6_data_wrangling

2017 年 3 月 16 日

1 Introduction to Data Science with Julia

2 目次

3 データ分析入門

3.1 データの読み込み

世にあふれるデータは Excel や CSV, SQL などいろいろな形式で保存されていますが、この講義では取り扱いが容易な CSV (Comma-Separated Values) ファイルを使用していきます。CSV ファイルは中身を見ればわかりますが、各要素がコンマで区切られたテキストファイルです。テキストファイルであるためメモ帳などで簡単に編集することが出来ます。

まずは CSV ファイルからデータを読み込み、平均や分散などの基本統計量を計算してみましょう。今回はサンプルとして、学生の ID と 2 つの試験の点数が保存された scores.csv というファイルを用意しました。

Julia の標準機能を使って csv ファイルを読み込むには readcsv 関数を使います。

```
In [1]: scores = readcsv("../data/scores.csv", Int, header=true) # デフォルトでは header=fa
```

```
Out[1]: (
        [1 31 61; 2 70 54; ⋯ ; 99 96 92; 100 73 93],

        AbstractString["ID" "exam1" "exam2"])
```

データを読み込むと変数 scores の第 1 要素が数値データに、第 2 要素が header が入ります。

```
In [2]: scores[1]
```

```
Out[2]: 100 × 3 Array{Int64,2}:
           1   31   61
```

```
  2  70  54
  3  61  36
  4  77  91
  5  46  29
  6  40  59
  7  80  99
  8  67  52
  9  80  72
 10   1   4
 11  23   4
 12  79  86
 13  16   1
  ⋮
 89  95  97
 90  15   2
 91  72  88
 92  45  45
 93   1   9
 94  50  39
 95  79  88
 96  20  26
 97  97  63
 98  15  23
 99  96  92
100  73  93
```

```
In [3]: scores[2]
```

```
Out[3]: 1 × 3 Array{AbstractString,2}:
         "ID"  "exam1"  "exam2"
```

summarystats を使って 2 つの試験の点数の平均などを調べてみましょう。

```
In [4]: import StatsBase
        StatsBase.summarystats(scores[1][:,2])
```

```
Out[4]: Summary Stats:
        Mean:         47.640000
        Minimum:      1.000000
        1st Quartile: 19.000000
        Median:       51.500000
        3rd Quartile: 71.000000
        Maximum:      97.000000
```

```
In [5]: StatsBase.summarystats(scores[1][:,3])
```

```
Out[5]: Summary Stats:
        Mean:          49.970000
        Minimum:       0.000000
        1st Quartile: 27.000000
        Median:        49.000000
        3rd Quartile: 76.000000
        Maximum:       99.000000
```

exam 1 と exam 2 の相関係数も計算してみましょう

```
In [6]: corcoeff = cor(scores[1][:,2], scores[1][:,3])
```

```
Out[6]: 0.8268224737640034
```

数字だけを見ていてもよくわからないので可視化もしてみましょう。

```
In [7]: import Plots
        Plots.gr(leg=false)
```

```
Out[7]: Plots.GRBackend()
```

```
In [8]: @show a, b = linreg(scores[1][:,2], scores[1][:,3]) # y = a  + b * x にフィッティン
        Plots.plot(scores[1][:,2], scores[1][:,3],
            linetype=:scatter,
            leg=false,
            title="Scores",
            xlabel="exam 1",
            ylabel="exam 2",
            xticks=0:20:100,
            yticks=0:20:100,
            xlims=(0,100),
            ylims=(0,100),
            aspect_ratio=1
            )
        Plots.plot!([0,100], a + b * [0,100])
```

```
(a,b) = linreg((scores[1])[:,2],(scores[1])[:,3]) = (8.874846128532958,0.862618679082011
```

```
In [ ]:
```

目次に戻る

3.2 DataFrames パッケージ

先程は readcsv を使ってデータを読み込みましたが、header があるにも関わらず全く活用しませんでした。readcsv で取り込んだ場合、exam1 は 2 列目で、exam2 は 3 列目だから... と header と対応する列番号が必要でした。これではデータ数が増えるとほしいデータ列が何列目なのか数えるだけでも大変です。header

があるのだからこれを活用したいものです。

DataFrames パッケージを使うと、各列を header で指定することが出来ます。DataFrames パッケージをつかって CSV ファイルを読み込む場合は readtable を使用します。

(注) DataFrames.jl からの派生で DataTables.jl というものもあり、今後 DataTables.jl が主流になっていくと思います。

```
In [9]: import DataFrames
```

```
In [10]: df = DataFrames.readtable("../data/scores.csv")
         DataFrames.head(df, 5) # head(df, n)  最初の n 行目までを表示。tail を使うと末尾を表示
```

```
Out[10]: 5 × 3 DataFrames.DataFrame
         | Row | ID | exam1 | exam2 |
         |-----|----|-------|-------|
         | 1   | 1  | 31    | 61    |
         | 2   | 2  | 70    | 54    |
         | 3   | 3  | 61    | 36    |
         | 4   | 4  | 77    | 91    |
         | 5   | 5  | 46    | 29    |
```

読み込んだデータは一見すると配列に似ていますが、実際は DataFrames パッケージ内で定義された DataFrame 型という型で配列と似て非なるものです。概ね配列と同様に扱えますが、扱い方が変わる部分もあるのでそのことを念頭においてください。

```
In [11]: typeof(df)
```

```
Out[11]: DataFrames.DataFrame
```

各列を抜き出すには普通の配列のように

```
df[:,2]
```

とするか、抜き出す列の列番号または header で指定して

```
df[2]

or

df[:exam1]
```

などとします。

```
In [12]: df[:,2]
```

```
Out[12]: 100-element DataArrays.DataArray{Int64,1}:
          31
          70
          61
          77
          46
```

```
 40
 80
 67
 80
  1
 23
 79
 16
  ☒
 95
 15
 72
 45
  1
 50
 79
 20
 97
 15
 96
 73
```

```
In [13]: df[2]
```

```
Out[13]: 100-element DataArrays.DataArray{Int64,1}:
 31
 70
 61
 77
 46
 40
 80
 67
 80
  1
 23
 79
 16
  ☒
 95
 15
 72
 45
  1
```

```
 50
 79
 20
 97
 15
 96
 73
```

```
In [14]: df[:exam1]
```

```
Out[14]: 100-element DataArrays.DataArray{Int64,1}:
 31
 70
 61
 77
 46
 40
 80
 67
 80
  1
 23
 79
 16
  ⋅
 95
 15
 72
 45
  1
 50
 79
 20
 97
 15
 96
 73
```

複数の列を抜き出す場合は

```
df[ [:exam1, :exam2] ] # 内側のカッコは必須
```

のようにします。

```
In [15]: df[ [:exam1, :exam2] ]
```

```
Out[15]: 100 × 2 DataFrames.DataFrame
```

Row	exam1	exam2
1	31	61
2	70	54
3	61	36
4	77	91
5	46	29
6	40	59
7	80	99
8	67	52
9	80	72
10	1	4
11	23	4
⋮		
89	95	97
90	15	2
91	72	88
92	45	45
93	1	9
94	50	39
95	79	88
96	20	26
97	97	63
98	15	23
99	96	92
100	73	93

describe を使うと各列の平均、五数要約、欠損値の数と割合が標準出力されます。

```
In [16]: DataFrames.describe(df)
```

```
ID
Min       1.0
1st Qu.   25.75
Median    50.5
Mean      50.5
3rd Qu.   75.25
Max       100.0
NAs       0
NA%       0.0%

exam1
Min       1.0
1st Qu.   19.0
Median    51.5
```

```
Mean      47.64
3rd Qu.   71.0
Max       97.0
NAs       0
NA%       0.0%

exam2
Min       0.0
1st Qu.   27.0
Median    49.0
Mean      49.97
3rd Qu.   76.0
Max       99.0
NAs       0
NA%       0.0%
```

既存の配列から DataFrame を作るには次のようにします。

```
In [17]: x = [1, 2, 3]
         y = [4, 5, 6]
         tmpdf = DataFrames.DataFrame(X = x, Y = y) #  a = b としたとき、左の a が列名になる。
                                                    # 列名と配列の変数名は同じでも構わない。すなわ

Out[17]: 3 × 2 DataFrames.DataFrame
         | Row | X | Y |
         |─────|───|───|
         | 1   | 1 | 4 |
         | 2   | 2 | 5 |
         | 3   | 3 | 6 |

In [18]: tmpdf[:X]

Out[18]: 3-element DataArrays.DataArray{Int64,1}:
          1
          2
          3
```

可視化も配列の時同様にすることができます。

```
In [19]: Plots.plot(df[:exam1], df[:exam2],
             linetype=:scatter,
             leg=false,
             title="Scores",
             xlabel="exam 1",
             ylabel="exam 2",
```

```
        xticks=0:20:100,
        yticks=0:20:100,
        xlims=(0,100),
        ylims=(0,100),
        aspect_ratio=1)
```

Plots にさらに箱ひげ図などの描写機能を加え、DataFrame 型にも対応している StatPlots パッケージを使うとより分析がはかどります。

```
In [20]: import StatPlots # 文法は Plots と概ね一緒
         StatPlots.gr()
```

```
Out[20]: Plots.GRBackend()
```

普通の Plots では出来きませんが、StatPlots は DataFrame 型に対応しているので次のようにも書けます

```
In [21]: StatPlots.plot(df, :exam1, :exam2, linetype=:scatter) # plot(df[:exam1], df[:ex
```

```
In [22]: StatPlots.histogram(df, :exam1, bins=0:5:100, label="exam 1")
         StatPlots.histogram!(df, :exam2, bins=0:5:100, alpha=0.5, label="exam 2", leg=t
         StatPlots.xlabel!("score")
```

```
In [23]: StatPlots.boxplot(df, ["exam 1" "exam 2"], :exam1, :exam2)
         StatPlots.ylabel!("score")
```

```
In [ ]:
```

目次に戻る

3.3 データ加工

現在では統計局や Kaggle などから様々なデータをダウンロード出来ますが、それらのデータが始めから解析しやすい形式になっているとは限りません。そのため、まずは解析がしやすくするためにデータを加工する必要が有ります。

まずは ID、性別、誕生日だけが入った単純なデータ people.csv を使ってデータ加工に慣れていきましょう。

```
In [24]: people = DataFrames.readtable("../data/people.csv")
         DataFrames.head(people)
```

```
Out[24]: 6 × 3 DataFrames.DataFrame
         | Row | ID | Sex      | Birthday     |
         |-----|----|----------|--------------|
         | 1   | 1  | "female" | "1984-08-30" |
         | 2   | 2  | "F"      | "1978-12-15" |
         | 3   | 3  | "Male"   | "1982-10-09" |
         | 4   | 4  | "Male"   | "1980-08-15" |
         | 5   | 5  | "F"      | "1980-01-12" |
         | 6   | 6  | "male"   | "1974-03-04" |
```

各列の要素の型を調べてみます。

```
In [25]: DataFrames.eltypes(people)
```

```
Out[25]: 3-element Array{Type,1}:
          Int64
          String
          String
```

ID 列は Int64 で、性別と誕生日の列は String 型のようです。Julia には日付を扱うための Date 型があるので、より Julia で扱いやすいように誕生日は String 型から Date 型へ変換したくなります。性別の方も書き方が何通りかあるようなので表し方を揃えたいところです。

初めに変換が簡単そうな誕生日の方からやっていきましょう。今回のように「年-月-日」という日付の表し方だと Date を使うだけで簡単に Date 型になります。

```
In [26]: Date(people[:Birthday][1])
```

```
Out[26]: 1984-08-30
```

ヨーロッパ式に 日・月・年 の順に書かれていた場合は次のように書きます。

```
In [27]: Date("12-31-2020", "m-d-y")
```

```
Out[27]: 2020-12-31
```

一見すると変換されていないようですが、型を調べるとちゃんと型変換がされています。

```
In [28]: @show typeof(people[:Birthday][1])
         @show typeof(Date(people[:Birthday][1]));
```

```
typeof((people[:Birthday])[1]) = String
typeof(Date((people[:Birthday])[1])) = Date
```

Date 型にすれば、2 つの日付の引き算をすると経過日数を計算することが出来ます。

```
In [29]: yourbirthday = "2000-1-1"
         print("今日はあなたが生まれてから ", Dates.today() - Date(yourbirthday), " です。")
```

```
今日はあなたが生まれてから 6284 days です。
```

```
In [ ]:
```

これから、誕生日の列を String 型から Date 型に変換していきますが、もともとの誕生日の列は String 型なので、変換した Date 型を直接代入することは出来ません。そのため、ここでは 1. birthday という列を新しく作る 1. birthday 列に変換後の誕生日を入れる 1. 元の Birthday 列を削除 1. birthday を Birthday に名前を変更

という手順を踏んでいくことにします。

```
In [30]: # 新しい列を作成
         # 初期化していないので各要素の日付に意味はありません。
         people[:birthday] = Array{Date}(size(people, 1))
```

```
Out[30]: 1000-element Array{Date,1}:
```

```
 0000-12-31
 47036885-03-21
 0001-01-03
 47036885-03-20
 264799-11-21
 271940-08-12
 271940-08-12
 271940-08-28
 0116-06-06
 0012-04-04
 233885-07-10
 23518443-02-10
 261588-07-09
 □
 276653-09-17
 271962-03-02
 271962-03-10
 271962-04-03
 276642-07-01
 267232-09-06
 1375874798-11-01
 271962-01-13
 271962-04-27
 271962-04-27
 271962-04-27
 271962-01-13
```

In [31]:
```
# 型を変換し代入する
for i in 1:size(people, 1)
    people[:birthday][i] = Date(people[:Birthday][i])
end
DataFrames.head(people)
```

Out[31]: 6 × 4 DataFrames.DataFrame

Row	ID	Sex	Birthday	birthday
1	1	"female"	"1984-08-30"	1984-08-30
2	2	"F"	"1978-12-15"	1978-12-15
3	3	"Male"	"1982-10-09"	1982-10-09
4	4	"Male"	"1980-08-15"	1980-08-15
5	5	"F"	"1980-01-12"	1980-01-12
6	6	"male"	"1974-03-04"	1974-03-04

In [32]:
```
# Birthday 列を削除
delete!(people, :Birthday)
```

```
DataFrames.head(people)
```

```
Out[32]: 6 × 3 DataFrames.DataFrame
| Row | ID | Sex      | birthday   |
|-----|----|----------|------------|
| 1   | 1  | "female" | 1984-08-30 |
| 2   | 2  | "F"      | 1978-12-15 |
| 3   | 3  | "Male"   | 1982-10-09 |
| 4   | 4  | "Male"   | 1980-08-15 |
| 5   | 5  | "F"      | 1980-01-12 |
| 6   | 6  | "male"   | 1974-03-04 |
```

```
In [33]: # 列名を変更
DataFrames.rename!(people, :birthday, :Birthday)
DataFrames.head(people)
```

```
Out[33]: 6 × 3 DataFrames.DataFrame
| Row | ID | Sex      | Birthday   |
|-----|----|----------|------------|
| 1   | 1  | "female" | 1984-08-30 |
| 2   | 2  | "F"      | 1978-12-15 |
| 3   | 3  | "Male"   | 1982-10-09 |
| 4   | 4  | "Male"   | 1980-08-15 |
| 5   | 5  | "F"      | 1980-01-12 |
| 6   | 6  | "male"   | 1974-03-04 |
```

```
In [34]: # 無事に変換できたか確認
DataFrames.eltypes(people)
```

```
Out[34]: 3-element Array{Type,1}:
 Int64
 String
 Date
```

注) 今回の様に、変換する列に欠損値が存在しない場合、新しく列を作らずとも一気に型変換できます。

```
In [35]: # people = readtable("people.csv")
# people[:Birthday] = Date(people[:Birthday])
# @show head(people)
# eltypes(people)
```

次に性別の方を加工していきましょう。性別の列を見るとどうやら男女の書き方が何通りかあるようです。

```
In [36]: people[:Sex]
```

```
Out[36]: 1000-element DataArrays.DataArray{String,1}:
 "female"
 "F"
```

```
"Male"
"Male"
"F"
"male"
"female"
"M"
"F"
"F"
"Female"
"Male"
"female"
□
"female"
"female"
"male"
"F"
"Female"
"male"
"female"
"female"
"Female"
"male"
"Male"
"male"
```

このままでは、何通りの書き方があるのかはっきりしないので、Set を使ってこの列から重複する要素を取り除きましょう。 Set 型は数学の集合の同様、要素の順序に意味はありません。

```
In [37]: Set(people[:Sex])
```

```
Out[37]: Set(String["F","female","male","Female","Male","M"])
```

これから、男性は M, Male, male の 3 通りの書き方が、女性は F, Female, female の 3 通りの書き方があることがわかります。3 通りもあると扱いが大変なので、ここでは男性は M、女性は F で統一していきましょう。

この性別の列の加工は皆さんが実際にコードを書いてみてください。

ヒント

```
x in 配列 （または集合）
    or
x ∈ 配列 （または集合）
```

とすると x が配列の中に含まれるかどうかを確認することが出来る。

```
In [38]: @show 1 in [1, 2, 3]
         @show 5 ∈ [1, 2, 3];
```

```
1 in [1,2,3] = true
5 ∈  [1,2,3] = false
```

```
In [39]: # 実際にコードを書いてみてください
```

```
In [ ]:
```

加工ができたら最後に加工したデータを保存しましょう。

```
In [40]: # 元のデータと同じ名前にすると上書きされてしまうので、必ず違うファイル名をつける。
         DataFrames.writetable("people_fix.csv", people)
```

上記の方法で保存したファイルをもう一度読み込むと誕生日の列はまた String 型で読み込まれます。そのため扱いやすくするにはまた Date 型へまた変換する必要が有ります。

せっかく扱いやすい型に加工したのだから、その型のまま保存・読み込みがしたいものです。そのようなときには JLD.jl パッケージを使うと型の情報を保持したまま比較的楽に変数を保存することが出来ます。

```
In [41]: import JLD
```

```
In [42]: # 変数を保存
         JLD.@save "people_fix.jld" people

         # 読み込むときは
         # JLD.@load "people_fix.jld"
```

```
In [ ]:
```

目次に戻る

3.4 データの結合

まずは変数の保存が正しく行われたか確認するために一度カーネルを restart して、下記のコマンドを実行して変数を読み込んでみましょう。

カーネルの restart の方法は上の Kernel のタブをクリックし、Restart をクリックしてください。これを行うと Julia を再起動したのと同じになります。そのため、今まで計算してきた変数などの情報は失われます。

```
In [43]: import JLD, DataFrames
         JLD.@load "people_fix.jld"
         DataFrames.head(people)
```

```
Out[43]: 6 × 3 DataFrames.DataFrame
         | Row | ID | Sex      | Birthday   |
         |-----|----|----------|------------|
         | 1   | 1  | "female" | 1984-08-30 |
         | 2   | 2  | "F"      | 1978-12-15 |
         | 3   | 3  | "Male"   | 1982-10-09 |
         | 4   | 4  | "Male"   | 1980-08-15 |
         | 5   | 5  | "F"      | 1980-01-12 |
```

```
| 6   | 6   | "male"   | 1974-03-04 |
```

```
In [44]: DataFrames.eltypes(people)
```

```
Out[44]: 3-element Array{Type,1}:
          Int64
          String
          Date
```

さらに、今回は people_satisfaction.csv という CSV ファイルも読み込みます。この people_satisfaction.csv 中には ID、あるサービスに対する満足度の 2 つのデータが入っています。ここで、people.csv と people_satisfaction.csv の ID が同じ人は同一人物だとします。

満足度は数値で入っており 1. 非常に不満 1. 不満 1. 普通 1. 満足 1. 非常に満足
を表します。

```
In [45]: people_satisfaction = DataFrames.readtable("../data/people_satisfaction.csv")
         DataFrames.head(people_satisfaction)
```

```
Out[45]: 6 × 2 DataFrames.DataFrame
         | Row | ID   | Satisfaction |
         |-----|------|--------------|
         | 1   | 1000 | 4            |
         | 2   | 999  | 3            |
         | 3   | 998  | 2            |
         | 4   | 997  | 3            |
         | 5   | 996  | 2            |
         | 6   | 995  | 2            |
```

次に people と people_satisfaction のデータを結合していきます。今回は共通項目として ID があるのでこれを基準にして結合します。 2 つのデータを結合する場合には join 関数を使います。

```
In [46]: people = DataFrames.join(people, people_satisfaction, on = :ID, kind = :outer)
         DataFrames.head(people)
```

```
Out[46]: 6 × 4 DataFrames.DataFrame
         | Row | ID | Sex      | Birthday   | Satisfaction |
         |-----|----|----------|------------|--------------|
         | 1   | 1  | "female" | 1984-08-30 | 3            |
         | 2   | 2  | "F"      | 1978-12-15 | NA           |
         | 3   | 3  | "Male"   | 1982-10-09 | 3            |
         | 4   | 4  | "Male"   | 1980-08-15 | 2            |
         | 5   | 5  | "F"      | 1980-01-12 | 2            |
         | 6   | 6  | "male"   | 1974-03-04 | 4            |
```

kind を変えると結合方法が変わります。詳しくは公式ドキュメントを読んでください。

今回の様に ID が 2 つのファイルで完全に同じ場合、

```
people[:Satisfaction] = sort(people_satisfaction)[:Satisfaction]
```

として代入することも出来ますが、常に一致する ID があるとは限らないので join 関数を使ったほうが無難です。

目次に戻る

3.5 欠損値

結合結果を見ると ID 2 の人の Satisfaction が NA となっていることがわかります。ここで NA とは欠損値を表します。アンケートで無回答だった場合や、実験でサンプルが取れなかった場合は欠損値になります。

欠損値がある場合、単純に平均などを求めることが出来ません。

```
In [47]: mean(people[:Satisfaction])
```

```
Out[47]: NA
```

データから欠損値を取り除くには dropna を使います。

```
In [48]: DataFrames.dropna(people[:Satisfaction])
```

```
Out[48]: 898-element Array{Int64,1}:
           3
           3
           2
           2
           4
           3
           3
           3
           4
           1
           1
           2
           1
           ⌧
           5
           2
           4
           2
           2
           2
           2
           2
           3
           2
           3
           4
```

```
In [49]: mean(DataFrames.dropna(people[:Satisfaction]))
```

```
Out[49]: 2.8106904231625833
```

欠損値があるデータを describe を使ってみると、欠損値を取り除いたデータでの平均などが表示されます。

```
In [50]: DataFrames.describe(people)
```

```
ID
Min       1.0
1st Qu.   250.75
Median    500.5
Mean      500.5
3rd Qu.   750.25
Max       1000.0
NAs       0
NA%       0.0%

Sex
Length  1000
Type    String
NAs     0
NA%     0.0%
Unique  6

Birthday
Length  1000
Type    Date
NAs     0
NA%     0.0%
Unique  949

Satisfaction
Min       1.0
1st Qu.   2.0
Median    3.0
Mean      2.8106904231625833
3rd Qu.   4.0
Max       5.0
NAs       102
NA%       10.2%
```

describe の出力結果より Satisfaction には欠損値が 102 個あることがわかります。この欠損値をどう扱うかというのは難しい問題ですが、ここでは欠損値を平均値に置き換えてみます。

```
In [51]: people[:Satisfaction_fix] = Vector{Float64}(size(people, 1))
```

```
        for iter in 1:size(people, 1)
            if DataFrames.isna(people[:Satisfaction][iter])
                people[:Satisfaction_fix][iter] = mean(DataFrames.dropna(people[:Satisf
            else
                people[:Satisfaction_fix][iter] = people[:Satisfaction][iter]
            end
        end
        DataFrames.head(people)
```

Out[51]: 6 × 5 DataFrames.DataFrame

Row	ID	Sex	Birthday	Satisfaction	Satisfaction_fix
1	1	"female"	1984-08-30	3	3.0
2	2	"F"	1978-12-15	NA	2.81069
3	3	"Male"	1982-10-09	3	3.0
4	4	"Male"	1980-08-15	2	2.0
5	5	"F"	1980-01-12	2	2.0
6	6	"male"	1974-03-04	4	4.0

要素が NA か否かを調べるときには isna 関数を使います。isna 関数は引数が NA ならば true を、そうでなければ false を返します。

In [ ]:

上の例では for 文を使って欠損値を平均値に置き換えましたが、convert を使うと欠損値をコード一行書きで置き換えることが出来ます。

```
In [52]: # meansatisfaction = mean(DataFrames.dropna(people[:Satisfaction]))
         # people[:Satisfaction_fix] = convert(Vector{Float64}, people[:Satisfaction], m
```

目次に戻る

4 練習問題

4.1 1.

people から年齢のヒストグラムを作成せよ。ただし、年齢は数え年とする。

In [ ]:

4.2 2.

2 つのデータセット names, jobs を ID を基準に結合せよ。 また、kind オプションを変えるとどのような結果が得られるのか確認せよ。 ※オプションは公式ドキュメント参照

```
using DataFrames
names = DataFrame(ID = [1, 2], Name = ["John Doe", "Jane Doe"])
jobs = DataFrame(ID = [1, 3], Job = ["Lawyer", "Doctor"])
```

In [ ]:

4.3 3.

下記のデータの欠損値を全て中央値に置き換えよ

```
srand(1)
ex = DataFrame()
ex[:sample] = @data([rand(@data([NA, 1, 2, 3, 4, 5])) for i in 1:100])
```

In [ ]:

目次に戻る