Published by:

TECHNICS PUBLICATIONS
TECHNOLOGY / LEADERSHIP

2 Lindsley Road, Basking Ridge, NJ 07920 USA
https://www.TechnicsPub.com

Edited by Jessica McCurdy-Crooks
Cover design by Lorena Molinari

First Printing 2020
Copyright © 2020 Zacharias Voulgaris

ISBN, print ed. 9781634628136
ISBN, ePub ed. 9781634628150
ISBN, PDF ed. 9781634628167

Library of Congress Control Number: 2020935538

Contents at a Glance

Introduction

When I wrote the book *Julia for Data Science* a few years ago, it was an innovative book where few people believed in this language enough to write a whole book on the subject. Also, existing books at that time focused on Julia as a programming language rather than as a tool for data science work. Fortunately, things have changed since then, especially as the language matured, and its merits became more known.

Nowadays there are plenty of books on Julia, and lately, even courses offered by a branch of Julia Computing called Julia Academy (www.juliaacademy.com). What's more, there are courses on Julia in various universities around the world. Recently, there was a book on Julia and its application to Statistics.

After attending the Julia conference in the summer of 2018 in London, where Julia 1.0 made its debut, I started considering writing another book on the language. I was thinking of a second edition to *Julia for Data Science*, but after looking into the latest trends and the vastness of Machine Learning, I decided it would be best to focus on this particular aspect of data science. After all, with the popularity of Artificial Intelligence (AI) methods in Machine Learning (ML), it seems that data engineering is not as important as it used to be, while there is so much material on ML methods, that if I were to write a second edition of the Julia for Data Science book, it would be too large!

Anyway, parallel to the books, videos, and other materials I developed since then, including an entry in Springer's online encyclopedia on Big Data, on the Julia language (https://bit.ly/2VkzjZJ), I did some (non-academic) research in data science. Naturally, I used Julia for this, as well as many proof-of-concept projects I carried out during that time. So, when I write about Julia, I have a very practical approach to the language. What's more, I'm a practicing data scientist, not a developer or a professor, so my take on the language is very hands-on and always related to data science from a holistic perspective.

There are many books that talk about the programming aspects of Julia and some of them do a good job at it, but few books talk about how it can be useful to a data scientist or a Machine Learning professional—now enter this book. This book will be a useful aid to data science professionals who wish to use Julia in their work, be it data science or any data analytics task in general.

Since you are reading this book, you must be at least a bit curious about the Julia programming language and its usefulness in Machine Learning. But chances are that you need to be convinced of Julia's power. Everyone talks about the merits of Python, Scala, R, and even MATLAB, when it comes to data science work—where does Julia fit?

In addition, Julia has fewer packages than the above languages, because it is very easy to code from scratch. Perhaps, that's why it appeals so much to developers, particularly those who are more creative and curious to explore. Not only is it easy to code, but the code is very fast, both as a prototyping language and for executing existing code. In Python or R, for example, custom scripts are not encouraged, especially when it comes to loops, due to slow performance. Besides, most of the packages in these languages are developed in some low-level language (usually C or C++), making the corresponding scripts lightning fast. In Julia, most of the packages are written in Julia itself since you don't need to rely on a low-level language for a performance boost.

In addition, Julia empowers its users to do their own thing. Not just building a Machine Learning model from scratch, which we'll do later on in this book, but also developing new algorithms, new heuristics, and even new data science systems leveraging Machine Learning algorithms to use in your work. Without the performance boost of Julia and the ease of use of its coding syntax and grammar, this wouldn't have been as feasible a task. Nevertheless, if you want to rely solely on existing packages for your work, you have this option too, as we'll see in a later part of this book.

Julia is also quite mature when it comes to scientific work, including data science. Even before its production-ready release (ver. 1.0), it was used in scientific research and had managed to become one of the languages that the MXNet framework supported with APIs (https://bit.ly/3c4yewh). Soon after that, there was even an API for Spark, making Julia a relevant language for different kinds of data science work (https://bit.ly/2zUw8k9). Now, things have escalated since the language has plenty more packages for processes related to data science, including many Machine Learning models.

Julia is also quite easy to learn. Although many programming languages require a certain familiarity with programming, Julia is fairly straightforward right off the shelf. If someone has coded in Python, R, or MATLAB, it will seem even easier to learn, since it shares a lot of the syntax, particularly with MATLAB. Besides, many of the high-level languages used in data science have a similar structure, something that Julia shares to a certain extent, even if it follows a somewhat different programming style, namely the functional paradigm.

Compatibility with other programming languages is another advantage that Julia has as a language. Julia scripts can be called from other languages, like Python and R, while you can call scripts from other languages (including C), via the Julia kernel, making the integration of Julia with existing pipelines feasible and practical, and greatly facilitating your learning of Julia since you don't have to jump to this new programming ecosystem and abandon everything you've developed in other languages already. Appendix B covers how other languages interact with Julia.

One of the key differentiators of Julia from other data science languages is *multiple dispatch*. This is a very useful feature of the language enabling the programmer to have functions with the same name but using different inputs. The idea is that the user of these functions doesn't have to remember numerous names for specialized versions of the same function, as Julia itself decides which one to use, based on the inputs provided. Multiple dispatches, therefore, allow for cleaner code and easier maintenance of scripts, enabling the user to leverage the functional programming paradigm, saving time and effort.

Julia's built-in package manager is another reason why this programming language is a good choice when it comes to data science work. Since the use of packages is commonplace in all high-level languages, Julia was designed with this in mind. As a result, it is possible to add, remove, and update packages of the language within its kernel, using simple commands. If you use a lot of external code for your work, this feature is particularly useful, especially today, when a lot of the code that used to be in the base package of Julia is found in external packages instead. Also, the current package manager module is much faster than it used to be, so updating existing packages is a walk in the park. Naturally, you can use the package manager even in IDEs like Jupyter, as long as you first load the corresponding package, namely, *Pkg*.

One of the biggest advantages of Julia, however, is not its code-related aspects, but the people who make it shine. Namely, there is a team of talented developers who work to fix bugs and develop new features for this language. Note that this is not some random start-up but one that involves a core of MIT graduates, as well as a professor who works at the well-known technical institution. What's more, there are developers worldwide who contribute to Julia's code base, as well as enthusiasts who discover and report issues with the existing code. Finally, there are lots of people who use Julia in their work, be it financial modeling, scientific research, and data science projects. All this makes Julia not just a language, but a vibrant community that continues to grow. The large number of Meetup groups for this language may also have contributed to this phenomenon.

Finally, Julia is also at the epicenter of deep learning-related innovation. From the deep learning package Knet to the Gen programming language (https://bit.ly/3b03M5W), Julia has been driving innovations in this AI-related area. Note that although Gen is branded as a wide scope AI language, it also handles probabilistic modeling, particularly Bayesian Statistics, while it also has an application focus in Computer Vision. MIT could have gone with any programming language, but they decided on Julia, something which is quite telling regarding the language's promising presence in our field. As for the Knet package for deep learning, it is the first deep learning framework written entirely in Julia, something that attests to the demand for alternatives to existing systems like MXNet and Tensorflow.

If all this hasn't convinced you of the numerous merits of the Julia language, perhaps you need to try it out for yourself. Many people who were hardcore R users shifted to Julia over the past few years, and there have been many other users who have formed a positive opinion of Julia after using it for a bit. Given how challenging it is to learn a new programming language (unless you are a coder already), this says a lot about the ease of use that Julia exhibits and its similarity to other high-level languages. Besides, there are numerous tutorials and introductory courses on the language, making it easy to learn. As for more advanced stuff regarding Julia, there is always this book!

The book is structured as follows. In chapter 1, we'll look at Julia as a programming language today and focus on what makes it stand out as a data science language, particularly on the Machine Learning front. In chapter 2, we'll explore how you can set up Julia on your computer or on the cloud. We'll also look into the IDE options in this chapter and explain why we delved into Jupyter in this book. Chapter 3 will examine the essential libraries of Julia for data science work. Packages related to visuals, data structures, and some very useful mathematical processes, will be covered in this part of the book. In chapter 4 we'll shift gears and look at Machine Learning—there is a need to clear some things up before we look at specific ML libraries using Julia, which we'll do in chapter 5. Chapter 6 explores examples and exercises, and chapter 7 shows you how to code a

Machine Learning model from scratch using Julia. To make it more interesting, the model we'll be working with is not one that can be found in any package, although it is one that has been proven to be effective and efficient.

Naturally, this book would be incomplete, if we didn't look into dimensionality reduction methods based on Machine Learning. That's something we'll delve into in chapter 8, through a closer examination of two of the most mature packages. In chapter 9, we'll examine some additional topics that play a role in Machine Learning, like parallelization and proper data engineering processes, which are oftentimes essential prerequisites to data modeling. In chapter 10, we'll look at how all this Machine Learning work impacts the business world and how to best use this know-how when liaising with other project stakeholders. We'll then examine some useful considerations to have about this subject in chapter 11, and we look at the future trends of Machine Learning and Julia in chapter 12. We conclude with some suggestions for what you can do next.

As a supplement to your learning of this subject, there will be a series of questions for most of the chapters, so that you can test your understanding of the topics presented. These questions can help you think about this subject critically and explore it in more depth, gaining a better understanding of Julia and its usefulness.

Beyond all this material, there is also an extensive glossary of the most important terms used in this book, as well as three appendices with supplementary material. The first appendix contains all the answers to the questions and exercises throughout the book. The second appendix looks at Julia's relationship with other programming languages and how you can leverage the bridge packages. Finally, the third appendix contains three heuristics, largely specialized but quite useful for data science work. All of them are implemented in Julia and are a good addition to the material presented previously, as they don't exist in any of the existing packages yet can add value to a data science project.

Naturally, since this is more of a hands-on book, it is accompanied by a series of Jupyter notebooks that contain all the examples presented in the text. On top of that, the data files of the datasets used, as well as any auxiliary scripts, are also provided. You can find all of this material at https://technicspub.com/julia. Note that future versions of Julia may not be 100% compatible with this code, so you may need to make alterations to it to ensure it works with the Julia version you are using. All the code here works well with Julia 1.1.

Upon reading and practicing the material presented in this book, you should be able to have a good grasp on Machine Learning and how you can leverage Julia for related projects. However, there is plenty more to learn on this topic, which is why this book is best seen as a starting point rather than a complete resource on the topic. After all, the field of Machine Learning is blooming, so there are new things coming up constantly. However, if you have a solid understanding of the topic, you should be able to learn the new methods and techniques quicker and better.

So, without any further ado, let's get started.

Chapter 1
Julia Today

Let's now look at how Julia fares as a programming language today. We'll start by briefly examining its programming paradigm, talk a bit about data science languages in general, and then look at Julia as a multi-purpose language. Next, we'll explore Julia as a data science language, and talk about the user community. We'll conclude by examining some useful resources for learning Julia basics.

The functional paradigm of programming

Programming can be done in various ways and today we have even more paradigms than the previous generation. Namely, we can choose among the following programming paradigms:

- procedural
- object-oriented
- functional

Procedural programming has to do with providing simple instructions to the computer (or any programmable machine) so that it can follow a particular procedure effectively. A procedural program contains a systematic order of commands, statements, and functions designed to complete a computational task or process. Languages like C, FORTRAN, Pascal, and Basic are procedural languages. Despite their long history in the computing world, some of these languages are still relevant today.

The key structures of procedural programming are the variable and the command. The former involves the storage of data and the latter its processing. Procedural programming is fast and fairly easy to learn, though not as easy to master since when doing complex tasks in a procedural language, it can take many lines of code. Procedural programming is often referred to as imperative programming. Despite its usefulness in computer science, procedural programming is rarely used in data science, unless it's a C script for a very specific supportive role.

As for object-oriented programming (OOP), this is the paradigm that involves the use of objects as the key structures of a program. An object is a software bundle that is characterized by states and behaviors. Much like physical objects, objects in OOP are defined by a series of properties and actions that can be taken related to these objects. This makes the OOP paradigm fairly intuitive and easy to work with, though mastering it is still somewhat time-consuming. After all, the plethora of data involved in these objects can be overwhelming and difficult to handle, especially if there are many objects that have similar attributes.

The actions related to the objects are represented by the various functions that can be applied to them. These are defined within the class that describes the most general form of an object. That's why we say that objects are instances of particular classes.

Let's look at a non-programming example, namely the car Fiat Punto with a particular license plate. This particular vehicle is an object of the class Fiat Punto, which describes all the cars of that Fiat make and of the Punto model. We can have a more general class called Fiat that describes all the vehicles and other products developed by this automotive company, thereby forming a hierarchy of classes. An object like a particular Fiat Punto can have functions like "drive", "park", and "fill up with gas." Each one of these functions applies to all the objects of this class, since other cars like this one can be driven, parked, and filled up with gas. However, an electric vehicle of the class Fiat Punto may have other functions, such as "charge battery" while the function "fill up with gas" wouldn't make any sense for cars belonging to that sub-class consisting of electric cars under the Fiat Punto umbrella.

Object-oriented programming is the most popular paradigm today and Julia follows that to some extent, since it allows for class structures and objects. Other languages that follow the OOP paradigm are Python, R, C++, C#, Java, and Javascript. However, debugging an OOP script can be time-consuming, particularly if it's complex, which is why a new paradigm was developed to alleviate this shortcoming of OOP: functional programming.

Functional programming is all about functions and their application on different inputs, to yield outputs. It's a fairly straightforward paradigm, though it has its limitations too. For example, a strictly functional programming script doesn't allow for a common workspace where variables can be shared among different functions. So creating more complex scripts can be a challenge, though this peculiarity of functional programming allows for faster execution speed and mitigates the chances of errors due to variable conflicts. Naturally, the scripts of a functional language are easier to debug and maintain, compared to those of other programming paradigms.

Julia follows the functional programming paradigm to a great extent, though it does allow for shared variables among different functions, bypassing this peculiarity of a nonexistent shared workspace. Also, the variables that are not used any more are discarded automatically (also known as garbage collection), which conserves memory resources and prevents memory leakage.

The functional programming paradigms are quite popular today due to performance. Although, many languages that make use of functional programming are also compatible with the OOP paradigm. A good example of such a language, other than Julia, is Scala. Additional languages that follow the functional paradigm include Haskell, Clojure, F#, and Elm.

Data science programming languages

What does all this have to do with data science and Machine Learning though? Few people are aware that many of the programming languages interact with each other. For performance reasons, it's often the case that a program is developed in one of them (e.g. Python) and implemented in another one (e.g. Java). This is known as the two-language problem, something that is a necessary evil when using a certain kind of language to prototype (namely a high-level language) and a different kind of language to deploy the program (namely a low-level language). The two-language problem creates additional latency and various kinds of liabilities such as coding errors, all of which can be avoided through the use of a language like Julia.

High-level languages are those that are easy to use since their syntax is closer to how we think and communicate. They are fairly easy to learn too, and because of their intuitive nature, developing a program in them can be fairly fast, even if you are not a professional programmer. High-level languages are key to data science since most data scientists don't care much about honing their programming skills, since there is so much more to learn in order to be able to work in this field. High-level languages include

Python, R, MATLAB, and Julia. MATLAB is not used as much due to its high license fees, but you may see some people using its open-source clone, Octave. However, both MATLAB and Octave are fairly slow and despite their ease of use, they are not used as much in data science today (especially Octave, as it's not as practical for more complex problems due to its lack of packages). Other high-level languages are also somewhat slow, since the code written in them needs to be translated into something the computer can understand. Julia, however, is an exception to this, as we'll see later on.

Low-level languages are those programming languages that are "close to the metal" meaning that they are near the language computers understand: machine language. These languages don't need much translation for the computer to understand, which enables them to be super-fast to run. In fact, most performance critical processes, such as those responsible for infrastructure-related tasks, are handled by low-level languages for this reason. That's why programmers often spend years learning these languages, as these languages are not as intuitive as high-level languages. Some low-level languages (e.g. Java) are used in data science, but only as supplementary tools due to the challenge in using them to prototype. When using a high-level language however, it's not too difficult to migrate to a low-level language to ensure high performance.

Julia as a multi-purpose language

Let's now zoom in on Julia as a multi-purpose language—it follows the functional paradigm but is also connected to the OOP one. First of all, Julia was not designed to be a niche language, since much like most programming languages, it exists to bridge the user and the computer efficiently. Perhaps that's why there weren't that many data science packages in it when it was first released. It was only after some of us discovered that it can be leveraged as a programming language in data science.

Whatever the case, since it has reached a critical mass of users, Julia has been applied to many problems, ranging from the strictly scientific to the more applied. It became particularly popular in economics and finance at one point, so much so that the best introduction to Julia was a niche blog run by an economics professional. Also, a little known fact, many financial companies use Julia. After all, in domains like this one, performance matters. So, a high performance high-level language is like a dream come true.

Julia has developed in many ways, however, beyond economics and finance applications. For example, Julia has been successfully used in self-driving cars. Tiny computers (these computers that work on a single board and are very popular today among DIY tech enthusiasts) can run Julia too, since it's a cross-platform language, much like every other serious language. As a result, it doesn't take much imagination to figure out that it can be leveraged in computer vision applications which are an essential part of self-driving vehicles. Check out this video corresponding to this project, from a time before Julia v. 1.0: https://bit.ly/2JV8wh1.

The fact that Julia is so versatile makes it a useful platform for professionals and hobbyists alike. That's why there is a large variety of packages for all kinds of applications—see the official Julia website: https://pkg.julialang.org/docs. Currently there are over 2,000 packages for Julia and the ones on this webpage are just the official ones. There are a few unofficial packages too, plus nowadays it's easier than ever to create your own packages in Julia. So, even though Julia is often seen in the context of data analytics applications, it is much more than that.

Julia as a data science language

Let's now look at how Julia fares as a data science language and how it can be leveraged in the development and testing of Machine Learning models. First of all, although Julia was not designed with data science in mind, it ticks all the boxes of a data science language. Namely:

- it is easy to code and quite intuitive in its syntax
- its scripts are easy to debug and maintain
- it has great plotting packages enabling good data visualization
- it has several other packages that delve into Statistics and Machine Learning
- it liaises with other programming languages that are used in data science
- it is not esoteric and thereby it is accessible to people not trained in programming

- there is at least one good book on it, showcasing its usefulness in data science
- there are several companies that value Julia-savvy data scientists

As a bonus, it is fairly fast, so it overcomes the two-language problem. Not many data science programming languages can do that. As for the data science languages that are high performing (e.g. Scala), they are not nearly as easy to use as Julia.

But has it been used in practice for data science tasks? The short answer is yes, definitely. The longer answer is yes, but not as much as it should. This creates a bit of a discrepancy since many data scientists are either too busy or too conservative to give it a chance. After all, Python and Scala are good for data science work, so why would someone want to jump ship on them? Even people who have attended Julia Meetups and other events, showcasing how promising this language is in data science work, are still often undecided. The most common reason given is that they are not sure about the packages being enough and good enough. This is a big topic and it deserves its own chapter, which is why we are not going to go into it right now. Suffice to say that these concerns are not justified and that you can carry out data science projects using Julia exclusively.

Julia is great for data science because it is not tied to any particular platform, especially a cloud one. Even though there is an indisputable partnership with a major tech company, enabling Julia to be an option in that company's cloud, Julia can be used anywhere. Also, the fact that it is not widely known which company this is, shows that Julia is still independent.

Of course, Julia Computing has its own cloud service which it promotes for the seamless use of the language on a server, but they still give you a choice as to where you deploy your Julia programs. Other languages, like Swift for example, are tied to a particular ecosystem (in this case Apple's), which may be limiting in some cases. Julia doesn't do that and is extremely unlikely to do that in the future, either. All this enables Julia to tie in to all kinds of frameworks and tech ecosystems, making it a versatile piece of technology, compatible with many pipelines.

Julia as the epicenter of a community of users

Beyond the technical aspects of the language, it's good to be aware of the community aspect of it too. After all, it was never intended to be a niche tool for a few programmers having too much time on their hands! Instead, it was created to be adopted on a larger scale and aid in numerous areas, not just computer science. At least that's what can be derived by reading the thesis from MIT's Dr. Jeff Bezanson, who is one of the creators of the Julia language.

Although the Julia community is fairly small compared to other programming language communities, it is growing at a fast pace. Beyond the Julia conference that takes place annually (http://www.juliacon.org), there are several other events and groups where Julians gather. From the online ones such as the GitHub groups and Slack channels, to the face-to-face ones such as Meetup groups, the Julia community is vibrant. You can learn more about it at the official web page: https://julialang.org/community.

What's particularly interesting when it comes to Julia enthusiasts, is the diversity of the community. Julia may have started in a fairly technologically advanced corner of the world (Cambridge, Massachusetts), but it has spread to every part of the globe, including less developed areas. At the time of this writing, there are Meetups all over North America, Europe, Asia, Oceania, and even South America (particularly Brazil). What's more, there is a lot of work being done in Julia Computing to ensure that the set of users of the language remains diverse and as inclusive as possible.

Now, as mentioned previously, you don't need much help to pick up this language since it's fairly intuitive and there are plenty of educational resources at your disposal. However, for someone new to it, or someone who is not particularly versed in programming, it may be a challenging task. Having a support group of people who are in the same boat, as well as people more experienced in Julia programming can be a big plus. Participating in hackathons and talks may be enough to motivate or even inspire you to invest some time in learning and mastering this language.

The value of the community goes beyond all this, however. After all, just like most programming languages, Julia is an open source project. This means that even the source code of the language itself is open to everyone. Of course, not everyone is capable of adding something useful to the code-base of the base package (the core of the language), but everyone can report bugs, run tests, and even contribute additional packages. If you are more courageous, you can give direct feedback to its contributors and even promote Julia in an educational platform. Whatever the case, community members are encouraged to refine and promote the language in whatever way they feel comfortable. Perhaps this is why it has grown so quickly, unlike its closed-source counterparts, such as MATLAB, which although very useful, is limited to a niche user group, mainly researchers and engineers in large companies.

What may be most inspiring about the Julia community is that everyone there, including its creators, are very approachable. Some of these people are the equivalent of celebrities in the programming world, yet they are down-to-earth and happy to share their stories like you'd share stories with a friend over a coffee or a drink. Perhaps that's the most powerful aspect of the community and what makes it stand out from other programming language user groups, since it's akin to a group of friends who share a common interest.

Useful resources for learning Julia

Before we finish this chapter, let's look at some useful resources to brush up your Julia know-how. All of these are free, so you don't need to spend any money on learning the language. If you do have some funds that you'd like to devote to that purpose, feel free to buy my *Julia for Data Science* book!

First of all, there is a simple tutorial for v. 1.0 of the language (a fairly new release which is compatible with the one used in this book, v.1.1.1): https://bit.ly/3a3Bwhn. This 150 minute video, developed by the company that created Julia, covers the basics of the language, with plenty of examples to clarify all the points made. It won't make you an expert in Julia, but through it you can learn the ropes of the language and better understand the more advanced Julia resources.

In addition, there is the official documentation of Julia: https://bit.ly/2RsA1D6. This more esoteric resource may not be for the fainthearted, but it does contain all the information related to the functionality of the language that you'll ever need. Also, it's practically impossible to find a more reliable resource, since it was created by the makers of the language and the people committed to maintaining and evolving Julia.

Finally, Julia Wikibook is another great resource when learning the language: https://bit.ly/3b6JDLq. Contrary to other third party resources, this one is as robust as it gets, without being a Julia Computing resource. The Wikibook is not related to Wikipedia in any way, though it does use the same framework (wiki) as the well-known online encyclopedia. However, it reads like a good book and it is regularly updated. This is one of the few resources used in this book that is also used in the *Julia for Data Science* book from 2016.

Beyond these resources, there are plenty more, some more reliable than others. Note, however, that there is no golden resource that can make things easy for you all the way to the mastery of the language. If you are serious about learning the ins and outs of Julia—not just in theory but in practice, you need to use it in projects that are of real value to you, forcing yourself to become familiar with its more subtle aspects. Unfortunately, there is no book that can help in that, since the best way to truly learn a powerful tool like Julia, is to use it to solve challenging problems. Fortunately, before long, this whole process will become more rewarding and somewhat less challenging.

Useful considerations

There are several layers to a sophisticated programming platform and Julia is not an exception to this, no matter how exceptional it is as a language. For instance, Julia is bound to change, as every other programming language changes—except for the various legacy languages that have stopped being relevant in the 1990s. This means that many packages that work fine today may be unusable if they are not maintained to be compatible with the newer and better releases of the language. We'll talk about this more in the next chapter.

What's more, Julia is now a professional language, quite distinct from other new languages. A good example of such a language is Nim, an elegant and powerful programming language that although it has been around for over a decade, it's still under the radar for most data professionals. Nim may be a fairly good language, but it's still more of a novelty and not something someone would learn and hope to use for work projects related to data science.

Note that Julia scripts are not the most secure code files. After all, the programs written in Julia are not compiled, so in order to run them, you need to have the source code exposed. That's why it's best to shield them with APIs, when possible, if you want to have the option of third parties making use of them all while respecting the privacy of your code. Besides, due to the intuitive nature of the language, even someone who doesn't know it well can still figure out your code by looking at it—so you may want to make sure it's secure, especially if it involves critical processes in your organization.

Finally, Julia is not a solo player when it comes to the data science world, unlike Go, for example. Julia can collaborate well with other data science languages, such as Python and R, using the corresponding bridge packages. Also, there are packages that link Julia with C and Java, so you can always leverage packages of all these languages in your Julia scripts. What's more, if you are in a Python or R kernel, you can import Julia scripts too. All this makes the integration of Julia scripts in existing pipelines something doable—though if you are really concerned with performance, it would make sense to migrate your code base to Julia sooner or later.

Summary

- There are different programming paradigms, the most important of which are procedural (imperative), object-oriented (OOP), and functional.
- Julia is a hybrid programming language which although it follows the functional paradigm closely, it has a lot of elements from the OOP one, too.
- Data science programming languages tend to be high-level and, as a result, not particularly fast.
- Julia manages to combine the merits of a high-level language with those of a low-level one, solving the two-language problem—that is, needing both kinds of languages for a data science problem. Also, Julia is an excellent multi-purpose language, having applications that go beyond data science, as well as a respectable number of packages.
- The user community of Julia is a vibrant one and is characterized by diversity, breadth, and accessibility, among other things. Particularly, if you are new to the language, you can benefit from it in various ways, while if you believe in the language and wish to help refine or promote it, there are plenty of ways to do that too.

- There are various resources for learning the basics of Julia, some of which are free. However, the best way to master it is through practice, beyond programming tutorials and specialized books.
- There are certain things you need to consider about Julia as a programming language, such as the fact that it's constantly evolving, its professional presence in the world as a production-ready language, the need to keep your Julia scripts secure when using them with third parties if the source code is part of an organization's IP, and the fact that Julia can collaborate with other programming languages through bridge packages.

Questions

1. What makes Julia a functional language?
2. Why is Julia relevant as a data science language?
3. Do you need to be a programmer to learn and master the Julia language?
4. Isn't Julia too new to be reliable as a programming language?
5. How does Julia compare with Java in terms of performance?

Chapter 2
Setting up Julia

As with other programming languages, Julia is a program, so it needs to be installed and run like any other application on your computer or the cloud. That's something that hasn't received enough attention, and although the average developer won't have any issues handling this process, many data scientists may not find this process as intuitive.

Since we have more urgent things to do, going through the lengthy documentation of Julia to figure out how to make it run on our machines is not a priority. That's why it's good to have a straightforward way of installing Julia, namely a simple step-by-step guide. Also, as the use of IDEs greatly facilitates data science work, we need to install the IDEs and ensure they are working well with Julia. This task may prove even more challenging than installing Julia, since various key steps need to be taken in order to do this properly, something that is not clear in the instructions of these IDEs.

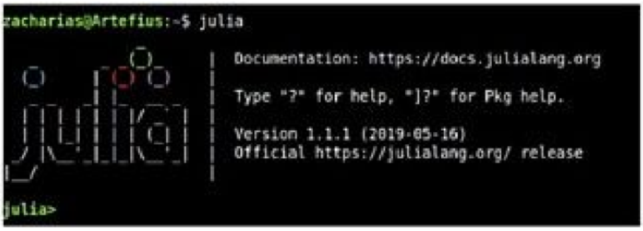
In this chapter, we'll start by looking at how you can install the kernel of the language—that is, install the actual programming language on your machine and then explore how you can do the same with the Jupyter IDE, which is the most popular IDE for data science work. We'll then look into other IDE options for Julia and where you can find them before we take a look at the JuliaBox possibility. Next we'll look at how you can handle Julia scripts and Jupyter notebooks for your work and examine some useful considerations to have about this topic.

Julia kernel installation

Let's start by looking at how you can install the Julia kernel on your machine. Fortunately, the Julia language is compatible with most operating systems, including Linux-based ones (e.g. Linux Mint) and FreeBSD. You can download the installation files here: https://julialang.org/downloads.

Unless you have a good reason to opt for the latest release of the language, you can make use of the latest stable release (usually the first one listed on the aforementioned web page). Then you need to find the most relevant file for your computer. If you are using a 64-bit machine that runs Windows, for example, you will need the .exe file that is on the right of the first row of the table. Note that most machines nowadays are 64-bit—unless you have an older computer.

You can find specific instructions on how you can install the Julia kernel for each operating system at https://julialang.org/downloads/platform.html. Note that if you want to uninstall the kernel for whatever reason, it's best to use the process provided at the end of the web page. If you decide to do that, however, make sure that you have kept your scripts elsewhere so that they are not deleted accidentally. You can replace the Julia kernel easily, but replacing your scripts may be next to impossible. Once you have installed the Julia kernel, you can execute the corresponding file and run Julia. For a Linux-based system, for example, you just type `julia` on the shell, as shown in Fig. 2.1.



```
zacharias@Artefius:~$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.1.1 (2019-05-16)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

Figure 2.1. Screenshot of the Julia kernel being run on a Linux-based system.

Note that the prompt changes to `julia>` once the kernel is loaded on your computer. From this point onward, you can run Julia commands on your computer. To exit the language and return to the shell, you can either type `exit()` or press `Control` and `D`.

Jupyter IDE installation

Let's now explore the Jupyter IDE, which is also the best option for data science work, especially when it comes to wrapper methods and merging the various components of a program. Jupyter has been around for a while, but only fairly recently has it gained popularity thanks to its usefulness in data science projects. Whether you use Python, R, or Julia, Jupyter can provide a very useful and intuitive interface for your programming work, making any programming task more efficient and comprehensible, especially when dealing with complex tasks involving lots of commands and outputs.

The Jupyter IDE is a notebook where you can mix formatted text, code, and the output of that code neatly. The output can include graphics as well, making the notebook a very comprehensible demonstration of your work, which you can showcase at a data science meeting, for example. The fact that Jupyter allows you to export the whole notebook as a PDF file, for example, can help greatly in showcasing your work to a larger audience.

In essence, Jupyter comprises two main screens, the home screen and the notebook one, both of which are viewed on your browser through a local server the Jupyter program uses. The first screen, which you can view in Fig. 2.2, shows you the different notebooks on your machine and when they were last modified. You can also view any folders there are there and navigate through them, much like a file explorer. On this screen, you can also rename the notebooks and even delete them. Finally, you can see which notebooks are currently running. Clicking on any one of these notebooks will open it, usually in a new tab on your browser.

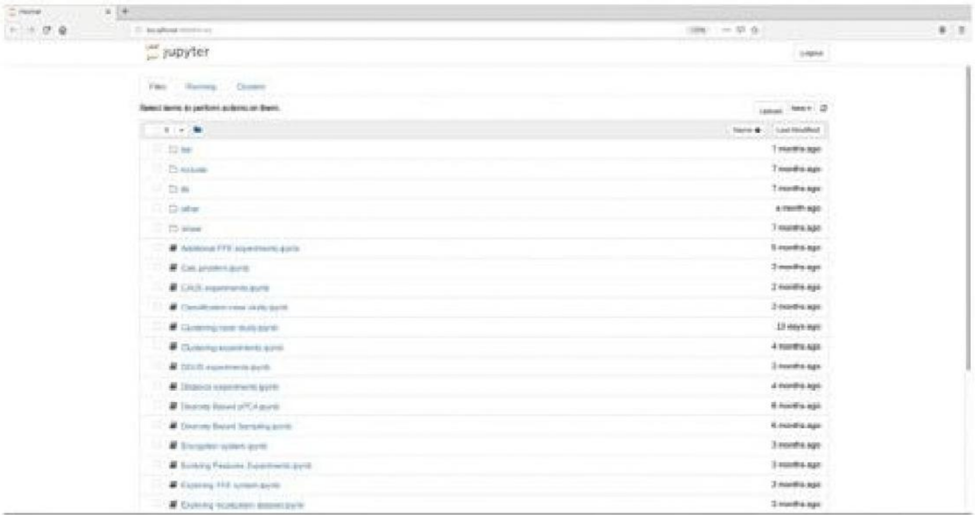


Figure 2.2. Screenshot of Jupyter's Home screen.

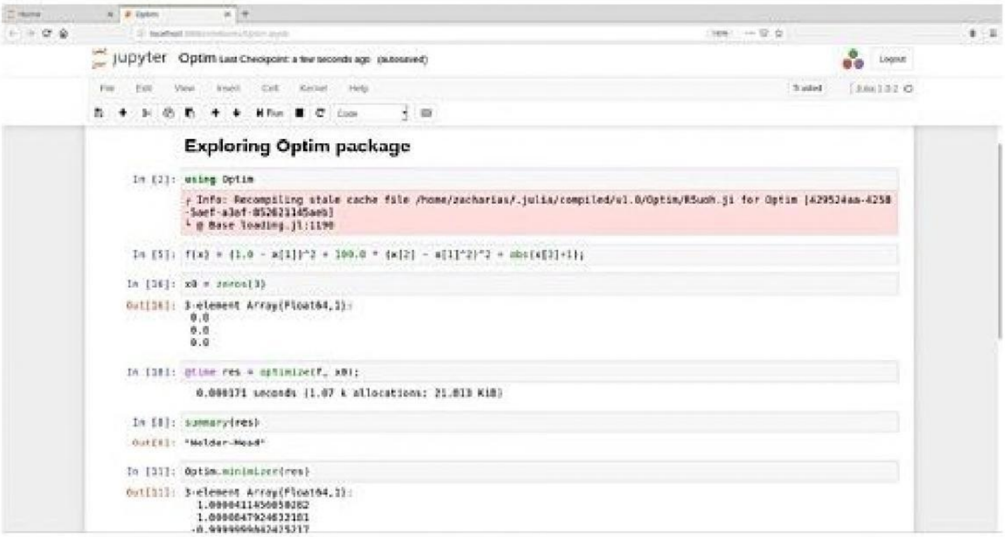
The Jupyter notebook is a bit more interesting. It looks like a document editor but allows for writing and executing code. All the lines that have been marked with `In[#]` are where executable code is included, while those marked with `Out[#]` are where the output of that code is shown (# is a number, related to the order in which that particular cell was executed). Note also that you can include headings to organize your work better and make the notebook easier to read. A sample of a Jupyter notebook appears in Fig. 2.3.

You can download and install Jupyter on your machine through Python by typing the following on the command prompt:

```
python3 -m pip install jupyter (for Python 3),
    or

python -m pip install jupyter (for Python 2)
```

Note that you may want to upgrade the pip program before installing Jupyter, by typing:

```
python3 -m pip install --upgrade pip

or

python -m pip install --upgrade pip
```


jupyter Optim Last Checkpoint: a few seconds ago (autosaved)
Logout
File Edit View Insert Cell Kernel Help
Trusted
Julia 1.0.2
Run
Code

Exploring Optim package

```
In [2]: using Optim
        ┌ Info: Recompiling stale cache file /home/zacharias/.julia/compiled/v1.0/Optim/R5uoh.ji for Optim [429524aa-4258
        -5aef-a3af-852621145aeb]
        └ @ Base loading.jl:1190

In [5]: f(x) = (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2 + abs(x[3]+1);

In [16]: x0 = zeros(3)

Out[16]: 3-element Array{Float64,1}:
          0.0
          0.0
          0.0

In [18]: @time res = optimize(f, x0);
          0.000171 seconds (1.07 k allocations: 21.813 KiB)

In [8]: summary(res)

Out[8]: "Nelder-Mead"

In [31]: Optim.minimizer(res)

Out[31]: 3-element Array{Float64,1}:
           1.0000411456058282
           1.0000847924632101
          -0.9999999662425217
```

Figure 2.3. Screenshot of a Jupyter notebook.

To make Jupyter usable with Julia, however, you need to also install the IJulia package on your machine, through the Julia prompt:

```
Pkg.add("IJulia")
```

Unless IJulia is properly installed, you will not be able to use Julia on your Jupyter Notebook, even if the latter runs fine. So, make sure you pay attention to this step before proceeding to the next one.

To run the Jupyter program, you need to type on the command prompt:

```
jupyter notebook
```

Note that although Jupyter notebook is still the most popular IDE for data science work, there is also a new IDE developed by the same company, called JupyterLab. This is still fairly new, so it's a bit premature to know for sure its efficacy in data science work. However, the design seems to be slick and user-friendly, while the core functionality of the Jupyter Notebook remains more or less the same. In essence, the GUI is the only key difference, while the multitude of windows it entails makes it useful only if you are using at least two monitors. Not a bad option but perhaps a bit hyped overall since there are few things it offers that you cannot do on a conventional Jupyter notebook.

You can learn more about Jupyter Notebook and JupyterLab at the official website: https://jupyter.org.

Other IDE options

Julia has gained enough traction to be acknowledged as a relevant-enough language to have a presence in other IDEs as well. Also Juno is a special IDE designed for Julia specifically. Although it has been unreliable and does not work seamlessly in all operating systems, Juno is one of the first IDEs to accompany the programming language. After all, when it does work, it is very practical and ideal for low-level programming in Julia. Also, before it became its own thing, it was part of Atom, another useful IDE for the language. Note that you can use Atom as an IDE, regardless of Juno. The former is quite stable and works well with Julia.

Atom is probably the best IDE for programming in general, covering a large variety of programming languages, while it can also be used to view text files efficiently. Lately, it includes Julia scripts too, though getting the Julia kernel to run on Atom can be quite a challenge, especially on a Linux-based operating system. However, you can still get the IDE to recognize Julia code and show it with some colors, something that can make every Julia script more comprehensible. You just need to make sure that the package `language-julia` is installed on Atom for this to happen. Note that Atom has a bunch of other useful packages you may want to install as well, though there are no other Julia-related packages that you need for using Atom as an IDE.

Other IDEs include Vim and SublimeText, both of which are exceptional text editors and not just for editing scripts. Note that beyond these editors, any other text editor can function as an IDE. However, without the Julia IDEs syntax formatting dictionary, it would be difficult to read the code properly, while debugging may be more time-consuming. Finally, there is a reference somewhere to Weave as an IDE. However, this is a misconception since Weave is a scientific report generator package designed by the same group that developed Juno, namely JunoLab.

JuliaBox

JuliaBox is another interesting option for running Julia, especially if you need to access the language remotely. Note that this is primarily a commercial option managed by the Julia Computing company, so if you are serious about using it, it would be best to think about your budget. Naturally, there is a free option so that you can try it out. In Fig. 2.4 you can see a screenshot of what it looks like right after the login, while in Fig. 2.5, you can view it after the Jupyter app is launched. As you can see, it looks very much like running Jupyter Notebook on your machine. Note, however, that the versions of Julia that it supports are not the latest ones. Yet, it does support multi-threading, so if you want to use the full spectrum of computing resources in it, you have that option, even for the free option.

Figure 2.4. Screenshot of the JuliaBox system, after login.

Figure 2.5. Screenshot of the JuliaBox system, after launching Jupyter. Note that the files system of every JuliaBox account comes equipped with some tutorials for the newcomers of the language.

Although JuliaBox is fairly popular among many Julia learners, particularly in universities, its main use case is deploying Julia programs in need of a large number of resources. So, in order to make the most of it, it is best to invest some money in it, though preferably after you have implemented and tested your scripts. This way, you can use it without having to worry too much about potential bugs in your code, something that can cause frustrating delays, which you would have to pay for if you have to do the debugging of your scripts on the Julia cloud.

Nevertheless, JuliaBox is the last piece of the puzzle in the data science pipeline, since it can enable you to do even the most challenging part of it, model deployment, in Julia, without having to rely on a tool from a different ecosystem. This wasn't the case a few years ago when Julia first began to make a case for data science and Machine Learning applications. And even if it's not too difficult to create an API in Julia, JuliaBox makes the whole process much easier for deploying your programs on an internet server.

Note that JuliaBox is not the only cloud option for Julia. You can run Julia on the Microsoft cloud (Azure), something you can learn more about in this article: https://bit.ly/3eghymM. Of course, such a move would make sense if you already have worked with Azure or wish to deploy a Julia script in combination with a cloud-based database or other programs that dwell there. For most data science applications, the JuliaBox option would be more than adequate.

Julia scripts and Jupyter notebooks

Handling Julia scripts and Jupyter notebooks is something few people will talk about as it seems obvious to them, even if it doesn't seem so straightforward to someone new to the language or to how Julia is used in a data science context.

First of all, Julia scripts are the `.jl` files that contain Julia code. Any such file will be recognized as having Julia language syntax by an IDE that supports Julia. This way, you can open them and view them in an understandable manner. Note that even if the extension is different, the Julia scripts you have in text files would still run on Julia. However, it's generally good practice to store all your Julia programs in `.jl` files. A key `.jl` file you need to be aware of is the `startup.jl` script, which is located in the `/etc/julia` subfolder of the folder Julia has been installed. So, if you want to execute certain commands as soon as Julia starts, that's where you'd put them. For example, you may want certain libraries to be loaded or change the working directory to a particular folder. You can do these things easily by adding the corresponding commands in the `startup.jl` script. Note that this may slow down Julia a bit, so it's good to remember where this file is kept and perhaps make a backup of it before you modify this script.

As for the Jupyter notebooks, these are the `.ipynb` files. This bizarre extension stems from IPython Notebook, as they were used primarily for Python scripts. As the makers of the Jupyter notebook platform would tell you, Jupyter was created primarily for three programming languages: Julia, Python, and R, which is where its name comes from (Ju from Julia, Py from Python, and r for R, with the te part as a filler syllable).

Anyway, Jupyter notebooks are also text files that are designed to be used by the Jupyter program (technically they are JSON files, so if you were to open such a file in a text editor like Atom, be sure to use the corresponding package to parse them properly). Any project you make using Julia is best stored in such files, though you may use `.jl` files in it too, preferably stored separately, so that they can be leveraged in other projects also. Note that for `.jl` files that you plan to use in a different project, it's advisable to structure all of the code in them as functions.

For larger projects, you'd have several files, be it `.jl` or `.ipynb` ones. It is a good practice to store all of these files in the same folder, particularly a folder dedicated to that project. For `.jl` files that are used in several projects, it's best to have them in a general folder for easier access.

Naturally, you'd want to give all the files of your projects intuitive names or names that you can easily remember. Otherwise, they are bound to get lost as you create additional files in your codebase. Also, for the most important Julia files you have, it's best to back them up regularly.

What's more, if you make changes to these scripts regularly, or if you work on these scripts in tandem with other people, you may want to use a version control system like Git or SVN. Also, as you may have noticed, the bulk of the code base of Julia is on Github, which is also the version control system most Julians use. You can use whatever version control system you like for your Julia work since they are about the same in terms of functionality and ease of use.

If you have lots of Julia scripts working together for a particular task, especially for a generic task, it's best to compile them into a package. Use the `Pkg.jl` package, a package for packages (a meta-package if you will). We'll look into this more in the next chapter, where we'll talk about Julia packages in general, focusing on the most important ones for data science projects.

It's a good practice to separate the scripts that you plan to use in various projects from the ones that are created for drills or one-time usage. Also, it's good to maintain the scripts you plan to use in the future, as the Julia language evolves from release to release, to ensure that they are always relevant and efficient. Bundling the most similar into larger scripts also makes sense as it makes it easier to manage them.

Useful considerations

Installing Julia and an IDE may be fairly straightforward, but there are certain things that you need to keep in mind nevertheless. For example, it's not uncommon to have multiple IDEs when using the language (or any programming language for that matter). That's something that we'd recommend, since the Atom IDE, for example, is much better at handling multiple scripts than a Jupyter notebook. However, you'll still need the latter if you want to do some data science work in a manner that enables easy collaboration and presentation. However, if you develop a new method that you plan to use in various projects, it's best to do that in another IDE, not Jupyter.

Also, when upgrading the Julia kernel, you need to be careful to ensure that the new release is working properly. If you just download the new release on your machine and run it, things may be fine if you do so from its folder, but the symbolic link to the new executable needs to be updated too. This omission may create serious confusion and make you waste a lot of time. Besides, once you get a grip on the latest release of the language, there is little point in keeping the older release around, unless you have legacy code lingering on your computer.

What's more, you need to be mindful of the package situation when running a different release of the language. After all, every new release is clean, and you have to install all the packages you need from scratch. It's not too challenging a process, but it does take some time, so you need to plan properly. Fortunately, since version 1.0 of Julia, the package installing and updating process is a breeze, significantly better than that of other data science programming languages.

Furthermore, it's important to check if a package you plan to use is working properly in the new release before you upgrade your Julia kernel. Otherwise, you may find that you need to revert to a previous release just so that you can run that package properly. Luckily, this doesn't happen often and when it does happen, it is usually a temporary issue. However, if you have crucial code that relies on a certain package, that's something you need to be aware of as it can save you time and effort.

Finally, all these IDEs and accessories of the Julia ecosystem may be great, but they are no substitute for effective programming and good practices, qualities that are both transferable and valuable when it comes to carrying out any data science project.

Summary

- The installation of the Julia kernel is a fairly straightforward process as long as you obtain the right file for your computer and follow the corresponding instructions.
- Installing an IDE for Julia is not essential but particularly useful, especially for data science work.
- Jupyter is the most popular IDE for data science projects and works seamlessly with Julia. Always install the IJulia package on Julia before trying to run Jupyter in combination with Julia, though.

- JupyterLab is another product by the same organization that developed Jupyter Notebook, sharing most of its traits but with a slicker interface. However, whether it improves the efficacy of data science work is something yet to be determined.
- Several other IDEs are compatible with Julia, such as Atom and Juno.
- JuliaBox is a popular cloud-based option for Julia, essentially a Jupyter notebook that you can access from any computer. Beyond JuliaBox, there is the Azure cloud option, too, for more niche use cases.
- Handling Julia scripts and notebooks is needed to maintain a certain security level for your work. It's best to keep all of these files in a separate folder, different than the one where Julia is installed. Also, regular backups of these files are highly recommended.

Questions

1. When would you use an IDE like Atom in your work?
2. Can you use a basic text editor for creating or editing your Julia scripts?
3. What kind of file would you use to store the code of a multi-purpose function you have created, to use in your data science projects?
4. Can you use Jupyter offline?
5. Is Jupyter better than other IDEs?
6. Can you run Julia on your mobile device?

Chapter 3

Julia Libraries

Let's now examine some key libraries for Julia that are useful in all kinds of data analytics projects. Libraries are usually referred to as packages, and in Julia, they are handled by a package manager program, which is part of the main language. Since there are plenty of packages now, it's good to prioritize which ones to use since some are better than others for a particular task. Also, as the language matures, some packages are left behind, and even if they were great once, they might not be so relevant anymore.

In this chapter, we'll look into packages for Julia, with a focus on functionality. We'll look at what packages are available in the Julia ecosystem and organize them into meaningful categories for a data scientist. Then, we'll look specifically at the packages related to data preparation (making the data ready for your models), data models, Statistics, Machine Learning utilities, AI, and some other packages that don't fit into any one of these categories (e.g. plotting packages).

A library of libraries

Fortunately, there are plenty of packages available in the Julia ecosystem for data science tasks. Contrary to what the critics of the languages say, nowadays, there is a package for everything you'd need in your data science work, even the most advanced models, as can be seen in Fig. 3.1. If you are comfortable with using mind-maps, you can use this diagram as a starting point to help you organize the packages you learn in a way that makes sense and is not too difficult to remember. Note that not all of these packages would be necessary for your data science work. Sometimes even a handful of packages suffice for a given project, while other times, it's easier to code something from scratch and use your scripts for your work. Whatever the case, knowing that these packages exist can be a useful aid, particularly when you are new to programming or the Julia language.

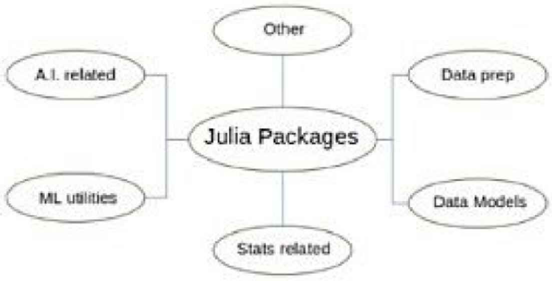


Figure 3.1. Main types of libraries in Julia languages, related to data science work.

For additional packages, the best place to start would be the official package repository of Julia: https://pkg.julialang.org/docs. Although not all packages are covered, the repository includes the ones that have been tried and tested enough for the Julia Computing company to give its stamp of approval. Also, these are the packages that the Julia community uses and provides feedback, so they are the most likely ones to be in good shape.

Data preparation libraries

Let's start our exploration of the Julia packages with the ones geared toward data preparation. Data preparation is the part of the data science pipeline that involves getting the data in a neat and usable state so that it can be fed into models, be it for exploring or for making predictions. The key data preparation packages in Julia are:

- **DataFrames** – the Julia equivalent of Python's Pandas package, and one that plays an instrumental role in data science tasks as it contains the Data Frame structure. Although not always essential for a data science project, it is very useful, and you can't do much without it if you plan to work with most of the data science packages since they have this package as a dependency.
- **ExcelReaders** – a package for accessing .xlsx documents and storing their contents into a data frame. This package does not create .xlsx files, however.

- **CategoricalArrays** – a package for handling categorical data, be it nominal or ordinal. The data may contain missing values too.
- **CSV** – a package for handling CSV files easily. Although you could create all the functions of this package yourself without being adept at programming, it's still a convenience when dealing with tabular data files.

Note that there are more packages that qualify as members of the data preparation family, but these are sufficient for most tasks. Besides, for more niche applications, you can find whatever package you need in the Julia package ecosystem. Also, all of the previous packages are available on GitHub, too, so you can access them using a GitHub account. However, it's easier to use them through Julia, either through the REPL or one of its IDEs.

Libraries for data models

Let's now look at some libraries geared toward data models. We won't go into much depth in them as they are going to be examined in some detail at a later chapter, along with examples. Depending on the methodology, they are most relevant to these packages:

- **Unsupervised Learning** – not making use of a target variable, such as for exploratory data analysis (EDA).
- **Supervised Learning** – making use of a target variable, be it for classification or regression problems.
- **Transparent models** – models that are easy to interpret and understand their functionality.
- **Black box models** – models that are difficult or impossible to interpret (figure out how they arrive at their conclusions exactly).

Based on this simple taxonomy, we can organize the data model packages as:

- Unsupervised Learning
    - **Clustering** – a package specializing in clustering methods for grouping data into meaningful groups,

    usually part of Exploratory Data Analysis.
    - **ManifoldLearning** – a package focusing on non-linear methods for reducing the dimensionality of a dataset.
- Supervised Learning (transparent model)
    - **DecisionTree** – decision trees and random forests package.
    - **GLM** – generalized linear model package for regression analysis.
- Supervised Learning (black box models)
    - **XGBoost** – a gradient boosting framework for ensembles using boosting, particularly comprising of decision trees.
    - **LIBSVM** – a package for Support Vector Machines (SVMs) in Julia.

Beyond these packages, there is another one that, although clearly under the supervised learning umbrella, it is difficult to classify using this taxonomy. The reason is that it contains models that span across both categories: the MLJ package and its sibling package, MLJModels. For reasons that go beyond the scope of this book, these two packages are often used in tandem.

MLJ and MLJModels are two packages used for various supervised learning models, spanning from simpler ones such as kNN, to more complex models such as Ridge Regressors, Learning Networks, and Ensembles. Although MLJModels is an extension of MLJ, MLJ is a unique data modeling package in Julia as it attempts to unify all existing models and methods within a single framework. However, as it's relatively new, it has some wrinkles that need to be ironed out before it can be considered production-ready. You can check it out at https://bit.ly/34swGZZ.

Note that it's best to have a decent understanding of the models behind the methods in these packages since the examples in the corresponding GitHub pages are lean and simple, perhaps too simple for someone unfamiliar with all this. However, this is not a criticism for these web pages or the people maintaining them, but rather an observation regarding the oversimplification of the data science field and the dangers this entails. We'll look into this in more detail in the next chapter.

Statistics libraries

Statistics packages are some of the most mature packages in Julia and the most widely used in various data analytics projects. The most important and reliable such packages are:

- **StatsBase** – a collection of multiple Statistics functions, essential in most data science projects. Probably one of the oldest packages in the Julia repository.
- **Distributions** – a package for all kinds of probability distributions, along with useful functions for performing various tasks with them, such as sampling, maximum likelihood estimation, and the *pdf* of each (as well as its logpdf). See the package's official documentation for more information: https://bit.ly/3b4vz58.
- **HypothesisTests** – a package containing several statistical methods for hypothesis testing, such as t-tests and chi-square tests.
- **MultivariateStats** – a great place to obtain various useful Statistics functions, including *principal components analysis* (PCA).

Probably the best place to start would be with the distributions package, as it's often used as a dependency in other data science packages. Whatever the case, each one of the Statistics packages here is worth learning, particularly if you have a sufficient understanding of Stats. Note that the GLM package would also fit in this category, but it seems more relevant as a data model package. After all, this taxonomy is used to facilitate the organization of the packages into a meaningful and memorable manner—to optimize their usefulness in data science projects.

Libraries related to ML utilities

Beyond Stats and model-related packages, there are some packages geared toward helping out with Machine Learning tasks. These are:

- **Distances** – a quite useful package for distance calculation, covering all major distance metrics—particularly essential for transductive systems, be it clustering algorithms or distance-based classifiers and regressors.

- **MLJ** – beyond Machine Learning models, this package includes a variety of utilities, too, such as grid search and K-fold cross-validation. Even if you don't use the MLJModels package, MLJ is useful in and of itself for data science projects.
- **ScikitLearn** – the well-known master package for all kinds of Machine Learning applications in Python, made available through the PyCall bridge package.
- **MLLabelUtils** – a quite useful package for pre- and post-processing of labels data for classification problems. Labels encoding may seem like a trivial task, but it can be time-consuming and prone to errors, something this package aims to help manage efficiently.
- **MLBase** – this is the most established package in Julia, related to Machine Learning, covering a wide range of tasks. You can obtain more information about it at https://bit.ly/2wyJmln.
- **AutoMLPipeline (AMLP)** – a creative approach to building machine learning pipeline structures, for more complex tasks. This tool can bring a great deal of value to a data science project. More information about it at https://bit.ly/2zLM6gt.

Although MLJ is the most promising, it's still not there yet, so we will focus on the other ML packages in this book. This package may someday be the Julia equivalent of Scikit-learn (even if the latter exists in Julia too as an API for the Python package). Still, until then, it's recommended to rely on the existing packages for any data science work.

AI-related libraries

AI-related packages aim to bring forward the more advanced models that are employed in data science, via Machine Learning methodologies. There are several, but the most important of them are:

- **Knet** – this deep learning framework is the first one that's been developed entirely in Julia (even the cost function methods are coded in the language). Created by Professor Deniz Yuret and his team, it covers a variety of ANNs, including MLPs, CNNs, and some models specializing in sentiment analysis.

- **Flux** – although this package is marketed for Machine Learning, it is one of the most well-made deep learning packages, developed using Julia exclusively. It also has GPU and AD support and is optimized for performance. Also, it is fully compatible with the Metalhead package. You can learn more about Flux at https://bit.ly/2RysQZX.
- **Metalhead** – one of the most creatively named packages in the Julia ecosystem, Metalhead is all about Computer Vision. So, if you are looking to perform classification on images using AI, this is the best place to start. More info at https://bit.ly/2Vg3IYQ.

Other AI-related packages in Julia corresponding to well-known frameworks such as MXNet are used with languages like Python and R. However, as this is not a book on these frameworks, we'll focus on the conventional Machine Learning packages instead. Besides, these are more promising and worth looking into, no matter how popular the more advanced AI-related libraries are these days.

Other libraries

Beyond the previous packages, there are a few more that are equally useful, if not more useful. Namely, this generic group of packages includes those related to specialized processes like graph analytics and of course plotting (data visualization). Also, there is a package that ensures that all packages in Julia work in harmony and are up-to-date. These packages are:

- **TSne** – a package implementing the T-SNE algorithm by the creators of the algorithm. Its creator and his team cover it at https://bit.ly/2ycKX0w. Note the stochastic nature of this algorithm and its scope, since this is not meant for everyday dimensionality reduction, like PCA and ICA, for example.
- **UMAP** – a package related to the Uniform Manifold Approximation and Projection algorithm, a powerful

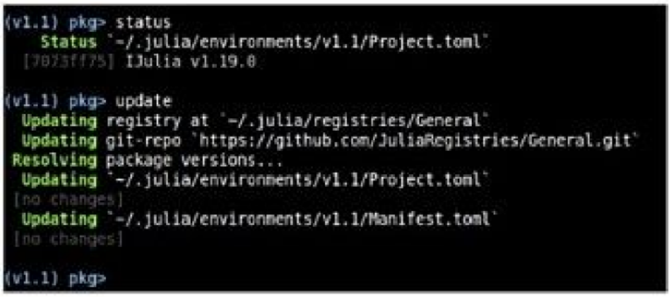
dimensionality reduction method. You can learn more about it at https://bit.ly/3ccLLkV. Just like T-SNE, UMAP is stochastic, but unlike T-SNE, it's much more useful in dimensionality reduction that goes beyond visualization. UMAP is often used in conjunction with Clustering methods.

- **Graphs** – the most complete graph analytics package. Note that these are mathematical graphs, and although there is a visualization aspect to this package, it is not relevant to plotting data in general, just graph-related data.
- **Gadfly** – one of the best plotting packages, written entirely in Julia. Note that the inputs of the corresponding methods require the use of the DataFrames package, mentioned in a previous section of this chapter.
- **PyPlot** – a great plotting package, borrowed from Python's Matplotlib, ideal for heat maps, among other plot types. If you are new to Julia and already familiar with Python, this is a good place to start in your visualization endeavors.
- **Pkg** – a powerful package for creating, installing, updating, and uninstalling packages in your Julia kernel. Learn more about it here: https://bit.ly/3efliF4. The most common usage of it is:

```
Pkg.add(CoolPackage)

Pkg.update()

Pkg.rm(CoolPackage)

Pkg.status()
```

Although the above format is valid, it is rarely used any more since this package is so common that a particular shortcut is employed instead. Namely, you can press the "]" key while you are on the REPL and gain access to it directly, so that you can run the above commands in a simpler format, as shown in Fig. 3.2. You can exit this mode by pressing the "backspace" key while you are on the Pkg prompt. Note that you need to import this package in a Jupyter notebook if you plan to add, remove, or update a package while in that environment. After all, the "]" shortcut only works on the REPL.

The upgraded usage of this package is one of the most important improvements of Julia since version 1.0. You should familiarize yourself with this package before learning any other package in the language, especially if you plan to use packages extensively for your work. Browse the many more packages available on the Julia package website.



```
(v1.1) pkg> status
    Status `~/.julia/environments/v1.1/Project.toml`
  [7073ff75] IJulia v1.19.0

(v1.1) pkg> update
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `~/.julia/environments/v1.1/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.1/Manifest.toml`
 [no changes]

(v1.1) pkg>
```

Figure 3.2. The Pkg package in action. You can enter this mode of usage by pressing the "]" key, while to exit it, you just need to press "backspace" while on the Pkg prompt. When actually updating packages, there is more text on the screen, depicting the progress of the update for each of the packages being updated.

Useful considerations

Although the package ecosystem of the Julia language is quite straightforward, some things are important to keep in mind to make the most of it and avoid any potential issues. First of all, Julia packages are always work in progress, and they rely on the feedback of users like you to remain relevant and useful as tools in data science work.

What's more, not all packages you come across in Julia are going to be fully compatible with each other. Take LightGraphs, for example, a package that aims to facilitate Graph Analytics tasks swiftly and intuitively, explained in my previous book on Julia. As great as this package may be, it is not compatible at all with the Graphs package, which is another, more well-established library of Graph Analytics methods.

In addition, packages are the product of some people who took time out of their days to create them, so they have their limitations, and may not be bug-free. They are not always the product of a team with sufficient resources to develop them in the professional manner you may be used to if you are coming to Julia from other more established programming languages. Take `ELM.jl`, for example, the package that put Extreme Learning Machines on the map for those data scientists who are unfamiliar with this kind of network-based system. The ELM package doesn't cover the various types of ELMs, which are what make ELMs such an intriguing technology.

Furthermore, Julia packages are a great facilitator of data analytics tasks, but they are no substitute for proper planning and meticulous application of the data science mindset. They may enable you to do interesting things and come off as someone who knows data science, but unless you have an understanding of the field that goes beyond methods and techniques, you are bound to remain at the surface and not fulfill your potential.

Finally, even if you are not an adept programmer or a Julia connoisseur, you can still take a stab at developing your tools using this language. They may not be optimal, and they may not be promoted as much by the more experienced Julians. Still, if they help you improve your efficiency at your data science work, they may be valuable regardless. If you are brave enough and have no copyright restrictions, you can even share your work with the Julia community and invite others to help you with these scripts, gradually evolving them into new packages which benefit everyone.

Summary

- Packages in Julia are sufficient in number and diversity, for facilitating various data science-related tasks, particularly those geared toward Machine Learning.
- There are different groupings of data science-related packages in Julia, such as the one proposed in this chapter: Data preparation, Data models, Statistics, ML utilities, AI-related, and Other. Whether this taxonomy is comprehensive enough or not, it can still be useful for organizing the various data science related packages in the Julia ecosystem and finding them faster when in need of them.

- Data preparation packages include DataFrames, ExcelReaders, CategoricalArrays, and CSV.
- Data model packages include Clustering, DecisionTree, GLM, XGBoost, LIBSVM, and MLJ/MLJModels. These cover a variety of models, ranging from the unsupervised learning ones to the supervised learning ones. Some of these correspond to transparent models (GLM and DecisionTree packages) while others to black box ones (ELM, XGBoost, and LIBSVM). The MLJ and MLJModels packages, which are usually used together, cover a broad spectrum of packages across different categories.
- Statistics packages include StatsBase, Distributions, HypothesisTests, and MultivariateStats. Of these four packages, the one that is most relevant and worth learning first is the Distributions one, though all of them are important. Some understanding of Statistics would be crucial for making the most of these packages, however, and for understanding the results of the methods they contain.
- ML Utilities packages include Distances, MLJ, ScikitLearn, MLLabelUtils, and MLBase. Of these, MLJ is probably the newest one and attempts to provide a unified framework for all the Machine Learning models available in Julia. ScikitLearn is more like a proxy for the well-known scikit-learn package in Python, while the MLLabelUtils package is useful for processing the labels data for classification-related projects.
- AI-related packages include Knet, Flux, and Metalhead. Of these, Knet is a deep learning framework comparable to the conventional ones used with Python, while Metalhead is specialized in computer vision applications.
- Other packages include T-Sne, Graphs, Gadfly, PyPlot, and Pkg. The most important of these are Gadfly (best plotting package in Julia) and Pkg (responsible for adding, updating, removing, and creating packages in the Julia installation you are using).

Questions

1. What's the point of using pre-made packages in Julia if we can