Getting started with Julia and machine learning

Anthony Blaom, Samuel Okon

July, 2022

Introducing Julia

Julia is an open source, general-purpose programming language, in the fourth year of its first **stable** release 1.0.

Introducing Julia

Julia is an open source, general-purpose programming language, in the fourth year of its first **stable** release 1.0.

Key take aways

- Julia let's you write **fast** code **fast**.

Introducing Julia

Julia is an open source, general-purpose programming language, in the fourth year of its first **stable** release 1.0.

Key take aways

- Julia let's you write **fast** code **fast**.
- Maximally hackable.

How to entertain your kids at home - the case for ML

Playmobil ©

Lego ©

3 models dinosaurs + 15 user contributed models

We support you in building your snake powered train

Introducing Julia

Julia is an open source, general-purpose programming language, in the fourth year of its first **stable** release 1.0.

Key take aways

- Julia let's you write **fast** code **fast**.
- Maximally hackable.
- Julia's design makes **extending** Julia code easy, promoting a relatively rapid expansion of third party libraries.

Julia's secret sauce

- **Just-in-time compilation**
- **Multiple dispatch**
- **Abstract type system** (typing is dynamic, nominative and parametric)

Workshop resources:

`github.com/ablaom/HelloJulia.jl`

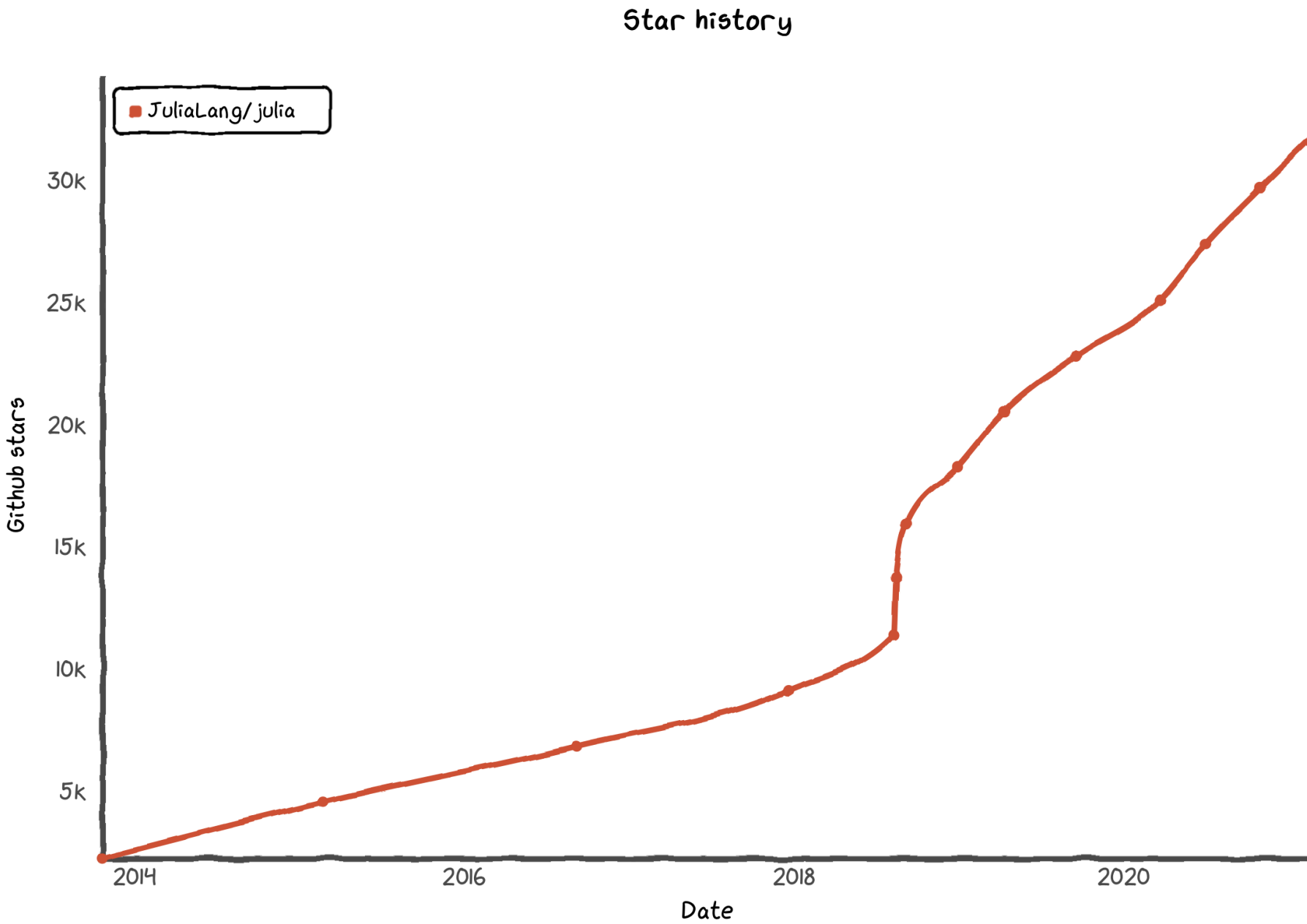
Secret sauce demo:

`github.com/ablaom/HelloJulia.jl`
`/tree/dev/notebooks/secret_sauce`

Package composability demo:

`github.com/ablaom/HelloJulia.jl`
`/tree/dev/notebooks/pkg_composability`

```
function mandel(z)
    c = z      # starting value and constant shift
    max_iterations = 20
    for n = 1:max_iterations
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
    return max_iterations
end
```


Star history
JuliaLang/julia
30k
25k
20k
15k
10k
5k
Github stars
2014
2016
2018
2020
Date

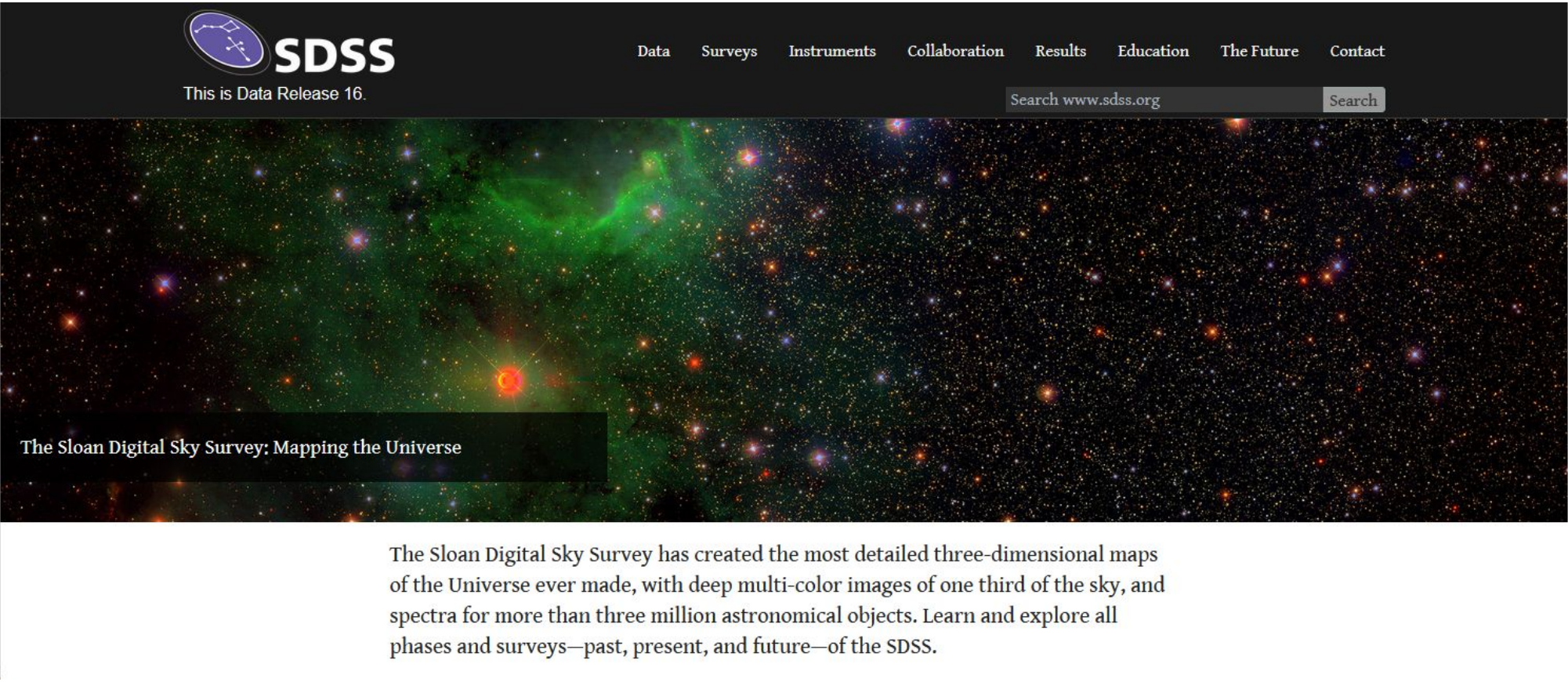
JuliaCon 2018: 90 talks and 350 attendees

University College London

Julia adoption accelerated at a rapid pace in 2020.

	Total Cumulative as of Jan 1, 2020	Total Cumulative as of Jan 1, 2021	Change
Number of Julia Downloads (JuliaLang + Docker + JuliaPro)	12,950,630	24,205,141	+87%
Number of Julia Packages	2,787	4,809	+73%
GitHub stars (Julia language repo + registered packages)	99,830	161,774	+62%
YouTube views (Julia language channel)	1,562,223	3,320,915	+113%
Published citations of Julia: A Fast Dynamic Language for Technical Computing (2012) + Julia: A Fresh Approach to Numerical Computing (2017)	1,680	2,531	+51%
Discourse posts	137,399	211,888	+54%
TIOBE Index Rank	#47	#23	+24

Sloan Digital Sky Survey



Cataloging the Visible Universe through Bayesian Inference at Petascale

Jeffrey Regier*, Kiran Pamnany†, Keno Fischer‡, Andreas Noack§, Maximilian Lam*, Jarrett Revels§, Steve Howard¶, Ryan Giordano¶, David Schlegel‖, Jon McAuliffe¶, Rollin Thomas‖, Prabhat‖

*Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
†Parallel Computing Lab, Intel Corporation
‡Julia Computing
§Computer Science and AI Laboratories, Massachusetts Institute of Technology
¶Department of Statistics, University of California, Berkeley
‖Lawrence Berkeley National Laboratory

***Abstract*—Astronomical catalogs derived from wide-field imaging surveys are an important tool for understanding the Universe. We construct an astronomical catalog from 55 TB of imaging data using Celeste, a Bayesian variational inference code written entirely in the high-productivity programming language Julia. Using over 1.3 million threads on 650,000 Intel Xeon Phi cores of the Cori Phase II supercomputer, Celeste achieves a peak rate of 1.54 DP PFLOP/s. Celeste is able to jointly optimize parameters for 188M stars and galaxies, loading and processing 178 TB across 8192 nodes in 14.6 minutes. To achieve this, Celeste exploits parallelism at multiple levels (cluster, node, and thread) and accelerates I/O through Cori's Burst Buffer. Julia's native performance enables Celeste to employ high-level constructs without resorting to hand-written or generated low-level code (C/C++/Fortran), and yet achieve petascale performance.**

***Keywords*-astronomy, Bayesian, variational inference, Julia, high performance computing**

I. Introduction

Modern astronomical surveys produce vast amounts of data. Our work uses images from the Sloan Digital Sky Survey (SDSS) [1] as a test case to demonstrate a new, highly scalable algorithm for constructing astronomical catalogs. SDSS produced 55 TB of images covering 35% of the sky and a catalog of 470 million unique light sources. Figure 1 shows image boundaries for a small region from SDSS. Advances in detector fabrication technology, computing power, and networking have enabled the development of a new generation of upcoming wide-field sky surveys orders of magnitude larger than SDSS. In 2020, the Large Synoptic Survey Telescope (LSST) will begin to obtain more than 15 TB of new images nightly [2]. The instrument will produce 10s–100s of PBs of imaging and catalog data over the lifetime of the project.

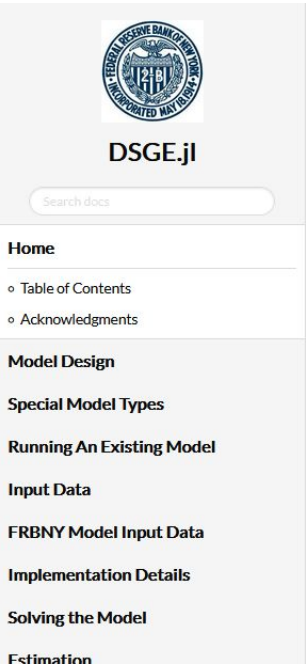
Producing an astronomical catalog is challenging in part because the parameters of overlapping light sources must be learned collectively: the optimal parameters for one light

JULIA LANGUAGE DELIVERS PETASCALE HPC PERFORMANCE

November 28, 2017 Rob Farber

Written in the productivity language **Julia**, the **Celeste project**—which aims to catalogue all of the telescope data for the stars and galaxies in in the visible universe—demonstrated the first Julia application to exceed 1 PF/s of double-precision floating-point performance (specifically 1.54 PF/s).

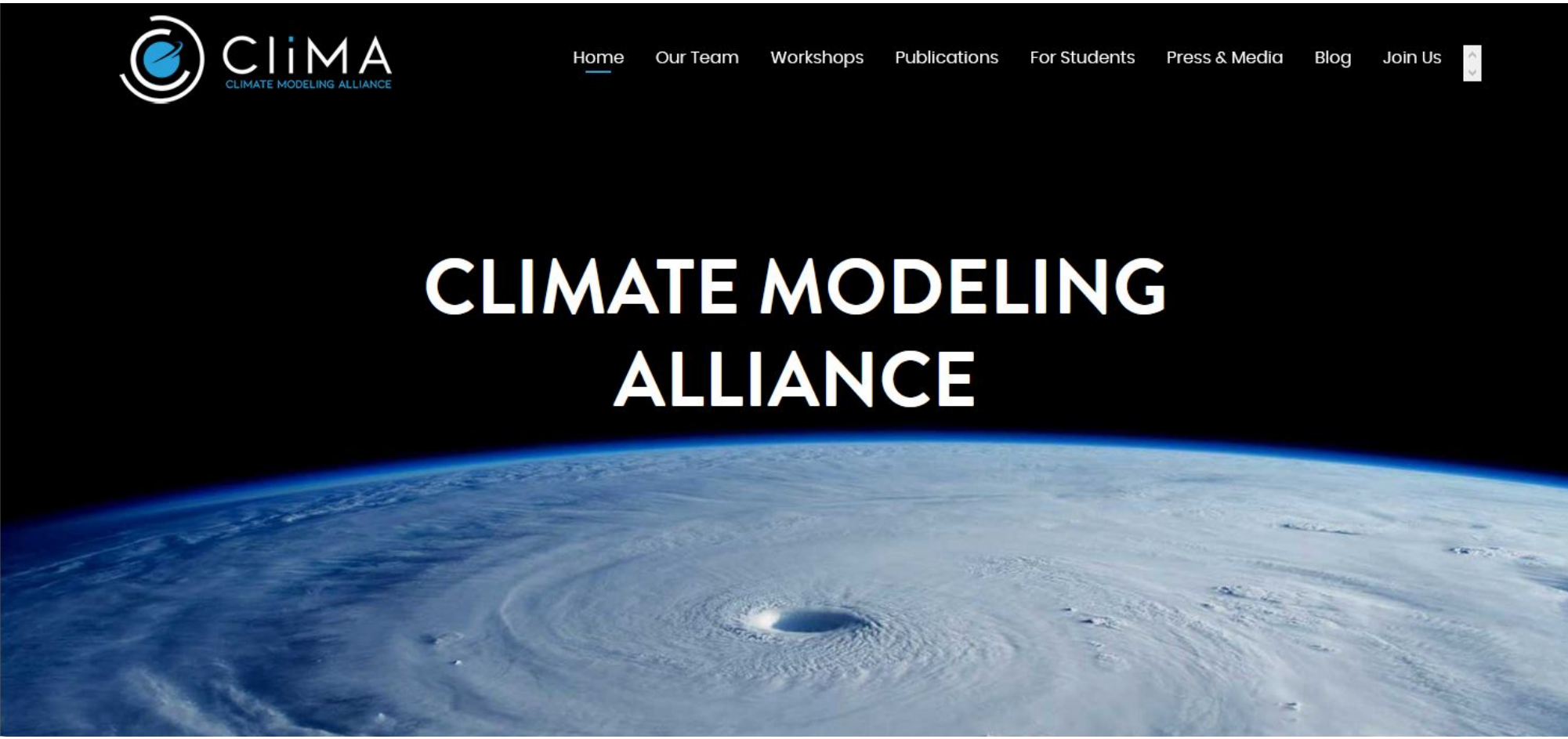
Federal Reserve Bank of New York

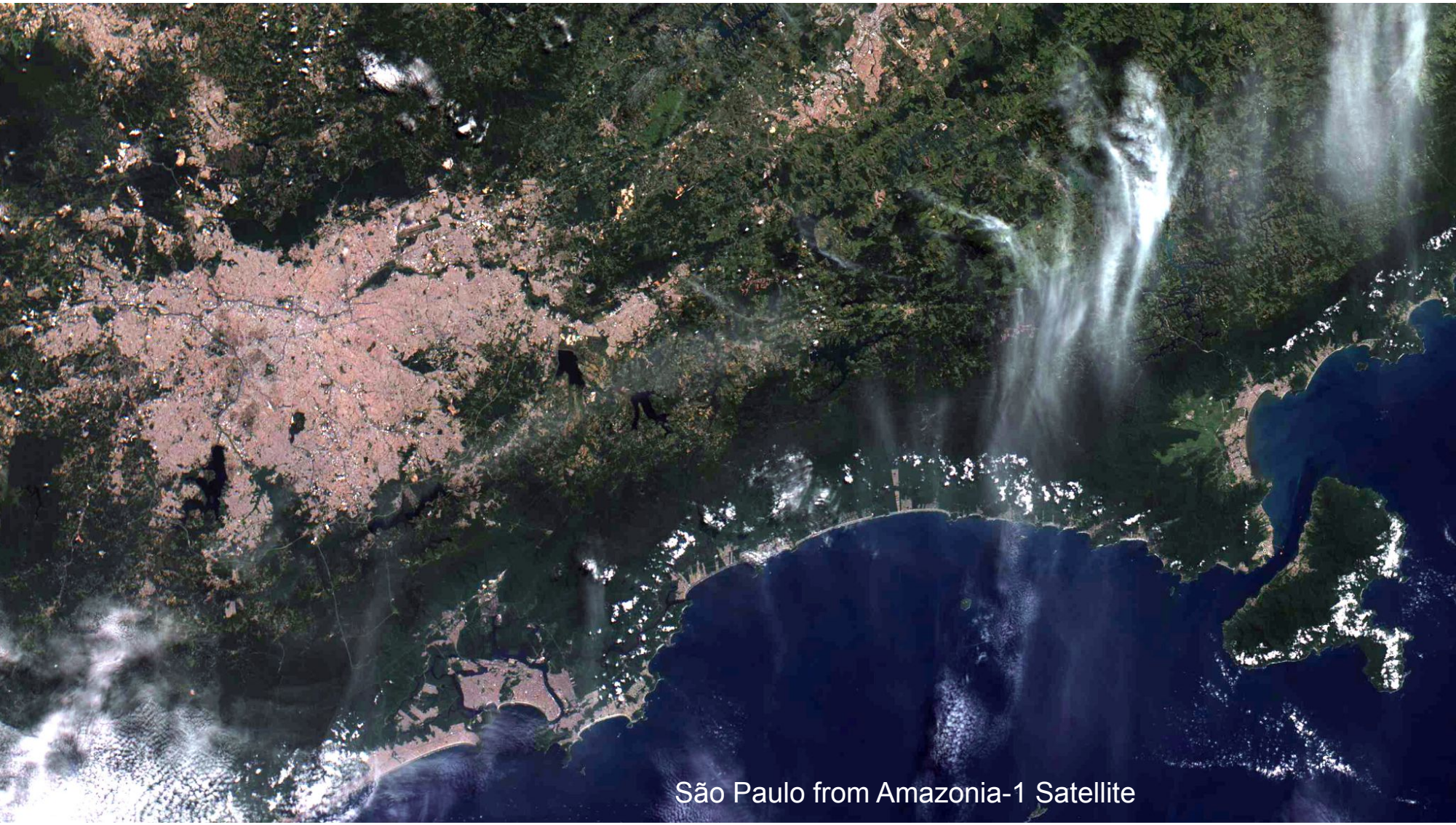


You may wonder why we wrote our code, which was originally in MATLAB and made available here, in Julia. MATLAB is a widely used, mature programming language that has served our purposes very well for many years. However, Julia has two main advantages from our perspective. First, as free software, Julia is more accessible to users from academic institutions or organizations without the resources for purchasing a license. Now anyone, from Kathmandu to Timbuktu, can run our code at no cost. Second, as the models that we use for forecasting and policy analysis grow more complicated, we need a language that can perform computations at a high speed. Julia boasts performance as fast as that of languages like C or Fortran, and is still simple to learn. (Read this post, written by the creators of the language, to understand why Julia fits the bill.) We want to address hard questions with our models—from understanding financial markets developments to modeling households' heterogeneity—and we can do so only if we are close to the frontier of programming.

We tested our code and found that the model estimation is about ten times faster with Julia than before, a very large improvement. Our ports (computer lingo for "translations") of certain algorithms, such as Chris Sims's gensys (which computes the model solution), also ran about six times faster in Julia than the MATLAB versions we had previously used. (These results should not be interpreted as a broad assessment of the speed of Julia relative to MATLAB, as they apply only to the code we have written.) This document written by the New York Fed and QuantEcon collaborators, who did the real work on the port, documents the speed improvements and offers an overview of the hurdles encountered in the translation of a large codebase from one language to another. We hope it will be of use to other central banks and researchers embarking on similar projects.

https://libertystreeteconomics.newyorkfed.org/2015/12/the-frbny-dsge-model-meets-julia.html


CliMA
CLIMATE MODELING ALLIANCE
Home
Our Team
Workshops
Publications
For Students
Press & Media
Blog
Join Us
CLIMATE MODELING
ALLIANCE



São Paulo from Amazonia-1 Satellite

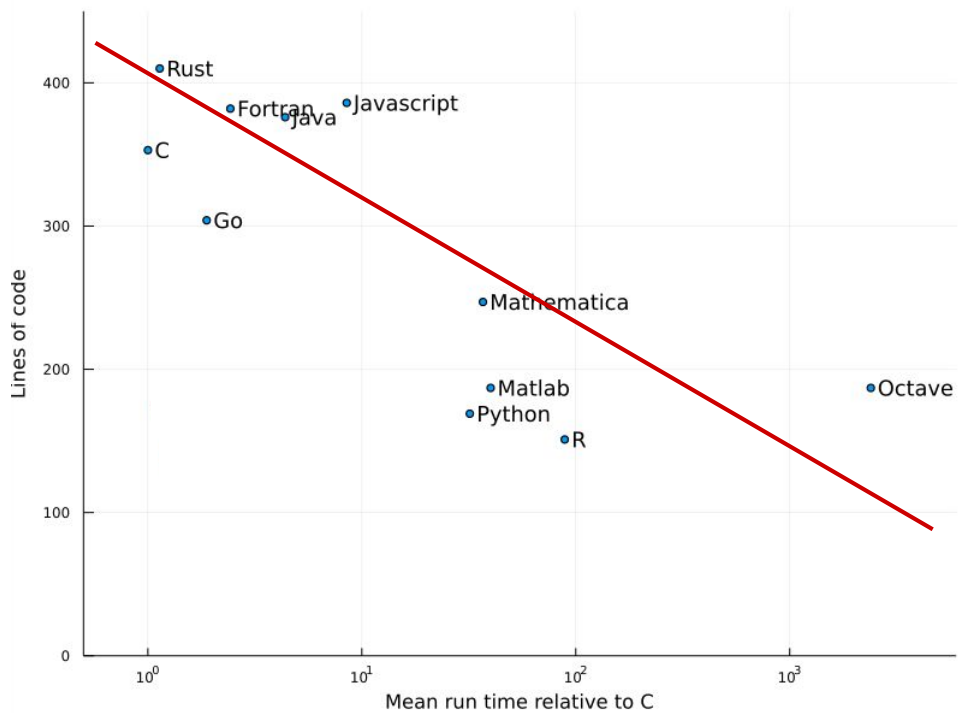
Popular libraries (packages)

These are all pretty mature:

- DataFrames.jl and CSV.jl - **in-memory data manipulation**
- Plots.jl (also easy to call R's ggplot())
- $\star$ JuMP.jl - **constrained optimization**
- $\star$ DifferentialEquations.jl
- StatsModels.jl, GLM.jl - **traditional stats models**
- Flux.jl - **deep learning**
- $\star$ Turing.jl, Soss.jl, ... **probabilistic programmming**
- MLJ.jl - multi-pardigm **machine learning** platform
- Pluto.jl - **"reactive" notebooks**

The two-language problem

Compiled, statically typed



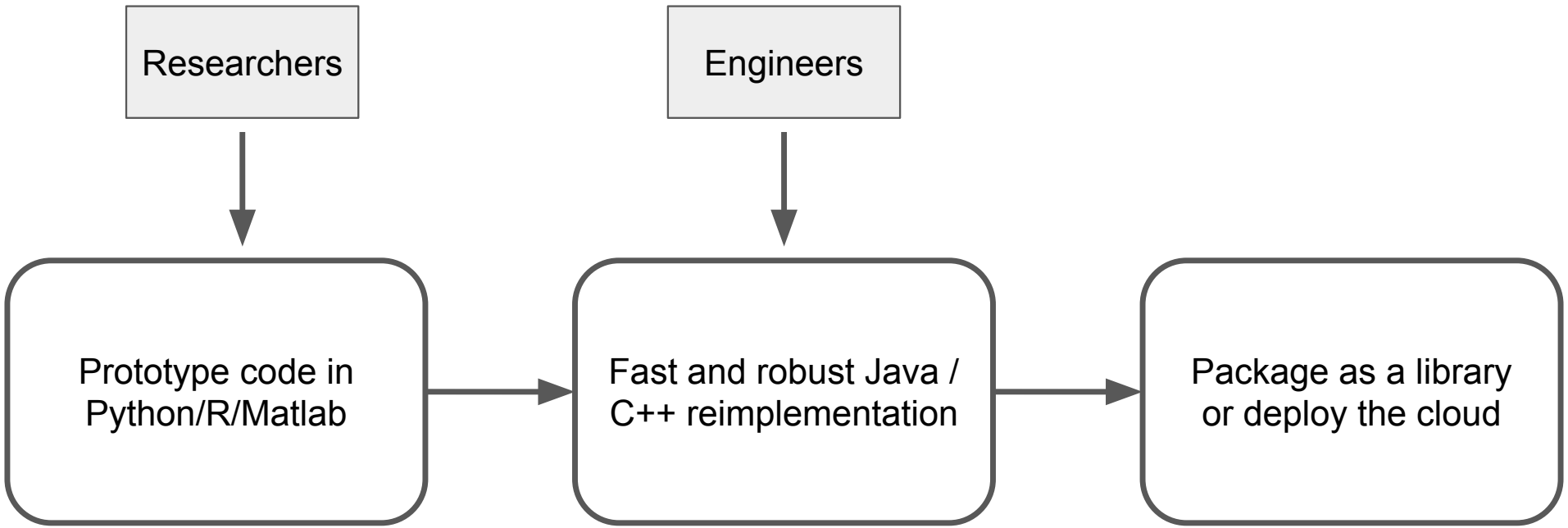
Interactive, dynamically typed

https://github.com/JuliaLang/Microbenchmarks

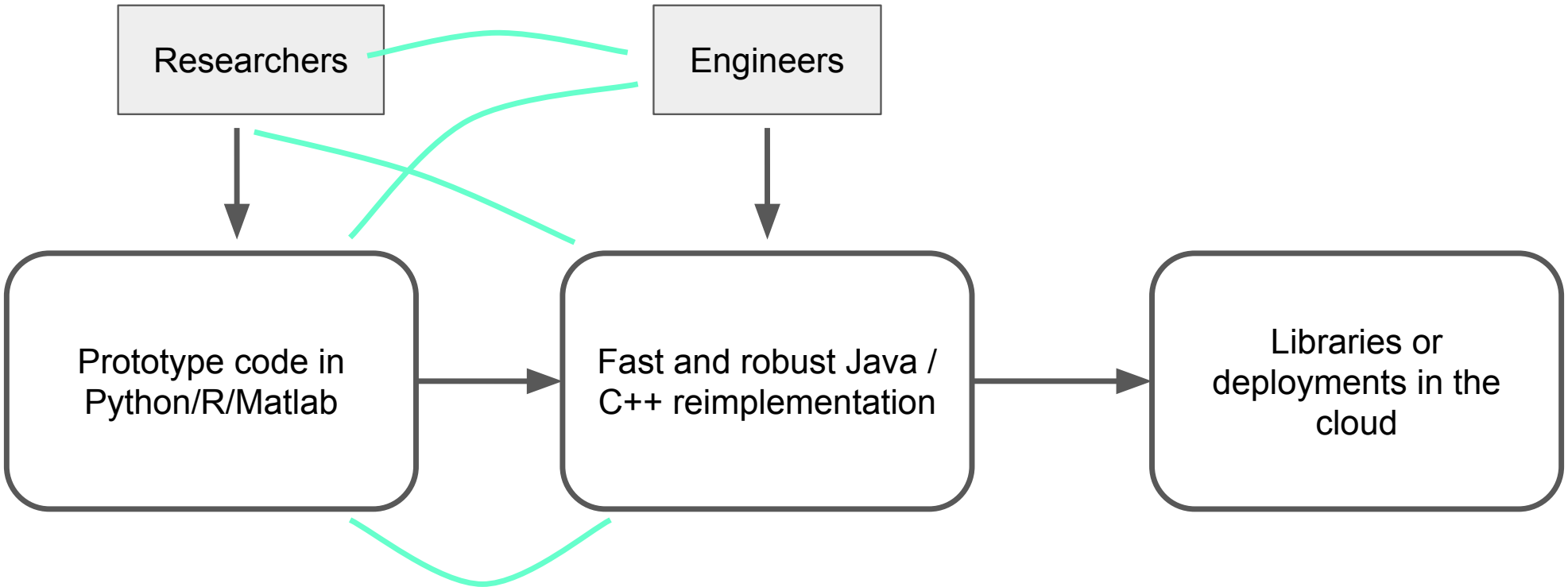
Why is two languages a problem ?

- Can't easily **modify** core algorithms
- Barrier to **understanding algorithms** even if not seeking to modify them
- Complicates installation of a complete software stack (package **dependency hell**, manual interventions)
- Barrier to **transparency** and **reproducibility**
- **Hampers innovation**, development of novel algorithms
- Barrier to **composing libraries** (e.g., automatic differentiation)

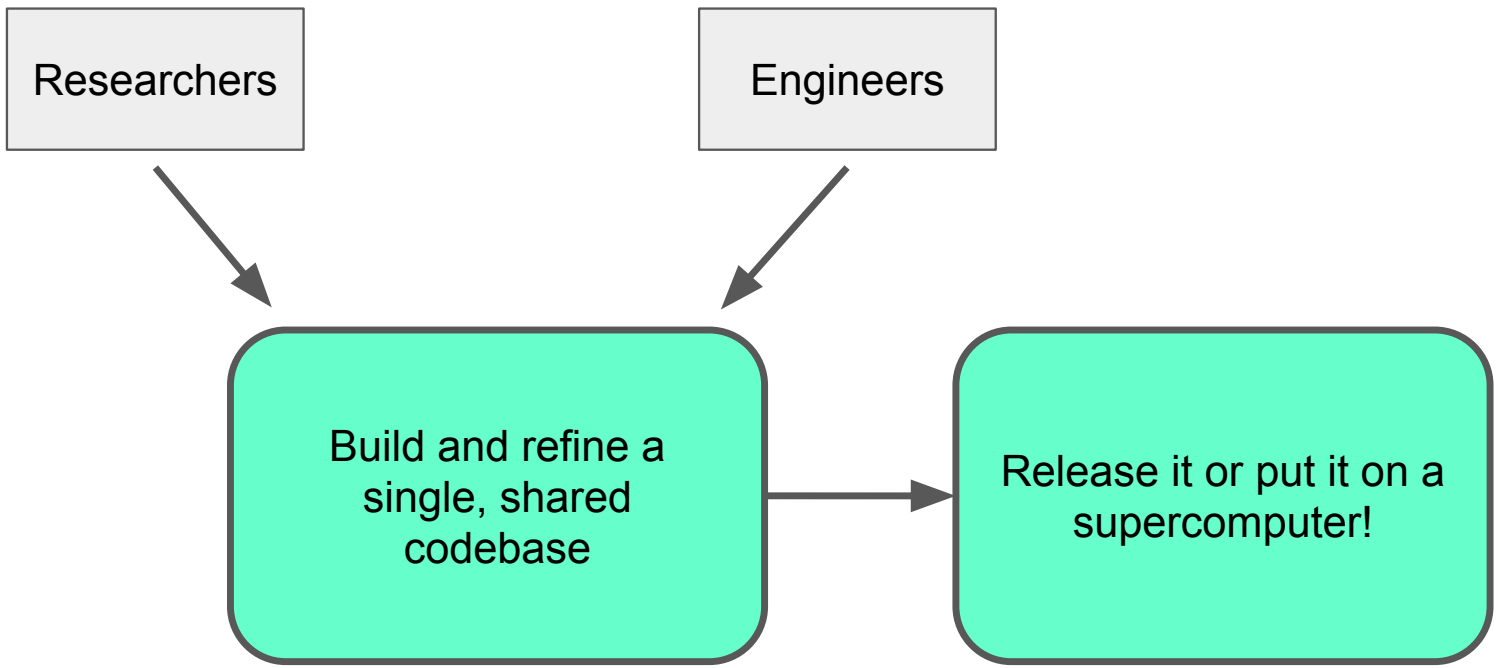
Data Science Pipeline



Data Science Pipeline: In Practice



Data Science Pipeline: A Fresh Approach



Fast code written fast: DifferentialEquations.jl

Comparison Of Differential Equation Solver Software

Subject/Item	MATLAB	SciPy	deSolve	DifferentialEquations.jl	Sundials	Hairer	ODEPACK/Netlib /NAG	JitCODE	PyDSTool	FATODE	GSL	BOOST	Mathematica	Maple
Language	MATLAB	Python	R	Julia	C++ and Fortran	Fortran	Fortran	Python	Python	Fortran	C	C++	Mathematica	Maple
Selection of Methods for ODEs	Fair	Poor	Fair	Excellent	Good	Fair	Good	Poor	Poor	Fair	Poor	Fair	Fair	Fair
Efficiency*	Poor	Poor****	Poor***	Excellent	Excellent	Good	Good	Good	Good	Good	Fair	Fair	Fair	Good
Tweakability	Fair	Poor	Good	Excellent	Excellent	Good	Good	Fair	Fair	Fair	Fair	Fair	Good	Fair
Event Handling	Good	Good	Fair	Excellent	Good**	None	Good**	None	Fair	None	None	None	Good	Good
Symbolic Calculation of Jacobians and Autodifferentiation	None	None	None	Excellent	None	None	None	None	None	None	None	None	Excellent	Excellent
Complex Numbers	Excellent	Good	Fair	Good	None	None	None	None	None	None	None	Good	Excellent	Excellent
Arbitrary Precision Numbers	None	None	None	Excellent	None	None	None	None	None	None	None	Excellent	Excellent	Excellent
Control Over Linear/Nonlinear Solvers	None	Poor	None	Excellent	Excellent	Good	Depends on the solver	None	None	None	None	None	Fair	None
Built-in Parallelism	None	None	None	Excellent	Excellent	None	None	None	None	None	None	Fair	None	None
Differential-Algebraic Equation (DAE) Solvers	Good	None	Good	Excellent	Good	Excellent	Good	None	Fair	Good	None	None	Good	Good
Implicitly-Defined DAE Solvers	Good	None	Excellent	Fair	Excellent	None	Excellent	None	None	None	None	None	Good	None
Constant-Lag Delay Differential Equation (DDE) Solvers	Fair	None	Poor	Excellent	None	Good	Fair (via DDVERK)	Fair	None	None	None	None	Good	Excellent
State-Dependent DDE Solvers	Poor	None	Poor	Excellent	None	Excellent	Good	None	None	None	None	None	None	Excellent
Stochastic Differential Equation (SDE) Solvers	Poor	None	None	Excellent	None	None	None	Good	None	None	None	None	Fair	Poor
Specialized Methods for 2nd Order ODEs and Hamiltonians (and Symplectic Integrators)	None	None	None	Excellent	None	Good	None	None	None	None	None	Fair	Good	None
Boundary Value Problem (BVP) Solvers	Good	Fair	None	Good	None	None	Good	None	None	None	None	None	Good	Fair
GPU Compatibility	None	None	None	Excellent	Good	None	None	None	None	None	None	Good	None	None
Analysis Addons (Sensitivity Analysis, Parameter Estimation, etc.)	None	None	None	Excellent	Excellent	None	Good (for some methods like DASPK)	None	Poor	Good	None	None	Excellent	None

Other features

- Fast user-defined composite types (C-like structs)
- Lisp-like macros/metaprogramming
- Call C functions directly
- Can wrap Python, R Java/Scala
- State-of-the-art distributed computing and multi-threading support

Other features (continued)

- First-class math support
- Exellent REPL (console)
- Built-in package manager
- Automatic differentiation (Zygote.jl)

Drawbacks

For most users all serious drawbacks derive from Julia's young age:

- Smaller number of libraries
- Smaller community of users
- Some interfaces and tooling less polished
- More bugs (in libraries, few in language itself)

Other drawbacks and annoyances:

- Limited support for exporting programs as stand-alone executables.
- The "time to first plot" problem.

machine learning $\neq$ deep learning

machine learning $\neq$ deep learning
(neural networks)

A plethora of machine learning models

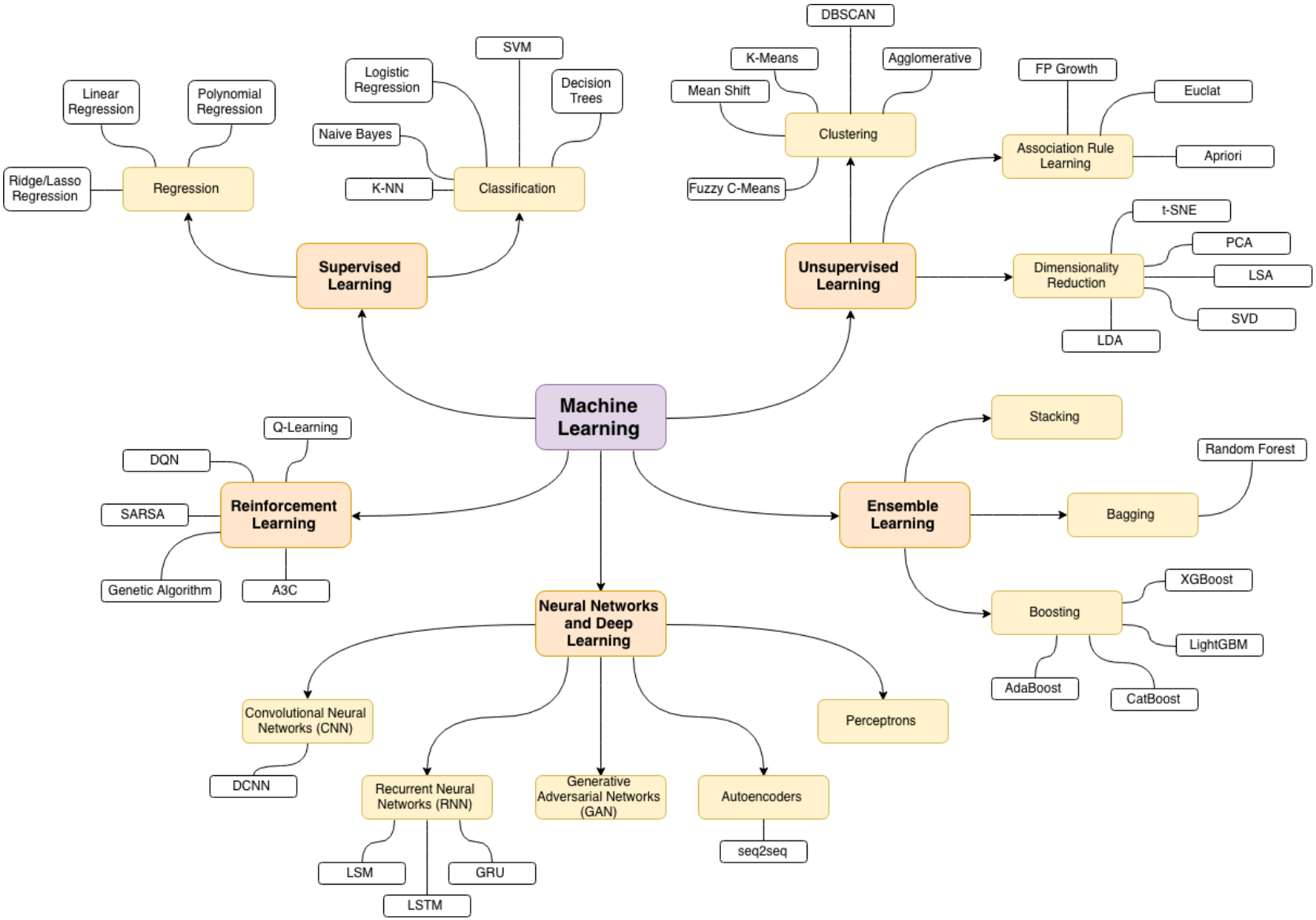


Figure source: Oleksii Trekhleb (`https://github.com/trekhleb/homemade-machine-learning`)

Supervised Learning

Learning to **predict** some target variable **y** from a knowledge of some other variables **X** (the *input features*).

Supervised Learning

Learning to **predict** some target variable **y** from a knowledge of some other variables **X** (the *input features*).

X	**y**
longitude, latitude, temperature, pressure	wind speed
individual DNA gene sequence	got diabetes
word frequencies in email	is junk?
an image of handwritten digit	$0, 1, 2, \ldots, 8$ or 9
num rooms, floor area, zip code	selling price
Titanic: sex, class, fare, where embarked	survived?

Supervised Learning

Learning to **predict** some target variable **y** from a knowledge of some other variables **X** (the *input features*).

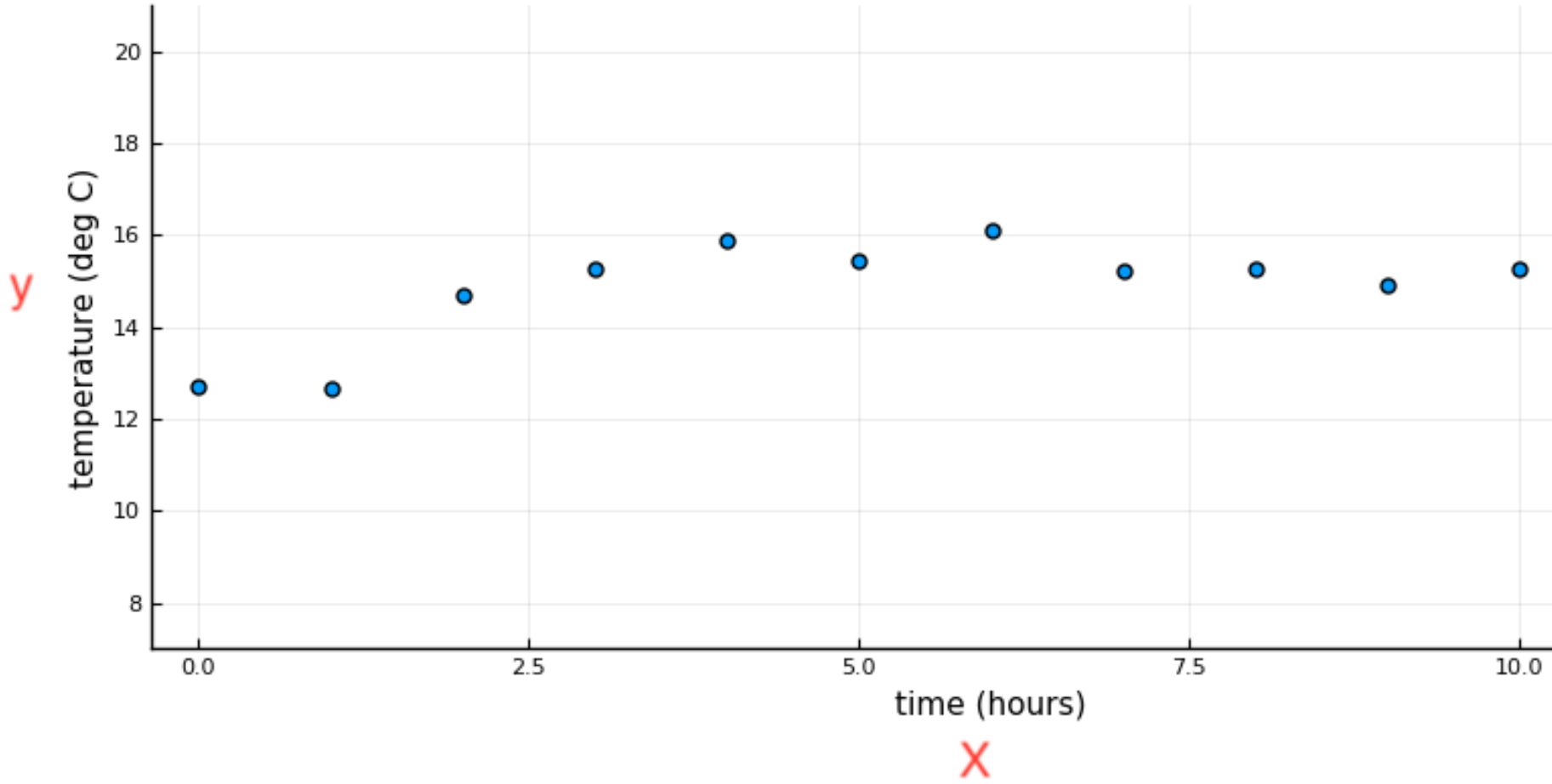
X

time Int64	room CategoricalValue{String,UInt32}
5	kitchen
5	bathroom
5	bedroom_1
5	living_room
6	kitchen
6	bathroom
6	bedroom_1
6	living_room
7	kitchen
7	bathroom
7	bedroom_1
7	living_room

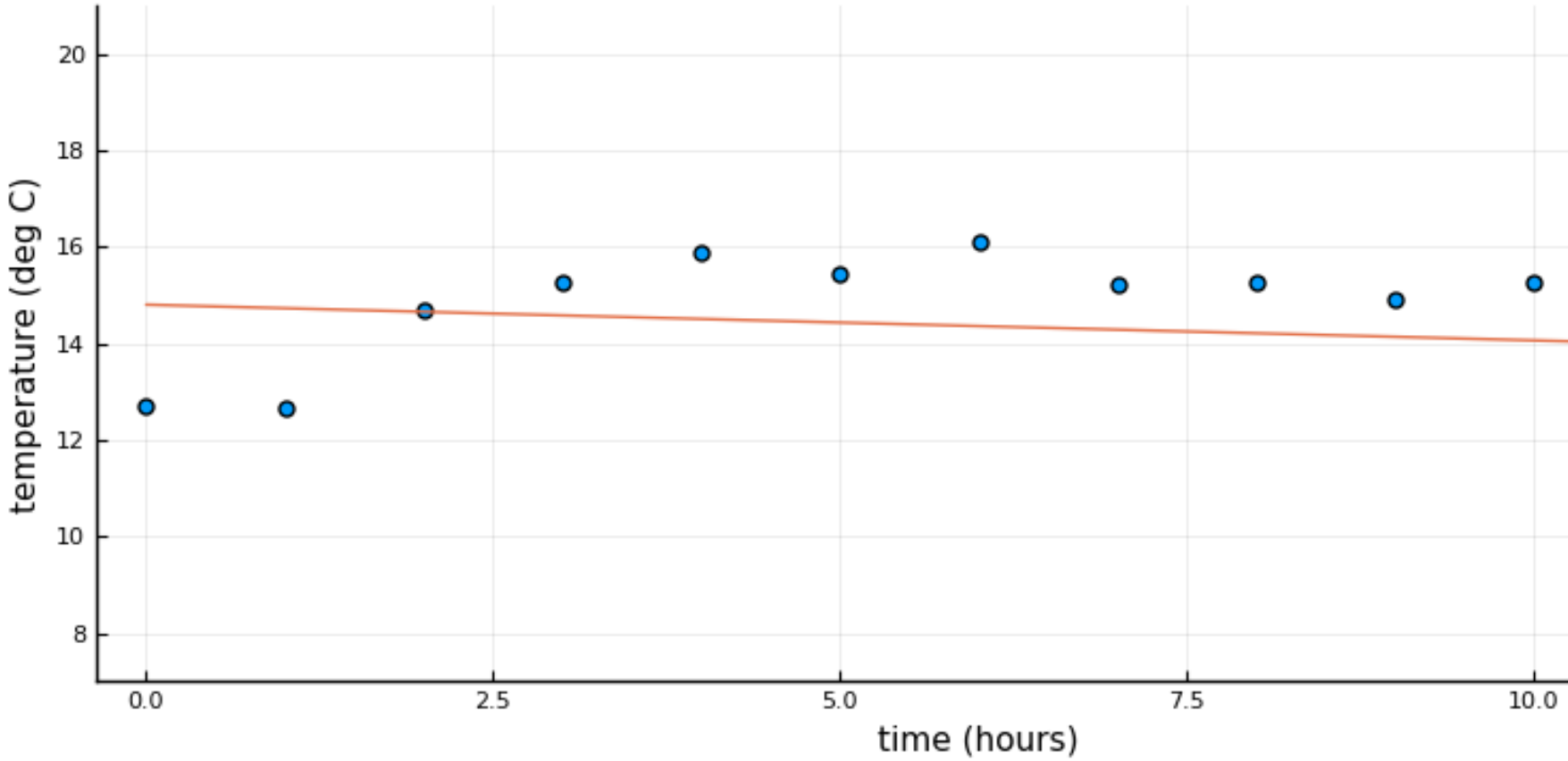
y

temperature Float64
18.5
18.3
18.3
17.4
16.6
20.7
18.9
20.2
20.4
19.9
16.2
17.3

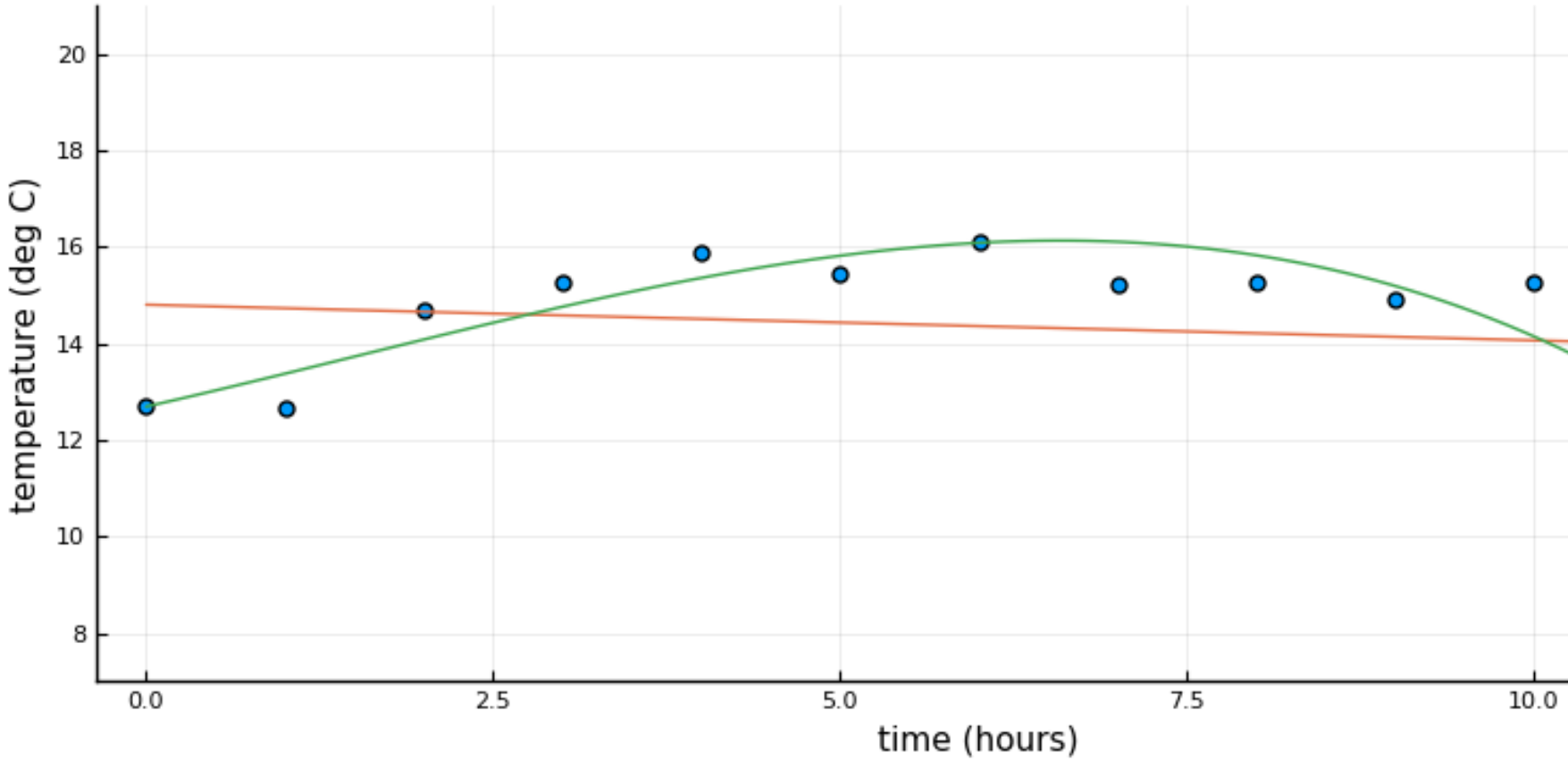
Supervised Learning



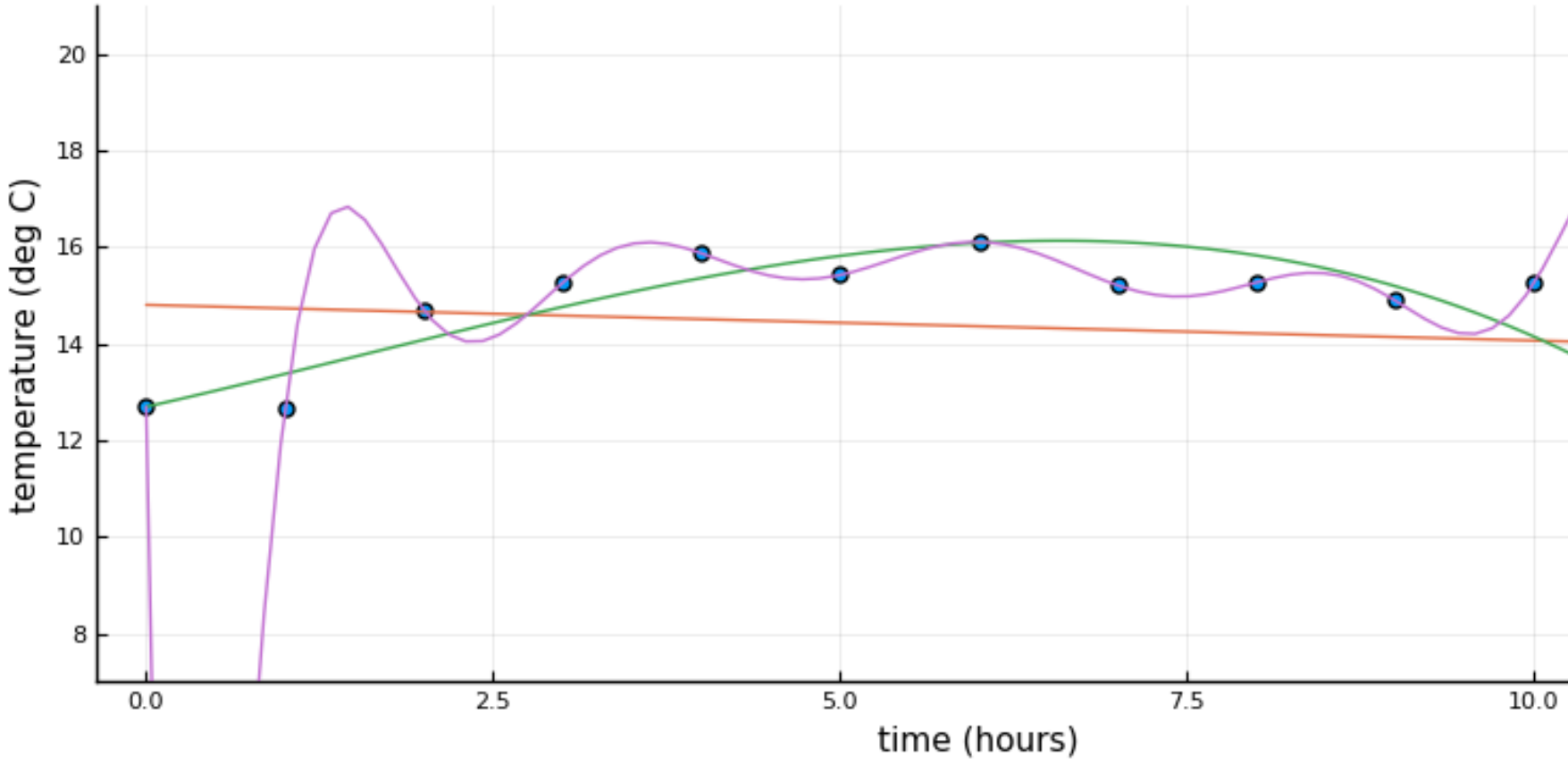
Supervised Learning



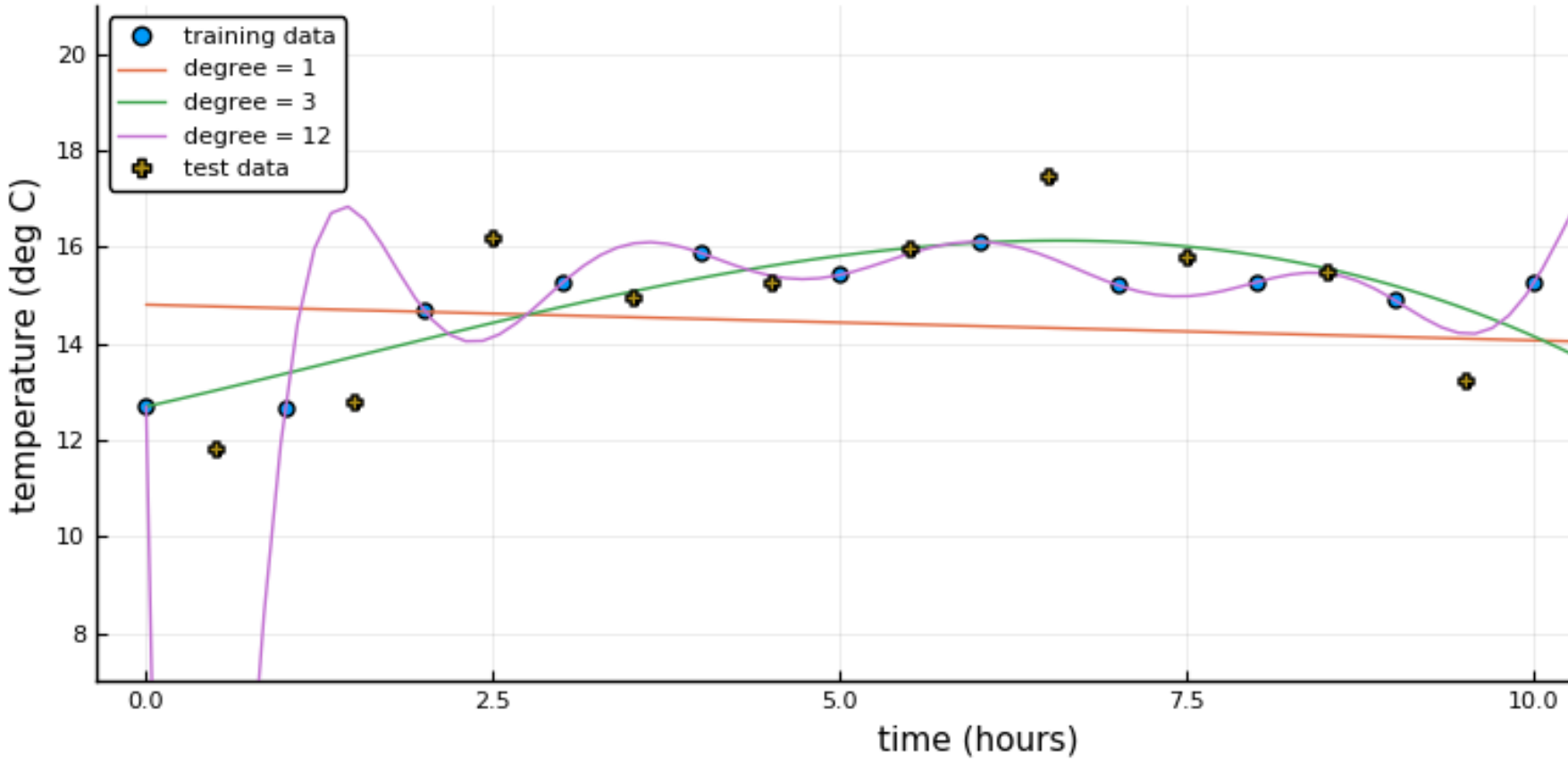
Supervised Learning



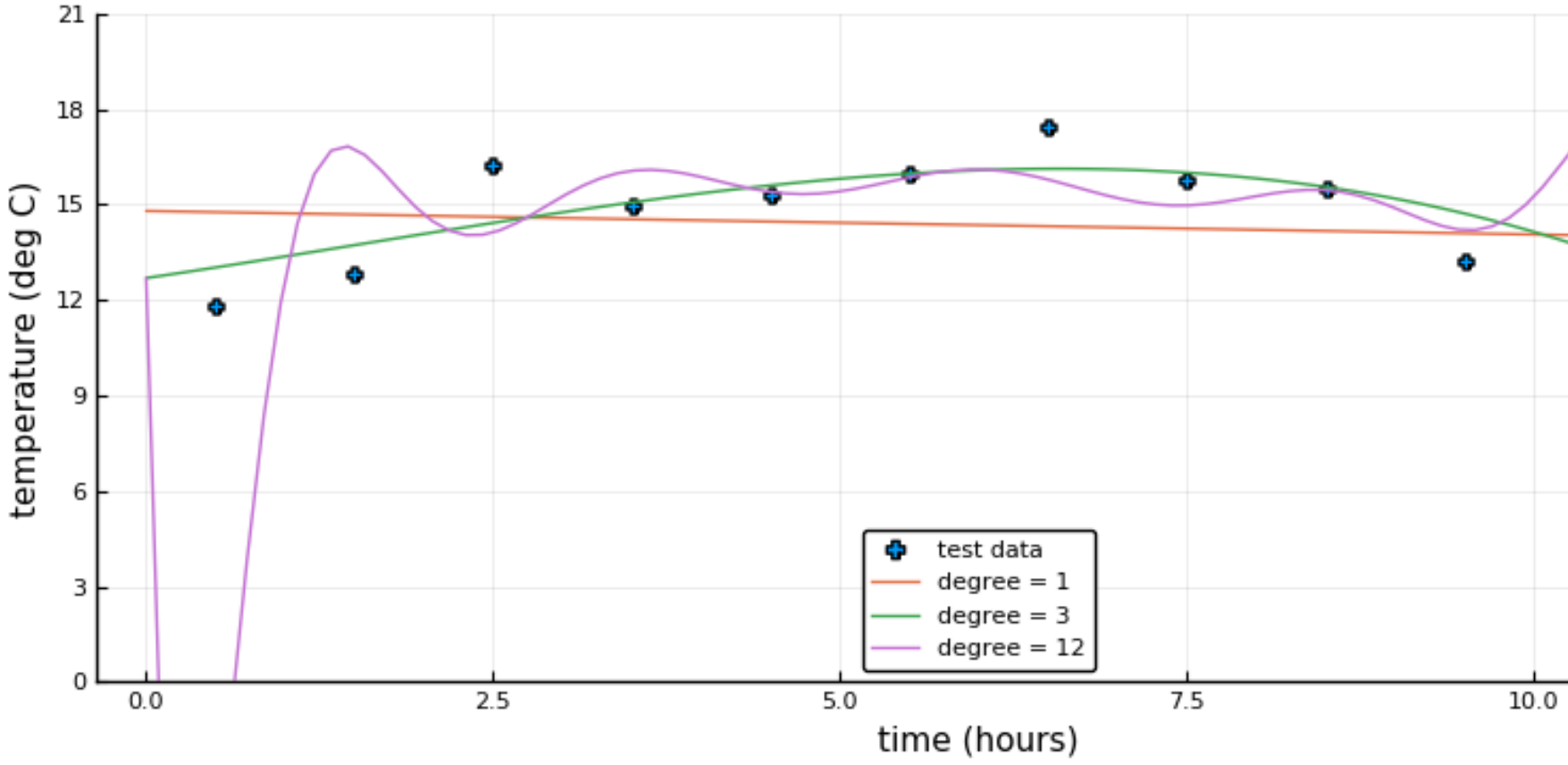
Supervised Learning



Supervised Learning



Supervised Learning



Supervised Learning

Setup

Got historical data with:

- **features** X (aka, inputs, patterns)
- **target** y (aka, labels)

Supervised Learning

Setup

Got historical data with:

- **features** X (aka, inputs, patterns)
- **target** y (aka, labels)

Basic machine learning workflow

1. Split historical data into **training** and **test** sets.

Supervised Learning

Setup

Got historical data with:

- **features** X (aka, inputs, patterns)
- **target** y (aka, labels)

Basic machine learning workflow

1. Split historical data into **training** and **test** sets.
2. Train the model using the **training** set

Supervised Learning

Setup

Got historical data with:

- **features** X (aka, inputs, patterns)
- **target** y (aka, labels)

Basic machine learning workflow

1. Split historical data into **training** and **test** sets.
2. Train the model using the **training** set
3. Get model predictions $\hat{y}$ given X for **test** set.

Supervised Learning

Setup

Got historical data with:

- **features** X (aka, inputs, patterns)
- **target** y (aka, labels)

Basic machine learning workflow

1. Split historical data into **training** and **test** sets.
2. Train the model using the **training** set
3. Get model predictions $\hat{y}$ given X for **test** set.
4. Compare $\hat{y}$ with y from the **test** set using some **measure** (e.g., mean squared error).

Supervised Learning

Setup

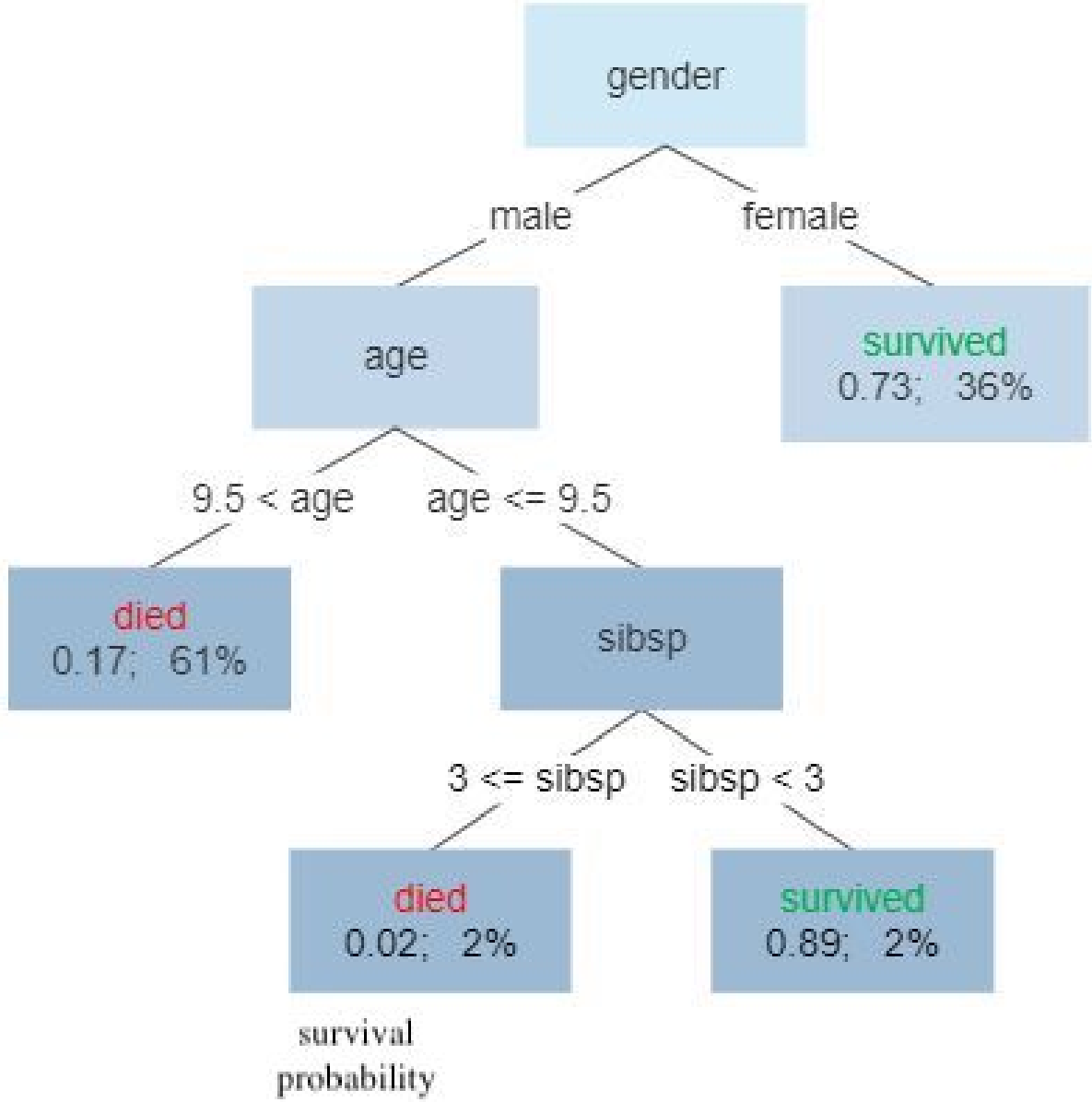
Got historical data with:

- **features** X (aka, inputs, patterns)
- **target** y (aka, labels)

Basic machine learning workflow

1. Split historical data into **training** and **test** sets.
2. Train the model using the **training** set
3. Get model predictions $\hat{y}$ given X for **test** set.
4. Compare $\hat{y}$ with y from the **test** set using some **measure** (e.g., mean squared error).
5. Change hyperparameters of model and repeat.

Survival of passengers on the Titanic



Tutorials

`github.com/ablaom/HelloJulia.jl`
`/tree/dev/notebooks/`