

Predictive Analytics - Regression and Classification
Prof. Sourish Das
Department of Mathematics
Chennai Mathematical Institute

Lecture - 62
Why Julia is Future for Data Science Projects?

Welcome to the 22nd lecture of these Predictive Analytics Regression and Classification course. Today, I am going to talk about why we need we want to use Julia along with R or Python. So, some of you have asked me question that R and Python, these are the two go to language when we have to do data science? And I agree that even a year back, I was mostly a R person, but at the same time, I also delivered commercial project using Python. Of course, I delivered many, many commercial project using R.

But recently about a year back, one of my senior colleague, Mr. Raja Shah, he kind of encouraged me to explore this new language called Julia. So, I decided, let me explore and I was quite impressed with this new language called Julia. It is mainly designed for doing scientific computation.

So, in this video, I will try to argue that why eventually the data science community should adopt Julia as go to language over R or Python. I am not saying that you completely switched to Julia overnight. Yeah, if you have a project going on with R and Python, please do so. But at the same time, learn Julia gradually moved to Julia. And I will argue why Julia is giving us something which is extremely important to consider for any data scientist.

(Refer Slide Time: 02:15)

Data Science with **julia**

NPTEL Course on Predictive Analytics



(Refer Slide Time: 02:18)

Mathematical Equation to R or Python Code : Lost in Translation

$\beta \sim \text{MvNormal}(\beta_0, \Sigma)$ $\sigma \sim \text{Inverse-Gamma}(a_0, b_0)$ $y \sim \text{MvNormal}(X\beta, \sigma)$	<pre>1 ## Simulate from Posterior distribution 2 set.seed(1) 3 library(mvtnorm) 4 N.sim = 30000 5 burnin = 0 6 sigma = rep(NA, N.sim) 7 beta.draws = matrix(NA 8 , nrow=N.sim 9 , ncol=k) 10 colnames(beta.draws) = nms 11 12 for(i in 1:(N.sim+burnin)){ 13 if(i %% 5000 == 0) cat('i=', i, '\n') 14 sigma_2_star = 1/rgamma(i, a_pi, b_pi) 15 S = sigma_2_star*Lambda_pi_inv 16 beta_star = rmvnorm(1, mean = beta_pi 17 , sigma = S) 18 if(i > burnin){ 19 sigma[i-burnin] = sqrt(sigma_2_star) 20 beta.draws[i-burnin,] = beta_star 21 } 22 } 23 theta = cbind(beta.draws, sigma) 24 head(theta)</pre>
---	--



So, typical, if you see any mathematical equation that we want to write and we want to code, there is a big gap in our understanding. So, if you see in this panel, in the left panel, this is essentially the three basic equations for Bayesian regression with a simple normal or Gaussian prior, where I am assuming that this is my likelihood model. These are the prior model on the sigma and this is the Gaussian prior on the coefficient beta.

So, this is a simple regression model. And now, if I have to implement this model, say in R, this will be the code. It is the same R code. It is the code which is run written by (Refer Time: 03:15) me. There are some functions also there where you can call these functions and do that. But what point I am trying to make here is the way we write our mathematical equation and way we code them, you know, programming language R and Python, many things get lost in translation.

(Refer Slide Time: 03:40)

Mathematical Equation to R or Python Code : **Lost in Translation**

- **Lost in translation**
- Reduces transparency within organisation / division / group
- Often fellow data scientists do not understand what are we doing!
- **Lack of Explainability** of AI - ML and Data Science



And as things get lost in translation, it reduces transparency within organization, within division, even within same peer group. Often, a fellow data scientist do not understand what are we doing? If you write a program you know in R or in Python and you share it with your friends, I can guarantee that your friends will stumble upon many lines that she or he will have trouble to understanding the code.

Similarly, if you receive a code from your friend, you will stumble upon and you will be not able to understand many lines of code what she or he did. So, that that is a that that is and it takes lot of time to understand exactly what is going on. And that contributes to something called lack of explainability in AI-ML and data science. So, I am I will try to make a limited point here. So, what is lack of explainability in AI-ML and data science?

So, nowadays, many of the deep learning neural network models, they are so complex and complicated. Many things are not very easily explainable to common people and or even scientists who are not trained as data scientist or computer scientist or statistician. For them also, these deep learning neural network model is like almost like a black box.

So, these are the these are the major criticism of AI-ML models in and they call it lack of explainability. The limited point I am trying to make here is the way we write our code that also contribute to lack of ML explainability in ML machine learning and data science projects.

(Refer Slide Time: 05:51)


cmj CHENNAI MATHEMATICAL INSTITUTE

NPTEL

Mathematical Equation to julia

$\beta \sim \text{MvNormal}(\beta_0, \Sigma)$	In []: <pre>#priors β ~ MvNormal(β0, Σ) σ ~ InverseGamma(0.1, 0.1) #likelihood y ~ MvNormal(x * β, σ)</pre>
$\sigma \sim \text{InverseGamma}(a_0, b_0)$	
$y \sim \text{MvNormal}(X\beta, \sigma)$	

Julia is a high level language **julia**



So, here comes Julia that gives us a interesting part. Now, here you can see this is a small code snippet from my Jupyter Notebook. The way we write code, our equation in Julia, we

can write our code exactly the same way. So, this is a very very important and interesting way we can write Julia code.

Julia is a very high level language. So, that makes the Julia very attractive, the expressivity, the way you express the way I write my mathematical equation on the board on my question or on my you know notebook.

I can write my program almost like same almost like same and that program runs and give me the answer and its correct answer. That is a beautiful, you know, I think it is a big leap forward for the way we do our data science project. So, that is the main reason I am very excited about Julia in data science. So, this expressivity makes Julia a high level language.

(Refer Slide Time: 07:10)

Performance Tradeoff

- C, Fortran, Java are high performance but low level language
- R, Python, Matlab, SAS, Stata are high level language but sacrifice performance
- These high level languages suffer from what is called **two-language** problem
- **julia** does not suffer from two-language problem. Because the compiler is written in **julia** itself.



Now, there is a big performance tradeoff. So, what is this performance tradeoff? We know that C, Fortran, Java, these are high performance, but low level language. So, you have to be really expert, very it is difficult to get an expert who is very good in C, Fortran or Java programming at the same time who is a very good data scientist. So, data scientist, most of the time data scientists go to languages R, Python, MATLAB, SAS, Stata, these are the high level language.

But what happen is they sacrifice performance. So, these high level languages suffer from what is called two-language problem. So, Julia does not suffer from two language problem because Julia's compiler is written in Julia itself. So, R, Python, MATLAB, their compiler are written in typically C in C or C plus plus mostly in C.

Now, what happened in these high level languages that if you really want to speed up your programming. So, that makes it. So, that makes these R, Python, MATLAB, what is called something called interpretable language. So, whatever you are writing it, eventually the compiler of these languages are in C or C plus plus. So, that makes it very slow, the mix the R, Python, MATLAB language is comparatively slow. MATLAB is much better designed programming language, but overall, they are not as fast as C or Fortran.

So, if you really want to write a program in MATLAB very fast, then what do you have to do? You have to write the core program in C or Fortran. And then you have to call that C or Fortran function in MATLAB or Python or R using C call or Fortran call or F call. So, but Julia does not suffer from two-language problem because Julia is not a interpretable language, Julia is a compiler level language.

(Refer Slide Time: 09:43)

Why **julia** is fast?

1. **Just-in-Time Compilation** : Compile code on the fly and optimise it based on data types
2. **Multiple Dispatch** : Determine the appropriate method to call function based on arg type
3. **Dynamic Typing** : allows flexibility and ease of use but it also allow compiler to optimise code
4. **Low level access to Hardware**: Such as SIMD instruction to speed up
5. **Built-in Parallelism**: Allow computation to split across the core
6. **Efficient Memory Management**: Uses garbage collector to manage memory, also allows explicit MM



So, why Julia is fast? So, essentially, Julia's compiler was written based on six major foundational structure, you know, major goal, the compiler tries to achieve six major goal one is just-in-time compilation. So, it compile the code on the fly and optimize it based on the data type. So, if you have a data type say float 64 or if you have a int 32 or if you have something else. So, it based on the data types, it runs the compilation code compilation and try to optimize its time in accordingly.

Another one of the beautiful feature of Julia is multiple dispatch. So, it determine appropriate method to call function based on argument type. So, what is how what happens is typically if you suppose if you have a argument, which has a different operation based on the argument type. So, if the argument is string, it will do some kind of operations and if the argument is float, then it will do certain other kind of operations.

So, what you will do? You will write a function saying if argument is this, you do this, if argument is that else, you do this. Now, in the Julia, you do not do that. In Julia, you write a function where you declare argument is string. And similarly, you write this another function with the same exact name, but you say that the argument is float. So, now you have the two function with the exact same name, but the argument type is different. And that makes a Julia you know maintaining the.

So, so, it might happen that suppose you have a argument A and it can have two kind of a possible input. One is say that you know it could be a screen, then it will do certain operation and it could be float. It will do another kind of operation. Now, you write a function where you say function name, say ABC function, ABC function, the argument is A string. It will do certain operation.

And then you write another function, exact same name ABC, but a is now float. So, both function will be defined and perfectly fine and Julia will just identify them based on their argument type, call that function and bang. But if then else, you do not have to do the if then else.

So, what is the advantage of this that you may have a string when in the when the A is string certain operations may not working or something, but still float is working. So, the your project will not get stalled. Only that part of the problem of the project will have a problem, but the rest of the project will be still up and running.

So, this is a beautiful feature I found and Python also have this feature, but Python you have to install a package called multiple dispatch and then Python will allow you to do multiple dispatch. It does not come as a base package. R I have not seen any package in R which allows multiple dispatch. So, it is a beautiful things and it actually makes your implementation very dynamic. It makes your implementation very you know easy going and all these things.

So, another third is dynamic typing. It allows flexibility and ease of use, but it also allows compiler to optimise the code. Low level access to hardware such as, you know, SIMD instruction to speed up built in parallelization allow computation to split across the core. So, if there is any free core available, Julia figures it out and it can go and try to use that core. Efficient memory management and uses the garbage collector to manage memory and allows explicit memory management.

(Refer Slide Time: 14:20)

Data Science Languages

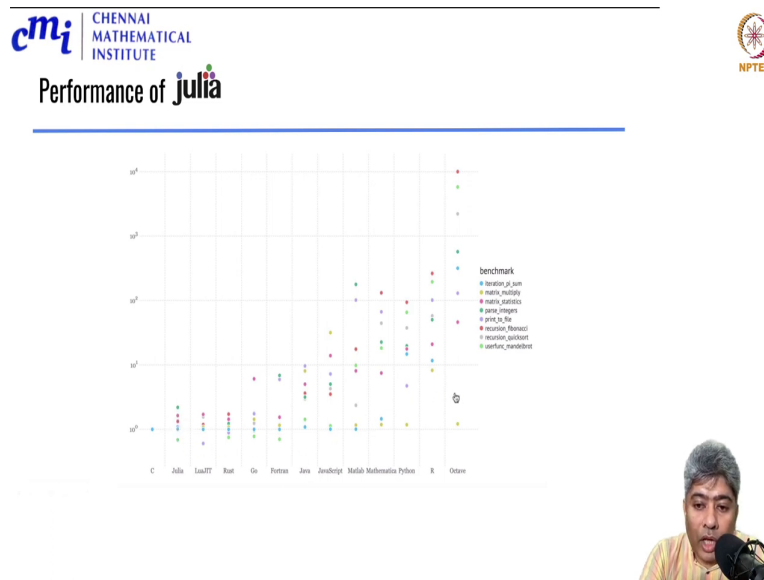
Language	Version 1.0 Release on	Compiler
SAS	1976	C
Matlab	1984	C/C++
Stata	1985	C
Python	1994	C/C++/multi-paradigm
R	2000	C/C++
Julia	2018	Julia



Now, if you see the evolution of the data science languages, the first major data science language was SAS that came in 1976. Its compiler was C, MATLAB was 1984 C C plus plus Stata was 85 C, Python, C C plus plus. Some other (Refer Time: 14:44) was also taken R was definitely C C plus plus and the last was 2000. The when I am talking about version 1.0

release the actual may have started a later little earlier. Julia's version 1.0 release was 2018 and the compiler was Julia itself.

(Refer Slide Time: 15:12)



So, that makes it Julia, you know, a very fast probably as fast as other languages. So, here is a comparison that I have taken directly from Julia's website. So, what they have done, they have taken some very nasty mathematical computation like iteration pi sum matrix multiplication, matrix statistics, parse integers, print to file it is a very complicated, like, you know, high complexity problem computation. Recursion Fibonacci, Fibonacci recursion is extremely difficult.

You need a dynamic programming to solve a recursion quick sort, use function, Mandelbrot brot. And if you do that where C is being kind of created as a base, so, every all programs are being kind of put in into 10 to the power 0, which is 1. And with respect to that, Julia, in fact,

Mandelbrot, Julia is even faster than C. Rest of them are somewhat incomparable. Lua, Jit and Rust are also similarly comparable. Go is a new language by Google. That is also very fast and comparable.

Fortran and in the same space, but if you see R and so, R all these things that with respect to R Julia were 10 times faster. And with respect to Python also on an average Julia was 10 times faster.

So, that is a beautiful feature that Julia is proposing that on an average, if we can make sure that Julia, the our data science projects implementing R or Python are maybe on an average 10 times faster, or even 5 times faster than R or Python, the our Julia project, 5 times faster than R or Python or 10 times faster than R or Python. Then I think it gives a huge advantage over R and Python so, R Python or even MATLAB.

(Refer Slide Time: 17:30)

Comparison between Python and Julia

1) Linear Algebra operations: `numpy`

Julia supports all of these built-in. Julia is superior.

2) Scientific Libraries: `scipy`

Julias SciML ecosystem is far superior to that of Scipy.

3) Data Science : `pandas`

Julia's DataFrames and TSFrames has superior data manipulation capabilities



So, I have compared here, I am presented a comparison between Python and Julia between different packages. For example, if you have the linear algebra operations, if you want to do linear algebra operations, you need to have a NumPy package. But Julia supports all the built-in and their performance, time performance, Julia is superior.

Scientific library, SciPy, Julia, Sciml, ecosystem is definitely far superior data science packages like Pandas, Julia's data frame and TS frame has superior data, manipulation capability, and TS frame is being developed by our group, by Chirag and Ajay and Ayush I will tell about them later.

(Refer Slide Time: 18:22)

Comparison between Python and Julia

4) Machine Learning: scikit-learn, pycaret, LightGBM

Julia has many basic ML capabilities - MLJ.jl, Lighthouse.jl, XGBoost.jl.

Python packages are more polished, but the Julia ones are easy to improve.

The momentum however is all in deep learning.



There is a machine learning package, bunch of like you know Python packages are, scikit-learn, PyCaret, Light, GBM, and comparable Julia packages are there, MLJ, Lighthouse, XGBoost.

(Refer Slide Time: 18:40)

Comparison between Python and Julia

5) Deep Learning: PyTorch, Tensorflow, Keras, Theano, Caffe

Julia packages are Flux.jl, Transformers.jl, Lux.jl

+ Only PyTorch is the breakout tool here.

+ TensorFlow is only if you are in the Google ecosystem

+ The others are effectively old and outdated.

+ Julia does not have the same level of ecosystem as PyTorch, but can integrate all the kernels from PyTorch. The Silicon Valley ML community has adopted Python - but these libraries are not written in Python. They are C++ under the hood and can be called from any language.



If the Python PyTorch, TensorFlow, Keras, Theano, they are still much much better when it comes to deep learning packages. And Julia is has Flux, Transformer, and Lux, and sort of a they are trying to catch up. I would say if you have a, currently, have a project which is a deep learning project, I will recommend still you stick with your Python thing because Julia may be he is trying to catch up, which is, but I am pretty confident in next 3 to 5 years time, Julia packages will be, if not better, then at least at part with Python.

(Refer Slide Time: 19:25)

Comparison between Python and Julia

6) Operation Research: Pyomo

JuMP - Julia is superior.

7) Dashboarding and UIs: Dash

Dash.jl, Genie.jl. Both are equivalent

8) 3d Graphics:

Makie. No equivalent in Python

9) GPU compiler: PyCuda

+ CUDA.jl, OneAPI.jl, AMDGPU.jl, Metal.jl.

+ Julia provides same API for NVIDIA, Intel, AMD and Apple GPUs.

+ Julia is superior due to its native compiler.

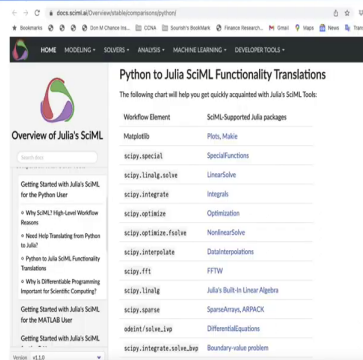


If you want to implement operation research projects like, you know, optimization things, that Pyomo is the kind of Python thing, Python package, but you jump is the best, I think, Julia jump is far superior than Pyomo. If you want dashboard UI, like Dash and Genie, both are equivalent, I would say, Dash is the Python, like, you know, UIs and Dashboard. 3D graphics, Makie is beautiful compared almost like; no Python is as good as Makie.

GPU compiler, if you do PyCUDA, like, you know, CUDA, OneAPI. So, all these Julia providers, like API, NVIDIA, Nvidia, Intel, AMD, Apple GPUs, all this Julia provides the same API. But Julia is superior due to its native compiler, even in the GPU compiler, Julia compilers does better than PyCUDA.

(Refer Slide Time: 20:40)

Python to Julia's SciML



Workflow Element	SciML-Supported Julia packages
Maths	Plots, Mpl
scipy.optimize	SpecialFunctions
scipy.linalg.solve	LinearSolve
scipy.integrate	Integrals
scipy.optimize	Optimization
scipy.optimize.fmin	NonlinearSolve
scipy.interpolate	DataInterpolations
scipy.fft	FFTW
scipy.linalg	Julia's Built-In Linear Algebra
scipy.sparse	SparseArrays, ARPACK
odeint / solve_ivp	DifferentialEquations
scipy.integrate.solve_ivp	Boundary value problem



So, in Python's Julia, SciML, what Python has done, that for all the Python, like, you know, functions, like, SciPy dot whatever you want, you have a corresponding Julia things that are named in there.

(Refer Slide Time: 20:58)

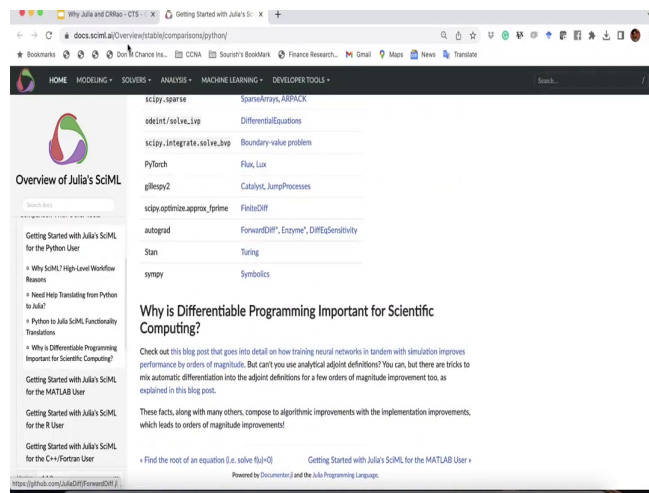
The screenshot shows a web browser displaying the SciML website. The page title is "Getting Started with Julia's SciML for the Python User". The main content area features a heading "Getting Started with Julia's SciML for the Python User" and a sub-heading "Why SciML? High-Level Workflow Reasons". The text explains that SciML is an open-source software for scientific machine learning with Julia, designed for Python users. It highlights several key reasons for its popularity:

- Performance** - The key reason people are moving from SciPy to Julia's SciML is performance. Even simple ODE solvers are much faster, demonstrating orders of magnitude performance improvements for differential equations, nonlinear solving, optimization, and more. And the performance advantages continue to grow as more complex algorithms are required.
- Package Management and Versioning** - Julia's package manager takes care of dependency management, testing, and continuous delivery in order to make the installation and maintenance process smoother. For package users, this means it's easier to get packages with complex functionality in your hands.
- Composable Library Components** - In Python environments, every package feels like a silo. Functions made for one file exchange library cannot easily compose with another. SciML's generic coding with JIT compilation these connections create new optimized code on the fly and allow for a more expansive feature set than can ever be documented. Take new high-precision number types from a package and stick them into a nonlinear solver. Take a package for Intel GPU arrays and stick it into the differential equation solver to use specialized hardware acceleration.
- Easier High-Performance and Parallel Computing** - With Julia's ecosystem, CUDA will automatically install the required binaries and `cu.jl` is then all that's required to GPU-accelerate large-scale linear algebra. MPI is easy to install and use. Distributed computing through password-less SSH. `MultiProc` is automatic and baked into many libraries, with a specialized algorithm to ensure hierarchical

The left sidebar contains navigation links such as "HOME", "MODELING", "SOLVERS", "ANALYSIS", "MACHINE LEARNING", and "DEVELOPER TOOLS". The right sidebar features the NPTEL logo.



(Refer Slide Time: 21:03)



And if you click on that, it will take directly take you there in that function. So, you can just have those functions there.

(Refer Slide Time: 21:07)

Press **esc** to exit full screen

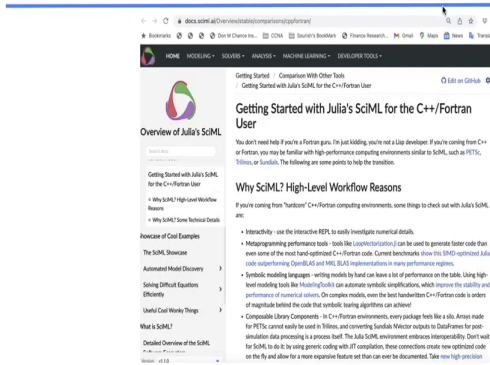
R to Julia's SciML

The screenshot shows a web browser displaying the SciML website. The main heading is "Getting Started with Julia's SciML for the R User". Below the heading, there is a sub-heading "Overview of Julia's SciML" and a list of links: "Getting Started with Julia's SciML for the R User", "How SciML's High-Level Workflow Reasons", "Meet the Translating Team to Julia!", "R to Julia SciML: Functionally Translating", "Meet the Power of Julia!", "Getting Started with Julia's SciML for the C++ Fortran User", "Demos of Cool Examples", "The SciML Showcase", "Automated Model Discovery", and "SciML: The AI Revolution". The main content area is titled "Why SciML? High-Level Workflow Reasons" and contains several bullet points:

- Performance** - The key reason people are moving from R to Julia's SciML, in short is performance. Even simple ODE solvers are much faster, demonstrating orders of magnitude performance improvements for differential equations, nonlinear solving, optimization, and more. And the performance advantages continue to grow as more complex algorithms are required.
- Composable Library Components** - In R environments, every package feels like a silo. Functions made for one file exchange library cannot easily compare with another. SciML's generic coding with JF compilation flow connections create new updated code on the fly and allow for a more expansive future set that can never be documented. Take new high precision number types from a package and stick them into a nonlinear solver. Take a package for fast GPU arrays and stick it into the differential equation solver to use specialized hardware acceleration.
- A Global Hierarchical Documentation for Scientific Computing** - R's documentation for scientific computing is scattered in a bunch of individual packages where the developers do not take to each other. This not only leads to documentation differences, but also 'style' differences: one package uses `is()`, while the other uses `is!()`. With Julia SciML, the whole ecosystem is considered together, and inconsistencies are handled at the global level. The goal is to be working in one environment with one language.
- Easier High-Performance and Parallel Computing** - With Julia's ecosystem, CUDA will automatically install if the required hardware and `libcudart.so` is there. All that's required is GPU accessible large-scale linear algebra. MPI is easy to install and use. Distributed computing through `distributed` has SGL. Multithreading is automatic and `Threads` makes `Threads` a `Task` instead of `Thread` to be more hierarchical across `Task` and `Task`.

(Refer Slide Time: 21:08)

C++ or Fortran to Julia's SciML



The screenshot shows the SciML website with the following content:

- Getting Started - Comparison With Other Tools**
 - Getting Started with Julia's SciML for the C++/Fortran User
- Overview of Julia's SciML**

You don't need help if you're a Fortran guru. For just kidding, you're not a Lisp developer. If you're coming from C++ or Fortran, you may be familiar with high-performance computing environments similar to SciML, such as PETSc, Trilinos, or Sundials. The following are some points to help the transition.
- Why SciML? High-Level Workflow Reasons**

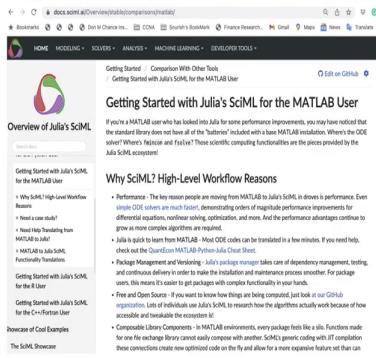
If you're coming from "hardcore" C++/Fortran computing environments, some things to check out with Julia's SciML are:

 - Interactivity** - use the interactive REPL to easily investigate numerical details.
 - High-performance performance tools** - tools like LoopVectorization can be used to generate faster code than even some of the most hand-optimized C++/Fortran code. Current benchmarks show this SPEC-optimized Julia code outperforming OpenBLAS and MKL BLAS implementations in many performance regimes.
 - Synthetic modeling languages** - writing models to hand can have a lot of performance on the table. Using high-level modeling tools like ModelingToolkit can automate symbolic simplifications, which improve the stability and performance of numerical solvers. On complex models, even the best hand-written C++/Fortran code is orders of magnitude behind the code that symbolic learning algorithms can deliver.
 - Composable Library Components** - In C++/Fortran environments, every package feels like a silo. Access made for PETSc cannot easily be used in Trilinos, and converting Sundials Vector outputs to DataFrames for post-simulation data processing is a pain in the butt. The Julia SciML environment embraces interoperability. Don't wait for SciML to do it by using generic coding with JIT compilation, these connections create new optimized code on the fly and allow for a more expansive feature set than can be documented. Take new high precision



(Refer Slide Time: 21:11)

Matlab to Julia's SciML



The screenshot shows a web browser displaying the SciML website. The page title is "Getting Started with Julia's SciML for the MATLAB User". The main content area is titled "Overview of Julia's SciML" and "Getting Started with Julia's SciML for the MATLAB User". It includes a list of "Why SciML? High-Level Workflow Reasons" such as Performance, Julia is quick to learn from MATLAB, Package Management and Versioning, and Free and Open Source. The page also features a sidebar with navigation links and a search bar.



(Refer Slide Time: 21:12)



cmi CHENNAI MATHEMATICAL INSTITUTE

Notable Uses of julia

Company


- Amazon
- Apple
- AstraZeneca
- Capital One
- Google
- IBM
- Intel
- JPMorgan
- Microsoft
- Moderna
- Pfizer

Organisations

- NASA
- US Fed Reserve
- *Instituto Nacional de Pesquisas Espaciais*, **INPE** (Brazilian Space Agency)

Universities/Institutes

- MIT
- Stanford
- UC Berkeley



So, similarly, C plus plus, Fortran, MATLAB, everything. And now, some of the notable uses of Julia, like, you know, Amazon, Apple, AstraZeneca, Capital One, IBM, Intel, JPMorgan, Microsoft, Moderna, Pfizer, these all companies are now gradually moving their data science project to Julia. It is not like they are doing overtime. Of course, that nobody will do that. I mean, they will run their mission critical projects still in R and Python. They do not want to disturb that.

But any project that is starting from the scratch, they are definitely considering Julia as a serious options. Among the other major organization, NASA, US Fed Reserve, Brazilian Space Agency, for sure, they are using Julia. Among the major universities, MIT, Stanford, UC Berkeley, they are all offering their linear algebra and other different courses in Julia.

(Refer Slide Time: 22:15)

- 1M + users
- Used by biggest companies in US, such as Moderna, JPMorgan, AstraZeneca, Pfizer,
- Moderna used Julia for vaccine FDA approval
- Taught at MIT, Berkeley, Stanford and other place
- Replacement for R, SAS, Matlab and many other aging platform
- JuliaHub is working with infosys already in the airline industry - confirmed by Dr. Viral Shah
(co-creator of Julia and CEO of JuliaHub)



So, so far, worldwide 1 million plus users used by biggest companies in US, such as Moderna, JPMorgan, AstraZeneca, Pfizer. Moderna used Julia for vaccine FDA approval and taught at MIT. It has been taught an MIT so many companies are trying to replace the R, SAS, MATLAB, many other aging platforms.

And Julia Hub is working with Infosys nowadays for an airline industry routing optimization problems and other problems. It is it was confirmed by Dr. Viral Shah, who is the co-creator of Julia and CEO of Julia Hub.

(Refer Slide Time: 23:01)

$$\text{var}(\hat{\theta}) \geq \frac{1}{I(\theta)} = \frac{1}{-E\left[\frac{\partial^2}{\partial \theta^2} \log f(X, \theta)\right]}$$

$$E[(\hat{\theta}(X) - \theta)^2] \leq E\left[\frac{1}{I(\theta)}\right]$$

CRRao

Single API for diverse Data Science
Models



Source: Nobel Prize Series



(Refer Slide Time: 23:12)

CRRao : A Consistent API for Data Science in Julia

- Dr FC Kohli Center at CMI
- + we are building a CRRao in Julia
- Funded by MIT
- Version 0.1.0 was released on 30th
December 2022



So, now, I will speak about the CRRao package that we are in CMI. We are developing this single API for diverse statistical models at the Dr. F. C. Kohli Center at CMI. We are building a CR, a package called CRRao in Julia. It was funded by MIT and version 0.1.0 was released in last December.

(Refer Slide Time: 23:29)

Simple Linear Regression

- Here we consider the linear regression for the `mtcars` datasets

$$MPG = \beta_0 + \beta_1 HP + \beta_2 WT + \epsilon, \quad \epsilon \sim N(0, \sigma^2)$$

- *MPG* : Miles per gallons
- *HP* : Horse Power
- *WT* : Weight



So, for example, if you want to, you know, develop a `mtcar`, you want to just fit in this predictive analytics model regression classification course. We want to fit say miles per gallon is equal to beta naught plus beta 1 horse power plus beta 2 weight plus epsilon.

(Refer Slide Time: 23:52)

CRRao : A Consistent API for Data Science in Julia

R

```
> attach(mtcars)
> df = mtcars
> fit <- lm(mpg~hp+wt,df)
> summary(fit)
```

Julia

```
In [10]: using CRRao, RDatasets, StatsModels
df = dataset("datasets", "mtcars")
model = fit(@formula(MPG ~ HP + WT+Gear), df, LinearRegression())
```



Now, in this model, this is how you will implement in R and in Julia, all you have to do just call like instead of `lm`, you just call `fit`. And then at the rate of formula, the exact way you have given it in R exact way you give it in the Julia, then the data frame. And then you have to say linear regression, that is it and when if it is logistic regression, then you say logistic regression. If it is Poisson equation, right, say Poisson equation. Because of the multiple dispatch, still `fit` will be there. So, for each fit, all you have to do `fit`, that is it.

(Refer Slide Time: 24:32)

CRRao : A Consistent API for Data Science in Julia

Language	Package / Function	Mean Time Taken
Python	statsmodels / ols	2106.60 μ s
Python	sklearn / fit	559.90 μ s
R	stats / lm	380.13 μ s
Julia	CRRao / fit	160.22 μ s

Speed makes Julia a sustainable, economically viable and green language. - Helps ESG compliance
Reduce cost in a range of $\frac{1}{2}$ - $\frac{1}{10}$ th

Oops sorry!
I mean Julia



Similarly, so, and then what we did, we compared the exact same model, exact same implementation in Python, R and Julia. And what we found that in Python sklearn, it took about 560 microseconds, milliseconds actually. And in R, it took about 380 milliseconds. And in Julia, it took about 160 milliseconds. So, what is it going on here? So, what we found it in this is a really interesting phenomenon that we I found and that makes me super excited and why I will tell you.

In industry nowadays, what is happening is nobody wants to maintain their server. So, they are moving all their data related work and data science related project in the cloud. Now, in cloud, the all cloud services are basically charging the their client for the services. The time they are using the their services. Now, imagine you have a service that for each microsecond, it charges the 1 dollar, right.

So, the same model, this is the exact same model for the exact same model when I am trying to fit in Python. It will cost me 560 dollar, whereas, in R, it will cost me 380 dollar, whereas, in Julia's CRRao will cost you only 160 dollar. So, immediately you can see Julia makes it economically viable. You can save lot of money; you can reduce your data science project cost significantly. If you just move your project gradually from R to Python, that will reduce your implementation cost significantly at least for 30, 40, 50, percent.

Now, once you reduce your economically, the implementation cost that makes the profitability and return on investment on the data science team and data science project more viable. At the same time, the data science projects become a much more sustainable language because, if you are implementing the same model at a one half of a time; that means, not only you paying less for your model implementation model training, essentially you are consuming less energy.

That means your electricity cost is going down. That means your carbon footprint to train your model is going down. So, that makes Julia is a go-to language and this will help all the Indian company companies, a Indian corporates to compliant with their ESG requirement. What is ESG? ESG is Environment and Sustainability and Governance.

It is a very important concept that is becoming very popular and most of the companies in India, in Europe, in US, they are trying to become ESG compliant. So, Julia will help the Indian industries to become ESG compliant.

(Refer Slide Time: 28:31)

CRRao : A Consistent API for Data Science in Julia

- Simple Linear Regression

```
In [10]: using CRRao, RDatasets, StatsModels
          df = dataset("datasets", "mtcars")
          model = fit(@formula(MPG ~ HP + WT+Gear), df, LinearRegression())
```

OLS method

- Bayesian Linear Regression

```
In [11]: using CRRao, RDatasets, StatsModels
          df = dataset("datasets", "mtcars")
          model = CRRao.fit(@formula(MPG ~ HP + WT+Gear), df, LinearRegression(), Prior_Ridge())
```

Quantum Monte Carlo method



So, in CRRao, you can implement this you know simple linear regression and the Bayesian linear regression as you see the exact same model, exact same DF (Refer Time: 28:45) linear regression and at the end, if you just give prior ridge, it will just go and implement the Bayesian linear regression. And it will itself will figure out if you do not give anything by default, it will implement OLS method.

But if you give prior ridge, it will implement quantum Monte Carlo method. It will figure out, ok, you are asking the user is asking for me to implement Bayesian linear regression and it will just go and implement the same.

(Refer Slide Time: 29:11)

cm | CHENNAI MATHEMATICAL INSTITUTE

NPTEL

Bayesian Regression in R vs Julia

```
1 data {
2
3   int<lower=0> N;
4   int<lower=0, upper=1> det(N);
5   vector<lower=0, [N] depth;
6   int<lower=0> K;
7   vector<lower=0, [K] depth_priors;
8 }
9
10
11 parameters {
12
13   // The (unobserved) model parameters that we want to recover
14   real alpha;
15   real beta;
16 }
17
18
19 model {
20
21   // A logistic regression model relating the defect depth to whether it will be detected
22   det ~ bernoulli_logit(alpha + beta * log(depth));
23 }
24
25 // Prior models for the unobserved parameters
26 alpha ~ normal(0, 1);
27 beta ~ normal(1, 1);
28 }
```

```
In [1]: using CRao, RDatasets, StatsModels
df = dataset("datasets", "mtcars")
model = CRao.fii(fformula(HPG ~ HP + Wt*Gear), df, LinearRegression(), Prior_Ridge())
```



So, here I have given you a code. So, this is the Stan code that I have written for, you know, Bayesian regression in R. It is about a 30 lines of code that I wrote in Stan. But then I thought I can write a small line of code, which will be like in one line implement the Bayesian linear regression with ridge prior.

But what I found that implementing anything on the top of Stan is very difficult because Stan makes it a three language problem because you have to have a very correct version of C in your system. Then you have to have a correct version of Stan and you have to have a correct version of R and then only you have to you will be able to make anything on the top of it.

So, three language version problem is even too much to handle. So, I just kind of gave up on that idea. But when I found Julia, I found that ok yes; I can build something very easy. And here is Bayesian regression with ridge prior implementation with Julia.

(Refer Slide Time: 30:20)

List of Models CRRao can solve

1. Linear Regression
2. Logistic Regression
3. Poisson Regression
4. Negative Binomial Regression



(Refer Slide Time: 30:28)

CRRao capabilities

- **Frequentist Methods**

- Likelihood Method :

- **Bayesian Method**

- Ridge Prior (Gaussian Prior / L2 Penalty)
- Laplace Prior (Lasso / L1 Penalty)
- Cauchy Prior
- T-Distributed Prior
- Uniform Prior

Computational Methods

OLS / IRLS/ Fisher's Scoring Method

Computational Methods

MCMC / HMC



So, currently CRRao implements linear regression, logistic regression, Poisson regression, negative binomial regression. It can implement frequentist method Bayesian method, both or OLS and Fisher's scoring method, MCMC and Hamiltonian Monte Carlo method.

(Refer Slide Time: 30:40)

CRRao is Learning

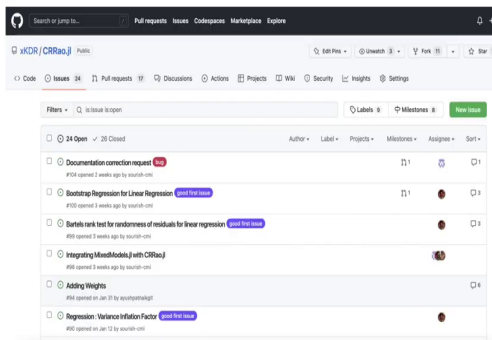
- Discriminant Analysis
- Survival Analysis
- Bootstrap Inference
- Hierarchical Bayesian Models
- Gaussian Process Regression
- Mixed Effect Models



Currently, we are trying to implement discriminant analysis, survival analysis, bootstrap inference. Well, bootstrap inference is almost 50 percent done. It will be released soon. Hierarchical Bayesian models, Gaussian process regression, mixed effect models. These are all we are trying to implement by the next year.

(Refer Slide Time: 31:07)

Open Development on GitHub



And this is our this is this development is completely open development on GitHub and all the issues that we want to implement. There are bunch of good first issues that I have kind of marked it. So, anybody who are interested contributing in this project, please do so.

You can pick up any of your first good first issues and start, you know, developing in either you can even you can start with comment and you can write a small piece of code and submit it in do a pull request or just put a, you know, thing. And then there is a if you really like our projects here a project, you can here is the star, you can, you know, click the star and support us.

(Refer Slide Time: 31:55)

More on **julia** Packages



(Refer Slide Time: 31:57)

More on **julia** Packages

Our Packages	GitHub Stars
TSFrames (0.2.0)	67
NighttimeLights (0.1.0)	13
Survey (0.1.0)	34
CRRao (0.1.0)	17
---	131 (Total)



So, more on Julia packages, our group is developing a few more packages, TS Frames, nighttime lights. So, nighttime lights is a package which handles the satellite, which tries to do data science on satellite images. So, in the satellite images, there is a particular brand which takes the captures the images of the nighttime lights.

And it is do the data science on the nighttime lights. So, this package is a very beautiful package. Then survey and CRRao. So, these are the packages that are we are building our GitHub stars right now around 130. But I am hoping that soon it will go up. Please support us. Click the star button and support us on GitHub.

(Refer Slide Time: 32:43)

Acknowledgement

Mentor



Dr. Viral Shah
Co-Creator of Julia
CEO, Julia Hub



Alan Edelman, Distinguished
Professor, MIT
Co-Creator of Julia



(Refer Slide Time: 32:56)

Team



Sidhant Choudhury,
CMI, BSc, GSoC



Sourish Das,
CMI



Mousum Dutta,
Founder MSA Labs,
Visiting Faculty CMI



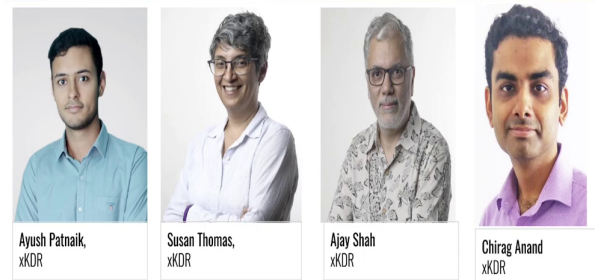
Shouvik Ghosh,
CMI, BSc



Our mentor, Dr. Viral Shah, who is a co-creator of Julia and Professor Alan Edelman, distinguished professor of MIT. And then our team, Sidhant is a CMI student. Sourish is also CMI student, BSc student. They are consistently contributing this in this package. Mousum is my friends and collaborator for long time.

(Refer Slide Time: 33:15)

Team



Ayush is in now Australia, Susan Ajay in Bombay and Chirag in Delhi. Chirag is one of the major implementer of TS Frame. And this is a, you can see there is a team this team is all over the country.

And thank you very much. And please do support us on the GitHub. I will share the link on the GitHub in the below in the YouTube description. Please do support us and click us and come and join our group of group to do develop CRRao package in Julia, adopt Julia. I am sure in next 3 to 5 years down the line. Julia will become a go-to language for everybody.

Thank you very much, see you in the next video.