataset.clusters
                 masks = featuremasks(dataset.dimension, 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[27]: plotslot_multi (generic function with 1 method)

In [28]: let
             dataset = Dataset()
             plotslot_multi(dataset)
         end

Out[28]:
```



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

```
In [30]: vector_matrix(data) = float(hcat(data...))

         let
             x = Vector{Int}[[1,2], [3,4], [5,6]]
             vector_matrix(x)
         end
```

```
Out[30]: 2x3 Array{Float64,2}:
          1.0  3.0  5.0
          2.0  4.0  6.0
```

```
In [31]: using MultivariateStats
```

```
In [32]: let
             dataset = Dataset()
             train = vector_matrix(dataset.input.data)
             fit(PCA, train; maxoutdim=2)
         end
```

```
Out[32]: PCA(indim = 200, outdim = 2, principalratio = 0.20142)
```

In [33]:
```
let
    dataset = Dataset()
    train = vector_matrix(dataset.input.data)
    model = fit(PCA, train; maxoutdim=2)

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

Out[33]:
```
2x356 Array{Float64,2}:
 2.53207  3.1561   2.61682   3.17531  2.50344  2.91659   2.38646   ...  2.92984  3.08314  2.930
 1.91864  1.0619   0.863023  1.40407  0.97461  0.979259  0.936551      1.6251   1.05003  1.11317
```

In [34]:
```
let
    dataset = Dataset()
    train = vector_matrix(dataset.input.data)
    model = fit(PCA, train; maxoutdim=2)

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

Out[34]:
```
395-element Array{Float64,1}:
 2.50747
 2.72736
 2.47102
 2.10188
 3.57765
 2.7311
 3.21828
 2.59773
 2.43684
 3.02356
 2.35511
 2.84099
 2.12469
 1.71799
 3.58645
 2.2573
 2.94551
 3.05155
 ⋮
 1.88989
 2.9358
 3.06371
 2.70279
 2.29331
 2.74743
 3.16218
 2.43391
 3.68556
 2.4085
 2.91859
 2.2133
```

```
2.11478
2.41177
2.85048
2.91603
2.70163
```

In [35]:
```
function plotpca(dataset)
    train = vector_matrix(dataset.input.data)
    model = fit(PCA, train; maxoutdim=2)

    g = Array(Layer, 0)

    for k=1:dataset.clusters
        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[35]:
```
plotpca (generic function with 1 method)
```

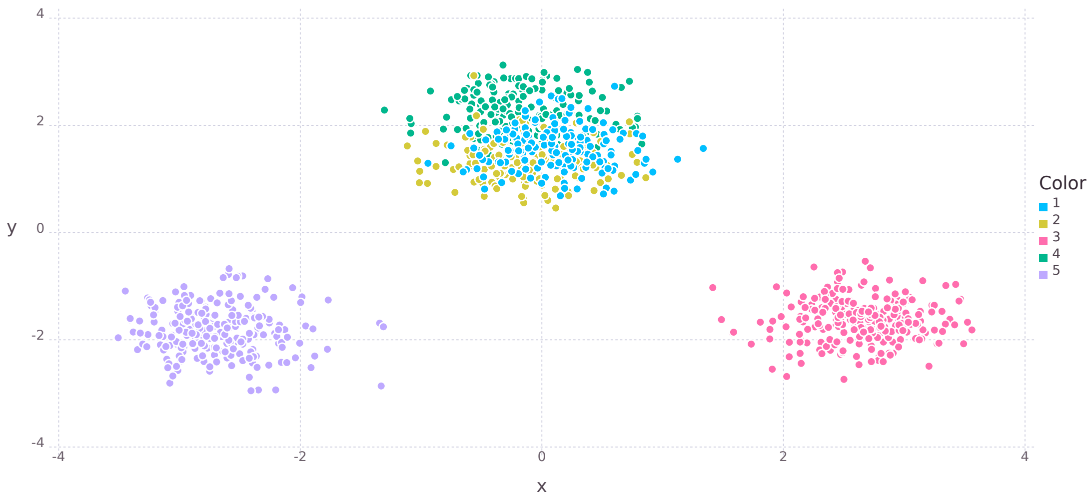
In [36]:
```
let
    dataset = Dataset()
    plotpca(dataset)
end
```

Out[36]:

In [37]:
```
let
    dataset = Dataset(clusters=5, size=1000, dimension=200, slot=40)
```

```
    plotpca(dataset)
end
```

Out[37]:



1.5 3. Evaluation

In [38]:
```
function distribution(dataset)
    clusters = Array(Float64, dataset.clusters)
    size = 0
    for k=1:dataset.clusters
        size += count(dataset, k)
        clusters[k] = size
    end
    clusters /= size
    clusters
end

let
    dataset = Dataset()
    distribution(dataset)
end
```

Out[38]:
```
3-element Array{Float64,1}:
 0.264
 0.648
 1.0
```

In [39]:
```
function choosek(distribution)
    r = rand()
    for k=1:length(distribution)
        if r <= distribution[k]
            return k
        end
```

```julia
    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[39]: 3-element Array{Float64,1}:

```
 0.3004
 0.19973
 0.49987
```

In [40]:

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

let
    dataset = Dataset()
    random_clustering(dataset)
end
```

Out[40]: 1000-element Array{Int64,1}:

```
 3
 3
 3
 3
 1
 2
 1
 1
 2
 1
 1
 3
 2
 2
 2
 3
 1
 1
 ⋮
 3
```

```
1
3
1
2
1
2
3
2
1
3
2
2
3
1
2
1
```

1.5.1 Confusion Matrix

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

```
In [41]: function confusion_matrix(dataset, prediction)
             matrix = zeros(Int, dataset.clusters, dataset.clusters)
             for p=1:length(prediction)
                 i = dataset.target[p]
                 j = prediction[p]
                 matrix[i,j] += 1
             end
             matrix
         end

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

```
Out[41]: 3x3 Array{Int64,2}:
          159  110  127
          104   75   99
          127   86  113
```

```
In [42]: let
             dataset = Dataset()
             confusion_matrix(dataset, dataset.target)
         end
```

```
Out[42]: 3x3 Array{Int64,2}:
          352    0    0
            0  329    0
            0    0  319
```

```
In [43]: let
             dataset = Dataset()
             prediction = random_clustering(dataset)
             matrix = confusion_matrix(dataset, prediction)
```

```
println(matrix, "\n")
sleep(0.2)

n = sum(matrix)
println("Size: ", n)

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

x = sum(trace)

println("Correct: ", x)

o = n - x

println("Mistakes: ", o)

acc = x / n

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

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

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

println(k, " - True Positive: ", ktp, ", ", round(100 * ktpp, 2), "%")

kfn = kn - ktp
kfnp = kfn / o

println(k, " - False Negative: ", kfn, ", ", round(100 * kfnp, 2), "% (errors)")

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

println(k, " - False Positive: ", kfp, ", ", round(100 * kfpp, 2), "% (errors)")

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

println(k, " - True Negative: ", 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)
```

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

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

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

            sleep(0.2)
        end
```

```
[121 95 132
 101 70 109
 129 107 136]

Size: 1000
Trace:
[121,70,136]
Correct: 327
Mistakes: 673
Accuracy: 32.7%
3 - Size: 372
3 - True Positive: 136, 36.56%
3 - False Negative: 236, 35.07% (errors)
3 - False Positive: 241, 35.81% (errors)
3 - True Negative: 387, 61.62%
3 - Accuracy: 52.3%
3 - Precision: 36.07%
3 - Recall: 36.56%
3 - F1-score: 0.36
```

```julia
In [44]: 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 [45]:

```
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.clusters)
    Evaluation(matrix, s, c)
end

function Base.show(io::IO, s::SampleEvaluation)
    println(io, "Size: ", s.size)
    println(io, "Correct: ", s.correct)
    println(io, "Mistakes: ", s.mistakes)
    println(io, "Accuracy: ", round(100 * s.accuracy, 2), "%")
end
```

```julia
function Base.show(io::IO, c::ClusterEvaluation)
    println(io, "Cluster ", c.cluster)
    println(io)
    println(io, "Size: ", 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, "True Positive: ", c.truePositive, " (", round(100 * c.truePositiveShare, 2),
    println(io, "True Negative: ", c.trueNegative, " (", round(100 * c.trueNegativeShare, 2),
    println(io, "False Negative: ", c.falseNegative, " (", round(100 * c.falseNegativeShare, 2
    println(io, "False Positive: ", 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, "Confusion Matrix:\n\n", r.matrix, "\n")
    print(io, r)
end

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

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

```
Confusion Matrix:

[172 117 116
 109 79 90
 109 111 97]

Size: 1000
Correct: 348
Mistakes: 652
Accuracy: 34.8%

Cluster 1

Size: 405
Accuracy: 54.9%
Precision: 44.1%
Recall: 42.47%
```

```
F-score: 0.43

True Positive: 172 (42.47%)
True Negative: 377 (63.36%)
False Negative: 233 (35.74%)
False Positive: 218 (33.44%)

Cluster 2

Size: 278
Accuracy: 57.3%
Precision: 25.73%
Recall: 28.42%
F-score: 0.27

True Positive: 79 (28.42%)
True Negative: 494 (68.42%)
False Negative: 199 (30.52%)
False Positive: 228 (34.97%)

Cluster 3

Size: 317
Accuracy: 57.4%
Precision: 32.01%
Recall: 30.6%
F-score: 0.31

True Positive: 97 (30.6%)
True Negative: 477 (69.84%)
False Negative: 220 (33.74%)
False Positive: 206 (31.6%)
```

```
In [46]: let
             dataset = Dataset()
             evaluation_summary(dataset, dataset.target)
             sleep(0.2)
         end
```

```
Size: 1000
Correct: 1000
Mistakes: 0
Accuracy: 100.0%

Cluster 1

Size: 234
Accuracy: 100.0%
Precision: 100.0%
Recall: 100.0%
F-score: 1.0

True Positive: 234 (100.0%)
True Negative: 766 (100.0%)
False Negative: 0 (NaN%)
```

```
False Positive: 0 (NaN%)

Cluster 2

Size: 353
Accuracy: 100.0%
Precision: 100.0%
Recall: 100.0%
F-score: 1.0

True Positive: 353 (100.0%)
True Negative: 647 (100.0%)
False Negative: 0 (NaN%)
False Positive: 0 (NaN%)

Cluster 3

Size: 413
Accuracy: 100.0%
Precision: 100.0%
Recall: 100.0%
F-score: 1.0

True Positive: 413 (100.0%)
True Negative: 587 (100.0%)
False Negative: 0 (NaN%)
False Positive: 0 (NaN%)
```

In [47]:
```
let
    n = 100
    k = 3
    c = 16
    c_y = 3

    tiny = Dataset(size=n, clusters=k, dimension=c, slot=c_y)
    summary(tiny)

    assignments = map(t -> rand() <= 0.7 ? k - t + 1 : rand(1:k), tiny.target)

    centermap = zeros(Int, k)
    target = tiny.target
    for i=1:k
        k_index = findin(target, i)
        centers = map(i -> assignments[i], k_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
```

```
Clusters: 3
Dimension (features): 16
Features per Cluster: 3
Probability of Activation: 0.8

Size: 100
Min Cluster size: 20
Max Cluster size: 40
Cluster 1 size: 31
Cluster 2 size: 34
Cluster 3 size: 35
[(1,3),(2,2),(3,1)]
```

In [48]:
```
function mapping(dataset::Dataset, assignments::Vector{Int}, k::Int)
    centermap = zeros(Int, k)
    for i=1:dataset.clusters
        k_index = findin(dataset.target, i)
        centers = map(i -> assignments[i], k_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
    dataset = Dataset()
    assignments = map(t -> rand() <= 0.7 ? dataset.clusters - t + 1 : rand(1:dataset.clusters)
    centermap = mapping(dataset, assignments, dataset.clusters)
    collect(enumerate(centermap))
end
```

Out[48]:
```
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 [49]:
```
if Pkg.installed("JLD") === nothing
    println("Installing JLD...")
    Pkg.add("JLD")
end
```

In [50]:
```
using JLD
```

In [51]:
```
let
    dataset = Dataset()
```

```
        save("dataset.jld", "large", dataset)
    end
```

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

Out[52]:
```
StatStruct(mode=100664, size=2006728)
```

In [53]:
```
let
    dataset = load("dataset.jld", "large")
    summary(dataset)
    sleep(0.2)
end
```

```
Clusters: 3
Dimension (features): 200
Features per Cluster: 40
Probability of Activation: 0.8

Size: 1000
Min Cluster size: 200
Max Cluster size: 400
Cluster 1 size: 276
Cluster 2 size: 267
Cluster 3 size: 457
```

In [54]:
```
rm("dataset.jld")
```

In [55]:
```
default_datasetdir = "../dataset"
```

Out[55]:
```
"../dataset"
```

In [56]:
```
function export_dataset(name, dataset; datasetdir=default_datasetdir)
    path = datasetdir * "/" * name
    isdir(path) && rm(path, recursive=true)
    mkpath(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

let
    dataset = Dataset()
    export_dataset("test", dataset)
    readdir(default_datasetdir * "/test")
end
```

```
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"
```

```
In [57]: function load_dataset(name; datasetdir=default_datasetdir)
             path = datasetdir * "/" * name
             load(path * "/dataset.jld", "dataset")
         end

         load_dataset("test")
```

```
Out[57]: Dataset(3,200,40,0.8,1000,200,400,Input([[1,1,1,1,1,1,1,1,1,0  ...  1,1,1,0,0,0,1,0,1,0],[1,1,
```

```
In [58]: rm(default_datasetdir * "/test", recursive=true)
```