Universidad de Valladolid

FACULTAD DE CIENCIAS

TRABAJO FIN DE GRADO
GRADO EN ESTADÍSTICA

Exploración del Lenguaje Julia y Aplicación a Modelos Lineales Generalizados frente a R

Autora: Elisa Merino González

Tutora: María Teresa González Arteaga

Año: 2024 - 2025

Resumen

El presente Trabajo Fin de Grado “Exploración del Lenguaje Julia y Aplicación a Modelos Lineales Generalizados frente a R” busca analizar el lenguaje de programación Julia en profundidad. Se trata de una herramienta relativamente joven que se publicita como sencilla e intuitiva y con grandes resultados en términos de eficiencia. Por ello, este estudio pretende comprobar si estas afirmaciones son ciertas, ya que supondría que Julia podría consolidarse como una gran alternativa al resto de lenguajes en el ámbito de la Estadística y el Análisis de Datos.

Se realiza una revisión de su origen y evolución, así como de sus principales características técnicas y objetivos. Una vez conocido el contexto de la herramienta, se desarrolla una guía introductoria a la programación en Julia, donde se exponen herramientas de apoyo como tutoriales y guías en línea, y se abarca desde lo más básico, como su instalación, hasta el manejo de archivos y la realización de visualizaciones. Comprobando que en efecto se trata de una herramienta intuitiva y con una curva de aprendizaje reducida para usuarios de otras herramientas, como R o Python.

El análisis continúa evaluando su rendimiento y usabilidad frente al lenguaje R, ampliamente consolidado en el ámbito de la Estadística. Con esa finalidad, tras una introducción a los Modelos Lineales Generalizados, se lleva a cabo una implementación práctica de los GLM en ambos lenguajes, comparando tiempos de ejecución, sintaxis y facilidad de uso.

Se concluye que, efectivamente, el lenguaje de Julia demuestra un rendimiento excepcional frente a R. Por lo que, a pesar de ser un lenguaje relativamente joven, si en los próximos años continúa creciendo y mejorando, podría resultar una gran alternativa, sobre todo a la hora de manejar grandes archivos de datos.

Abstract

This Final Degree Project "Exploration of the Julia Language and Its Application to Generalized Linear Models Compared to R" aims to conduct an in-depth analysis of the Julia programming language. Julia is a relatively young tool promoted as simple, intuitive, and highly efficient. Therefore, this study seeks to verify the accuracy of these claims, as they could position Julia as a strong alternative to other programming languages within the fields of Statistics and Data Analysis.

The work includes a review of Julia's origins and evolution, as well as its main technical features and objectives. Once the contextual background is established, the thesis offers an introductory guide to programming in Julia, presenting supporting resources such as tutorials and online guides. This guide covers the basics—from installation to file handling and data visualization—demonstrating that Julia is indeed intuitive and has a gentle learning curve for users familiar with tools like R or Python.

The analysis then evaluates Julia's performance and usability in comparison to R, a language that is firmly established in the statistical community. To that end, following an introduction to Generalized Linear Models, a practical implementation of GLMs is carried out in both languages, comparing execution times, syntax, and ease of use.

The results confirm that Julia delivers exceptional performance compared to R. Consequently, despite being a relatively new language, if Julia continues to evolve and improve in the coming years, it may become a compelling alternative—particularly for handling large data sets.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En la actualidad, con el desarrollo de las ciencias en general y de la Estadística en particular, es inconcebible llevar a cabo ningún estudio científico sin la ayuda de las aplicaciones informáticas. Son muchos los programas que permiten al profesional de la Estadística avanzar en las distintas investigaciones; entre ellos tenemos: MATLAB, Octave, R, SciPy, SciLab, C y Fortran.

En general, todos se presentan como herramientas útiles para facilitar al investigador su trabajo. A la hora de elegir la alternativa más adecuada para una investigación concreta, el profesional tiene que tener en cuenta dos aspectos que no siempre van de la mano:

- La facilidad de uso del programa que viene determinada por el grado de sencillez del lenguaje que este utiliza.
- La potencia y eficacia de la aplicación.

Lo ideal para el investigador sería utilizar un programa cómodo y fácil de usar que presente resultados fiables de forma rápida. Sin embargo, entre las distintas alternativas disponibles para el investigador, aparecen programas muy complicados de utilizar en los que el proceso de introducción de datos y programación es muy laborioso, pero que el rendimiento computacional es muy potente (C y Fortran), junto con programas más orientados al análisis estadístico y la manipulación de datos de uso muy cómodo, aunque menos optimizados en términos de eficiencia (MATLAB, Octave, R, SciPy, SciLab).

Es en este contexto en el que aparece "Julia", un programa bastante fácil de utilizar, con un lenguaje sencillo e intuitivo y que produce buenos resultados. Este programa de licencia libre, relativamente nuevo, está siendo cada vez más utilizado por los investigadores.

1.1. Objetivos

En el presente trabajo se ha planteado como objetivo realizar un estudio detallado del programa Julia y sus características principales, comparándolo con R —una de las herramientas más consolidadas y robustas en análisis estadístico— con el fin de evaluar si Julia podría suponer una mejora y, eventualmente, una alternativa viable a R.

Para ello, la metodología que se ha usado ha consistido en realizar un análisis bibliográfico sobre el programa Julia y su relación con otros programas, así como de distintos textos relativos al programa R y a los Modelos Lineales Generalizados. Todos los textos utilizados aparecen detallados en la bibliografía del TFG y debidamente referenciados a lo largo del trabajo. Posteriormente a este análisis bibliográfico, se ha procedido a la instalación y uso del programa, elaborando una pequeña guía con información útil para iniciarse en este lenguaje, que incluye comandos básicos para comenzar a trabajar, así como enlaces a documentación oficial y herramientas complementarias que pueden resultar de interés.

Una vez adquiridos los conocimientos básicos de Julia, se ha procedido a la comparación de la herramienta con R. Este análisis ha consistido en estudiar las similitudes y diferencias que existen al ajustar Modelos Lineales Generalizados, tanto en metodología como en resultados, así como en rendimiento.

1.2. Estructura de la memoria

La estructura del presente trabajo fin de grado es la siguiente:

- En primer lugar, a continuación de esta introducción, se explora el origen de Julia, sus características técnicas y su evolución temporal, todo ello en el capítulo 2.
- Se sigue en el capítulo 3 con una pequeña guía de programación de Julia, que abarca desde el proceso de instalación hasta procedimientos de estadística básicos.
- En el Capítulo 4 se habla de los Modelos Lineales Generalizados y de su aplicación con ambos programas.
- El Capítulo 5 se dedica al análisis práctico del proceso de ajuste de los Modelos Lineales Generalizados, comparando las salidas generadas, los resultados estimados y el rendimiento en R y en Julia.
- Y, por último, en el Capítulo 6 se exponen las conclusiones extraídas y futuras líneas de investigación.

Capítulo 2

Lenguaje de programación Julia: Origen y características

Julia es un lenguaje de programación creado por Jeff Bezanson, Stefan Karpinski, Viral B. Shah, y Alan Edelman. En febrero de 2012, anunciaron en su página oficial que llevaban dos años y medio desarrollándolo. [2]

Los desarrolladores buscaban unificar en un mismo lenguaje todas las cualidades de los lenguajes que utilizaban, permitiéndoles resolver los problemas que se les presentaban en sus proyectos, sin la necesidad de ir cambiando entre ellos. Y, de esta manera, poder centrarse únicamente en la resolución de problemas complejos, eliminando la complejidad añadida de trabajar entre varios lenguajes. De esta necesidad surgió la idea de crear un lenguaje que fuera eficiente, versátil y fácil de usar: Julia.

En la figura 2.1 se muestra el logo de Julia. Se trata de un logo simple, que busca reflejar la filosofía del lenguaje, transmitiendo eficiencia y accesibilidad:

Figura 2.1: Logo del lenguaje de programación Julia. Fuente: [27]

El inconveniente al que se enfrenta Julia, es que se trata de un lenguaje joven. Su primera versión se lanzo en 2018, por lo que solo tiene seis años de desarrollo, y en comparación con lenguajes como R, aún no presenta la misma cantidad de recursos. Aún así, desde su lanzamiento el lenguaje ha ido creciendo y mejorando, demostrando que tiene un gran potencial.

A continuación, se presenta el origen de la herramienta y los objetivos que impulsaron su creación. Posteriormente, se muestran algunas de las cualidades técnicas presentes en la primera versión lanzada al público, así, como algunas de las mejoras que se han incorporado en la última versión. Por último, se expone una linea temporal con los momentos más destacables en el crecimiento de Julia, y se profundiza en algunas de las características tecnológicas más destacables de este lenguaje.

2.1. Origen

Los desarrolladores de Julia tenían unos objetivos extremadamente ambiciosos. Tras años de experiencia utilizando prácticamente todos los lenguajes de programación científica y matemática que existían hasta el momento (entre los que se encontraban Lisp, Python, C, R, Mathematica, Ruby, Matlab y Perl) llegaron a la conclusión de que cada uno tenía sus ventajas y sus inconvenientes. Aunque, cada uno de los lenguajes resultaba fantástico para ciertas áreas, también presentaba limitaciones significativas en otras, llegando incluso a resultar inútil en situaciones puntuales. Esta división en la funcionalidad de los lenguajes suponía que, a la hora de abordar proyectos más complicados, los desarrolladores se veían obligados a tener que combinar el uso de distintos lenguajes, lo que añadía más complejidad a los proyectos.

Decidieron que querían crear un lenguaje que funcionara de manera óptima en todos los campos en los que trabajaban. Entre esos campos se incluyen: la computación científica, el aprendizaje automático, la minería de datos, el álgebra lineal a gran escala, y la computación distribuida y paralela. [2]

En las palabras de los creadores:

> "*Queremos un lenguaje que sea de código abierto, con licencia liberal. Queremos la velocidad de C con el dinamismo de Ruby. Queremos un lenguaje que sea homoicónico, con macros verdaderos tipo Lisp, pero con notación matemática, obvia y familiar como Matlab. Queremos algo usable para programación general como Python, tan fácil para estadística como R, tan natural para procesamiento de cadenas como Perl, tan potente para álgebra lineal como Matlab, tan bueno uniendo programas juntos como un shell. Algo que sea sumamente sencillo de aprender, pero que mantenga a los hackers más serios felices. Queremos que sea interactivo y que sea compilado.*" [2]

Donde expresaban que querían crear un lenguaje con lo mejor de todos los mundos, es decir, capaz de hacer todo lo que sus predecesores eran capaces. Pero, que además, debía ser sencillo de aprender y utilizar, de código abierto, con licencia libre, interactivo y compilado. A parte, también querían que fuese capaz de realizar todas la tareas de forma rápida y eficiente. Que fuese una herramienta muy potente pero sin la necesidad de escribir bloques de código excesivamente grandes.

Versión 1.0

Después de casi una década de desarrollo, en agosto de 2018 se anunció oficialmente la liberación de Julia 1.0 como lenguaje de código abierto. Los desarrolladores lograron la mayoría de sus objetivos, creando un lenguaje rápido, versátil y eficiente para la computación matemática. Con una base estable, consideraron que el lenguaje estaba "completo", y a partir de esta versión, las mejoras se enfocarían en nuevos paquetes y herramientas.[13]

Algunas cualidades tecnológicas en las que trabajaron y mejoras que incluyeron para el lanzamiento de Julia 1.0, con respecto a versiones anteriores (la versión 0.6 más concretamente) [13]:

- La introducción de un administrador de paquetes, mejorando el rendimiento al encargarse de gestionar todo lo relacionado con estos.
- Integración de una nueva representación canónica para valores ausentes (*missing*). Propone una nueva solución general, flexible y rápida.
- En esta versión el tipo String permite contener datos arbitrarios, es decir, de este modo el programa es más tolerante y no falla por la presencia de algún carácter inválido.
- Compromiso de estabilidad de API, es decir, el código escrito en Julia 1.0 funciona para versiones posteriores, en 1.1, 1.2, ...
- *"Broadcasting"*: Mejora en la metodología de difusión, aumentando el rendimiento en los cálculos.

2.2. Versión actual

Desde el lanzamiento de la versión 1.0.0 los desarrolladores han continuado trabajando en mejorar la herramienta de forma constante. A lo largo de los años, han publicado nuevas versiones con mejoras específicas. Actualmente, la versión más reciente es Julia 1.11.5, lanzada en abril de 2025.

Algunos de los cambios técnicos e incorporaciones de la versión 1.11 son los siguientes [5]:

- Reimplementación del objeto *Array* en código Julia, reemplazando su anterior implementación completamente en C, con el nuevo tipo *Memory*. Trabajar con un este objeto permite mejorar el rendimiento y reducir el uso de memoria.
- Mejoras en las sugerencias y el autocompletado de código del entorno REPL
- Un nuevo tipo de datos *ScopedValues*, el cual proporciona una alternativa al uso de variables globales para manejar parámetros de configuración.
- Actualización de la macro *@time*, que ahora informa sobre conflictos de bloqueo dentro de la ejecución de la llamada que esta siendo medida.

2.3. Linea temporal

En la figura 2.2 está la cronología de la evolución de Julia, representada en una línea temporal donde se destacan los momentos de crecimiento del lenguaje.

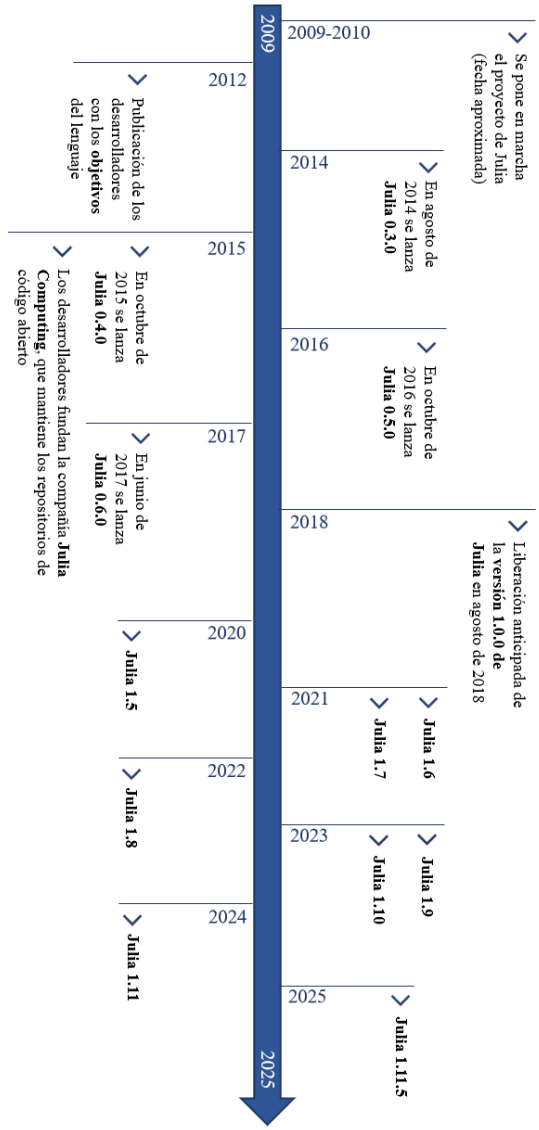


Figura 2.2: Cronología de la evolución de Julia. Fuente: Elaboración propia

2.4. Características

En los apartados anteriores se han indicado varias cualidades y características del lenguaje Julia y cómo se han ido incorporando a lo largo del tiempo. A continuación, se profundiza en algunas de sus propiedades más destacables.

2.4.1. Compilador JIT

Una de las principales características a señalar de Julia es que utiliza un sistema de compilación *Just-In-Time* (JIT) basado en LLVM (anteriormente conocido como *Low Level Virtual Machine, o Máquina Virtual de Bajo Nivel*). Los compiladores JIT se utilizan en lenguajes de programación y entornos de ejecución (como máquinas virtuales), y destacan por compilar el código fuente, o *bytecode*, a código de máquina nativo en tiempo de ejecución. [3, 12]

Julia lleva a cabo la compilación JIT utilizando LLVM, cuyo logo se puede ver en la figura 2.3. Actualmente, su nombre ya no es acrónimo de *Low Level Virtual Machine* sino que se trata del nombre completo del proyecto, ya que, según ha ido evolucionando, LLVM ya no guarda ninguna relación con las máquinas virtuales tradicionales. LLVM es una infraestructura de compilación compuesta por una colección de herramientas, tecnologías y módulos reutilizables capaces de soportar tanto la compilación estática como la dinámica. [22] El uso de esta técnica de

Figura 2.3: Logo de LLVM. Fuente: [21]

compilación permite a Julia optimizar la ejecución de los programas, reduciendo el consumo de memoria y mejorando el rendimiento en comparación con la compilación estática, donde la traducción del código ocurre antes de la ejecución. Además, le proporciona las ventajas de un lenguaje dinámico manteniendo el rendimiento de un lenguaje estáticamente compilado, llegando a alcanzar velocidades comparables con lenguajes tan potentes como C o Fortran. Por ejemplo, el lenguaje R también utiliza un sistema de compilación JIT (basado en GNU). [23, 18]

2.4.2. Sistema de tipado dinámico y de inferencia de tipos

Otro aspecto característico de Julia es que utiliza un sistema de tipado dinámico. Esto significa que las variables pueden ir cambiando de tipo de dato. Puedes empezar asignando a una variable un número, y posteriormente un texto o un booleano, y los valores se irán guardando en la variable sin que se dé ningún error. Podemos ver un ejemplo en la figura 2.4 donde le damos el valor *4* a la variable *v* y justo después se le asigna la cadena *"hola"*. Al comprobar de que tipo es la variable tras cada asignación con la función *typeof()*, se ve como cambia de entero a *String* sin que se dé ninguna excepción. Pyhton, JavaScript, Ruby o R son ejemplos de lenguajes que también utilizan el tipado dinámico. [8]

```
julia> v = 4
4

julia> typeof(v)
Int64

julia> v = "hola"
"hola"

julia> typeof(v)
String
```

Figura 2.4: Ejemplificación tipado dinámico en Julia

Además, el procesador de Julia cuenta con un avanzado sistema de inferencia de tipos. Lo que le permite la posibilidad de no declarar el tipo de una variable de forma explícita y que sea este sistema el que lo deduzca en tiempo de ejecución. De esta manera, escribir código en este lenguaje se vuelve más sencillo y la ejecución es más rápida, por lo que contribuye a una mejor eficiencia.

Tanto en la figura 2.4 como en la figura 2.5 se puede observar claramente cómo Julia infiere los tipos de las variables sin necesidad de declararlos de forma explícita. Sin embargo, si se desea que una variable sea de un tipo específico, se puede indicar utilizando la notación '*variable ::tipo-de-variable = valor*', como se muestra en la figura 2.5. De esta forma la variable solo podrá almacenar valores del tipo declarado. En resumen, Julia tiene un tipado dinámico por defecto con inferencia de tipos, pero también permite declararlos estáticamente cuando es necesario.

```
julia> z = 9
9

julia> typeof(z)
Int64

julia> x::Int = 3
3

julia> x = a
ERROR: UndefVarError: `a` not defined
Stacktrace:
 [1] top-level scope
   @ REPL[29]:1

julia> x = 9
9

julia> typeof(x)
Int64
```

Figura 2.5: Ejemplificación tipado opcional con inferencia de tipos en Julia

2.4.3. Multiple dispatch

Al programar, lo más habitual es dividir el código fuente en partes o bloques más pequeños, como subrutinas, funciones, o métodos, para facilitar el uso del mismo en diferentes implementaciones. Cuando se llama a una función, se debe elegir cuál de los varios métodos definidos se va a ejecutar; este proceso de selección es lo que se conoce como *dispatch*.

Julia utiliza como paradigma el patrón de diseño *multiple dispatch*, que podemos traducir como envío (despacho) múltiple. Consiste en elegir a qué método se va a llamar, teniendo en cuenta todos los argumentos pasados a la función, y sus tipos. Es un mecanismo que resulta más flexible que los utilizados en lenguajes de programación tradicionales, los cuales solo se fijan en el primer argumento para realizar esta elección. Considerando que en Julia se utiliza un sistema de tipado dinámico en el que los tipos de las variables pueden determinarse en tiempo de ejecución (como se explicó en el apartado anterior), el *multiple dispatch* permite que el comportamiento de una función sea distinto según los tipos de argumentos que reciba. Este paradigma resulta especialmente útil para estructurar y organizar programas, y destaca en código matemático. [28]

Además de Julia, otro lenguaje que también utiliza este paradigma es Common Lisp. Por otro lado, lenguajes como Python, C, JavaScript o R no soportan de base el envío múltiple, pero cuentan con bibliotecas y extensiones que permiten implementarlo. [6]

2.4.4. Otras características tecnológicas relevantes

Para concluir, se incluye a continuación otras características técnicas significativas de este lenguaje de programación [23, 11]:

- Licencia libre y de código abierto.
- Gestor de paquetes integrado.
- Macros tipo Lisp y otras herramientas para la meta programación.
- Interoperabilidad con otros lenguajes. Permite llamadas a funciones de otros lenguajes por medio de distintos paquetes: de Python por medio del paquete *PyCall*, de java con *JavaCall*, de Matlab con *MATLAB*, y de R con los paquetes *Rif* y *Rcall*
- Llamadas directas a funciones de C y Fortran, sin necesidad de usar entornos o APIs especiales.
- Línea de comandos altamente potente.
- Computación tanto paralela como distribuida de manera nativa.
- Capacidad de generar código de forma automática
- Soporte eficiente para Unicode, incluyendo UTF-8.

Capítulo 3

Introducción a la programación con Julia

En este capítulo se muestra una pequeña guía de programación en Julia, indicando tareas esenciales para poder trabajar con el lenguaje. Se abarca su instalación, las interfaces con las que se puede acceder, plataformas para localizar paquetes que puedan resultar útiles, manejo de archivos de datos y estadística básica. Cabe aclarar que este capítulo no pretende ser un manual, por lo que no se abarcan todas las posibles funcionalidades de Julia.

No obstante, si se pretende conocer más a fondo la herramienta, se puede acceder a la documentación de Julia a través del siguiente enlace:

- Documentación de Julia: https://docs.julialang.org/en/v1/

En esta dirección web los desarrolladores del proyecto "Julia" dejan a disposición de todos los usuarios el manual actualizado desde el lanzamiento de su última versión. La documentación se puede explorar a través de la página web de forma sencilla, por medio de un índice y una barra de búsqueda, o si se prefiere, también esta disponible en formato PDF. [29].

3.1. Instalación

En el canal oficial de YouTube del lenguaje de programación Julia, podemos encontrar distintos vídeos dedicados a la descarga e instalación del lenguaje. Uno de los vídeos más recientes y simples de seguir, es el siguiente [19]:

- Tutorial: https://www.youtube.com/watch?v=t67TGcf4SmM

El primer paso para descargar la herramienta es acceder a la página oficial de Julia, y tal como se muestra en la figura 3.1, haciendo click en el botón *Download* te lleva a la página de descarga.

Una vez en la página de descarga se muestran las distintas opciones que hay según el dispositivo en el que se esté trabajando y se selecciona la alternativa más adecuada. Como se observa en la figura 3.2, Julia es compatible con todos los sistemas operativos. En este proyecto se supone la utilización de un dispositivo Windows de 64-bit.

Figura 3.1: Descarga de Julia desde la página oficial. Fuente: [27]

Current stable release: v1.10.5 (August 27, 2024)

Release notes | GitHub tag | SHA256 checksums | MD5 checksums

Platform	64-bit	32-bit
Windows [help]	installer, portable	installer, portable
macOS x86 (Intel or Rosetta) [help]	.dmg, .tar.gz	
macOS (Apple Silicon) [help]	.dmg, .tar.gz	
Generic Linux on x86 [help]	glibc (GPG)	glibc (GPG)
Generic Linux on ARM [help]	AArch64 (GPG)	
Generic Linux on PowerPC [help]	little endian (GPG)	
Generic FreeBSD on x86 [help]	.tar.gz (GPG)	

Source Tarball (GPG) Tarball with dependencies (GPG) GitHub

Figura 3.2: Opciones de descarga de la última versión de Julia. Fuente: [26]

Se accede a la carpeta que se ha descargado, y se procede con la instalación. Al lanzar el programa se abre un terminal de comandos, o REPL (Read-Eval-Print-Loop, traducido “bucle de leer-evaluar-imprimir”), como se muestra en la figura 3.3.

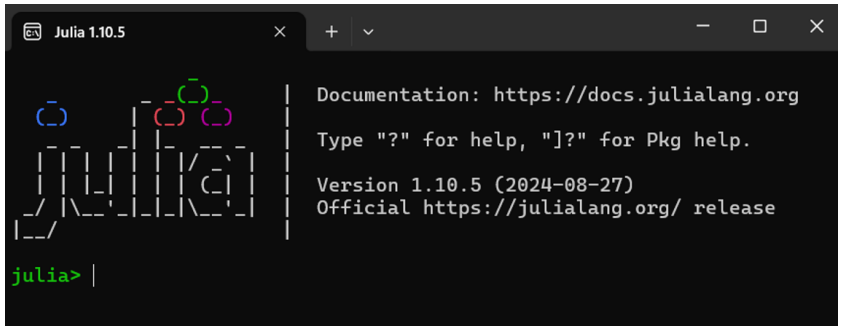


Figura 3.3: Inicio de la aplicación de Julia. Fuente: Elaboración propia

3.2. IDEs de Julia

3.2.1. Visual Studio Code

Una vez instalado el lenguaje Julia en el dispositivo, es recomendable utilizar un editor de código como **Visual Studio Code**.

Figura 3.4: Logo Visual Studio Code. Fuente: [30]

Este editor cuenta con extensiones de gran utilidad, entre ellas, dispone de una que proporciona soporte básico para *Jupyter Notebook*. Esta herramienta está diseñada para facilitar la escritura y lectura de código en todos los lenguajes compatibles con *Jupyter Notebook* incluido Julia.

Además, ofrece varias extensiones que dan soporte específico al lenguaje de Julia, siendo una de las más destacables: *Julia Lenguage Support*. Aporta funciones extremadamente útiles a la hora de programar, y que mejoran significativamente el flujo de trabajo, se pueden destacar las siguientes:

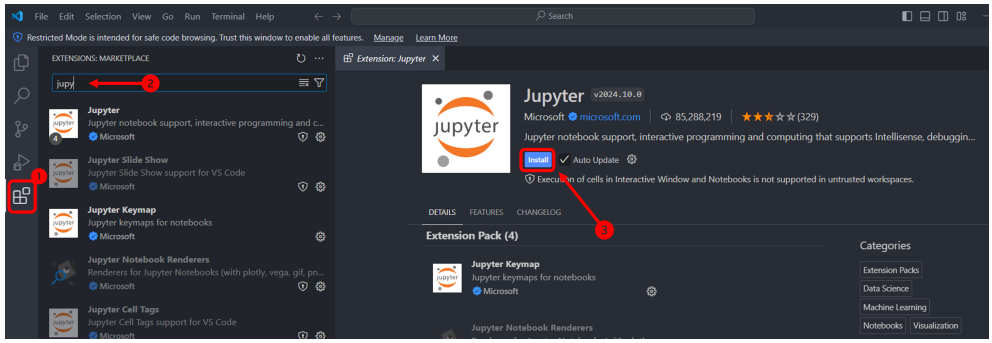
- Auto-completado de código
- Resaltado de sintaxis
- Ayuda contextual al pasar el cursor
- REPL de Julia integrado

Se puede acceder a la descarga e información de **Visual Studio Code** y sus extensiones a través de los siguientes enlaces:

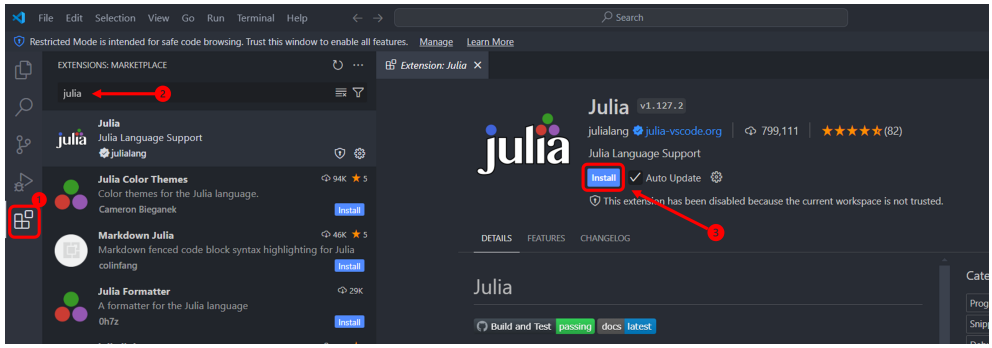
- Descarga: https://code.visualstudio.com/

- Extensión *Jupiter Notebook*: https://marketplace.visualstudio.com/items?itemName=ms-toolsai.jupyter
- Extensión *Julia Language Support*: https://marketplace.visualstudio.com/items?itemName=julialang.language-julia

Una vez instalado **Visual Studio Code**, la forma más cómoda de acceder y descargar sus extensiones es desde la propia aplicación de escritorio, como se muestra en la figura 3.5.



(a) Extensión Jupiter Notebook



(b) Extensión Julia Language Support

Figura 3.5: Instalación de extensiones en Visual Studio Code

3.2.2. IDEs en la nube

También es posible programar en Julia directamente desde la web. La plataforma **JuliaHub**, diseñada para dar soporte y ejecutar proyectos en el lenguaje Julia, entre otras cosas, ofrece desarrollo interactivo en la nube a través de dos aplicaciones: **Julia IDE** y **Pluto.jl**.[14]

Para acceder a estas herramientas, se debe iniciar sesión en **JuliaHub**. Como se muestra en la figura 3.6, en la sección 'aplicaciones' (*Applicstions*), se incluyen todas las posibilidades. La elección queda a preferencia del usuario, según sus intereses y necesidades.

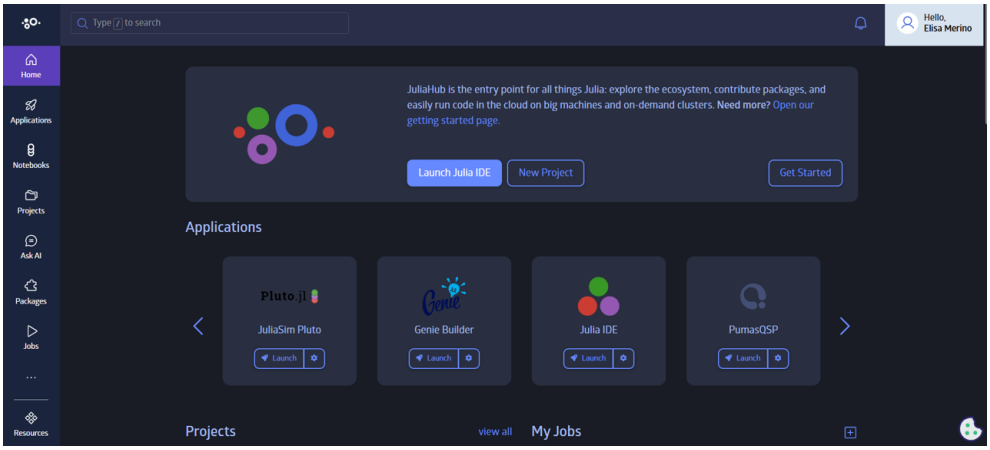


Figura 3.6: Página sesión iniciada en JuliaHub. Fuente: [14]

Se puede acceder a la plataforma **JuliaHub** a través del siguiente enlace:

- Enlace a JuliaHub: https://juliahub.com/

Julia IDE

Para acceder a **Julia IDE**, se hace click en el botón de lanzar Julia IDE (*Launch Julia IDE*) como se observa en la figura 3.7. Una vez iniciada la aplicación, la plataforma ofrece la opción de conectar con ella (*Connect to Julia IDE*). Al seleccionar el botón, la aplicación se abre en una nueva pestaña del navegador web y puede tardar unos minutos en conectar.

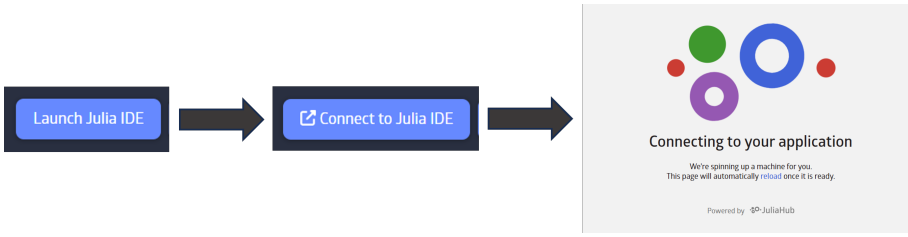


Figura 3.7: Proceso para acceder a Julia IDE. Fuente: [14]

En cuanto termina el proceso de conexión, como se incluye en la figura 3.8, se abre una interfaz web de **Visual Studio Code**, sobre la que se basa **Julia IDE**.

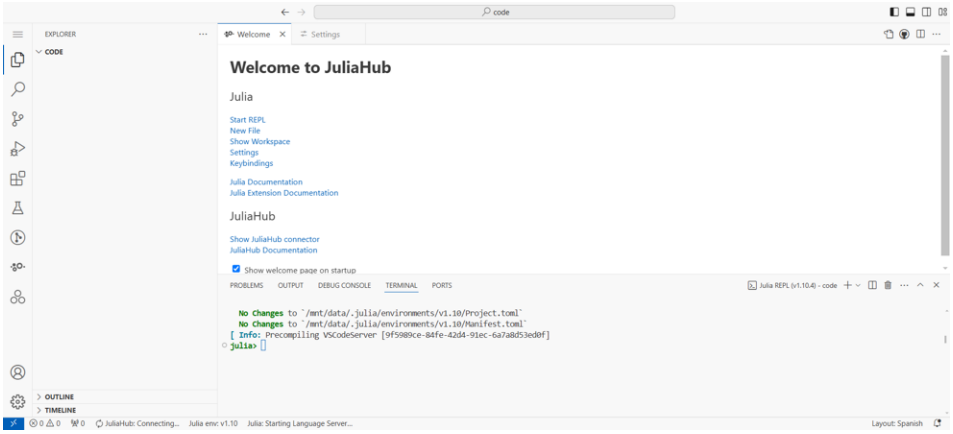


Figura 3.8: Julia IDE, sobre la interfaz web de Visual Studio Code. Fuente: [14]

Pluto Notebook

El proceso de conexión con los cuadernos en linea de **Pluto.jl** es el mismo que para **Julia IDE**, como se representa en la figura 3.9.

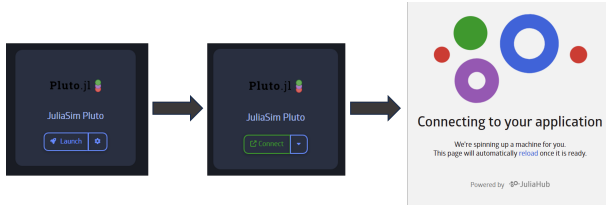


Figura 3.9: Proceso para acceder a Pluto.jl. Fuente: [14]

Se puede comprobar como es la interfaz de un cuaderno de **Pluto.jl** en linea en la figura 3.10.

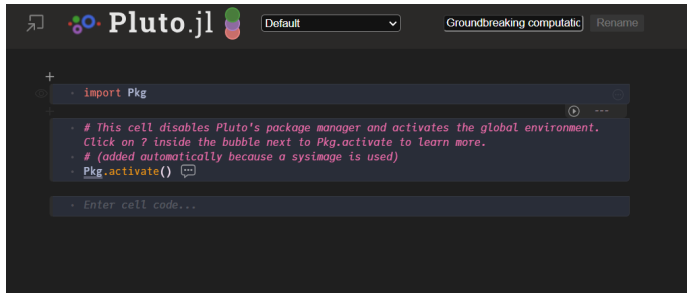


Figura 3.10: Cuaderno de Pluto.jl en la nube. Fuente: [14]

3.3. Paquetes

La biblioteca estándar de Julia cuenta con una gran cantidad de paquetes, lo que puede hacer complicada la tarea de encontrar el paquete adecuado para una necesidad específica. Existen distintos servicios en línea que facilitan ésta búsqueda, como **JuliaHub** y **Julia Packages**, que permiten la localización de paquetes. A continuación, se ve en detalle cómo utilizar estos buscadores.

3.3.1. JuliaHub

JuliaHub, además de dar acceso a la programación en la nube, presenta diversas funcionalidades y servicios de gran utilidad, como la búsqueda de la documentación de paquetes de Julia de código abierto.

En la figura 3.11, se observa que en la esquina superior derecha de la página de inicio de JuliaHub se incluye un buscador. En él se introducen etiquetas o palabras clave relacionadas con las funcionalidades que se buscan, y como resultado, se obtiene una lista con información de cada paquete que presenta alguna coincidencia.

Cabe destacar que este buscador no sirve únicamente para la búsqueda de paquetes. También te da la opción de obtener documentación del lenguaje de Julia, así como código, símbolos o cuadernos públicos, con programas que se han publicado en JuliaHub.

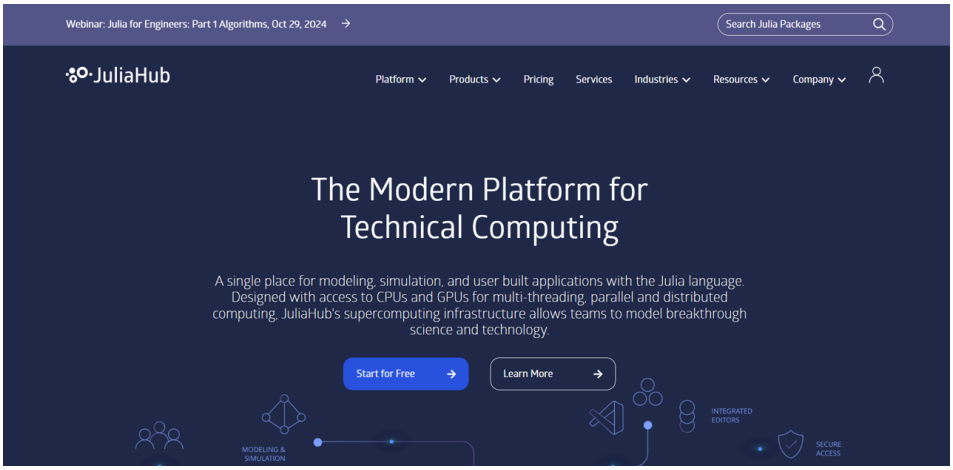


Figura 3.11: Página de inicio de JuliaHub. Fuente: [14]

Por ejemplo, si se quiere hacer un ajuste de modelos lineales generalizados pero no se conoce qué paquete incluye las funciones necesarias, como muestra la figura 3.12, se puede introducir el término 'glm' y seleccionar 'Paquetes'. En este caso, al buscar 'glm' se obtienen varios paquetes relacionados con distribuciones estadísticas, pero el que resulta más adecuado es el paquete ***GLM***. Desde los resultados de búsqueda, se puede comprobar cuántas veces se ha descargado o la versión del paquete, y al seleccionarlo, se obtiene información como la orden para instalarlo

(*Pkg.add("GLM")*), la fecha de salida de la versión del paquete, o enlaces directos a su documentación.

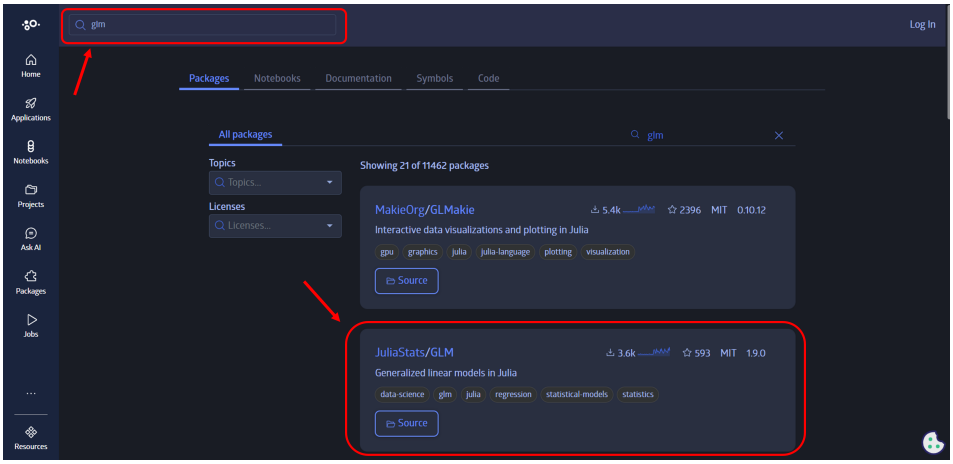


Figura 3.12: Resultado de búsqueda de paquetes por medio de palabras clave en JuliaHub. Fuente: [14]

- Enlace a JuliaHub: https://juliahub.com/

3.3.2. Julia Packages

Otro motor de búsqueda de paquetes es *JuliaPackages*. En esta página se puede encontrar información de todos los paquetes disponibles del lenguaje de Julia. Como se observa en la figura 3.13 sección (1), tiene la opción de realizar la búsqueda por medio de palabras clave. También se puede navegar a través de su índice y buscar por categorías.

Siguiendo el ejemplo del apartado anterior, para buscar el paquete adecuado de Julia que permita realizar ajustes de modelos lineales generalizados, se tiene la opción de buscar por la palabra clave 'glm', y al igual que en JuliaHub se obtiene una lista de paquetes que presentan coincidencias. Al seleccionar el paquete de interés, se presentan las características del paquete, como tiempo desde la última actualización, enlaces a la documentación o el autor del paquete.

Por otro lado, también se puede navegar por las distintas categorías. En este caso se puede encontrar el paquete idóneo buscando en la categoría *Programming & Stadistics*, como se ve en la sección (2) de la figura 3.13.

- Enlace a JuliaPackages: https://juliapackages.com/

Ambos buscadores de paquetes funcionan de forma efectiva y proporcionan información similar, por lo que la elección entre cuál de ellos utilizar se puede basar en preferencias personales del usuario.

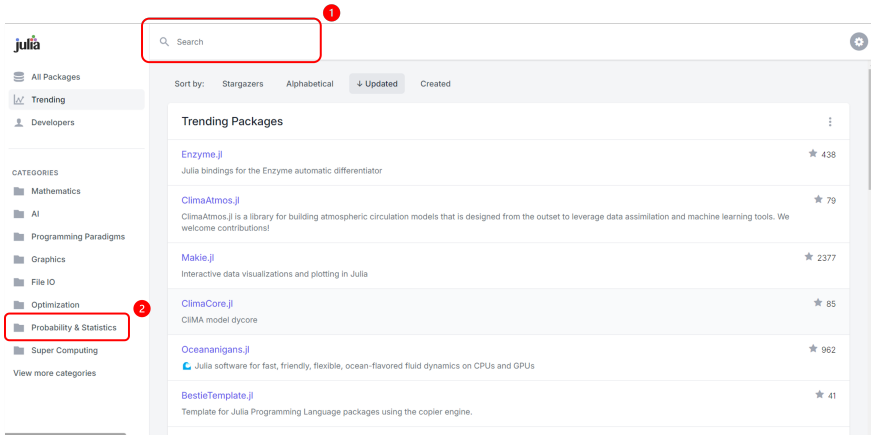


Figura 3.13: Inicio en la herramienta en linea de Julia Packages. 1. Búsqueda por palabras clave. 2. Búsqueda por categorías. Fuente: [24]

3.3.3. Descripción de paquetes relevantes

Algunos de los paquetes incluidos dentro de la biblioteca estándar de Julia son los siguientes[29]:

- ***DelimitedFiles***: permite el manejo de archivos de datos en formato tabla (lectura/escritura) de forma sencilla. Al estar incluido dentro de la biblioteca estándar de Julia, facilita su integración en programas simples sin necesidad de importar paquetes externos.
- ***Statistics***: incluye funciones de estadística básica, como *std()*, para calcular la desviación típica muestral, o *var()*, para calcular la varianza muestral.
- ***LinearAlgebra***: permite la realización de cálculos de álgebra lineal, y la ejecución de rutinas de forma rápida.

Otros paquetes que pueden resultar de gran interés en el ámbito de la Ciencia de Datos y de la Estadística son:

- ***DataFrames***: uno de los paquetes más utilizados para manejar datos en formato de tabla. Similar a *dplyr* en R.
- ***Distributions***: contiene funciones para trabajar con una gran colección de distribuciones de probabilidad. Es posible su integración con otros paquetes de Julia, como ***StatsPlots***.
- ***StatsPlots***: extensión del paquete de Julia ***Plots***, que se centra en la creación de gráficos estadísticos. Con la combinación de ***Distributions*** y ***DataFrames*** concede una gran sencillez y flexibilidad para la visualización de datos estadísticos.
- ***RDatasets***: da acceso a los conjuntos de datos que contiene *datasets*, la librería base de R , incluyendo *iris*, *cars* y *co2* entre otros. También dispone de la librería ***VegaDatasets***, la cual contiene conjuntos de datos como los disponibles en *Vega* y en *Vega-Lite* de Python.
- ***RCall***: permite llamar a funciones de R desde Julia, facilitando la interoperabilidad de ambos lenguajes.

3.3.4. Instalación de un paquete

El gestor de paquetes integrado de Julia, ***Pkg***, permite la instalación y manejo de paquetes de forma eficiente. En el siguiente fragmento de código [1], se incluyen algunas de sus funcionalidades más destacables.

[1]:
```
import Pkg

#Añadir paquete:
Pkg.add("paquete")

#Mostrar paquetes instalados:
Pkg.status()

#Borrar paquete:
Pkg.rm("paquete")

#Actualizar paquetes:
Pkg.update()
```

La librería base de Julia incluye una gran variedad de módulos, pero estos se mantienen desactivados por defecto con el fin de evitar sobrecarga de la sesión con funciones innecesarias y tipos de variables.

A continuación, se muestran dos formas de activar un paquete, en este caso, el paquete ***Statistics***:

[2]:
```
import Statistics

Statistics.mean(1:20)
```

```
10.5
```

[3]:
```
using Statistics

mean(1:20)
```

```
10.5
```

El uso de *import* [2], obliga a incluir el nombre del paquete como prefijo de las funciones, lo que puede resultar útil para evitar conflictos de nombres. Por otro lado, con *using* [3], las funciones y métodos del paquete a importar están disponibles directamente.

Por otro lado, la instalación de un paquete no incluido en la librería base de Julia, por ejemplo, el paquete ***DataFrames***, se realiza como se muestra en el fragmento de código [4]. En este caso, antes de importar un paquete, se debe instalar con la función *add* del gestor de paquetes.

[4]:
```
import Pkg

Pkg.add("DataFrames")
import DataFrames
```

3.4. Carga y Escritura de Datos en Julia

Un aspecto fundamental al trabajar con cualquier lenguaje de programación es saber cómo manejar archivos de datos. A continuación, se explica cómo trabajar con archivos en formatos comunes, como *txt* o *csv*.

3.4.1. Lectura de archivos

Archivos *csv*

Para el manejo de archivos en formato *csv*, Julia dispone de la librería con el mismo nombre: ***CSV***. La función más utilizada de este paquete es *CSV.File*, la cual se encarga de leer el archivo completo y detectar el número de filas y columnas, así como el tipo de dato para cada columna.

La función *CSV.File* en lugar de cargar todo el archivo de una vez, permite la lectura fila por fila ahorrando memoria. Sin embargo, devuelve un objeto de tipo *'CSV.File'* (como se muestra a continuación, en el fragmento de código [2] con la función *typeof()*), que puede resultar poco práctico a la hora de manejar el dato.

Con el fin de optimizar el uso de esta librería, se recomienda combinarla con el paquete ***Dataframes***, que ofrece herramientas avanzadas para trabajar con datos tabulares, como el filtrado, la agrupación y la agregación de datos, lo que facilita el análisis y la manipulación de grandes volúmenes de información. Su diseño y funcionalidad son similares a las bibliotecas de R: *data.table*, *data.frame* y *dplyr*.

La función *CSV.read()* carga el archivo de datos y al introducir DataFrame como parámetro transforma los datos en un DataFrame.

Para realizar la demostración, se ha utilizado un archivo de datos sobre las ventas de Apple en 2024, obtenido de la plataforma en línea Kaggle [17].

En primer lugar, se importan las librerías ***Dataframes*** y ***CSV***.

[1]:
```
using  DataFrames
using CSV
```

A continuación, se carga el archivo *csv* y se verifica el tipo de objeto resultante utilizando la función *typeof()*.

[2]:
```
appleSales = CSV.File("C:/Users/emg51/OneDrive/Documentos/
 ↪apple_sales_2024.csv")
typeof(appleSales)
```

```
CSV.File
```

El resultado muestra que el objeto cargado es de tipo *'CSV.File'*.

[3]:
```
first(appleSales,2)
```

```
2-element Vector{CSV.Row}:
CSV.Row: (State = String15("Chongqing"), Region =␣
↪String15("Greater China"), var"iPhone Sales (in million units)"␣
↪= 7.46, var"iPad Sales (in million units)" = 6.75, var"Mac Sales␣
↪(in million units)" = 1.19, var"Wearables (in million units)" =␣
↪5.88, var"Services Revenue (in billion \$)" = 15.88)
CSV.Row: (State = String15("Germany"), Region =␣
↪String15("Europe"), var"iPhone Sales (in million units)" = 8.63,␣
↪var"iPad Sales (in million units)" = 14.06, var"Mac Sales (in␣
↪million units)" = 7.03, var"Wearables (in million units)" = 7.
↪42, var"Services Revenue (in billion \$)" = 10.12)
```

Utilizando la función *first()*, se muestran las dos primeras filas del archivo *csv*.

Por otro lado, se vuelve a cargar el archivo de datos, pero esta vez usando la función *CSV.read()*, que lee el archivo *csv* y permite convertirlo en un *DataFrame*.

[4]:
```
tablaAppleSales = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪apple_sales_2024.csv", DataFrame)
typeof(tablaAppleSales)
```

```
DataFrame
```

El resultado muestra que el objeto resultante es de tipo *'DataFrame'*.

[5]:
```
first(tablaAppleSales,10)
```

	State	Region	iPhone Sales (in million units)	iPad Sales (in million units)	
	String15	String15	Float64	Float64	
1	Chongqing	Greater China	7.46	6.75	...
2	Germany	Europe	8.63	14.06	...
3	UK	Europe	5.61	14.09	...
4	Shanghai	Greater China	7.82	7.97	...
5	Thailand	Rest of Asia	16.7	8.13	...
6	Chongqing	Greater China	12.18	10.97	...
7	UK	Europe	25.47	7.41	...
8	New York	North America	22.37	6.74	...
9	Mexico	Rest of World	20.8	6.79	...
10	Italy	Europe	9.06	9.14	...

Finalmente, se muestran las primeras diez filas de los datos en formato tabla.

Otros archivos de datos

La función de Julia *CSV.read()*, al igual que *read.csv()* en R, permite la lectura de archivos con la extensión .csv, así como de otros archivos de texto delimitados,

.dat, .txt, . . . Siempre que presenten una estructura similar a la de un *csv*.

Cuando los datos están separados por espacios, su procesamiento en R tiende a ser más eficiente en comparación con Julia. Ya que Julia requiere realizar pasos adicionales para obtener el mismo resultado. Para representar esta situación, se ha creado un archivo de datos específico para realizar el ejemplo.

En primer lugar, se importan las librerías necesarias.

[6]:
```
using CSV
using DataFrames
```

Se lee el archivo con la función de Julia *readlines()* y se almacenan las líneas en la variable *lines*.

[7]:
```
file_path = "C:/Users/emg51/OneDrive/Documentos/Datos.dat"
lines = readlines(file_path)
```

```
7-element Vector{String}:
" i var1   y   x1  x2"
"      1     2  3.1 0.1  6"
"      2     14 9.2 0.9  7"
"      3     7  4   0.7  8"
"      4     3  5   0.5  10"
"      5     3  3   0.2  5"
""
```

En este ejemplo, las columnas de datos no están delimitadas por un número fijo de espacios en cada fila. Por ello, si se utiliza directamente la función *CSV.read* con un espacio como separador, se generan columnas adicionales que contienen únicamente valores *missing*. Esto ocurre porque la función trata cualquier secuencia de espacios consecutivos como delimitadores de valores, incluso cuando no hay datos presentes entre ellos. La función *read.csv()* de R no tiene problema en esta situación, y sin necesidad de tratar los datos, lee el archivo de forma correcta.

Para solucionarlo, se procesan las líneas leídas para eliminar los espacios al inicio de cada fila con la función *lstrip()*, y se reemplazan las secuencias de múltiples espacios por un único espacio con *replace()*.

[8]:
```
processed_lines = [replace(lstrip(line), r"\s+" => " ") for line␣
 ↪in lines]
```

```
7-element Vector{String}:
"i var1 y x1 x2"
"1 2 3.1 0.1 6"
"2 14 9.2 0.9 7"
"3 7 4 0.7 8"
"4 3 5 0.5 10"
```

```
"5 3 3 0.2 5"
""
```

Una vez que los datos están procesados, se escriben en un archivo temporal con la función *write()*.

[9]:
```
temp_file_path = "C:/Users/emg51/OneDrive/Documentos/Datos_temp.
↪dat"
write(temp_file_path, join(processed_lines, "\n"))
```

Finalmente, se puede utilizar la función *CSV.read()* como se ha visto en el apartado anterior, introduciendo el archivo temporal y especificando el carácter separador mediante el parámetro *delim*.

[10]:
```
datos = CSV.read(temp_file_path, DataFrame; delim=' ')
```

	i	var1	y	x1	x2
	Int64	Int64	Float64	Float64	Int64
1	1	2	3.1	0.1	6
2	2	14	9.2	0.9	7
3	3	7	4.0	0.7	8
4	4	3	5.0	0.5	10
5	5	3	3.0	0.2	5

3.4.2. Escritura de archivos

Archivos *csv*

Para escribir un archivo de datos en formato *csv*, la mejor opción es utilizar la función de ***CSV***: *CSV.write()*. Esta función permite escribir datos estructurados a archivos en formato *csv*. Para que funcione correctamente, cualquier matriz de datos puede usarse como entrada, pero debe convertirse previamente en una tabla con la función *Tables.table()*, o ser un objeto de tipo *DataFrame*.

Para representar cómo se usa la función *CSV.write()* se emplean variables de apartados anteriores. En primer lugar, se utiliza la variable *datos* de tipo *DataFrame*.

[1]:
```
typeof(datos)
```

```
DataFrame
```

A continuación, se utiliza la función *CSV.write()* con la ruta de destino y el objeto *DataFrame* como entrada.

[2]:
```
CSV.write("C:/Users/emg51/OneDrive/Documentos/dataframe.csv",␣
↪datos)
```

```
"C:/Users/emg51/OneDrive/Documentos/dataframe.csv"
```

Para un ejemplo de escritura de matrices de datos estructurados, utilizamos un objeto de tipo vector.

[3]:
```
typeof(processed_lines)
```

```
Vector{String} (alias for Array{String, 1})
```

En este caso, a la entrada de la matriz de datos se le aplica la función *Tables.table()* para que *CSV.write()* funcione correctamente.

[4]:
```
CSV.write("C:/Users/emg51/OneDrive/Documentos/dataframe_2.csv",␣
↪Tables.table(processed_lines))
```

```
"C:/Users/emg51/OneDrive/Documentos/dataframe_2.csv"
```

Otros archivos de datos

En la librería ***DelimitedFiles*** de Julia se encuentra la función *writedlm()*, la cual permite escribir datos estructurados, como matrices o arrays, en archivos delimitados por caracteres (comas, tabulaciones, puntos y comas, ...). Y es compatible con formatos como *txt* o *dat*.

A continuación, se expone una demostración de cómo se utilizaría esta función.

El primer paso fundamental, importar la biblioteca necesaria.

[5]:
```
using DelimitedFiles
```

En este ejemplo se utiliza la variable de tipo array ya utilizada en apartados anteriores *processed_lines*. Para llamar a la función *writedlm()* basta con introducir como entrada la ruta de destino del archivo a escribir, y el array de datos.

[6]:
```
writedlm("C:/Users/emg51/OneDrive/Documentos/archivo.dat",␣
↪processed_lines)
```

3.5. Estadística básica con Julia

Por último, en esta sección se abordan las funcionalidades más útiles de Julia para realizar análisis estadísticos básicos. Para ello, se utilizan tanto librerías de su biblioteca estándar como paquetes más especializados, muchos de los cuales se han descrito brevemente en el apartado de paquetes de este capítulo 3.3. Se abarca desde el cálculo de operaciones básicas y estadísticos descriptivos con ***Statistics***, visualizaciones con ***StatsPlots*** y ***Plots***, hasta análisis más avanzados con ***DataFrames*** y ***Distributions***.

3.5.1. Estadísticos descriptivos

La librería de Julia ***Statistics*** proporciona un amplio rango de funciones fundamentales para el análisis de datos, como puede ser la media, la mediana, ...

A continuación, se explica cómo se utiliza la función media en Julia:

Para calcular la media de un conjunto de elementos, se utiliza *mean()*. En el siguiente ejemplo se calcula la media de un vector de enteros del 1 al 5.

[1]:
```
using Statistics
mean(1:5)
```

```
3.0
```

También permite introducir como parámetro una función, la cual se aplica a cada elemento del conjunto antes de realizar la media. En el siguiente ejemplo se introduce la función *exp(x)* que devuelve la exponencial de x, e^x, y la aplica al vector v por lo que se calcula: $\frac{e^1+e^2+e^3+e^4+e^5}{2}$

[2]:
```
mean(exp,v)
```

```
46.64083679725964
```

Otras funciones que permite calcular la librería ***Statistics*** de Julia son las que aparecen en el cuadro 3.1. Su uso es similar al que se ha descrito para la media.

Estadístico	Función
Mediana	*median()*
Varianza	*var()*
Desviación típica	*std()*
Percentiles y cuartiles	*quantile()*
Coeficiente de correlación	*cor()*
Covarianza	*cov()*

Cuadro 3.1: Algunas funciones de la librería ***Statistics*** [16]

Además, ***Statistics*** se puede complementar con la librería ***StatsBase***, la cual se debe instalar previamente ya que no se encuentra entre las librerías esenciales de Julia. Esta herramienta permite completar funcionalidades ya existentes en la librería de base ***Statistics*** y, además, tiene funciones adicionales, como por ejemplo *mode()* para calcular la moda.

Un caso en el que ***StatsBase*** complementa a ***Statistics*** es al utilizar la función *quantile()*. En esta función se debe introducir como segundo parámetro un valor del 0 al 1, o un vector de valores del 0 al 1, y *quantile()* devuelve los percentiles correspondientes:

[3]:
```
x = collect(1:10)
quantile(x, [0.1, 0.5, 0.9])
```

```
3-element Vector{Float64}:
 1.9
```

```
5.5
9.1
```

Con ***StatsBase***, permite introducir únicamente un vector de datos, y devuelve automáticamente todos los cuartiles sin tener que especificar nada más.

[4]:
```
using StatsBase
quantile(x)
```

```
5-element Vector{Float64}:
  1.0
  3.25
  5.5
  7.75
 10.0
```

Por otro lado, se echa de menos en la librería ***Statistics*** de Julia obtener un resumen estadístico como el que se obtiene con la función *summary()* de R. Pero se puede obtener algo similar a través de la función equivalente *describe()*, dentro de la librería ***DataFrames***.

[5]:
```
using CSV
using DataFrames

datos = CSV.read("C:/Users/emg51/OneDrive/Documentos/Datos.txt",␣
 ↪DataFrame; delim=',')
describe(datos)
```

	variable	mean	min	median	max	nmissing	eltype
	Symbol	Float64	Real	Float64	Real	Int64	DataType
1	i	3.0	1	3.0	5	0	Int64
2	var1	5.8	2	3.0	14	0	Int64
3	y	4.86	3.0	4.0	9.2	0	Float64
4	x1	0.48	0.1	0.5	0.9	0	Float64
5	x2	7.2	5	7.0	10	0	Int64

Esta función devuelve un pequeño resumen del conjunto de datos, con el nombre de las variables y para cada una de ellas: la media, el valor mínimo, la mediana, el valor máximo, el número de valores ausentes y el tipo.

3.5.2. Gráficos y visualización

Con el fin de desarrollar visualizaciones y gráficos, Julia dispone de la librería ***Plots*** para gráficos más generales, y ***StatsPlots***, una extensión de ***Plots***, diseñada específicamente para ejecutar gráficos estadísticos y análisis exploratorios de datos. En la tabla 3.2, se observan algunas de las funcionalidades más interesantes.

Gráfico	Función
Gráfico de lineas x vs y	*plot(x, y, ...)*
Histograma	*histogram(x, y, ...)*
Gráfico de barras	*bar(x,y, ...)*
Gráfico de densidad	*density(x, y, ...)*
Diagrama de cajas	*boxplot(x, y, ...)*
Gráfico de dispersión	*scatter(x, y, ...)*
Gráficos de correlaciones	*corrplot([y1, y2, ...], ...)*

Cuadro 3.2: Algunas funciones de la librería ***StatsPlots*** [4]

A continuación, se realiza un ejemplo de visualización utilizando el archivo de datos iris. Para ello, es necesario instalar las librerías ***StatsPlots*** y ***RDatasets***

[1]:
```
import Pkg
Pkg.add("RDatasets")
Pkg.add("StatsPlots")
```

Dentro de ***RDatasets*** se pueden encontrar todos los archivos de datos de la librería base ***datasets*** de R. Para cargar en Julia el archivo que se desea utilizar, tal como se muestra en el siguiente fragmento de código [2], en la función *dataset()* se introduce el nombre del paquete donde se encuentra el archivo y el nombre del archivo, en este caso *"iris"*.

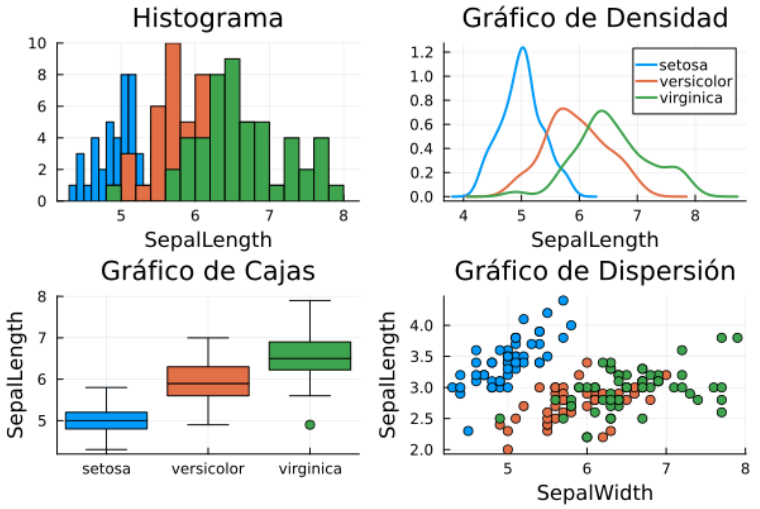
[2]:
```
import RDatasets
iris = RDatasets.dataset("datasets","iris")
```

En Julia para llamar a las variables de un *DataFrame* hay dos opciones, se puede escribir el nombre del *DataFrame* seguido de un punto y el nombre de la variable (*iris.SepalLength*), o utilizando el macro *@df* (importado desde la librería***StatsPlots***), tal como se muestra en el siguiente fragmento de código [3], que carga el *DataFrame* antes de aplicar la función, permitiendo trabajar directamente con sus columnas.

Las funciones de ***StatsPlots*** se comportan de forma similar. Con introducir las variables a representar basta, pero permite modificar diversos parámetros para una mayor personalización de las visualizaciones: *xlabel* e *ylabel* permiten añadir el nombre a cada eje x e y del gráfico, *title* permite añadir un título y *legend* permite añadir o quitar la leyenda (por defecto siempre es true), entre otros.

En este ejemplo, se muestran distintas representaciones gráficas de la variable longitud de sépalo para las distintas especies de iris. Para realizar la representación según una variable categórica, se indica a través del parámetro *group*. En el fragmento de código [3], se crean cuatro gráficos sobre la variable *SepalLength* agrupando por la variable *Species*, y se guarda cada uno de ellos como *p1, p2, p3* y *p4*. Por último, con *plot()* se unen los cuatro gráficos en una única visualización y se indica que la disposición de los gráficos sea de $2 * 2$.

[3]:
```
using StatsPlots

p1 = @df iris histogram(:SepalLength,  group=:Species, bins=15,␣
 ↪title= "Histograma", xlabel="SepalLength", legend=false)
# @df iris plot(:SepalLength, group=:Species,␣
 ↪seriestype="histogram")
p2 = @df iris density(:SepalLength, group=:Species, title=␣
 ↪"Gráfico de Densidad", xlabel="SepalLength",linewidth=2)

p3 = @df iris boxplot(:Species,:SepalLength, group=:Species,␣
 ↪title="Gráfico de Cajas", ylabel="SepalLength", legend=false)

p4 = @df iris scatter(:SepalLength, :SepalWidth, group=:Species,␣
 ↪title="Gráfico de Dispersión", ylabel="SepalLength",␣
 ↪xlabel="SepalWidth", legend=false)
# @df iris plot(:SepalLength, :SepalWidth, group=:Species,␣
 ↪seriestype="scatter")
plot(p1,p2,p3,p4, layout=(2,2))
```



Algunos de los gráficos mostrados se pueden realizar directamente con la función *plot()*, indicando el tipo de representación con el parámetro *seriestype*. Como se muestra comentado en el código [3]. Para hacer un histograma se le da el valor *"histogram"* y para hacer un gráfico de dispersión se le da el valor *"scatter"*.

3.5.3. Distribuciones de probabilidad

En esta última subsección, se abarca brevemente cómo trabajar con distribuciones estadísticas. Con este fin, Julia presenta la librería ***Distributions*** que contiene una amplia variedad de distribuciones con las que trabajar, entre ellas se encuentran: *Normal*(μ, σ), *Binomial(n,p)*, *Bernoulli(p)*, *Poisson*(λ), *Exponential*(λ), *Gamma*(α, θ), *Multinomial(n,p)* ... Además, cuenta con herramientas para determinar propiedades de las distribuciones y trabajar con ellas, permitiendo por ejemplo el cálculo de la función de densidad o la generación de muestras aleatorias a partir de una distribución específica. [15]

Se muestran algunas de estas funciones en la tabla 3.3.

Función	Descripción
pdf(d, x)	Función de densidad/probabilidad
logpdf(d,x)	Logaritmo de la densidad/probabilidad
cdf(d, x)	Función de distribución acumulativa
fit(D, X)	Ajuste por máximo verosimilitud
skewness(d)	Asimetría
kurtosis(d)	Curtosis
loglikelihood(d, data)	Log-verosimilitud

Cuadro 3.3: Algunas funciones de la librería ***Distributions*** [15]

Si se busca profundizar en el modelado probabilístico y el uso de distribuciones en Julia, el siguiente enlace proporciona una guía introductoria al manejo de la librería ***Distributions***:

- Introducción a ***Distributions***: https://juliastats.org/Distributions.jl/latest/starting/

A lo largo de este capítulo se han explorado los conceptos básicos de Julia, para llevar este conocimiento un paso más allá, en el siguiente capítulo se profundiza en los Modelos Lineales Generalizados (GLM), una herramienta clave en el análisis estadístico, y se ve cómo implementarlos tanto en Julia como en R.

Capítulo 4

GLM con R y Julia

Los **Modelos Lineales Generalizados**, también conocidos como **GLM** (Generalized Linear Models), son un conjunto de modelos introducidos por Nelder y Wedderburn en 1972. En este capítulo, se explican estos modelos, proporcionando una definición de los GLM y de sus componentes. Posteriormente, se muestra cómo ajustar a un conjunto de datos los modelos con la herramienta R y con el lenguaje Julia.

4.1. Modelos Lineales Generalizados. GLM

Para la presentación de los modelos **GLM** se toma como referencia los libros de Agresti [1], y Dobson y Barnett [7].

4.1.1. Definición

Los GLM son una extensión de las regresiones lineales estándar, que buscan abarcar los datos con una mayor flexibilidad, para así poder cubrir situaciones donde las regresiones tradicionales no se ajustan adecuadamente.

Para poder aplicar un modelo de regresión lineal clásico, los datos deben cumplir ciertos supuestos para que el modelo sea válido. Se debe satisfacer la hipótesis de normalidad para su variable dependiente, la hipótesis de homocedasticidad (homogeneidad de varianzas), la linealidad entre las variables independientes y la variable dependiente, así como la independencia de los residuos. El cumplimiento de estos supuestos puede resultar demasiado restrictivo para muchos casos, lo que limita la aplicabilidad de las regresiones clásicas.

Los GLM nos permiten modelizar aquellos casos en los que no se satisface uno o varios de dichos supuestos. Al generalizar las regresiones lineales ordinarias, se obtienen estructuras más flexibles, y se dan unos modelos que permiten abarcar muchas más situaciones. Se extienden a distribuciones con respuestas no normales y funciones no lineales de la media.

4.1.2. Componentes

En un modelo lineal generalizado podemos distinguir tres componentes principales:

- **Componente aleatoria:** La componente aleatoria de un GLM se corresponde con la variable respuesta $\boldsymbol{y} = (y_1, \ldots, y_n)^T$ de la distribución.
  Las observaciones $(y_1, \ldots, y_n)$ de esta variable respuesta deben ser independientes, y su función de densidad debe pertenecer a una distribución de la familia exponencial (por ejemplo, pertenecen a la familia exponencial la normal, la binomial, la Poisson o la Gamma).
- **Predictor lineal:** La segunda componente es una función lineal predictora que combina las variables independientes.
  El predictor lineal se denota con la letra griega $\boldsymbol{\eta}$. Para n observaciones y p variables explicativas, la matriz del modelo (también conocida como matriz de diseño en estudios experimentales) es $\boldsymbol{X}$ de tamaño $n \times p$ con los valores $\{x_{ij}\}$, $i = 1, \ldots, n$, $j = 1, \ldots, p$. El predictor lineal se define como la combinación lineal de las columnas de la matriz del modelo con unos parámetros desconocidos $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_p)^T$, obteniendo la expresión en forma matricial:
  $$\boldsymbol{\eta} = \boldsymbol{X}\boldsymbol{\beta} \tag{4.1}$$
- **Función de enlace:** La función de enlace se encarga de relacionar (enlazar) sus otras dos componentes, es decir, conecta la componente aleatoria con el predictor lineal.
  La función de enlace $g(\cdot)$ se trata de una función diferenciable y monótona que va a establecer una relación linealizada entre $\boldsymbol{\eta}$ y la esperanza de la variable dependiente, $E(\boldsymbol{y}) = \mu$:
  $$g[E(\boldsymbol{y})] = g(\mu) = \boldsymbol{X}\boldsymbol{\beta} \tag{4.2}$$
  También se puede escribir como $\mu = g^{-1}(\boldsymbol{\eta})$, siendo $g^{-1}(\cdot)$ la *función respuesta*.
  Cada una de las distribuciones de la familia exponencial tiene su parámetro natural θ, el cual se conoce como parámetro canónico. La función de enlace, g, se llama *enlace canónico* cuando transforma la esperanza de la variable dependiente, μ, en el parámetro natural, θ:
  $$\theta = g(\mu) \tag{4.3}$$

A continuación, en la tabla 4.1, se muestran los enlaces canónicos para algunas de las funciones de densidad más utilizadas.

Distribución	Función de enlace	$\boldsymbol{\eta} = g(\mu)$
Normal	Identidad	$\theta = \mu$
Binomial	Logit	$\theta = \ln(\frac{\mu}{1-\mu})$
Poisson	Logarítmica	$\theta = \ln(\mu)$
Gamma	Inversa	$\theta = \mu^{-1}$

Cuadro 4.1: Enlaces canónicos más utilizados

4.1.3. Estimación de los parámetros

La función de densidad de la variable aleatoria Y con el parámetro natural θ es $f_Y(\, y\, ;\, \theta\,)$. Hay distintas maneras de expresar la densidad dentro de la familia de distribuciones exponenciales. Según Agresti [1], $f_Y(\, y\, ;\, \theta\,)$ pertenece a esta familia si tiene la siguiente forma:

$$f(\, y\, ;\, \theta\, , \phi) = \exp\left(\frac{y\theta - b(\theta)}{a(\phi)} + c(y;\phi)\right) \tag{4.4}$$

Donde $a(\cdot)$, $b(\cdot)$ y $c(\cdot)$ son funciones conocidas, θ es el parámetro natural, como ya se ha mencionado anteriormente, y ϕ es un parámetro de dispersión. Para una explicación más detallada, consulta el anexo C.

Se busca estimar los parámetros $\boldsymbol{\beta}$ de la ecuación 4.1, utilizando la estimación de la máxima verosimilitud, donde las ecuaciones de log-verosimilitud para un GLM son:

$$L(\boldsymbol{\beta}) = \sum_{i=1}^{n} L_i = \sum_{i=1}^{n} \log f(y_1; \theta_i, \phi) = \sum_{i=1}^{n} \frac{y_i\theta_i - b(\theta_i}{a(\phi)} + \sum_{i=1}^{n} c(y_i, \phi), \tag{4.5}$$

para n observaciones independientes.

Al diferenciar las ecuaciones de log-verosimilitud, se obtienen las ecuaciones de verosimilitud:

$$\frac{\partial L(\boldsymbol{\beta})}{\partial \beta_j} = \sum_{i=1}^{n} \frac{(y_i - \mu_i)x_{ij}}{var(y_i)} \frac{\partial \mu_i}{\partial \eta_i} = 0, \quad j = 1, 2, \ldots, p, \tag{4.6}$$

en el que $\eta_i = \sum_{j=1}^{p} \beta_j x_{ij} = g(\mu_i)$ para la función de enlace g.

Los estimadores obtenidos son consistentes y presentan una distribución asintótica normal: $\hat{\beta} \sim N(\boldsymbol{\beta}, \mathcal{I}^{-1})$, siendo I la matriz de información de Fisher, la cual mide la cantidad de información contenida en los datos sobre los parámetros.

Si se llama V a la matriz diagonal de las varianzas y D a la matriz diagonal que contiene los elementos $\partial\mu_i/_{\partial\eta_i}$. Y, teniendo en cuenta la expresión para el predictor lineal 4.1, con matriz del modelo X, la forma matricial de la ecuación de log-verosimilitud es la siguiente:

$$X^T D V^{-1}(y - \mu) = 0, \tag{4.7}$$

donde $\boldsymbol{\beta}$ se encuentra presente de forma implícita a través de μ, con $\mu_i = g^{-1}(\sum_{j=1}^{p} \beta_j x_{ij})$.

Las ecuaciones de verosimilitud son funciones no lineales de $\boldsymbol{\beta}$, y se deben resolver de forma iterativa, no presentan una solución cerrada. Se relacionan con la distribución de y_i a través de μ_i y $var(y_i)$. Estas últimas, a su vez, se relacionan por la fórmula $var(y_i) = v(\mu_i)$, para una función v.

4.1.4. Contrastes de hipótesis sobre los parámetros

Para la inferencia de un GLM hay tres métodos estándar basados en la función de log-verosimilitud. Estos tests contrastan la hipótesis $H_0 : \beta = \beta_0$ frente a $H_1 : \beta \neq \beta_0$ y sus estadísticos son:

- Razón de verosimilitud:
$$-2\log\Lambda = -2\log(\ell_0/\ell_1) = -2(L_0 - L_1), \tag{4.8}$$
donde la razón $\Lambda = \ell_0/\ell_1 \leq 1$, L_0 y L_1 son las funciones de log-verosimilitud maximizadas bajo H_0 y H_1, y bajo condiciones regulares sigue una distribución chi-cuadrado nula en $n \to \infty$, con un grado de libertad;
- Wald:
$$z = \frac{\hat{\beta} - \beta_0}{SE}, \tag{4.9}$$
donde SE es el error estándar estimado de $\hat{\beta}$, y z^2 sigue una distribución chi-cuadrado con un grado de libertad.
- Score:
$$z^2 = \frac{[\partial L(\beta)/\partial\beta_0]^2}{-E[\partial^2 L(\beta)/\partial\beta_0^2]}, \tag{4.10}$$
donde z^2 sigue una chi-cuadrado.

4.1.5. *Deviance*

Para comprobar la bondad de ajuste de un GLM se utiliza el estadístico conocido como *deviance*. Este estadístico compara el modelo elegido con el modelo saturado.

El modelo saturado se trata de un modelo teórico similar al modelo que se esta estudiando, pero que se ajusta perfectamente a los datos observados, es decir, $\hat{\mu} = y$ (a diferencia del modelo nulo con $\hat{\mu} = \bar{y}$). Se utiliza de base para compararlo con otros modelos con más parámetros.

Teniendo en cuenta $L(\hat{\mu}; y)$ como la máxima log-versimilitud del modelo a ajustar, y $L(y; y)$ la máxima log-verosimilitud del modelo saturado, la *deviance* se define como:
$$D(y;\hat{\mu}) = -2\log\left[\frac{\ell_0}{\ell_{sat}}\right] = -2[L(\hat{\mu}; y) - L(y; y)], \tag{4.11}$$

donde ℓ_0 es la máxima verosimilitud para el modelo a comparar, y ℓ_{sat} es la máxima verosimilitud para el modelo saturado.

Considerando $L(\hat{\mu}; y) \leq L(y; y)$, la *deviance* debe ser $D(y; \hat{\mu}) \geq 0$. Cuanto mayor es su valor, peor es el ajuste. Se compara $D(y; \hat{\mu})$ con una χ^2_{n-p}.

4.1.6. Residuos

Tras ajustar el modelo y realizar las pruebas de bondad de ajuste, es importante comprobar los residuos. Nos dan una visión más concreta de en qué lugares se ajusta peor el modelo, y dónde se encuentran observaciones atípicas.

Cuando se ajustan modelos GLM se dispone de distintos tipos de residuos:

Residuos de *Pearson*

En un modelo con función de varianza $v(\mu)$, el residuo para la observación y_i y su valor ajustado $\hat{\mu}$, es:

$$e_i = \frac{y_i - \hat{\mu}_i}{\sqrt{v(\hat{\mu}_i)}}. \tag{4.12}$$

Residuos de la *deviance*

Expresando la *deviance* como $D(y;\hat{\mu}) = \sum_i d_i$, donde
$$d_i = 2\omega_i[y_i(\tilde{\theta}_i - \hat{\theta}_i) - b(\tilde{\theta}_i) + b(\hat{\theta}_i)],$$
con $\hat{\theta}_i$ estimador del parametro natural θ_i, correspondiente a la media estimada $\hat{\mu}_i$, y $\tilde{\theta}_i$ la estimación de θ_i para el modelo saturado.

El residuo de la *deviance* es:
$$\sqrt{d_i} \times signo(y_i - \hat{\mu}_i). \tag{4.13}$$

Residuos estandarizados

Para los residuos de *Pearson* se estima $\sqrt{v(\hat{\mu}_i)} = \sqrt{var(y_i - \hat{\mu}_i)}$. Para estandarizar los residuos se aproxima la varianza a $var(y_i - \hat{\mu}_i) \approx v(\mu_i)(1 - h_{ii})$, donde h_{ii} es el elemento diagonal de la matriz $\boldsymbol{H_w}$ para la observación i (su *leverage*).

El residuo estandarizado se obtiene dividiendo los residuos $(y_i - \hat{\mu}_i)$ entre su error estándar:
$$r_i = \frac{y_i - \hat{\mu}_i}{\sqrt{v(\hat{\mu}_i)(1 - \hat{h}_{ii})}} = \frac{e_i}{\sqrt{1 - \hat{h}_{ii}}}, \tag{4.14}$$

donde $\hat{h}_{ii}$ es el estimador de h_{ii}, y e_i es el residuo de *Pearson*.

4.2. GLM en R

Para ajustar modelos lineales generalizados en R, se tiene la orden *glm()* incluida dentro del paquete base. La ayuda de R nos permite saber cómo se utiliza y los argumentos que requiere de forma más detallada, aplicando el comando: *help(glm)*.

La función se puede aplicar de la siguiente manera:

$$\begin{aligned} glm\ \ &(formula,\ family = gaussian,\ data,\ weights,\ subset,\ na.action,\\ &x = FALSE,\ y = TRUE,\ singular.ok = TRUE,\ contrasts = NULL,\ ...) \end{aligned}$$

Donde, los argumentos más interesantes son: *'formula'*, que se trata del argumento en el que se introduce el objeto de clase 'formula', con una representación simbólica del modelo que se quiere ajustar; *'family'*, introduce la distribución de la familia exponencial de la variable respuesta, la cual determinará la *función de enlace* que se va a utilizar; y *'data'* es el objeto que contiene los datos con las variables del modelo.

Además, esta función de R admite especificar otros argumentos, como el argumento *'start'*, en el que se puede introducir un vector con valores iniciales para los coeficientes β, o *'intercept'*, un argumento que al definirlo como falso posibilita

generar un modelo sin término independiente (por defecto *'intercept'* siempre es verdadero).

Las opciones que da R para definir el objeto *'family'*, en el cual se especifica la distribución del modelo a ajustar con su *función de enlace*, son:

- gaussian(link = "identity")
- binomial(link = "logit")
- Gamma(link = "inverse")
- inverse.gaussian(link = "$1/mu^2$")
- poisson(link = "log")
- quasi(link = "identity", variance = "constant")
- quasibinomial(link = "logit")
- quasipoisson(link = "log")

Por defecto, si no se especifica, R aplica la función de enlace de la distribución Gausiana. La salida es un objeto de clase *'glm'*, que contiene los datos del ajuste, como *coefficients* que contiene el vector con los coeficientes estimados, *residuals* con los residuos, *fitted.values* con los valores estimados de la media o *family* con el objeto de clase *'family'* que se ha utilizado.

Se puede acceder directamente a cada componente del objeto *glm*, o utilizar la función *summary()* para obtener un pequeño resumen del ajuste. También, es posible aplicar la función *anova()* para obtener una tabla con el análisis de la *deviance* del modelo. Con la función *resid()* se obtienen los residuos del modelo, y con *predict()* los valores ajustados.

4.3. GLM en Julia

A diferencia de R, en Julia no se puede aplicar un modelo lineal generalizado a partir de sus órdenes básicas. En su lugar, cuenta con la librería ***GLM***. En el manual de Julia se proporciona información detallada de cómo usar el paquete ***GLM***. Se puede acceder a esta información a través del siguiente enlace:

- Documentación ***GLM.jl***: https://juliastats.org/GLM.jl/dev/

En esta librería, además de incluir la función *glm()* para ajustar modelos lineales generalizados, se ofrecen diversas herramientas para ajustar modelos como la función *lm()* que se centra en modelos lineales. Muchas de estas herramientas y métodos presentan nombres similares a los métodos de R (*coef(), predict(), vcov(), deviance(), etc.*), lo que hace que sean bastante intuitivos de utilizar.

Como se muestra en el anexo B, con el macro *@doc* podemos obtener la documentación sobre la función *glm()* o desde el REPL escribiendo: *?glm*.

La función se puede aplicar de dos formas distintas:

- El primer método es indicar la fórmula del modelo, que debe ser un objeto *'Formula'* del paquete ***StatsModels*** (*@formula(y ~ x)*), y la tabla con los datos.

$$glm\ \ (formula, data, distr :: UnivariateDistribution,$$
$$link :: Link = canonicallink(distr); < keywordarguments >)$$

- El segundo método es introducir la matriz de variables explicativas X por un lado, y el vector con la variable dependiente y.

$$glm\ \ (X :: AbstractMatrix, y :: AbstractVector, distr :: UnivariateDistribution,$$
$$link :: Link = canonicallink(distr); < keywordarguments >)$$

En ambos casos se indica la distribución con el parámetro *'distr'* y el enlace canónico con *'link'*.

Además, la función *glm()* permite modificar otros argumentos como *'maxiter'* que se trata del número máximo de iteraciones, definido a 30 por defecto. O el argumento *'start'*, por defecto no tiene valores definidos, permite introducir un vector con los valores iniciales para β (este vector debe tener la misma longitud que el número de columnas en la matriz del modelo).

Por otro lado, Julia también permite ajustar modelos lineales generalizados de la siguiente forma: *fit(GeneralizedLinearModel, ...)*. Aplicando *fit()* indicando el tipo de modelo que se desea ajustar, es una forma alternativa con la que se obtiene el mismo resultado que con la función *glm()*.

Las opciones más típicas que da Julia para definir el objeto *'distr'*, en el cual se especifica la distribución del modelo a ajustar con su función de enlace *'link'* , son:

- *Bernoulli()* con enlace *LogitLink()*
- *Binomial()* con enlace *LogitLink()*
- *Gamma()* con enlace *InverseLink()*
- *Geometric()* con enlace *LogLink()*
- *InverseGaussian()* con enlace *InverseSquareLink()*
- *Poisson()* con enlace *LogLink()*
- *NegativeBinomial()* con enlace *NegativeBinomialLink()*, frecuentemente utilizado con el enlace *LogLink()* (la función *glm()* de R no permite ajustar la binomial negativa, pero se puede aplicar a partir de la función *glm.nb()* de la libreria ***MASS***)
- *Normal()* con enlace *IdentityLink()*

Julia contiene implementadas las siguientes funciones de enlace: *CauchitLink()*, *CloglogLink()*, *IdentityLink()*, *InverseLink()*, *InverseSquareLink()*, *LogitLink()*, *LogLink()*, *NegativeBinomialLink()*, *PowerLink()*, *ProbitLink()* y *SqrtLink()*

La función *glm()* devuelve un objeto con información detallada del modelo ajustado, de tipo *'GeneralizedLinearModel'* definido en el paquete ***GLM***.

Utiliza una implementación del método de mínimos cuadrados iterativamente reponderados (IRLS, del inglés Iteratively Reweighted Least Sqares), pero a diferencia de R, que también utiliza IRLS, Julia es capaz de trabajar más rápidamente y con conjuntos de datos más grandes. [25]

En el capítulo 5 se va a comparar la eficiencia de ambos lenguajes, y para ello se simularán muestras de datos de distintos tamaños para posteriormente modelarlos en R y en Julia.

Capítulo 5

Etudio de casos de ajustes de modelos GLM con R y Julia

En este capítulo se realiza un análisis práctico del proceso de ajuste de los Modelos Lineales Generalizados en los lenguajes R y Julia, con el propósito de comparar su comportamiento y rendimiento.

El capítulo comienza con la descripción del conjunto de datos empleado, para continuar con una descripción detallada de las salidas generadas por cada herramienta y la comparación de los resultados obtenidos. Finalmente, se realizan unas pruebas de rendimiento computacional para llevar la comparación entre ambos lenguajes.

5.1. Datos

Se introduce un ejemplo con el archivo de datos *TitanicSurvival* presente entre los datasets de R. Este archivo reúne datos acerca de los pasajeros del desastre del Titanic de 1912.

El archivo de datos proporcionado por R contiene 1309 observaciones y 4 variables, donde el nombre de cada fila corresponde al nombre del pasajero, y cada columna es una de estas variables [9] (en el Anexo D, sección D.1, se realiza una descripción más detallada):

- survived: Estado del pasajero, informa de si sobrevivió al naufragio o no (*no*/*yes*).
- y: Transformación de la variable survived a una variable númerica binaria (0/1).
- sex: Sexo del pasajero (*female*/*male*).
- age: Edad del pasajero en años, a excepción de los niños que se expresa en fracciones de año (con 263 valores ausentes).
- passengerClass: Clase del pasajero (*1st*/*2nd*/*3rd*).

```
head(titanic)
```

```
##                                 survived    sex     age passengerClass y
## Allen, Miss. Elisabeth Walton        yes female 29.0000            1st 1
## Allison, Master. Hudson Trevor       yes   male  0.9167            1st 1
## Allison, Miss. Helen Loraine          no female  2.0000            1st 0
## Allison, Mr. Hudson Joshua Crei       no   male 30.0000            1st 0
## Allison, Mrs. Hudson J C (Bessi       no female 25.0000            1st 0
## Anderson, Mr. Harry                  yes   male 48.0000            1st 1
```

Una vez descritos los datos que se van a utilizar, se aplica la función *glm()* de R y la función *glm()* de Julia y se analiza la salida de cada una de ellas para destacar similitudes y diferencias entre ambas.

5.2. Aplicación

Para comenzar, al aplicar la función de R, si se indica únicamente la fórmula y el conjunto de datos, se ajusta un modelo lineal generalizado con familia Normal y función de enlace identidad (gaussian(link = "identity")).

Por otro lado, en Julia si no se especifica la familia y la función de enlace, el modelo no se aplica y la función da error. Es decir, no contiene unos valores por defecto para estos parámetros y deben ser indicados cada vez que se desea utilizar la función *glm()*.

Teniendo en cuenta que la variable dependiente y se trata de una variable binaria, que indica si el pasajero sobrevivió (1) o no (0), a continuación, se compara la salida de ambos lenguajes con el ajuste del modelo con la familia Binomial.

La fórmula del modelo a ajustar es:

$$g(\pi) = \log\left(\frac{\pi}{1-\pi}\right) = \beta_0 + \beta_1 \cdot sexmale + \beta_2 \cdot age + \beta 3 \cdot passengerClass2nd + \beta_4 \cdot passengerClass3rd$$

5.2.1. Salida en R

```
MR <- glm(y ~ sex + age + passengerClass, data = titanic,
          family = binomial(link = "logit"))
MR
```

```
##
## Call:  glm(formula = y ~ sex + age + passengerClass,
##     family = binomial(link = "logit"), data = titanic)
##
## Coefficients:
##       (Intercept)            sexmale                age
##           3.52207           -2.49784           -0.03439
##  passengerClass2nd passengerClass3rd
```

```
##          -1.28057           -2.28966
##
## Degrees of Freedom: 1045 Total (i.e. Null);  1041 Residual
## Null Deviance:       1415
## Residual Deviance: 982.5     AIC: 992.5
```

Al aplicar la función *glm()* en R, se obtiene un objeto de tipo *list* y clase *glm*. Si se llama a este objeto, se pueden conocer las estimaciones de los coeficientes obtenidos para el modelo, los grados de libertad, la *deviance* del modelo nulo, la *deviance* residual del modelo ajustado y el valor del estadístico *AIC* del modelo.

En R, se puede obtener información adicional del modelo llamando directamente a los elementos contenidos dentro del objeto de salida. Por ejemplo, los residuos del modelo se obtienen con: *MR$residuals*, o los valores ajustados del modelo se encuentran en *MR$fitted.values*. Como alternativa, R ofrece funciones que permiten extraer la misma información. Por ejemplo, *resid(MR)* devuelve los residuos del modelo, y *coef(MR)* obtiene los valores de los coeficientes estimados.

```
summary(MR)
```

```
##
## Call:
## glm(formula = y ~ sex + age + passengerClass,
##     family = binomial(link = "logit"), data = titanic)
##
## Coefficients:
##                     Estimate Std. Error z value Pr(>|z|)
## (Intercept)         3.522074   0.326702  10.781  < 2e-16 ***
## sexmale            -2.497845   0.166037 -15.044  < 2e-16 ***
## age                -0.034393   0.006331  -5.433 5.56e-08 ***
## passengerClass2nd -1.280570   0.225538  -5.678 1.36e-08 ***
## passengerClass3rd -2.289661   0.225802 -10.140  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1414.62  on 1045  degrees of freedom
## Residual deviance:  982.45  on 1041  degrees of freedom
## AIC: 992.45
##
## Number of Fisher Scoring iterations: 4
```

Además, en R se puede aplicar la función *summary()* al modelo, obteniendo información adicional como: los errores estándar, valores z, p-valores y el número de iteraciones.

5.2.2. Salida en Julia

Al aplicar la función *glm()* de Julia, a diferencia de R, se debe introducir la fórmula explícitamente como un objeto de tipo *formula*. Otra diferencia entre R y Julia

es que, para indicar la familia y la función de enlace, en Julia se expresa en dos parámetros distintos:

```
[1]: MJulia = glm(@formula(y ~ sex + age + passengerClass), titanic,␣
     ↪Binomial(), LogitLink(), verbose=true)
```

```
Iteration: 1, deviance: 982.483312134374, diff.dev.:5.
↪223447259856698
Iteration: 2, deviance: 982.4531069160355, diff.dev.:0.
↪030205218338551276
Iteration: 3, deviance: 982.4531049512136, diff.dev.:1.
↪9648218767542858e-6

StatsModels.TableRegressionModel{GeneralizedLinearModel{GLM.
↪GlmResp{Vector{Float64}, Binomial{Float64}, LogitLink}, GLM.
↪DensePredChol{Float64, LinearAlgebra.CholeskyPivoted{Float64,␣
↪Matrix{Float64}, Vector{Int64}}}}, Matrix{Float64}}

y ~ 1 + sex + age + passengerClass

Coefficients:
------------------------------------------------------------------
                       Coef.      Std. Error       z  Pr(>|z|)
------------------------------------------------------------------
(Intercept)           3.52207      0.326702   10.78    <1e-26
sex: male            -2.49784      0.166037  -15.04    <1e-50
age                  -0.0343932    0.006331   -5.43    <1e-07
passengerClass: 2nd  -1.28057      0.225538   -5.68    <1e-07
passengerClass: 3rd  -2.28966      0.225802  -10.14    <1e-23
------------------------------------------------------------------
----------------------
Lower 95%   Upper 95%
----------------------
 2.88175     4.1624
-2.82327    -2.17242
-2.82327    -2.17242
-0.0468018  -0.0219847
-1.72262    -0.838523
-2.73222    -1.8471
----------------------
```

Una vez ajustado el modelo, en la salida se obtiene: el tipo del objeto generado por la función *glm()*, *TableRegressionModel*, la fórmula del modelo ajustado, y una tabla con: los estimadores de los coeficientes del modelo, los errores estándar, los valores z, p-valores, y los límites inferiores y superiores de los intervalos de confianza al 95 %.

En Julia no hay una función equivalente a la función de R *summary()* que saque información adicional del modelo, pero comparando la salida en ambos lenguajes, se observa que la salida simple de la función de Julia proporciona muchos de los datos que se obtienen con *summary()* en R (los coeficientes del modelo, los errores estándar, los valores z y los p-valores). Además, la salida de Julia aporta los intervalos de confianza de los estimadores de los coeficientes.

Para conocer el número de iteraciones requeridas se puede modificar el parámetro *verbose* dentro de la función *glm()* a verdadero. La salida contendrá información sobre cada una de las iteraciones realizadas, en concreto la *deviance* residual y la diferencia de *deviance* con la iteración previa.

Por otro lado, la salida de R muestra directamente los valores de la *deviance* residual, del estadístico *AIC* y los grados de libertad. En Julia, estos datos se pueden obtener, de forma similar a R, a partir de distintas funciones aplicadas al objeto de salida: *deviance(MJulia)*, *aic(MJulia)*, *dof_residual(MJulia)*. Y al igual que R, también presenta funciones que permiten obtener los valores de los estimadores de los coeficientes, *coef(MJulia)*, y los valores ajustados del modelo, *predict(MJulia)* (para más detalles, examinar la sección D.3 del Anexo D).

Pero, como desventajas, Julia no contiene una función que devuelva los valores residuales del modelo, deben calcularse programándolos directamente, y no presenta los códigos de significancia que proporciona la salida de *summary()* en R, que resultan de gran utilidad a la hora de escoger las variables del modelo.

5.2.3. Comparación de resultados

Para analizar y comparar los resultados obtenidos en ambos lenguajes, se ha recogido la información del ajuste - presentado en los apartados previos - en los cuadros 5.1 y 5.2.

En cuanto a los modelos ajustados, en la tabla 5.1, tanto R como Julia alcanzan las mismas estimaciones de los coeficientes, de los errores estándar y de los valores z. La mayor diferencia observable es que al mostrar los p-valores, R ofrece una mayor precisión que Julia para valores extremadamente pequeños, mostrando cifras en notación científica con mayor detalle.

Binomial

	Estimadores		Error Estándar		Valor Z		Pr(>|z|)	
	R	Julia	R	Julia	R	Julia	R	Julia
β_0	3.5221	3.5221	0.3267	0.3267	10.781	10.78	<2e-16	<1e-26
β_1	-2.4978	-2.4978	0.1660	0.1660	-15.044	-15.04	<2e-16	<1e-50
β_2	-0.0344	-0.0344	0.0063	0.0063	-5.433	-5.43	5.56e-08	<1e-07
β_3	-1.2806	-1.2806	0.2255	0.2255	-5.678	-5.68	1.36e-08	<1e-07
β_4	-2.2897	-2.2897	0.2258	0.2258	-10.140	-10.14	<2e-16	<1e-23

Cuadro 5.1: Resultados de la estimacióndel modelo GLM Binomial en R y en Julia

El criterio de información de Akaike, en la tabla 5.2, es el mismo en ambos ajustes. En cambio, la función de Julia *dof_residual()* (más información en el Anexo D sección D.3) devuelve 1042 grados de libertad cuando realmente son 1041, ya que se tiene 1046 observaciones y el modelo contiene 5 parámetros. Esto se debe a que Julia presenta el siguiente error: para algunas de las familias exponenciales, como Bernoulli, Binomial o Poisson, se incrementa en 1 los grados de libertad del modelo ajustado. Esto ya ha sido señalado por algunos usuarios de Julia [10].

	Binomial		Poisson	
	R	Julia	R	Julia
AIC	992.45	992.4531	1410.1	1410.1198
Grados de Libertad	1041	1042	1041	1042
Deviance Residual	982.45	982.4531	546.12	546.1198
Nº Iteraciones	4	3	5	4

Cuadro 5.2: Información obtenida de los modelos ajustados en R y en Julia

Por último, a pesar de que R requiere un mayor número de iteraciones que Julia, la *deviance* residual del modelo es la misma en ambos lenguajes. Esta diferencia puede ser una de las razones por las que se considera Julia como un lenguaje más eficiente, ya que ambos han alcanzado unos resultados muy similares con diferencias mínimas y Julia ha ejecutado una iteración menos.

5.3. Comparación de eficiencias

Tras analizar las metodologías, salidas y resultados obtenidos al aplicar Modelos Lineales Generalizados en R y en Julia, esta sección tiene como objetivo evaluar si Julia cumple con su promesa de estar altamente orientado a la eficiencia computacional, especialmente al trabajar con grandes cantidades de datos.

5.3.1. Descripción

Se han diseñado pruebas que consisten en medir el tiempo que tarda cada lenguaje en ajustar modelos GLM sobre conjuntos de datos de distintos tamaños. Los datos han sido simulados en R utilizando la librería *simglm*. Se han generado seis conjuntos de datos, que van desde 1.000 hasta 1.000.000 observaciones. Esta amplia escala busca comprobar si, como se ha planteado, Julia optimiza su rendimiento al trabajar con volúmenes de datos más grandes.

Para los ajustes con la familia Binomial, las simulaciones replican la estructura de las variables explicativas del ejemplo anterior, *titanic*. Se simula una variable respuesta binaria y, y tres variables explicativas: *age*, *sex* y *passengerClass*.

Por otro lado, con el objetivo de que cualquier diferencia en la eficiencia se atribuya únicamente a la familia estadística utilizada, y no a la estructura de los datos, para realizar los ajustes con la familia de Poisson, las simulaciones contienen las mismas tres variables explicativas:*age*, *sex* y *passengerClass*. Y la variable respuesta y se simula como una variable de conteo discreta con valores enteros, siguiendo una distribución Poisson.

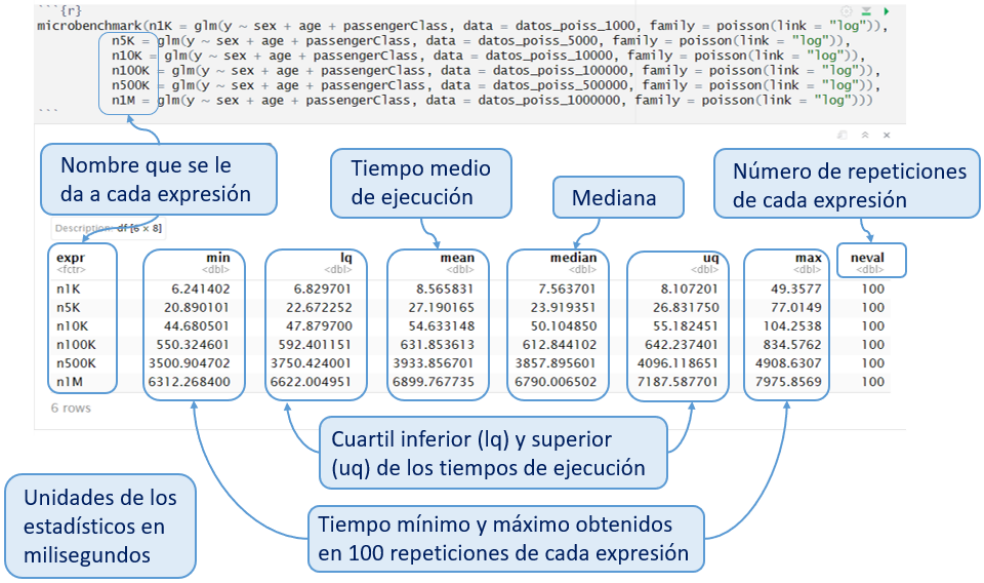
Después, los datos se han exportado para poder aplicar exactamente los mismos modelos sobre los mismos datos en ambos lenguajes.

La medición de tiempos en R se ha realizado con la librería *microbenchmark*, cuya función del mismo nombre permite medir los tiempos de ejecución de varias expresiones en paralelo. En Julia, se ha llevado a cabo con la librería ***BenchmarkTools***, la cual funciona de forma distinta, ya que solo permite medir el tiempo de una expresión por ejecución. A pesar de esta diferencia, ambas herramientas ofrecen resultados comparables.

En ambos lenguajes, las herramientas de evaluación del rendimiento —*microbenchmark* en R y ***BenchmarkTools*** en Julia— han repetido 100 veces cada ajuste para cada conjunto de datos. A partir de estas ejecuciones, se calculan estadísticos descriptivos de los tiempos de ejecución por expresión, como el mínimo, la media y la mediana, entre otros.

Para complementar el análisis, se han realizado los primeros ajustes utilizando la familia Binomial, y, posteriormente, se han vuelto a repetir las pruebas ajustando modelos con la familia de Poisson. Este doble enfoque permite evaluar si el tipo de modelo lineal generalizado influye en los tiempos de ejecución observados.

En las figuras 5.1 y 5.2, se presentan unos ejemplos de salidas de las funciones *microbenchmark()* en R y *benchmark()* en Julia, respectivamente. Para más detalles, se encuentra el procedimiento completo en la sección D.4 del Anexo D.



expr	min	lq	mean	median	uq	max	neval
n1K	6.241402	6.829701	8.565831	7.563701	8.107201	49.3577	100
n5K	20.890101	22.672252	27.190165	23.919351	26.831750	77.0149	100
n10K	44.680501	47.879700	54.633148	50.104850	55.182451	104.2538	100
n100K	550.324601	592.401151	631.853613	612.844102	642.237401	834.5762	100
n500K	3500.904702	3750.424001	3933.856701	3857.895601	4096.118651	4908.6307	100
n1M	6312.268400	6622.004951	6899.767735	6790.006502	7187.587701	7975.8569	100

Figura 5.1: Ejemplo de salida de *microbenchmark* en R

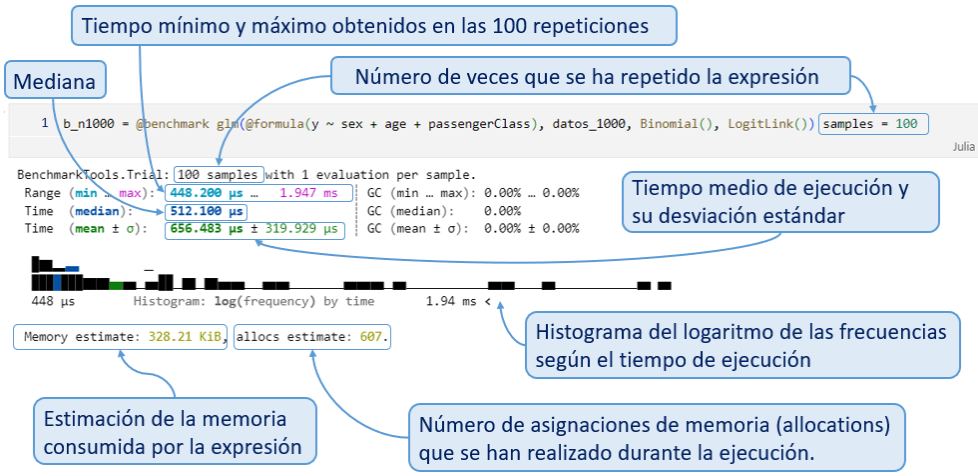


Figura 5.2: Ejemplo de salida de *benchmark* con ***BenchmarkTools*** en Julia

5.3.2. Resultados

Finalmente, se han recogido los resultados obtenidos en las tablas 5.3 y 5.4. Se han dispuesto los tiempos resultantes en milisegundos, donde las columnas representan cada tamaño muestral y las filas cada estadístico descriptivo para R y Julia.

Binomial

		n = 1K	n = 5K	n = 10K	n = 100K	n = 500K	n = 1000K
Mínimo	R	3,66	13,50	26,75	268,19	1449,41	3072,25
	Julia	0,45	1,59	3,13	39,43	211,77	412,51
Media	R	4,97	18,97	38,86	339,23	1712,40	3546,30
	Julia	0,66	2,22	4,28	44,96	282,28	501,41
Mediana	R	4,67	17,29	32,36	317,70	1649,85	3319,84
	Julia	0,51	2,03	3,77	43,77	236,72	450,11
Máximo	R	9,57	43,98	151,53	547,11	2476,27	5391,78
	Julia	1,95	3,80	20,19	80,87	781,83	774,75

Cuadro 5.3: Resultados prueba de ajustes de GLMs con la familia **Binomial**. Unidades: milisegundos

A primera vista se evidencia que para todos los casos de uso, el lenguaje de Julia siempre es más rápido que R. Siendo la media de tiempo del ajuste Binomial (tabla 5.3) para el conjunto de datos más pequeño aproximadamente 7,5 veces mayor en R comparado con Julia, y 7 veces mayor en el conjunto de datos más grande. Lo cual demuestra que consigue mantener un rendimiento sobresaliente al trabajar con volúmenes de datos altamente elevados, pues la diferencia de velocidad es de 0.5 aumentando 999 mil unidades el tamaño de la muestra.

Poisson							
		n = 1K	n = 5K	n = 10K	n = 100K	n = 500K	n = 1000K
Mínimo	R	6,38	23,21	44,09	518,43	3021,41	5359,04
	Julia	0,56	2,62	4,86	74,32	441,40	894,77
Media	R	8,26	28,85	62,71	589,92	3331,27	5793,78
	Julia	0,77	3,31	6,65	78,94	489,45	1082,00
Mediana	R	7,75	28,16	58,93	572,32	3309,03	5755,10
	Julia	0,65	2,98	6,04	77,92	465,22	962,89
Máximo	R	28,41	54,03	219,41	736,99	5342,47	6443,58
	Julia	1.99	5,37	11,11	107,52	742,27	2362,00

Cuadro 5.4: Resultados prueba de ajustes de GLMs con la familia **Poisson**. Unidades: milisegundos

Del mismo modo, para el ajuste de Poisson (tabla 5.4) la media de tiempo del tamaño más pequeño en R es 10,7 veces mayor que la de Julia, y 5,3 veces mayor en el conjunto de datos más grande, confirmando de nuevo que el rendimiento de Julia es excepcional.

En 5.5, se recogen las medias de los tiempos de ejecución obtenidos de todos los ajustes. Las columnas corresponden a cada familia de modelos (Binomial y Poisson) implementados en R y en Julia, mientras que las filas indican los distintos tamaños muestrales utilizados.

	Binomial		Poisson	
	R	Julia	R	Julia
n = 1K	4,97	0,66	8,26	0,77
n = 5K	18,97	2,22	28,85	3,31
n = 10K	38,86	4,28	62,71	6,65
n = 100K	339,23	44,96	589,92	78,94
n = 500K	1712,40	282,28	3331,27	489,45
n = 1000K	3546,30	501,41	5793,78	1082,00

Cuadro 5.5: Medias obtenidas en las pruebas de ajustes de GLM con la familia Binomial y la Poisson. Unidades: milisegundos

Se aprecia que, al ajustar el modelo de Poisson, los tiempos de ejecución aumentan ligeramente en comparación con los obtenidos con la familia Binomial en ambos lenguajes. Esto demuestra que el tipo de modelo ajustado influye en el rendimiento de ejecución de ambos lenguajes. Aun así, Julia mantiene un desempeño superior en todos los escenarios evaluados.

En las figuras 5.3 y 5.4 se observan las representaciones gráficas de los tiempos medios de ejecución obtenidos frente a los distintos tamaños muestrales para la Binomial y la Poisson respectivamente.

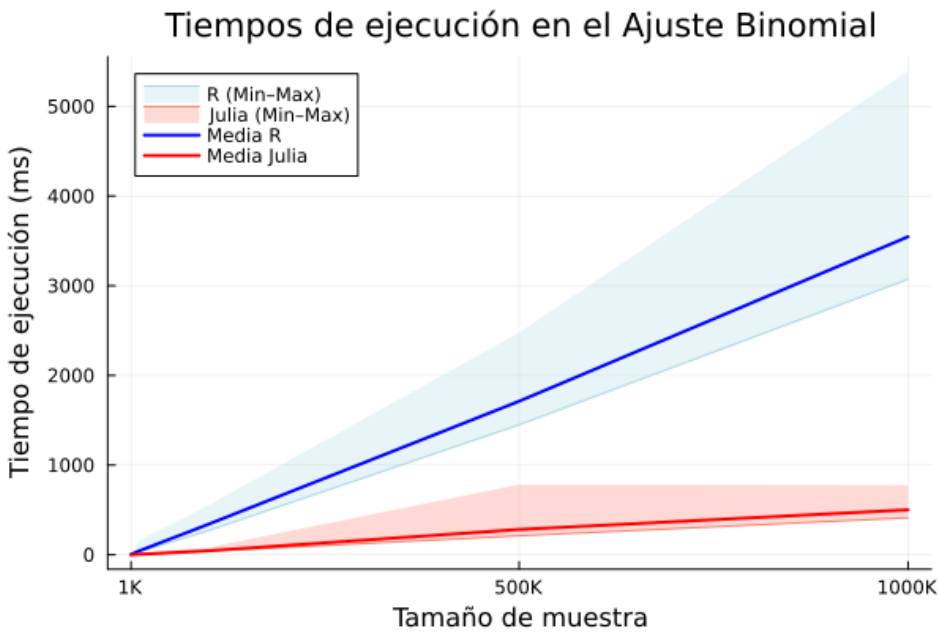


Figura 5.3: Representación gráfica de los resultados obtenidos en los ajustes de modelos Binomiales

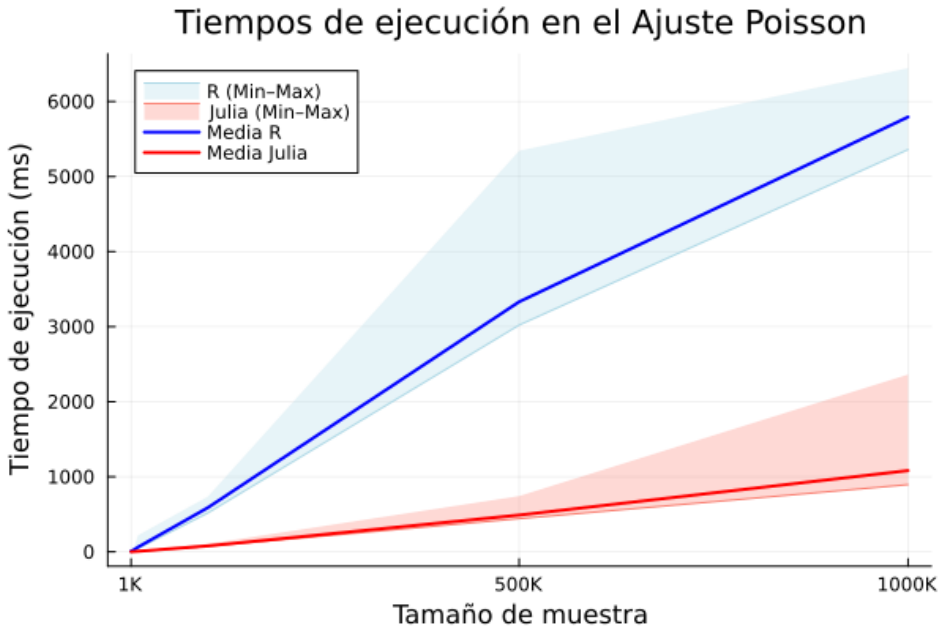


Figura 5.4: Representación gráfica de los resultados obtenidos en los ajustes de modelos de Poisson

Capítulo 6

Conclusiones

Por último, en esta sección se exponen las conclusiones derivadas de la realización del proyecto y se presentan las líneas de trabajo a futuro.

Como se propone en el capítulo 1, el objetivo de este proyecto es realizar un estudio detallado del lenguaje de programación Julia y de sus características principales con el fin de evaluar si podría suponer una alternativa viable al resto de lenguajes de programación que llevan usándose durante años en las ciencias de análisis de datos.

En cuanto a la usabilidad, Julia se ha mostrado como un lenguaje accesible e intuitivo. Su sintaxis resulta familiar para quienes ya tienen experiencia en otros lenguajes de programación, lo que facilita el proceso de aprendizaje. Además, muchas de sus funciones replican comportamientos presentes en herramientas populares, lo que reduce la curva de adaptación para usuarios de otras herramientas.

A esto se suma una comunidad activa y en crecimiento que ofrece un gran número de recursos de apoyo, como blogs especializados, tutoriales en línea, plataformas interactivas y foros, lo que favorece un aprendizaje autodidacta y dinámico.

Durante las comparaciones realizadas con el lenguaje R, Julia ha demostrado ser una alternativa altamente competente en el ámbito estadístico, obteniendo salidas y resultados muy similares en el ajuste de Modelos Lineales Generalizados.

También se ha visto que Julia aún presenta ciertas limitaciones, como la ausencia de algunas funcionalidades consolidadas en R o pequeños desajustes en determinadas bibliotecas. No obstante, debe tenerse en cuenta que R cuenta con una larga trayectoria de desarrollo orientado específicamente a la Estadística, lo que le ha permitido consolidar una amplia gama de herramientas, bibliotecas especializadas y documentación altamente detallada.

Por su parte, Julia, a pesar de ser un lenguaje más joven y de propósito más general, ha sido diseñado con un fuerte énfasis en la eficiencia computacional. En este sentido, las pruebas realizadas con la función glm() han confirmado que Julia cumple con dicha premisa, mostrando un rendimiento superior en términos de tiempo de ejecución, especialmente con volúmenes de datos elevados.

En consecuencia, aunque R seguirá siendo una herramienta de referencia en el análisis estadístico, se ha llegado a la conclusión de que resulta altamente recomendable explorar el uso de Julia en escenarios donde la eficiencia y el manejo de grandes conjuntos de datos sean factores críticos.

6.1. Trabajo Futuro

Teniendo en cuenta que Julia es un lenguaje de programación relativamente reciente, con poco más de una década desde su aparición, resulta especialmente interesante observar su evolución futura y las mejoras que pueda incorporar.

Si Julia continúa ampliando sus funcionalidades estadísticas y desarrolla un ecosistema tan robusto y accesible como el de R, podría convertirse en una alternativa aún más competitiva e incluso llegar a sustituirlo en ciertos contextos profesionales. Por ello, seguir de cerca el desarrollo de nuevos paquetes, mejoras en la documentación y ampliación de la comunidad de usuarios será clave en los próximos años.

Asimismo, resultaría interesante ampliar el análisis del lenguaje Julia hacia otras funcionalidades más allá del ajuste de Modelos Lineales Generalizados (GLM), que ha sido el foco principal de este trabajo. Por ejemplo, se podría evaluar su rendimiento en la generación de visualizaciones de datos complejos, así como su capacidad para implementar y ejecutar otros tipos de modelos estadísticos o de aprendizaje automático. Este tipo de análisis permitiría obtener una visión más completa del potencial de Julia como herramienta integral para el análisis de datos.

Además, se puede ampliar el análisis comparando las velocidades que se obtienen al aplicar paralelización sobre múltiples núcleos tanto en R como en Julia, para estudiar si Julia ofrece ventajas respecto a R, ya que por cuestiones de tiempo no se ha podido profundizar en este aspecto.

Apéndice A

Recursos web

Ayudas en línea disponibles para agilizar el aprendizaje del lenguaje:

- Página principal de Julia: https://julialang.org/
- Página de descarga: https://julialang.org/downloads/
- Foro: https://discourse.julialang.org/
- Canal de YouTube: https://www.youtube.com/user/JuliaLanguage
- Paquetes: https://julialang.org/packages/
- Recursos de aprendizaje: https://julialang.org/learning/
- Lectura y escritura de blogs sobre Julia: https://forem.julialang.org/
- Tutorial primeros pasos con Julia: https://hedero.webs.upv.es/julia-basico/1-primerospasos/[20]

Apéndice B

Funciones de interés

names()

Al introducir el nombre de un paquete o librería devuelve una lista con todas las funciones que contiene:

[1]:
```
names(GLM)
```

```
101-element Vector{Symbol}:
 Symbol("@formula")
 :AbstractContrasts
 :AbstractTerm
 :Bernoulli
 :Binomial
 ⋮
 :terms
 :unprotect
 :vcov
 :vif
 :width
```

Si el paquete contiene muchas funciones una manera de poder verlas todas es imprimirlas con un bucle como se muestra a continuación:

[2]:
```
for i in names(GLM) println(i) end
```

? o @doc

Se puede acceder a la documentación de una función desde el REPL de Julia escribiendo un signo de interrogación '?' y el nombre de la función, como se muestra en la imagen B.1.

Para acceder a la ayuda fuera de la REPL se puede utilizar el macro @doc:

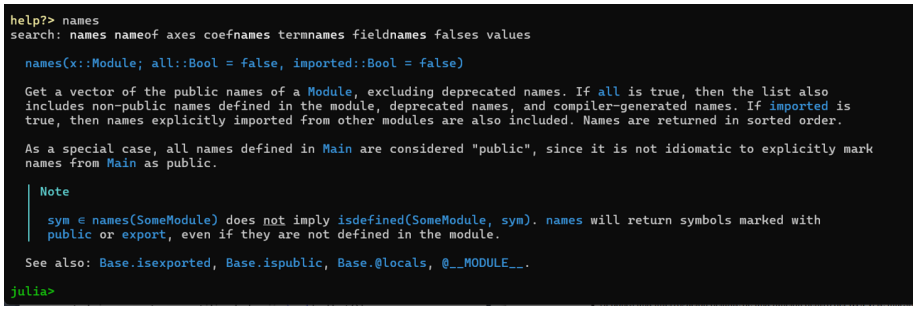


Figura B.1: Acceso a la ayuda de Julia desde el REPL

[3]:
```
@doc names
```

collect()

Para crear un vector que contenga los valores enteros del 1 al 5, hay que utilizar la función *collect()*, esto se debe a que si se utiliza únicamente $v = 1 : 5$ se crea un objeto de tipo rango (*UnitRangeInt64*), que únicamente contiene una representación compacta del intervalo, con la posición inicial, la final y su longitud, sin crear un array explícito con todas las posiciones:

[4]:
```
v = 1:5
typeof(v)
```

```
UnitRange{Int64}
```

Esto le permite a Julia ahorrar memoria, lo que se traduce en operaciones más ligeras al trabajar con objetos de este tipo. Pero si se necesita trabajar con vectores resulta de gran utilidad la función *collect()*:

[5]:
```
w = collect(1:5)
typeof(w)
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

skipmissing()

Cuando se aplica una función a un array o matriz que contiene valores ausentes, NaN o missing, esta devuelve respectivamente NaN o missing. Para que funcione bien se puede aplicar a los datos la función *skipmissing()*, que al encontrarse con un valor ausente lo omite.

[6]:
```
v = [1, 2, 3, missing, missing, 6, 7, 8]
```

```
8-element Vector{Union{Missing, Int64}}:
1
2
```

```
3
missing
missing
6
7
8
```

[7]:
```
sum(v)
```

```
missing
```

[8]:
```
sum(skipmissing(v))
```

```
27
```

zeros()

La función *zeros()* te devuelve un array con el número de ceros que se introduzca:

[9]:
```
zeros(5)
```

```
5-element Vector{Float64}:
0.0
0.0
0.0
0.0
0.0
```

varinfo()

La función *varinfo()* proporciona información sobre las variables creadas en el espacio de trabajo (o *workplace*) actual.

Un modo de liberar memoria cuando alguna variable ya no es necesaria es vaciar su contenido de la siguiente manera: julia>var = nothing. Ya que, si se elimina la variable, puede provocar problemas según el entorno de trabajo. En caso de que se busque limpiar el entorno de trabajo por completo, se debe terminar la sesión de Julia y empezar de nuevo otra sesión.

[10]:
```
varinfo()
```

name	size	summary
Base		Module
Core		Module
Main		Module
datos	1.047 KiB	5×5 DataFrame
v	80 bytes	5-element Vector{Int64}
x	120 bytes	10-element Vector{Int64}
y	296 bytes	32-element Vector{Int64}
z	120 bytes	10-element Vector{Int64}

- Manejo de DataFrames
- Los comentarios de una línea se indican con el símbolo #. Para incluir bloques de comentarios que ocupen más de una linea, se delimita el texto que se quiere comentar entre: #=, al principio, y, =#, al final del texto.
- Los archivos escritos en Julia deben nombrarse con la extensión .jl. Una opción para cargar un archivo de Julia es ejecutando en el REPL la siguiente función: julia>include("programa_en_julia.jl")
- en RDatasets.packages(): ggplot2 An Implementation of the Grammar of Graphics

Apéndice C

La familia exponencial

Como se menciona en el capítulo 4, sección 4.1.2, *la componente aleatoria* pertenece a una distribución de la familia exponencial, por lo que esta familia de distribuciones toma un papel muy importante dentro de estos modelos.

La función de densidad de la variable aleatoria Y con el parámetro natural θ es $f_Y(\, y\,;\, \theta\,)$. Hay distintas maneras de expresar la densidad dentro de la familia de distribuciones exponenciales. Según Agresti [1], $f_Y(\, y\,;\, \theta\,)$ pertenece a esta familia si tiene la siguiente forma:

$$f(\, y\,;\, \theta\,, \phi) = \exp\left(\frac{y\theta - b(\theta)}{a(\phi)} + c(y; \phi)\right) \tag{C.1}$$

Donde $a(\cdot)$, $b(\cdot)$ y $c(\cdot)$ son funciones conocidas, θ es el parámetro natural, como ya se ha mencionado anteriormente, y ϕ es un parámetro de dispersión.

Por lo general, se suele dar que la función conocida $a(\cdot)$ de ϕ es $a(\phi) = \frac{\phi}{w}$, con w conocido.

Cada distribución de la familia exponencial con $f_Y(\, y\,;\, \theta\,)$ tendrá valores de $a(\cdot)$, $b(\cdot)$ y $c(\cdot)$ diferentes. Por ejemplo, la distribución normal, que pertenece a esta familia, tiene la función de densidad:

$$f(\, y\,;\, \theta\,, \phi) = \exp\left(\frac{y\mu - \frac{\mu^2}{2}}{\sigma^2} - \frac{1}{2}\left(\frac{y^2}{\sigma^2} + \log(2\pi\sigma^2)\right)\right) \tag{C.2}$$

Sabiendo que para la distribución normal se tiene que $\theta = \mu$ y $\phi = \sigma^2$, a partir de las ecuaciones (C.1) y (C.2) se puede sacar que las funciones conocidas, $a(\cdot)$, $b(\cdot)$ y $c(\cdot)$ son las siguientes:

- $a(\phi) = \phi$,
- $b(\theta) = \frac{\theta^2}{2}$,

- $c(y;\phi) = -\frac{1}{2}\left(\frac{y^2}{\phi} + \log(2\pi\phi)\right)$.

Media y varianza

Considerando que Y tiene la función de densidad $f_Y(\,y\,;\,\theta\,,\phi)$ (C.1), se puede obtener la media y la varianza de la distribución a partir de la función de log-verosimilitud $L = \log f(Y;\theta,\phi)$, los cuales cumplen, generalmente:

(1) $E(\,{}^{\partial L}/_{\partial\theta}\,) = 0$,

(2) $var(\,{}^{\partial L}/_{\partial\theta}\,) = -E(\,{}^{\partial^2 L}/_{\partial\theta^2}\,)$.

Aplicando la log-verosimilitud a (C.1) se tiene:

$$L \;=\; \log f_Y(\,y\,;\,\theta\,,\phi) \;=\; \frac{y\theta - b(\theta)}{a(\phi)} + c(y,\phi) \tag{C.3}$$

La primera derivada parcial de la log-verosimilitud:

$$\left(\frac{\partial}{\partial\theta}\right)\log f_Y(\,y\,;\,\theta\,,\phi) \;=\; \frac{y - b'(\theta)}{a(\phi)} \tag{C.4}$$

Y la segunda derivada:

$$\left(\frac{\partial^2}{\partial\theta^2}\right)\log f_Y(\,y\,;\,\theta\,,\phi) \;=\; \frac{-b''(\theta)}{a(\phi)} \tag{C.5}$$

Teniendo en cuenta la primera derivada, (C.4), y la función (1), se deduce que la media es:

$$E\left(\frac{y - b'(\theta)}{a(\phi)}\right) = 0 \qquad \Rightarrow \qquad E(Y) = b'(\theta) \tag{C.6}$$

Y del mismo modo, con las funciones de la primera y segunda derivada, (C.4) y (C.5), y la función (2), se deduce que la varianza es:

$$var\left(\frac{y - b'(\theta)}{a(\phi)}\right) = -E\left(\frac{-b''(\theta)}{a(\phi)}\right) \quad \Rightarrow \quad var(Y) = b''(\theta)a(\phi) \tag{C.7}$$

Apéndice D

Código complementario del capítulo 5

D.1. Datos Titanic

```
library(carData)
```

```
titanic <- TitanicSurvival
head(titanic)
```

```
##                                survived    sex     age passengerClass
## Allen, Miss. Elisabeth Walton       yes female 29.0000            1st
## Allison, Master. Hudson Trevor      yes   male  0.9167            1st
## Allison, Miss. Helen Loraine         no female  2.0000            1st
## Allison, Mr. Hudson Joshua Crei      no   male 30.0000            1st
## Allison, Mrs. Hudson J C (Bessi      no female 25.0000            1st
## Anderson, Mr. Harry                 yes   male 48.0000            1st
```

```
summary(titanic)
```

```
##  survived      sex           age          passengerClass
##  no :809   female:466   Min.   : 0.1667   1st:323
##  yes:500   male  :843   1st Qu.:21.0000   2nd:277
##                         Median :28.0000   3rd:709
##                         Mean   :29.8811
##                         3rd Qu.:39.0000
##                         Max.   :80.0000
##                         NA's   :263
```

La variable *age* contiene 263 valores ausentes, y se tienen 1309 observaciones en total. Por ello, se ha considerado que para el ejemplo que se va a realizar es una

muestra lo suficientemente grande como para eliminar aquellos individuos que no tienen el valor edad. A mayores, para que la función de R *glm()* se aplique correctamente, se transforma la variable respuesta a una variable numérica y, donde "no" pasa a tomar el valor entero 0, y "yes" el valor 1:

```
titanic <-titanic[!is.na(titanic$age),]
titanic$y <- as.numeric(titanic$survived) -1
summary(titanic)
```

```
##  survived      sex           age          passengerClass       y
##  no :619   female:388   Min.   : 0.1667   1st:284        Min.   :0.0000
##  yes:427   male  :658   1st Qu.:21.0000   2nd:261        1st Qu.:0.0000
##                         Median :28.0000   3rd:501        Median :0.0000
##                         Mean   :29.8811                  Mean   :0.4082
##                         3rd Qu.:39.0000                  3rd Qu.:1.0000
##                         Max.   :80.0000                  Max.   :1.0000
```

```
write.csv(titanic,file="C:/Users/emg51/OneDrive/Documentos/titanic")
```

D.2. GLM en R

```
MR <- glm(y ~ sex + age + passengerClass, data = titanic,
          family = binomial(link = "logit"))
summary(MR)
```

```
##
## Call:
## glm(formula = y ~ sex + age + passengerClass,
##     family = binomial(link = "logit"), data = titanic)
##
## Coefficients:
##                     Estimate Std. Error z value Pr(>|z|)
## (Intercept)         3.522074   0.326702  10.781  < 2e-16 ***
## sexmale            -2.497845   0.166037 -15.044  < 2e-16 ***
## age                -0.034393   0.006331  -5.433 5.56e-08 ***
## passengerClass2nd -1.280570   0.225538  -5.678 1.36e-08 ***
## passengerClass3rd -2.289661   0.225802 -10.140  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1414.62  on 1045  degrees of freedom
## Residual deviance:  982.45  on 1041  degrees of freedom
## AIC: 992.45
##
## Number of Fisher Scoring iterations: 4
```

D.3. GLM en Julia

Para realizar la demostración de la aplicación del modelo en Julia, se utilizan los datos ya transformados desde R directamente, los cuales se han guardado en un archivo *csv* en local.

[1]:
```
using GLM, DataFrames, StatsBase, CSV
titanic = CSV.read("C:/Users/emg51/OneDrive/Documentos/titanic",␣
↪DataFrame)
```

[2]:
```
MJulia = glm(@formula(y ~ sex + age + passengerClass), titanic,␣
↪Binomial(), LogitLink(), verbose=true)
```

```
Iteration: 1, deviance: 982.483312134374, diff.dev.:5.
↪223447259856698
Iteration: 2, deviance: 982.4531069160355, diff.dev.:0.
↪030205218338551276
Iteration: 3, deviance: 982.4531049512136, diff.dev.:1.
↪9648218767542858e-6

StatsModels.TableRegressionModel{GeneralizedLinearModel{GLM.
↪GlmResp{Vector{Float64}, Binomial{Float64}, LogitLink}, GLM.
↪DensePredChol{Float64, LinearAlgebra.CholeskyPivoted{Float64,␣
↪Matrix{Float64}, Vector{Int64}}}}, Matrix{Float64}}

y ~ 1 + sex + age + passengerClass

Coefficients:
──────────────────────────────────────────────────────────────────
                      Coef.      Std. Error       z  Pr(>|z|)
──────────────────────────────────────────────────────────────────
(Intercept)           3.52207      0.326702   10.78    <1e-26
sex: male            -2.49784      0.166037  -15.04    <1e-50
age                  -0.0343932    0.006331   -5.43    <1e-07
passengerClass: 2nd  -1.28057      0.225538   -5.68    <1e-07
passengerClass: 3rd  -2.28966      0.225802  -10.14    <1e-23
──────────────────────────────────────────────────────────────────
──────────────────────
Lower 95%   Upper 95%
──────────────────────
 2.88175     4.1624
-2.82327    -2.17242
-2.82327    -2.17242
-0.0468018  -0.0219847
-1.72262    -0.838523
-2.73222    -1.8471
──────────────────────
```

Julia no presenta una función similar a *summary()* de R, pero se puede obtener la misma información a través de distintas funciones:

[3]:
```
aic(MJulia) #funcion de StatsBase
```

```
992.4531049512144
```

[4]:
```
coef(MJulia)
```

```
5-element Vector{Float64}:
3.5220740107259565
-2.4978446665438767
-0.03439323084618117
-1.280569737833445
-2.2896605554713747
```

[5]:
```
stderror(MJulia)
```

```
5-element Vector{Float64}:
0.32670220036080083
0.1660365007216366
0.006331000587600012
0.22553819017052185
0.22580191957314344
```

[6]:
```
confint(MJulia)
```

```
5×2 Matrix{Float64}:
2.88175     4.1624
-2.82327    -2.17242
-0.0468018  -0.0219847
-1.72262    -0.838523
-2.73222    -1.8471
```

[7]:
```
vcov(MJulia) #Matriz estimada de varianza-covarianza de los␣
↪coeficientes estimados
```

```
5×5 Matrix{Float64}:
0.106734    -0.0232603   -0.00167786   -0.0447099    -0.0569236
-0.0232603   0.0275681    7.02555e-5    0.00461582    0.00811704
-0.00167786  7.02555e-5   4.00816e-5    0.000489083   0.00072628
-0.0447099   0.00461582   0.000489083   0.0508675     0.0312531
-0.0569236   0.00811704   0.00072628    0.0312531     0.0509865
```

En Julia al introducir en la función*predict()* como único parámetro un modelo, calcula los valores ajustados como la función *fitted()* de R.

[8]:
```
predict(MJulia)
```

```
1046-element Vector{Float64}:
0.9258533055349972
0.7296211038548692
0.9693290312831151
0.4981081137279094
⋮
0.675619230369142
0.10184854743696088
0.10028621537065195
0.09424808759468355
```

[9]:
```
deviance(MJulia)
```

```
982.4531049512136
```

[10]:
```
dof_residual(MJulia)
```

```
1042.0
```

[11]:
```
dof(MJulia)
```

```
5
```

D.4. Eficiencia de los GLM

D.4.1. Simulación de datos

```
coef(MR)
```

```
##       (Intercept)             sexmale                 age passengerClass2nd
##         3.52207401        -2.49784467        -0.03439323        -1.28056974
## passengerClass3rd
##        -2.28966056
```

```
mean(MR$residuals)
```

```
## [1] 0.1401719
```

```
var(MR$residuals)
```

```
## [1] 9.47736
```

```
mean(titanic$age)
```

```
## [1] 29.88113
```

```r
sd(titanic$age)
```

```
## [1] 14.4135
```

Respuesta Binomial

```r
#install.packages("simglm")
library(simglm)
library(magrittr)
```

```r
simular_n_titanic  <- function(n){
  set.seed(1234)
    sim_arguments <- list(
      formula = y ~ 1 + sex + age + passengerClass,
      fixed = list(sex = list(var_type = 'factor',
                              levels = c('male', 'female')),
                   age = list(var_type = 'continuous',
                              mean = 29.88,
                              sd = 14.41),
                  passengerClass = list(var_type = 'factor',
                              levels = c('1st', '2nd','3rd'))),

      error = list(mean = 0, variance = 10),
      sample_size = n,
      reg_weights = c(3.52, -2.5, -0.03, -1.28, -2.29),
      outcome_type = 'binary'
    )

    sim_data = simulate_fixed(data = NULL, sim_arguments) %>%
    simulate_error(sim_arguments) %>%
    generate_response(sim_arguments)

    sim_data$age = pmax(1, pmin(sim_data$age, 80))

  return(sim_data)


}
```

```r
datos_1000 <- simular_n_titanic(1000)
datos_5000 <- simular_n_titanic(5000)
datos_10000 <- simular_n_titanic(10000)
datos_100000 <- simular_n_titanic(100000)
datos_500000 <- simular_n_titanic(500000)
datos_1000000 <- simular_n_titanic(1000000)
```

```
write.csv(datos_1000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_1000")
write.csv(datos_5000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_5000")
write.csv(datos_10000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_10000")
write.csv(datos_100000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_100000")
write.csv(datos_500000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_500000")
write.csv(datos_1000000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_1000000")
```

Respuesta Poisson

```
simular_n_poisson  <- function(n){
  set.seed(1234)
    sim_arguments <- list(
      formula = y ~ 1 + sex + age + passengerClass,
      fixed = list(sex = list(var_type = 'factor',
                              levels = c('male', 'female')),
                   age = list(var_type = 'continuous',
                              mean = 29.88,
                              sd = 14.41),
                  passengerClass = list(var_type = 'factor',
                              levels = c('1st', '2nd','3rd'))),

      error = list(mean = 0, variance = 10),
      sample_size = n,
      reg_weights = c(3.52, -2.5, -0.03, -1.28, -2.29),
      outcome_type = 'poisson'
    )

    sim_data = simulate_fixed(data = NULL, sim_arguments) %>%
    simulate_error(sim_arguments) %>%
    generate_response(sim_arguments)

    sim_data$age = pmax(1, pmin(sim_data$age, 80))

  return(sim_data)

}
```

```
datos_poiss_1000 <- simular_n_poisson(1000)
datos_poiss_5000 <- simular_n_poisson(5000)
datos_poiss_10000 <- simular_n_poisson(10000)
```

```
datos_poiss_100000 <- simular_n_poisson(100000)
datos_poiss_500000 <- simular_n_poisson(500000)
datos_poiss_1000000 <- simular_n_poisson(1000000)
```

```
write.csv(datos_poiss_1000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_poiss_1000")
write.csv(datos_poiss_5000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_poiss_5000")
write.csv(datos_poiss_10000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_poiss_10000")
write.csv(datos_poiss_100000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_poiss_100000")
write.csv(datos_poiss_500000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_poiss_500000")
write.csv(datos_poiss_1000000,
    file="C:/Users/emg51/OneDrive/Documentos/datos_poiss_1000000")
```

D.4.2. Benchmark R

Puebas Binomial

```
#install.packages("microbenchmark")
library(microbenchmark)
```

```
microbenchmark(
    n1K = glm(y ~ sex + age + passengerClass,
        data = datos_1000, family = binomial(link = "logit")),
    n5K = glm(y ~ sex + age + passengerClass,
        data = datos_5000, family = binomial(link = "logit")),
    n10K = glm(y ~ sex + age + passengerClass,
        data = datos_10000, family = binomial(link = "logit")),
    n100K = glm(y ~ sex + age + passengerClass,
        data = datos_100000, family = binomial(link = "logit")),
    n500K = glm(y ~ sex + age + passengerClass,
        data = datos_500000, family = binomial(link = "logit")),
    n1M = glm(y ~ sex + age + passengerClass,
        data = datos_1000000, family = binomial(link = "logit")))
```

```
## Unit: milliseconds
##  expr       min         lq        mean     median         uq       max neval
##   n1K    3.6586    4.19385    4.968231    4.66925    5.29400    9.5649   100
##   n5K   13.5000   16.67405   18.971396   17.29065   19.86895   43.9844   100
##  n10K   26.7490   30.92920   38.857809   32.35855   38.60290  151.5321   100
## n100K  268.1923  294.14665  339.233727  317.70130  377.26265  547.1099   100
## n500K 1449.4140 1576.57860 1712.396249 1649.84685 1776.36285 2476.2719   100
##   n1M 3072.2480 3211.92330 3546.303043 3319.84250 3695.26535 5391.7843   100
```

Pruebas Poisson

```
microbenchmark(
    n1K = glm(y ~ sex + age + passengerClass,
        data = datos_poiss_1000, family = poisson(link = "log")),
    n5K = glm(y ~ sex + age + passengerClass,
        data = datos_poiss_5000, family = poisson(link = "log")),
    n10K = glm(y ~ sex + age + passengerClass,
        data = datos_poiss_10000, family = poisson(link = "log")),
    n100K = glm(y ~ sex + age + passengerClass,
        data = datos_poiss_100000, family = poisson(link = "log")),
    n500K = glm(y ~ sex + age + passengerClass,
        data = datos_poiss_500000, family = poisson(link = "log")),
    n1M = glm(y ~ sex + age + passengerClass,
        data = datos_poiss_1000000, family = poisson(link = "log")))
```

```
## Unit: milliseconds
##   expr         min          lq       mean      median          uq
##    n1K    6.383801    7.376251    8.25889    7.745401    8.325001
##    n5K   23.205901   27.406251   28.84799   28.161850   29.306601
##   n10K   44.093900   55.894301   62.70557   58.934251   60.569902
##  n100K  518.432901  556.863450  589.91714  572.320701  609.558401
##  n500K 3021.413001 3209.070451 3331.27298 3309.030501 3407.113401
##    n1M 5359.041301 5584.839851 5793.78130 5755.097851 5993.573351
##       max  neval
##   28.4121    100
##   54.0321    100
##  219.4067    100
##  736.9985    100
## 5342.4666    100
## 6443.5772    100
```

D.4.3. Benchmark Julia

[1]:
```
using BenchmarkTools, GLM
```

[2]:
```
datos_1000 = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪datos_1000", DataFrame)
datos_5000 = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪datos_5000", DataFrame)
datos_10000 = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪datos_10000", DataFrame)
datos_100000 = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪datos_100000", DataFrame)
datos_500000 = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪datos_500000", DataFrame)
```

```
datos_1000000 = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪datos_1000000", DataFrame)
```

[3]:
```
datos_poiss_1000 = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪datos_poiss_1000", DataFrame)
datos_poiss_5000 = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪datos_poiss_5000", DataFrame)
datos_poiss_10000 = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪datos_poiss_10000", DataFrame)
datos_poiss_100000 = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪datos_poiss_100000", DataFrame)
datos_poiss_500000 = CSV.read("C:/Users/emg51/OneDrive/Documentos/
 ↪datos_poiss_500000", DataFrame)
datos_poiss_1000000 = CSV.read("C:/Users/emg51/OneDrive/
 ↪Documentos/datos_poiss_1000000", DataFrame)
```

Pruebas Binomial

[4]:
```
b_n1000 = @benchmark glm(@formula(y ~ sex + age +␣
 ↪passengerClass), datos_1000, Binomial(), LogitLink()) samples␣
 ↪= 100
```

```
BenchmarkTools.Trial: 100 samples with 1 evaluation per sample.
 Range (min ... max):  448.200 µs ...   1.947 ms   GC (min ...␣
 ↪max): 0.00% ... 0.00%
 Time  (median):     512.100 µs               GC (median):    0.
 ↪00%
 Time  (mean ± σ):   656.483 µs ± 319.929 µs   GC (mean ± σ):  0.
 ↪00% ± 0.00%
```


```
 Memory estimate: 328.21 KiB, allocs estimate: 607.
```

[5]:
```
b_n5000 = @benchmark glm(@formula(y ~ sex + age +␣
 ↪passengerClass), datos_5000, Binomial(), LogitLink()) samples␣
 ↪= 100
```

```
BenchmarkTools.Trial: 100 samples with 1 evaluation per sample.
 Range (min ... max):  1.587 ms ...   3.796 ms   GC (min ... max):␣
 ↪0.00% ... 0.00%
 Time  (median):     2.028 ms               GC (median):    0.00%
 Time  (mean ± σ):   2.217 ms ± 528.311 µs   GC (mean ± σ):  0.00%␣
 ↪± 0.00%
 Memory estimate: 1.48 MiB, allocs estimate: 607.
```

[6]:
```
b_n10000 = @benchmark glm(@formula(y ~ sex + age +␣
↪passengerClass), datos_10000, Binomial(), LogitLink()) samples␣
↪= 100
```

```
BenchmarkTools.Trial: 100 samples with 1 evaluation per sample.
 Range (min ... max):  3.128 ms ... 20.185 ms   GC (min ... max):␣
↪0.00% ... 75.23%
 Time  (median):     3.769 ms               GC (median):    0.00%
 Time  (mean ± σ):   4.281 ms ±  1.859 ms   GC (mean ± σ):  3.55%␣
↪±  7.52%
```


```
 Memory estimate: 2.93 MiB, allocs estimate: 607.
```

[7]:
```
b_n100000 = @benchmark glm(@formula(y ~ sex + age +␣
↪passengerClass), datos_100000, Binomial(), LogitLink())␣
↪samples = 100
```

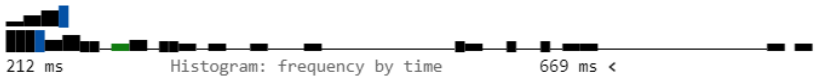
```
BenchmarkTools.Trial: 100 samples with 1 evaluation per sample.
 Range (min ... max):  39.426 ms ... 80.867 ms   GC (min ... max):␣
↪0.00% ... 37.83%
 Time  (median):     43.772 ms               GC (median):    5.13%
 Time  (mean ± σ):   44.959 ms ±  5.111 ms   GC (mean ± σ):  5.86%␣
↪±  5.44%
```


```
 Memory estimate: 29.03 MiB, allocs estimate: 608.
```

[8]:
```
b_n500000 = @benchmark glm(@formula(y ~ sex + age +␣
↪passengerClass), datos_500000, Binomial(), LogitLink())␣
↪samples = 100 seconds = 300
```

```
BenchmarkTools.Trial: 100 samples with 1 evaluation per sample.
 Range (min ... max):  211.765 ms ... 781.830 ms   GC (min ...␣
↪max):  0.00% ... 56.86%
 Time  (median):     236.717 ms               GC (median):     5.
↪53%
```

```
 Time  (mean ± σ):   282.283 ms ± 108.266 ms   GC (mean ± σ):  16.
↪66% ± 16.65%
```




```
 Memory estimate: 145.05 MiB, allocs estimate: 608.
```

[9]:
```
b_n1000000 = @benchmark glm(@formula(y ~ sex + age +␣
↪passengerClass), datos_1000000, Binomial(), LogitLink())␣
↪samples = 100 seconds = 600
```

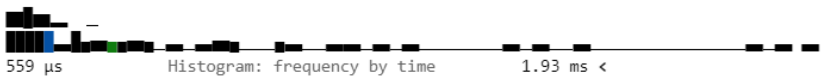
```
BenchmarkTools.Trial: 100 samples with 1 evaluation per sample.
 Range (min ... max):  412.510 ms ... 774.748 ms   GC (min ...␣
↪max):  0.00% ... 41.15%
 Time  (median):     450.108 ms                GC (median):     5.
↪62%
 Time  (mean ± σ):   501.413 ms ±  97.971 ms   GC (mean ± σ):  15.
↪61% ± 13.49%
```


```
 Memory estimate: 290.07 MiB, allocs estimate: 608.
```

Pruebas Poisson

[10]:
```
b_poiss_n1000 = @benchmark glm(@formula(y ~ sex + age +␣
↪passengerClass), datos_poiss_1000, Poisson(), LogLink())␣
↪samples = 100
```

```
BenchmarkTools.Trial: 100 samples with 1 evaluation per sample.
Range (min ... max):  559.400 µs ...   1.994 ms   GC (min ... max):
 ↪ 0.00% ... 0.00%
Time  (median):     648.250 µs               GC (median):    0.00%
Time  (mean ± σ):   767.092 µs ± 294.731 µs   GC (mean ± σ):  0.
↪00% ± 0.00%
```




```
Memory estimate: 331.02 KiB, allocs estimate: 645.
```

[11]:
```
b_poiss_n5000 = @benchmark glm(@formula(y ~ sex + age +␣
↪passengerClass), datos_poiss_5000, Poisson(), LogLink())␣
↪samples = 100
```

```
BenchmarkTools.Trial: 100 samples with 1 evaluation per sample.
Range (min ... max):  2.621 ms ...   5.367 ms   GC (min ... max):␣
↪0.00% ... 0.00%
Time  (median):     2.978 ms               GC (median):    0.00%
Time  (mean ± σ):   3.313 ms ± 701.665 µs   GC (mean ± σ):  0.00%␣
↪± 0.00%
```


```
Memory estimate: 1.48 MiB, allocs estimate: 638.
```

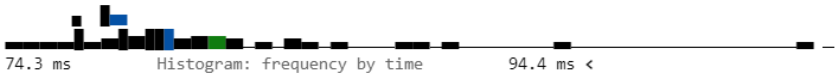
[12]:
```
b_poiss_n10000 = @benchmark glm(@formula(y ~ sex + age +␣
↪passengerClass), datos_poiss_10000, Poisson(), LogLink())␣
↪samples = 100
```

```
BenchmarkTools.Trial: 100 samples with 1 evaluation per sample.
Range (min ... max):  4.857 ms ... 11.109 ms   GC (min ... max): 0.
↪00% ... 0.00%
Time  (median):     6.035 ms               GC (median):    0.00%
Time  (mean ± σ):   6.651 ms ±  1.458 ms   GC (mean ± σ):  0.00% ±␣
↪0.00%
```


```
Memory estimate: 2.93 MiB, allocs estimate: 638.
```

[13]:
```
b_poiss_n100000 = @benchmark glm(@formula(y ~ sex + age +␣
↪passengerClass), datos_poiss_100000, Poisson(), LogLink())␣
↪samples = 100
```

```
BenchmarkTools.Trial: 64 samples with 1 evaluation per sample.
Range (min ... max):  74.324 ms ... 107.518 ms   GC (min ... max):␣
↪0.00% ... 18.23%
Time  (median):     77.920 ms               GC (median):    1.43%
Time  (mean ± σ):   78.941 ms ±   4.751 ms   GC (mean ± σ):  2.71%␣
↪±  3.77%
```




```
Memory estimate: 29.04 MiB, allocs estimate: 646.
```

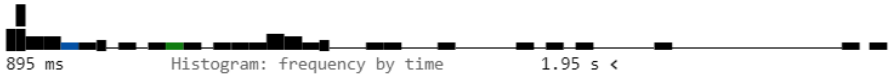
[14]:
```
b_poiss_n500000 = @benchmark glm(@formula(y ~ sex + age +␣
↪passengerClass), datos_poiss_500000, Poisson(), LogLink())␣
↪samples = 100 seconds = 300
```

```
BenchmarkTools.Trial: 100 samples with 1 evaluation per sample.
Range (min ... max):  441.397 ms ... 742.272 ms   GC (min ... max):
 ↪ 0.00% ... 33.56%
Time  (median):     465.216 ms                GC (median):    2.65%
Time  (mean ± σ):   489.449 ms ±  67.196 ms   GC (mean ± σ):  7.
 ↪24% ±  9.14%
```


```
Memory estimate: 145.05 MiB, allocs estimate: 660.
```

[15]:
```
b_poiss_n1000000 = @benchmark glm(@formula(y ~ sex + age +␣
 ↪passengerClass), datos_poiss_1000000, Poisson(), LogLink())␣
 ↪samples = 100 seconds = 600
```

```
BenchmarkTools.Trial: 100 samples with 1 evaluation per sample.
Range (min ... max):  894.773 ms ...    2.362 s   GC (min ... max):
 ↪ 2.01% ... 17.71%
Time  (median):     962.889 ms                GC (median):    2.70%
Time  (mean ± σ):     1.082 s ± 236.584 ms   GC (mean ± σ):  9.
 ↪13% ±  9.74%
```




```
Memory estimate: 290.07 MiB, allocs estimate: 646.
```

D.4.4. Representación gráfica de los resultados

[1]:
```
using DataFrames, Plots
```

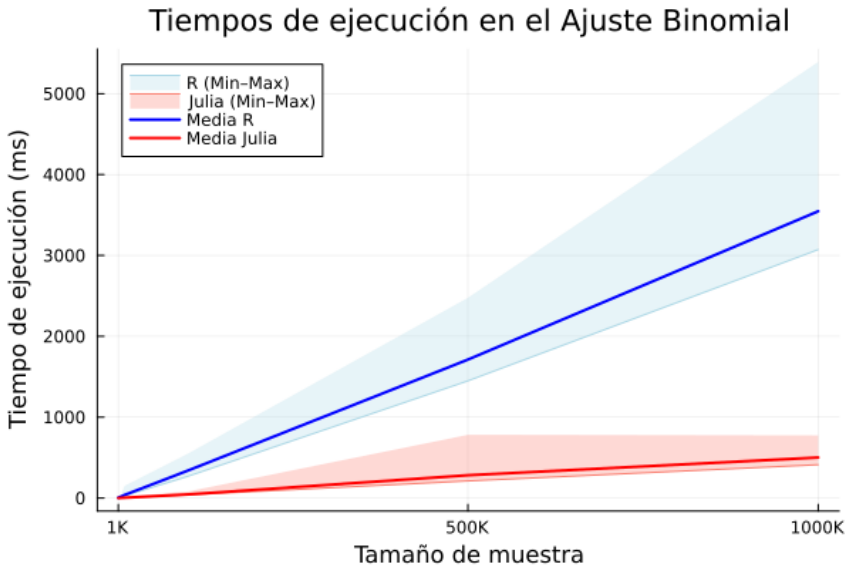
[2]:
```
datos_binomial = DataFrame(
    Lenguaje = repeat(["R", "Julia"], inner = 6, outer = 1),
    n = repeat([1000, 5000, 10000, 100000, 500000, 1000000],␣
 ↪outer = 2),

    Media = [
     4.97, 18.97, 38.86, 339.23, 1712.40, 3546.30,  # R
     0.66,  2.22,  4.28,  44.96,  282.28,  501.41],  # Julia
    Minimo = [
     3.66, 13.50, 26.75, 268.19, 1449.41, 3072.25,  # R
     0.45,  1.59,  3.13,  39.43,  211.77,  412.51],  # Julia
    Mediana = [
     4.67, 17.29, 32.36, 317.70, 1649.85, 3319.84,  # R
     0.51,  2.03,  3.77,  43.77,  236.72,  450.11], # Julia
```

```
    Maximo = [
     9.57, 43.98, 151.53, 547.11, 2476.27, 5391.78, # R
     1.95,  3.80,  20.19, 80.87,  781.83,  774.75 ]  # Julia
)
```

	Lenguaje	n	Media	Minimo	Mediana	Maximo
	String	Int64	Float64	Float64	Float64	Float64
1	R	1000	4.97	3.66	4.67	9.57
2	R	5000	18.97	13.5	17.29	43.98
3	R	10000	38.86	26.75	32.36	151.53
4	R	100000	339.23	268.19	317.7	547.11
5	R	500000	1712.4	1449.41	1649.85	2476.27
6	R	1000000	3546.3	3072.25	3319.84	5391.78
7	Julia	1000	0.66	0.45	0.51	1.95
8	Julia	5000	2.22	1.59	2.03	3.8
9	Julia	10000	4.28	3.13	3.77	20.19
10	Julia	100000	44.96	39.43	43.77	80.87
11	Julia	500000	282.28	211.77	236.72	781.83
12	Julia	1000000	501.41	412.51	450.11	774.75

[3]:
```
filtro_R = datos_binomial.Lenguaje .== "R"
filtro_Julia = datos_binomial.Lenguaje .== "Julia"
xticks = ([1000,  500000, 1000000], ["1K",  "500K","1000K"])

# Crear el gráfico con la banda de R
plot(datos_binomial.n[filtro_R],
    datos_binomial.Minimo[filtro_R],
    fillrange = datos_binomial.Maximo[filtro_R],
    fillalpha = 0.3,
    color = :lightblue, label = "R (Min-Max)",
    xlabel = "Tamaño de muestra",
    ylabel = "Tiempo de ejecución (ms)",
    legend = :topleft, xticks=xticks,
    title = "Tiempos de ejecución en el Ajuste Binomial"
)

# Banda para Julia
plot!(datos_binomial.n[filtro_Julia],
    datos_binomial.Minimo[filtro_Julia],
    fillrange = datos_binomial.Maximo[filtro_Julia],
    fillalpha = 0.3,
    color = :salmon, label = "Julia (Min-Max)"
)

# Línea de Media para R
plot!(datos_binomial.n[filtro_R],
    datos_binomial.Media[filtro_R],
```

```
    color = :blue, linewidth = 2,
    label = "Media R"
)

# Línea de Media para Julia
plot!(datos_binomial.n[filtro_Julia],
    datos_binomial.Media[filtro_Julia],
    color = :red, linewidth = 2,
    label = "Media Julia"
)
```



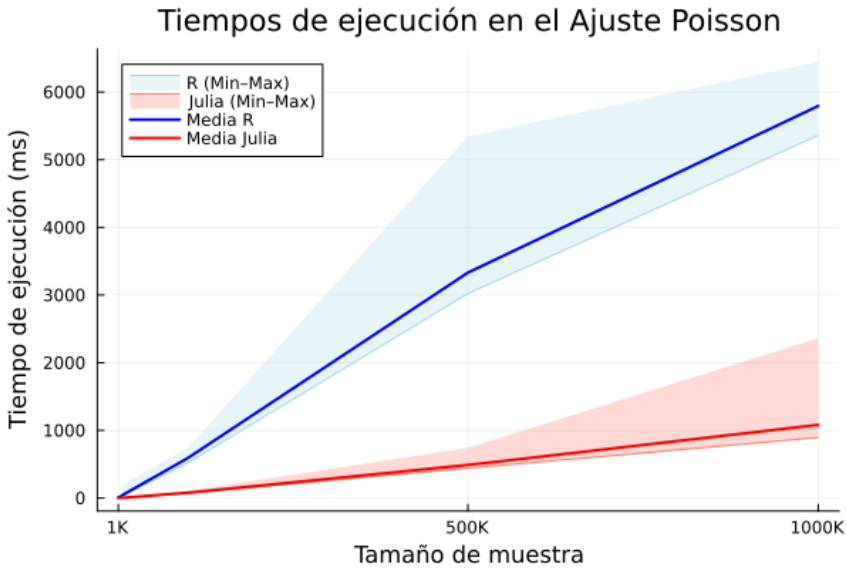
[4]:
```
datos_poisson = DataFrame(
    Lenguaje = repeat(["R", "Julia"], inner = 6, outer = 1),
    n = repeat([1000, 5000, 10000, 100000, 500000, 1000000],␣
↪outer = 2),
    Media = [
     8.26, 28.85, 62.71, 589.92, 3331.27, 5793.78,  # R
     0.77,  3.31,  6.65,  78.94,  489.45, 1082.00 ],  # Julia
    Minimo = [
     6.38, 23.21, 44.09, 518.43, 3021.41, 5359.04,  # R
     0.56,  2.62,  4.86,  74.32,  441.40,  894.77 ],  # Julia
    Mediana = [
     7.75, 28.16, 58.93, 572.32, 3309.03, 5755.10,  # R
     0.65,  2.98,  6.04,  77.92,  465.22,  962.89 ],  # Julia
```

```
    Maximo = [
     28.41, 54.03, 219.41, 736.99, 5342.47, 6443.58, # R
     1.99,  5.37,  11.11, 107.52, 742.27, 2362.00 ]   # Julia
)
```

	Lenguaje	n	Media	Minimo	Mediana	Maximo
	String	Int64	Float64	Float64	Float64	Float64
1	R	1000	8.26	6.38	7.75	28.41
2	R	5000	28.85	23.21	28.16	54.03
3	R	10000	62.71	44.09	58.93	219.41
4	R	100000	589.92	518.43	572.32	736.99
5	R	500000	3331.27	3021.41	3309.03	5342.47
6	R	1000000	5793.78	5359.04	5755.1	6443.58
7	Julia	1000	0.77	0.56	0.65	1.99
8	Julia	5000	3.31	2.62	2.98	5.37
9	Julia	10000	6.65	4.86	6.04	11.11
10	Julia	100000	78.94	74.32	77.92	107.52
11	Julia	500000	489.45	441.4	465.22	742.27
12	Julia	1000000	1082.0	894.77	962.89	2362.0

[5]:
```
# Crear el gráfico con la banda de R
plot(datos_poisson.n[filtro_R],
    datos_poisson.Minimo[filtro_R],
    fillrange = datos_poisson.Maximo[filtro_R],
    fillalpha = 0.3,
    color = :lightblue, label = "R (Min-Max)",
    xlabel = "Tamaño de muestra",
    ylabel = "Tiempo de ejecución (ms)",
    legend = :topleft, xticks=xticks,
    title = "Tiempos de ejecución en el Ajuste Poisson"
)

# Banda para Julia
plot!(datos_poisson.n[filtro_Julia],
datos_poisson.Minimo[filtro_Julia],
    fillrange = datos_poisson.Maximo[filtro_Julia],
    fillalpha = 0.3,
    color = :salmon, label = "Julia (Min-Max)"
)

# Línea de Media para R
plot!(datos_poisson.n[filtro_R],
    datos_poisson.Media[filtro_R],
    color = :blue, linewidth = 2,
    label = "Media R"
)
```

```
# Línea de Media para Julia
plot!(datos_poisson.n[filtro_Julia],
    datos_poisson.Media[filtro_Julia],
    color = :red, linewidth = 2,
    label = "Media Julia"
)
```



Bibliografía

[1] A. Agresti. *Foundations of Linear and Generalized Linear Models*. Wiley Series in Probability and Statistics. Wiley, 2015. ISBN: 9781118730058. URL: https://books.google.es/books?id=dgIzBgAAQBAJ.

[2] Jeff Bezanson et al. *Why We Created Julia*. Blog post. 2012. URL: https://julialang.org/blog/2012/02/why-we-created-julia/ (visitado 01-08-2024).

[3] Jeffry Chaves. *Compilación Justo a Tiempo (JIT)*. 2023. URL: https://jeffrychaves.com/diccionario/compilacion-justo-a-tiempo-jit/ (visitado 12-08-2024).

[4] JuliaPlots Community. *StatsPlots.jl Documentation*. 2025. URL: https://docs.juliaplots.org/latest/generated/statsplots/ (visitado 12-04-2025).

[5] The Julia contributors. *Julia 1.11 Highlights*. Blog post. 2024. URL: https://julialang.org/blog/2024/10/julia-1.11-highlights/ (visitado 01-06-2025).

[6] Wikipedia contributors. *Multiple dispatch*. 2024. URL: https://en.wikipedia.org/wiki/Multiple_dispatch (visitado 16-08-2024).

[7] A. J. Dobson y A. G. Barnett. *An Introduction to Generalized Linear Models*. 4th. Chapman y Hall/CRC, 2018. DOI: 10.1201/9781315182780. URL: https://doi.org/10.1201/9781315182780.

[8] EDteam. *¿Qué son los lenguajes tipados y no tipados? Explicación sencilla*. 2023. URL: https://ed.team/blog/que-son-los-lenguajes-tipados-y-no-tipados-explicacion-sencilla (visitado 12-08-2024).

[9] John Fox, Sanford Weisberg y Brad Price. *carData: Companion to Applied Regression Data Sets*. R package version 3.0-5. 2022. URL: https://CRAN.R-project.org/package=carData.

[10] Gragusa. *About $dof_{r}esidual$*. GitHub issue #509, JuliaStats/GLM.jl. 2022. URL: https://github.com/JuliaStats/GLM.jl/issues/509 (visitado 10-06-2025).

[11] Leah Hanson. *Learn Julia in Y Minutes*. 2024. URL: https://learnxinyminutes.com/docs/es-es/julia-es/ (visitado 17-09-2024).

[12] IBM. *JIT Compiler*. 2023. URL: https://www.ibm.com/docs/es/sdk-java-technology/8?topic=reference-jit-compiler (visitado 12-08-2024).

[13] Julia developers. *Announcing the release of Julia 1.0*. Blog post. 2018. URL: https://julialang.org/blog/2018/08/one-point-zero/ (visitado 01-08-2024).

[14] *JuliaHub: Julia for Cloud Computing*. 2024. URL: https://juliahub.com/ (visitado 05-10-2024).

[15] JuliaStats. *Distributions.jl Documentation*. 2025. URL: https://juliastats.org/Distributions.jl/latest/ (visitado 19-04-2025).

[16] JuliaStats. *Statistics.jl Documentation*. 2025. URL: https://juliastats.org/Statistics.jl/dev/#Statistics.stdm (visitado 22-03-2025).

[17] Kaggle. *Datasets on Kaggle*. URL: https://www.kaggle.com/datasets (visitado 12-02-2025).

[18] Tomas Kalibera y Luke Tierney. *Just-In-Time Compilation in R*. UseR! DSC 2017 Presentation. 2017. URL: https://www.r-project.org/dsc/2017/slides/JIT_for_R.pdf (visitado 08-07-2025).

[19] The Julia Programming Language. *Download the Julia Programming Language (in under 2.5 minutes)*. 2021. URL: https://www.youtube.com/watch?v=t67TGcf4SmM (visitado 23-09-2024).

[20] The Julia Programming Language. *Programación básica con Julia. Capitulo 1. Primeros Pasos*. 2022. URL: https://hedero.webs.upv.es/julia-basico/1-primerospasos/ (visitado 05-10-2024).

[21] LLVM Project. *LLVM Logo and Trademark Usage*. 2024. URL: https://llvm.org/Logo.html (visitado 12-08-2024).

[22] LLVM Project. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. 2024. URL: https://llvm.org/ (visitado 12-08-2024).

[23] Adrián Rodríguez Mira. *Lenguaje de programacion Julia: ideal para Machine Learning*. Blog. 2020. URL: https://www.tokioschool.com/noticias/lenguaje-programacion-julia/ (visitado 31-07-2024).

[24] *Paquetes de Julia*. 2024. URL: https://juliapackages.com/ (visitado 05-10-2024).

[25] Coursera Staff. *Julia vs. R: Comparación entre dos lenguajes de programación*. 2025. URL: https://www.coursera.org/articles/julia-vs-r (visitado 24-05-2025).

[26] The Julia Language. *Julia Downloads: Current Stable Release*. 2024. URL: https://julialang.org/downloads/#current_stable_release (visitado 23-09-2024).

[27] The Julia Language. *The Julia Programming Language*. 2024. URL: https://julialang.org (visitado 31-07-2024).

[28] The Julia Project. *Methods · The Julia Language, version 1.10.4*. 2024. URL: https://docs.julialang.org/en/v1/manual/methods/ (visitado 16-08-2024).

[29] The Julia Project. *The Julia Programming Language Documentation, Version 1.10.5*. 2024. URL: https://docs.julialang.org/en/v1/ (visitado 18-09-2024).

[30] *Visual Studio Code*. Editor de código fuente desarrollado por Microsoft. URL: https://code.visualstudio.com/ (visitado 29-11-2024).