Scientific Computing and (Big) Data Analysis with Julia

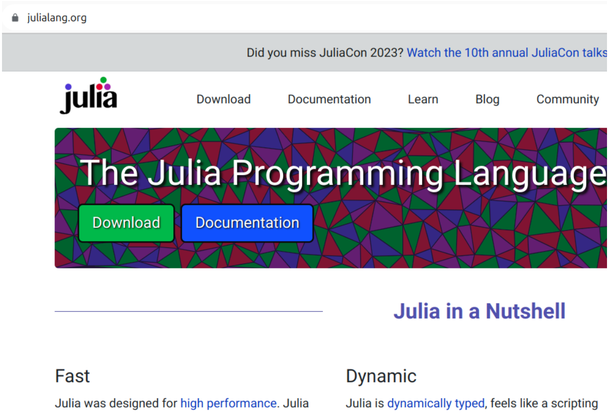
Prof.Dr.Mehmet Hakan Satman
mhsatman@istanbul.edu.tr

Istanbul University

2023.12.01

Julia Programming Language

julialang.org



Julia Programming Language

High Performance Computing

- Julia was designed for high performance. Julia programs automatically compile to efficient native code via LLVM, and support multiple platforms (Windows, MacOS, Linux, etc.)[1].

[1] `https://julialang.org/`

- Julia is dynamically typed, feels like a scripting language, and has good support for interactive use, but can also optionally be separately compiled[2].

[2] https://julialang.org/

Julia Programming Language

Composable

- Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns. The talk on the Unreasonable Effectiveness of Multiple Dispatch explains why it works so well[3].

[3] https://julialang.org/

Julia Programming Language

Open Source

- Julia is an open source project with over 1,000 contributors. It is made available under the MIT license. The source code is available on GitHub[4].

[4] https://julialang.org/

Julia Programming Language

First things first!

```
println("Hello, world")
```

Julia Programming Language

Welcome!

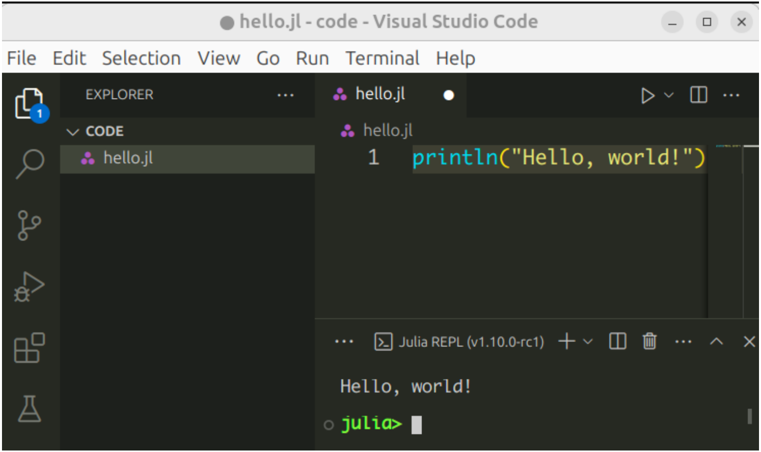
```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.10.0-rc1 (2023-11-03)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> println("Hello, world!")
Hello, world!

julia>
```

Julia Programming Language

The editor: Visual Studio Code



Linear Regression

The formulation

$$y = \beta_0 + \beta_1 x + \varepsilon \tag{1}$$

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x \tag{2}$$

$$\hat{\boldsymbol{\beta}} = (X'X)^{-1}X'y \tag{3}$$

Linear Regression
Sample Data

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{bmatrix}, y = \begin{bmatrix} 2 \\ 5 \\ 5 \\ 8 \\ 12 \end{bmatrix} \tag{4}$$

Linear Regression

The Matrix Solution

```
using LinearAlgebra

x = [1, 2, 3, 4, 5]
y = [2, 5, 5, 8, 12]
X = hcat(ones(5), x)
betahats = inv(X'X)X'y
println(betahats)
```

```
julia> include("reg-matrix.jl")
[-0.5, 2.3]
```

Linear Regression

Pseudo Inverse - Numerical Fit

```
x = [1, 2, 3, 4, 5]
y = [2, 5, 5, 8, 12]

betahats = hcat(ones(5), x) \ y
println(betahats)
```

```
julia> include("reg-simple.jl")
[-0.5, 2.3]
```

Linear Regression

The GLM package

```
using GLM

x = [1, 2, 3, 4, 5]
y = [2, 5, 5, 8, 12]

result = lm(hcat(ones(5), x), y)

println(result)
```

Linear Regression

The GLM package - Results

```
julia> include("reg-glm.jl")

 Coefficients:
─────────────────────────────────
        Coef.   Std. Error    t      Pr(>|t|)   Lower 95%   Upper 95%
─────────────────────────────────
x1     -0.5     1.25565     -0.40    0.7171    -4.49605     3.49605
x2      2.3     0.378594     6.08    0.0090     1.09515     3.50485
─────────────────────────────────
```

```
julia> GLM.r2(result)
0.9248251748251748
```

XOR

eXclusive OR

x_1	x_2	y
1	1	0
1	0	1
0	1	1
0	0	0

Table: $y = \text{xor}(x_1, x_2)$

Symbolic Regression

```
using SymbolicRegression, MLJ

x = (
    x1 = Float64[1, 1, 0, 0],
    x2 = Float64[1, 0, 1, 0]
)

y = Float64[0, 1, 1, 0]
```

Symbolic Regression

```
model = SRRegressor(
    niterations = 50,
    binary_operators = [+, -, *],
    unary_operators = [abs],
    should_simplify = true,
    save_to_file = false)
```

Symbolic Regression

```
mach = machine(model, x, y)
fit!(mach)
report(mach)
@info predict(mach, x)
```

Symbolic Regression

```
Hall of Fame:
-------------------
Complexity  Loss       Score      Equation
1           2.500e-01  3.604e+01  y = 0.5
4           0.000e+00  1.201e+01  y = abs(x1 - x2)
---------------------
[ Info: [0.0, 1.0, 1.0, 0.0]
```