

INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO



Cómputo estadístico: Julia, Python y R

TESIS

QUE PARA OBTENER EL TÍTULO DE

LICENCIADA EN MATEMÁTICAS APLICADAS

PRESENTA

VALERIA AURORA PÉREZ CHÁVEZ

ASESOR: ERNESTO JUVENAL BARRIOS ZAMUDIO

«Con fundamento en los artículos 21 y 27 de la Ley Federal del Derecho de Autor y como titular de los derechos moral y patrimonial de la obra titulada “**Cómputo estadístico: Julia, Python y R**”, otorgo de manera gratuita y permanente al Instituto Tecnológico Autónomo de México y a la Biblioteca Raúl Baillères Jr., la autorización para que fijen la obra en cualquier medio, incluido el electrónico, y la divulguen entre sus usuarios, profesores, estudiantes o terceras personas, sin que pueda percibir por tal divulgación una contraprestación.»

VALERIA AURORA PÉREZ CHÁVEZ

FECHA

FIRMA

Agradecimientos

Agradezco a facu por ser tan chingona y a Mike por pasarme el formato. Salu2.

A Nora, que su constante apoyo en la última etapa fue crucial para que no perdiera la cabeza y me mantuviera enfocada. Muchas gracias, nunca lo olvidaré.

EL SERCH

Índice general

1. Introducción	1
2. Julia	4
2.1. Reproducibilidad	5
2.2. Instalación	6
2.3. Símbolo del sistema	6
2.3.1. <i>Multithreading</i>	7
2.4. Básicos de Julia	8
2.4.1. Operaciones básicas	8
2.4.2. <i>Strings</i> (secuencias de caracteres)	10
2.4.3. Funciones	11
2.4.4. Vectores y Matrices	12
2.4.5. Instalación de un paquete	14
2.4.6. <i>DataFrames</i>	15
2.4.7. Análisis de regresión	17
3. Python	22
3.1. Listas	23
3.2. Paquetes	24
3.2.1. NumPy	24

3.2.2.	pandas	26
3.2.3.	os	26
3.2.4.	scikit-learn	26
3.2.5.	itertools	27
3.3.	Jupyter Notebook	28
3.3.1.	Julia	29
3.3.2.	R	29
4.	R	33
4.1.	Función lm	34
5.	Ajuste de polinomios	37
5.1.	El problema	38
5.1.1.	Método de Mínimos Cuadrados	38
5.2.	Los datos	39
5.3.	Planteamiento del problema	39
5.4.	Número de condición y precisión de la solución	41
5.5.	Solución usando Julia	45
5.5.1.	<i>GLM</i>	46
5.5.2.	Factorización QR versión económica	47
5.5.3.	Descomposición de valores singulares	50
5.5.4.	<i>Polynomials</i>	54
5.6.	Solución usando R	55
5.7.	Solución usando Python	57
5.8.	Resultados y Conclusiones	59
6.	Modelos de Regresión Lineal	63
6.1.	El modelo	64
6.2.	Los datos	66
6.3.	Planteamiento del problema	67

6.4. Factores y Sub-ajustes	71
6.4.1. Código en Julia	73
6.5. Resultados y Conclusiones	77
7. Discriminación de modelos	82
7.1. Preliminares y definiciones	83
7.2. Metodología General	85
7.3. Criterio MD	87
7.4. Algoritmo de intercambio	92
7.4.1. Algoritmo básico	93
7.4.2. Incorporación de excursiones	94
7.4.3. Puntos candidatos	95
7.5. Función MDopt	95
7.6. Implementación de MDopt en los lenguajes	97
7.6.1. JuliaCall	99
7.6.2. reticulate	101
7.7. Ejemplos y resultados	102
7.7.1. Ejemplo 1 - Proceso de moldeo por inyección . .	102
7.7.2. Ejemplo 2	112
7.8. Conclusiones	119
8. Conclusiones	121
A. Código	125
A.1. Código del modelo de regresión lineal que utiliza el Censo	125
A.1.1. R	125
A.1.2. Python	129
A.2. Tiempos de ejecución de los modelos de regresión lineal múltiple utilizados en el Censo	134

Capítulo 1

Introducción

“I always promote Julia among friends and colleagues in Latin America, even when it has been difficult to convince them because of the scarce resources of Julia in Spanish. I firmly believe in open access knowledge without barriers (either language barriers, accessibility, or others), and I will always advocate for that”, [The Julia Community \(2022\)](#). Las palabras de la chilena Pamela Bustamente, usuaria de Julia, engloban el motivo del desarrollo de esta tesis.

Mi camino con Julia comenzó a principios del 2021 cuando tuve la oportunidad de trabajar en el Instituto Mexicano del Seguro Social (IMSS). Julia fue la herramienta que utilice para desarrollar un proyecto fundamentado en estadística bayesiana y requería de una gran cantidad de simulaciones. No tarde mucho tiempo en encontrarme con las dificultades que menciona Pamela y algunas más. Sin embargo, Julia debe tener otras cualidades que frecuentemente lo destaquen como un lenguaje prometedor que cada día va tomando más fuerza en la comunidad de programadores.

Al principio, dichas cualidades eran un misterio para mí. Mi

interrogante principal fue sobre la necesidad de crear un nuevo lenguaje. ¿Por qué usar Julia y no Python o R?, ¿Cuál fue la motivación de su creación? y, después de encontrarme con una falta de recursos, ¿Cómo es posible que 10 años más tarde exista tan poca ayuda de este lenguaje? Esta tesis es mi esfuerzo por mostrar un nuevo lenguaje, sus alcances y hacer una comparativa con lo que ya se conoce. Asimismo, mi trabajo queda como punto de partida para futuros usuarios hispanohablantes e invitación a utilizar Julia.

Este trabajo no es un manual de Julia ni de ningún otro lenguaje. Eso ya existe. Lo que se busca es explicar pros y contras que se encontraron al utilizar Julia, Python y R en tres ejercicios distintos. El título de la tesis se refiere justamente a dicha comparación ya que se omite mencionar teoría de cómputo estadístico en sí. Python y R se eligieron por su uso y extenso desarrollo en la ciencia de datos, el tema principal de este trabajo. Asimismo, el uso de los tres lenguajes permite exponer la experiencia de usuario al trabajar con los tres lenguajes.

La tesis se divide en dos partes. El propósito de la primera parte es proveer una imagen general de los tres lenguajes así como las funciones que se utilizaron en la segunda parte. Primero, se expone la instalación de Julia en un sistema operativo Windows para después explicar aspectos básicos del lenguaje. También, se da una introducción a dos paquetes fundamentales para este trabajo. Esto se hace pensando que Julia es el lenguaje más reciente y se busca que el lector navegue fácilmente por el código mostrado. Después, se presentan los paquetes y funciones que se utilizaron en Python y en R de manera más general suponiendo que el lector ya está familiarizado con ellos.

La segunda parte de la tesis consta de tres proyectos cuyo objetivo es mostrar un aspecto diferente en los lenguajes. El primer proyecto

toma datos del National Institute of Standards and Technology (NIST) para medir la precisión numérica de cada lenguaje al hacer el ajuste de un polinomio de grado 10. El segundo ejercicio usa los datos del Censo de Población y Vivienda de México del 2020 realizado por el Instituto Nacional de Estadística y Geografía (INEGI). El objetivo de este ejercicio es el manejo y manipulación de una gran cantidad de datos. Finalmente, el tercer proyecto presenta la programación de un algoritmo de búsqueda que se utiliza en la discriminación de modelos en diseños de experimentos. El enfoque de este ejercicio es la búsqueda de modelos, no el diseño de experimentos. Los cálculos son más intensivos por lo que el ejemplo busca medir la capacidad y velocidad de cómputo de los lenguajes.

En suma, el capítulo 2 presenta una primera introducción a Julia. Los capítulos 3 y 4 describen los paquetes y funciones utilizados en Python y R, respectivamente. El capítulo 5 muestra la primera aplicación de los lenguajes al resolver un problema propuesto por NIST. Posteriormente, en el capítulo 6 se presenta el uso de modelos de regresión lineal en una gran cantidad de datos. El capítulo 7 expone un algoritmo utilizado para la búsqueda de modelos en el contexto de diseño de experimentos. Finalmente, en el capítulo 8 se presenta un resumen del trabajo así como la experiencia de usuario y opinión personal acerca del uso de Julia, Python y R.

A continuación, se comienza este trabajo con la presentación de Julia.

Capítulo 2

Julia

“Julia es un lenguaje de programación gratis y de código abierto desarrollado por Jeff Bezanson, Alan Edelman, Viral B. Shan y Stefan Karpinski en el MIT”, [Carrone et al. \(2021\)](#). Su propósito general es ser tan rápido como C, mientras mantiene la facilidad de lenguaje de R o Python. Es una combinación de sintaxis simple con alto rendimiento computacional. Su slogan es “Julia se ve como Python, se siente como Lisp, corre como Fortran”, [Carrone et al. \(2021\)](#).

La creación de Julia generó una gran impresión en la comunidad científica ya que busca resolver lo que se conoce como la “Ousterhout’s dichotomy” (la dicotomía de Ousterhout’s). El problema establece que los lenguajes de programación se pueden dividir en dos categorías: programación de sistemas (*system programming*) y lenguaje de secuencias de comandos (*scripting languages*). Los lenguajes de la primera categoría tienden a ser difíciles de usar pero muy rápidos mientras que los que forman parte de la segunda categoría son fáciles de usar pero lentos en ejecución. Julia busca tomar lo mejor de ambas categorías y utiliza un compilador *just in time (JIT)*. Esto significa

que Julia se compila al momento de la ejecución y guarda el código con los tipos de objeto que encuentra. Si el mismo código se ejecuta con objetos de diferente tipo, Julia recompila y guarda las instrucciones nuevas. Esto hace que el lenguaje no invierta tiempo en decidir el tipo de objeto con el que se está trabajando.

Esta combinación de características hace que Julia sea un lenguaje de programación que ha tomado fuerza en la comunidad científica últimamente. Ya que es un lenguaje poco conocido, en esta sección se explica como instalar Julia en una computadora con sistema operativo Windows y algunos de los básicos del lenguaje.

2.1. Reproducibilidad

Antes de empezar, se enfatiza que esta tesis es completamente reproducible. Peng y Hicks del departamento de bioestadística en la Universidad John Hopkins definen “un análisis de datos publicado es reproducible si el conjunto de datos y el código utilizados para crear el análisis de datos está disponible para que otros lo analicen y estudien de manera independiente”, Peng and Hicks (2021). A pesar de que en el artículo enfatizan que esta definición puede ser un tanto ambigua, sí resaltan que la reproducibilidad es un medio para revisar y, posteriormente, confiar en el análisis de otros.

Uno de los beneficios de tener un trabajo de investigación reproducible es que “los lectores obtienen los datos y el código computacional, los cuales son valiosos al grado de que pueden ser reutilizados o rediseñados para futuros estudios o investigaciones”, Peng and Hicks (2021). El código programado para esta tesis se encuentra en la plataforma de GitHub, específicamente en la liga https://github.com/valperez/Tesis_Julia. Los datos utilizados también

se pueden encontrar en la liga anterior o en la fuente que se indica al mencionarlos.

2.2. Instalación

Este trabajo se presenta como si fuera hecho en un ambiente de Windows. La instalación y uso en los sistemas Mac y Linux es muy similar y no se mencionará.

Al momento de la escritura de esta tesis la versión de Julia disponible es la v1.7.3. El primer paso es descargar Julia desde la página <https://julialang.org/downloads/>.

Para el sistema operativo Windows se tiene la opción de un instalador de 64-bits o uno de 32-bits. El tipo de sistema que tiene un ordenador se verifica en Start > Configuración > Sistema > Acerca de. Se debe seleccionar el installer y no el portable. Una vez descargado, se debe seleccionar el archivo .exe y seguir los pasos de instalación.

2.3. Símbolo del sistema

Una vez instalado, se puede ejecutar Julia desde el símbolo del sistema o desde alguna interfaz gráfica como Atom, Visual Studio Code o Jupyter Notebook. De este último se presenta una explicación detallada en el capítulo 3.3.

Una de las ventajas de utilizar Julia desde el símbolo del sistema (también conocido como *Command Prompt* o *cmd*) es que se pueden controlar algunos parámetros del lenguaje. Mi sugerencia es que se comience a usar Julia directo desde la interfaz nativa. Posteriormente, cuando se entienda lo básico y los programas generados requieran un mayor nivel computacional, entonces se puede migrar a usar el *cmd*

para correr Julia.

2.3.1. *Multithreading*

Una de las razones por la que Julia tiene gran velocidad es por su capacidad para multihilo (*multithreading* en inglés). Esto significa que puede correr diferentes tareas de manera simultánea en varios hilos. La meta de los autores de Julia fue crear un lenguaje de programación con un rendimiento tan alto que pudiera ejecutar varias tareas de manera simultáneas. Debido a que uno de los objetivos de esta tesis es mostrar la eficiencia y velocidad de Julia, es crucial conocer la característica del *multithreading* y cómo utilizarla.

Si se está ejecutando Julia por medio del `cmd` es necesario modificar la cantidad de hilos que se va a utilizar antes de ejecutar Julia. En Windows, esto se modifica con el comando `set Julia_NUM_THREADS=4` (Bezanson et al., 2014). Si se está trabajando con otro sistema operativo, la página <https://docs.julialang.org/en/v1/manual/multi-threading/> puede ser una guía para modificar la cantidad de hilos. En este ejemplo se cambiaron los hilos a 4, pero se puede asignar cualquier número. Sin embargo, se recomienda que éste no exceda de la cantidad de procesadores lógicos de la computadora.

Si se está usando Julia en algún otro entorno de programación la modificación del número de hilos se hace de forma diferente. Cada programa tiene su manera de hacerlo y usualmente las instrucciones vienen en el manual del mismo. Para observar que el cambio se ejecutó de manera correcta (independientemente del editor de texto elegida) basta con correr el comando `Threads.nthreads()` y observar que la respuesta sea el número deseado.

2.4. Básicos de Julia

“Como el compilador de Julia es diferente a los intérpretes usados para lenguajes como Python o R se puede percibir que el funcionamiento de Julia no es intuitivo en un principio. Una vez que se entienda como funciona Julia, es fácil escribir código que es casi tan rápido como C”, [Bezanson et al. \(2014\)](#). En esta sección se presenta una introducción a la sintaxis del lenguaje que se tomó del manual oficial de Julia, [Bezanson et al. \(2014\)](#).

La asignación de variables se hace con un signo de igualdad =. El ejemplo más sencillo de esto es ejecutar

```
julia> x = 2  
2
```

donde se asigna a x el valor de 2.

2.4.1. Operaciones básicas

La tabla [2.1](#) muestra la sintaxis usada para las operaciones básicas en Julia.

Expresión	Nombre	Descripción
$+x$	suma unaria	la operación identidad
$-x$	resta unaria	asigna a los valores sus inversos aditivos
$x + y$	suma binaria	realiza adición
$x - y$	resta binaria	realiza sustracción
$x * y$	multiplicación	realiza multiplicación
x / y	división	realiza divisiones
$x \div y$	división de enteros	x/y truncado a un entero
$x \setminus y$	división inversa	equivalente a dividir y / x
$x \wedge y$	potencia	eleva x a la potencia y
$x \% y$	residuo	equivalente a <code>rem(x,y)</code>
$!x$	negación	realiza lo contrario de x
$x \& \& y$	<i>and</i> lógico	verifica si x y y se cumplen
$x y$	<i>or</i> lógico	verifica si al menos uno, x o y , se cumplen

Tabla 2.1. Operaciones básicas en Julia

Operaciones básicas en vectores

En Julia cada operación binaria tiene su correspondiente operación punto (*dot operation* en inglés). Estas funciones están definidas para efectuarse elemento por elemento en vectores y matrices. Para ejecutarlas basta agregar un punto antes del operador binario. Por ejemplo,

```
julia> [1 9 9 7] .^ 2
1x4 Matrix{Int64}:
1 81 81 49
```

eleva cada uno de los elementos del vector al cuadrado.

Julia maneja los números imaginarios utilizando el sufijo `im`. Sin embargo, no se utilizaron en este trabajo así que se omitirá dar mayor explicación.

2.4.2. *Strings* (secuencias de caracteres)

Además de números, Julia puede asignar una secuencia de caracteres (mejor conocido como *string*) a variables usando comillas dobles. Se puede acceder a caracteres específicos de un string utilizando corchetes cuadrados `[]` y a cadenas seguidas de caracteres usando dos puntos `:`. Por ejemplo,

```
julia> string = "Esta tesis es interesante"
julia> string[6]
't': ASCII/Unicode U+0074 (category Ll: Letter, lowercase)

julia> string[4:8]
"a tes"
```

Además, Julia tiene la opción de concatenación de múltiples strings. Esto se hace utilizando un asterisco `*` que une cada uno de los strings. Por ejemplo,

```
julia> grado = "licenciada"
julia> nexa = "en"
julia> carrera = "matematicas aplicadas"
julia> espacio = " "
julia> grado*espacio*nexo*espacio*carrera
"licenciada en matematicas aplicadas"
```


2.4.3. Funciones

En Julia una función es un objeto que asigna una tupla de argumentos a un valor de retorno (Bezanson et al., 2014). La sintaxis básica para definir funciones en Julia es

```
julia> function nombre(x, y)
    instruccion_1
    instruccion_2
    .
    .
    .
    instruccion_k
end
```

Además, se puede agregar la palabra **return** para que la función regrese un valor. Por ejemplo, si se quisiera tener una función a la que se le dan dos números y regrese el número mayor, las instrucciones serían de la forma:

```
julia> function numero_mayor(x, y)
    if (x > y)
        return x
    else
        return y
    end
end
```

Para llamar a la función basta con escribir `numero_mayor(x, y)` asignando o sustituyendo valores por x y y . Por ejemplo,

```
julia> numero_mayor(4, 9)
9
```

2.4.4. Vectores y Matrices

Un vector columna de n componentes se define como un conjunto ordenado de n números escritos de la siguiente manera:

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

En Julia se utilizan los corchetes cuadrados [] y las comas para definir un vector columna. Por ejemplo,

```
julia> A = [1, 9, 9, 7]
4-element Vector{Int64}
 1
 9
 9
 7
```

da como resultado un vector de 4 elementos de tipo Int64. Julia es un lenguaje exigente con los tipos de objetos, por lo que aprender las características y funciones singulares de cada objeto es uno de los atributos que forman un buen usuario.

Si se quisiera definir un vector renglón se haría exactamente lo mismo excepto que se omitiría el uso de las comas. En el ejemplo anterior, Julia tomó el objeto A como una matriz, no como un vector. Una matriz A

de $m \times n$ es un arreglo rectangular de mn números dispuestos en m renglones y n columnas.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2j} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \dots & a_{ij} & \dots & a_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mj} & \dots & a_{mn} \end{pmatrix}$$

En Julia, hay dos formas de definir matrices. La primera es utilizando los corchetes cuadrados [] para comenzar y terminar la matriz. Las columnas están separadas por espacios y las filas por punto y coma. La segunda opción es similar a la primera con la diferencia de que en lugar de punto y coma se cambia de renglón. Esta opción puede parecer tediosa ya que requiere que las columnas estén alineadas. Sin embargo, es una forma más visual de ver las matrices. Los siguientes comandos muestran ambas opciones.

```
julia> A_1 = [1 2 3; 4 5 6]
2x3 Matrix{Int64}
1 2 3
4 5 6
julia> A_2 = [1 2 3
               4 5 6]
2x3 Matrix{Int64}
1 2 3
4 5 6
```

De manera análoga con los vectores, para llamar un solo elemento de la matriz se utilizan los corchetes cuadrados. Continuando con el ejemplo

anterior, para obtener el número 5 de la matriz `A_2`, se introduciría el comando

```
julia> A_2[2, 2]
5
```

Julia necesita de la instalación del paquete `LinearAlgebra` para hacer operaciones básicas y factorización de matrices. El catálogo de funciones es bastante extenso para incluirlo en este trabajo, pero se puede encontrar en <https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/>.

2.4.5. Instalación de un paquete

Similar a los paquetes en R, Julia utiliza paquetes para expandir su funcionalidad. Julia usa Git como repositorio para sí mismo y para sus paquetes. Un paquete registrado en Julia tiene un manual donde se explica el funcionamiento del mismo y un repositorio en GitHub donde cualquier usuario puede reportar errores encontrados. La lista completa de paquetes registrados en Julia se encuentra en <https://juliapackages.com/>. En esta sección se explica la instalación y uso de los mismos.

El único paquete que ya viene por default en la instalación de Julia es `Pkg`, ya que su función es instalar otros paquetes. El comando `using` activa un paquete ya descargado, mientras que `Pkg.add()` agrega un paquete nuevo. A continuación se presenta la guía básica para descargar cualquier paquete en Julia usando como ejemplo al ya mencionado `LinearAlgebra`.

```
using Pkg
Pkg.add("LinearAlgebra")
using LinearAlgebra
```

La instalación de un paquete solo se debe hacer una vez. Si se requiere usar en alguna sesión posterior basta con ejecutar la instrucción `using` y el nombre del paquete. En las siguientes secciones se explica y ejemplifica el uso de dos paquetes muy usados en esta tesis.

2.4.6. *DataFrames*

Un *dataframe* es una tabla estructurada de dos dimensiones que se usa para tabular distintos tipos de datos. Julia tiene un paquete llamado **DataFrames** que permite trabajar con dataframes de creación propia o exportados de alguna fuente externa.

Crear un dataframe

Los dataframes pueden ser usados para manejar grandes cantidades de información exportada o pueden ser creaciones propias en el lenguaje. Para crear un dataframe desde cero en Julia se debe comenzar la instrucción con la palabra **DataFrame** y abrir un paréntesis. Después, se escribe el nombre de la primera columna, un signo de igualdad y los datos que corresponden a esa variable. Se repite el proceso con la cantidad de columnas que se requieran. Por ejemplo, para hacer un dataframe con las claves únicas y los nombres de cinco mujeres el código sería el siguiente:

```
julia> using DataFrames
julia> df = DataFrame(id = 1:5,
                      nombre = ["Valeria", "Paula",
                                "María José", "Sofía", "Mónica"])
```

Los nombres de las columnas de un dataframe se vuelven una especie de atributo. Para referirse a la columna “col” del dataframe

“df” basta escribir `df.col`. Si se quisiera agregar una columna nueva se deben asignar datos a `df.colNueva`. Por ejemplo, si quisiera agregar una columna llamada `color` al dataframe del ejemplo anterior, el código sería el siguiente:

```
julia> df.color = ["morado", "azul", "verde",  
                  "negro", "rojo"]
```

Importar datos en un dataframe

Como ya se mencionó, los dataframes son utilizados para contener grandes cantidades de información. Usualmente esta información no es generada en Julia por lo que hay importarla. Esto se puede hacer con el paquete `CSV`. La descarga de este paquete se consigue siguiendo los pasos descritos en la sección 2.4.5.

Una vez instalado el programa, se necesita utilizar el comando `CSV.read` y la ruta de la ubicación del archivo para importar los datos. La figura 2.1 muestra el resultado de leer el archivo llamada `ejemplo.csv`.

```
julia> using CSV, DataFrames  
julia> df = CSV.read("~/ejemplo.csv", DataFrame)
```

Los dataframes tienen una cantidad vasta de funciones que incluyen agregar o eliminar información, seleccionar columnas o renglones, transformar su contenido, etc. Las especificaciones de dichas funciones se explican con mayor profundidad en el manual oficial del paquete en la página <https://dataframes.juliadata.org/stable/>.

```
julia> using CSV, DataFrames

julia> df = CSV.read("C:/Users/Valeria/Documents/ITAM/Tesis/Julia con R/ejemplo.csv", DataFrame)
5x6 DataFrame
  Row  id      nombre  color  deporte  lugar_residencia  estatura
  Int64 String15 String7 String15 String31      Float64
1     1   Valeria   morado  atletismo  Nuevo Leon        1.7
2     2    Paula    azul    hiking    Estado de Mexico  1.75
3     3  Maria Jose verde  atletismo  Ciudad de Mexico  1.63
4     4    Sofia   negro  funcional  Oaxaca            1.66
5     5   Monica    rojo    baile     Veracruz          1.58
```

Figura 2.1. Ejemplo de importación de un dataframe

2.4.7. Análisis de regresión

El análisis de regresión es una herramienta estadística para estudiar las relaciones entre distintas variables. “Regresión es un método que permite a los investigadores resumir como predicciones o valores promedio de un resultado varían a través de variables individuales definidas como predictores o regresores”, [Gelman et al. \(2021\)](#).

De forma concisa, la regresión es una expresión que intenta explicar como una variable depende de otras. En Julia, esto se puede hacer con ayuda del paquete GLM que significa *Generalized Linear Models* o modelos lineales generalizados. Una de sus funciones es ajustar modelos lineales, pero se puede usar para modelos más complejos. Como todos los paquetes, primero se debe instalar con los pasos descritos en la sección 2.4.5.

En este paquete una de las funciones principales se llama `lm` que se utiliza para ajustar un modelo lineal a un conjunto de datos. En el manual oficial [Bates et al. \(2022\)](#) está descrita la manera en que se pueden generar modelos más avanzados. Uno de los autores de este paquete, Douglas Bates tiene una larga trayectoria en el cómputo estadístico. En 1992 publicó el libro llamado *Statistical Models in S*

cuyo autor principal es John Chambers, otro grande del cómputo estadístico. Además, Bates es parte del llamado *R Core Team* que es el grupo de colaboradores con acceso a la fuente del lenguaje R. Uno de sus trabajos más recientes es desarrollar modelos estadísticos en Julia creando así el paquete GLM.

La función `lm` se utiliza en repetidas ocasiones en este trabajo, por lo que se explica a continuación. La función es `lm(formula, data, allowrankdeficient=false; [wts::AbstractVector], dropcollinear::Bool=true)` donde

- **formula:** usa los nombres de las columnas del dataframe para referirse a las variables predictoras. Debe ser un objeto de tipo `formula`.
- **data:** el dataframe que contenga los datos de los predictores de la fórmula.
- **allowrankdeficient:** permite o no que la matriz de datos tenga rango completo.
- **wts:** es un vector que especifica la ponderación de las observaciones.
- **dropcollinear:** controla si `lm` acepta una matriz que no sea de rango completo. Si el parámetro se define como `true` entonces solo se usan el conjunto de las primeras columnas linealmente dependientes.

Regresión lineal simple

El modelo de regresión lineal más simple es el que tiene un solo predictor

$$y = a + bx + \epsilon.$$

Para ejemplificar este modelo de regresión se usaron los datos del capítulo 7.1 del libro escrito por [Gelman et al. \(2021\)](#). La información fue recabada por Douglas Hibbs con el objetivo de predecir las elecciones de Estados Unidos basándose solamente en el crecimiento económico. Los datos se ven de la siguiente manera:

```
julia> elections = CSV.read("C:/Users/Valeria/Documents/ITAM/Tesis/Julia con R/Regression_and_other_stories/ROS-Examples-master/ROS-Examples-master/ElectionsEconomy/data/hibbs.csv", DataFrame)
16x5 DataFrame
```

Row	year Int64	growth Float64	vote Float64	inc_party_candidate String15	other_candidate String15
1	1952	2.4	44.6	Stevenson	Eisenhower
2	1956	2.89	57.76	Eisenhower	Stevenson
3	1960	0.85	49.91	Nixon	Kennedy
4	1964	4.21	61.34	Johnson	Goldwater
5	1968	3.02	49.6	Humphrey	Nixon
6	1972	3.62	61.79	Nixon	McGovern
7	1976	1.08	48.95	Ford	Carter
8	1980	-0.39	44.7	Carter	Reagan
9	1984	3.86	59.17	Reagan	Mondale
10	1988	2.27	53.94	Bush, Sr.	Dukakis
11	1992	0.38	46.55	Bush, Sr.	Clinton
12	1996	1.04	54.74	Clinton	Dole
13	2000	2.36	50.27	Gore	Bush, Jr.
14	2004	1.72	51.24	Bush, Jr.	Kerry
15	2008	0.1	46.32	McCain	Obama
16	2012	0.95	52.0	Obama	Romney

Figura 2.2. Encabezado de los datos sobre las elecciones en EUA recabados por Douglas Hibbs

En este modelo se busca que el voto sea resultado del crecimiento económico. El código para hacer esto, después de la importación de datos mostrado en la figura 2.2 es

```
elections_lm = lm(@formula(vote ~ growth), elections)
```

El resultado es una tabla con los coeficientes, su desviación estándar, el valor t , el valor $-p$ y su intervalo de confianza del 95 %. En este ejemplo, el resultado que da Julia es $y = 46.3 + 3.1x$, el cual coincide con los valores reportados en el libro.

Regresión lineal múltiple

La regresión lineal múltiple es el caso general de la regresión lineal simple. La diferencia es que en el primero hay múltiples predictores que deben cumplir ciertos criterios. [Gelman et al. \(2021\)](#) define este tipo de regresión como

$$y_i = \beta_1 X_{i1} + \cdots + \beta_k X_{ik} + \epsilon_i, \text{ para } i = 1, \dots, n$$

donde los errores ϵ_i son independientes e idénticamente distribuidos de manera normal con media 0 y varianza σ^2 . La representación matricial equivalente es

$$y_i = X_i \beta + \epsilon_i, \text{ para } i = 1, \dots, n \quad (2.1)$$

donde X es una matriz de $n \times k$ cuyo renglón es X_i .

Para ejemplificar este tipo de modelo se uso un ejemplo que consta de dos predictores y la interacción entre ellos. Esta vez se utilizaron los datos del capítulo 10.3 de [Gelman et al. \(2021\)](#) que muestran la relación entre los resultados de exámenes de niños (`kid_score`), el coeficiente intelectual IQ de sus madres (`mom_iq`) y si sus madres terminaron o no la preparatoria (`mom_hs`).

Se buscó determinar si existe una relación significativa entre la educación y el coeficiente de las madres con los resultados de los exámenes de sus hijos. Por lo tanto, los predictores son las variables en relación con la madre mientras que la respuesta es el desempeño de los niños. El código en Julia se ve de la siguiente manera

```
julia> using DataFrames, GLM, CSV
julia> data_kid = CSV.read("~/Tesis/data/kid_iq.csv",
    DataFrame)
```

```
julia> fm = @formula(kid_score ~ mom_hs + mom_iq +  
    mom_hs*mom_iq)  
julia> kidscore_lm = lm(fm, data_kid)
```

Que da como resultado el modelo ajustado

```
kid_score = -11.48 + 51.26* mom_hs + 0.97*mom_iq -  
0.48*mom_hs*mom_iq
```

Uno de los aspectos por resaltar en este ejemplo es que, similar a como se hace en R, para incluir la relación entre dos predictores se usa un asterisco entre ellos al momento de definir la fórmula de la regresión.

En el caso donde alguno de los regresores sea de tipo categórico, la fórmula se mantiene igual pero es necesario hacerle cambios a la base de datos en sí. Si Julia no reconoce estas columnas como categóricas entonces se debe cambiar su tipo en el dataframe. Se explica este caso más a fondo en el capítulo [6.4](#).

Por otro lado, se puede intentar usar el paquete **CSVFiles** para leer los archivos ya que hace mejor trabajo identificando el tipo de variables. Sin embargo, este paquete todavía está en desarrollo.

Capítulo 3

Python

“Python es un lenguaje de programación que permite trabajar rápido e integrar sistemas más eficientemente” es una frase que podría parecer sencilla, pero es esa misma simplicidad la que la hace destacar como lo primero que se observa en la página oficial de Python <https://www.python.org/>. El creador de Python, Guido van Rossum, comenzó a desarrollar el lenguaje a finales de los ochentas para, finalmente, hacerlo público en 1991. Esto lo hace un lenguaje más antiguo que Julia y R. Sin embargo, su desarrollo y efectividad ha sido tal que empresas líderes mundiales en tecnología como Youtube y Google lo utilizan hoy en día.

Los recursos de apoyo disponibles para el uso de Python son vastos y de todos los medios posibles. Se decidió incluirlo en este trabajo ya que su uso en ciencia de datos es cada vez más frecuente. Un ejemplo de ello es la publicación de libros escritos por reconocidos desarrolladores de software como lo son *Python Data Science Handbook* de Jake VanderPlas y *Python for Data Analysis* de Wes McKinney.

Python se incluye también en este documento por su uso similar

con Julia y R lo que permite mostrar su sencillez y facilidad de programación. En este trabajo se utilizó Python para los tres proyectos de manejo de datos que se explican a partir del capítulo 5. En dichos capítulos se presenta el código utilizado en los ejercicios por lo que en las siguientes secciones se comentan los paquetes principales y la interfaz que se utilizaron.

3.1. Listas

“Una *lista* es una colección de elementos en un orden particular”, [Matthes \(2019\)](#). La lista es la estructura de datos elemental y principal de Python ya que permite almacenar diferentes tipos de elementos en un solo objeto. Una lista se crea usando paréntesis cuadrados []. Por ejemplo, si se quisiera hacer una lista de animales en el zoológico el comando sería

```
animales = ["zebra", "leon", "jirafa", "elefante"]  
animales[1]
```

Los elementos de una lista pueden accederse mediante su índice y el uso de corchetes cuadrados. Por ejemplo, el comando `animales[1]` regresa "león". Una característica clave de las listas en Python es que, a diferencia de R y Julia, las listas comienzan a numerar sus elementos desde el cero.

Existen más estructuras de datos en Python cuyas características cumplen distintos criterios. En este trabajo se eligió trabajar con listas ya que son un objeto ordenado, mutable y que permite valores duplicados. Se recomienda ver la documentación de Python, [Python Software Foundation \(2022\)](#), para una explicación más detallada de los métodos propios de las listas.

3.2. Paquetes

Al igual que en Julia, en Python se desarrolló el lenguaje original separado de los paquetes. La diversidad del desarrollo de los paquetes es tal que facilita hacer casi cualquier tipo de análisis matemático posible.

Para usar cualquier instrucción de un paquete se tiene primero que nombrar su apodo y después llamar a la función. El apodo del paquete se lo otorga el usuario al momento de importarlo. Por ejemplo,

```
import numpy as np
```

importa el paquete NumPy con el apodo `np`. Si se quisiera llamar a la función `array` de este paquete se tendría que escribir el comando `np.array`. En el desarrollo de códigos extensos esto podría parecer tedioso e incluso innecesario. Sin embargo, lo considero una gran ventaja ya que quita la ambigüedad de reconocer que función corresponde a que paquete. En las siguientes secciones se presentan los paquetes de Python utilizados en este trabajo.

3.2.1. NumPy

NumPy es el paquete fundamental para computación científica en Python ya que es el que proporciona los objetos multidimensionales como lo son los vectores y matrices. Estos objetos se pueden crear sin necesidad de NumPy, pero hacerlo mediante el paquete otorga algunas ventajas. La primera de ellas es que los arreglos creados en NumPy tienen dimensiones inmutables; la segunda es que sus elementos deben pertenecer del mismo tipo de dato; la tercera es que facilitan operaciones matemáticas en grandes cantidades de datos; y, finalmente, la cuarta es que es uno de los paquetes preferidos por la comunidad de Python lo cual lo hace objeto de una gran cantidad de

publicaciones sobre su uso y aplicación.

En esta tesis se usó NumPy para crear y manipular arreglos, así como hacer un ajuste de un polinomio usando el método de mínimos cuadrados. A continuación se presenta la lista completa de comandos que se utilizaron con su respectiva explicación proveniente del paquete oficial de NumPy, [La Comunidad NumPy \(2022\)](#).

- `np.array([lista])`: Crea un arreglo con los valores de la `lista`.
- `np.insert(arr, obj, values)`: Inserta los valores `values` en el arreglo `arr` antes del índice `obj`.
- `np.arange(start, stop)`: Crea un arreglo con valores espaciados uniformemente desde `start` hasta el número antes de `stop`.
- `np.transpose(a)`: Transpone el objeto `a`.
- `np.concatenate(a1, a2, ...)`: Une la secuencia de arreglos en uno ya existente.
- `np.ones(shape)`: Crea una matriz de tamaño `shape` llena con unos.
- `np.diag(v)`: Extrae la diagonal de la matriz `v` o crea una matriz diagonal de tamaño `v`.
- `np.linalg.inv(a)`: Calcula la inversa multiplicativa de la matriz `v`.
- `np.random.choice(a, size = None, replace = True, p = None)`: Genera una muestra aleatoria de `a` de tamaño `size` con o sin reemplazo.
- `np.polyfit(x, y, deg)`: Hace un ajuste polinomial de grado `deg` a los puntos `(x, y)` usando el método de mínimos cuadrados.

3.2.2. pandas

pandas es el segundo paquete en importancia de **Python** ya que ofrece manipulación y análisis de datos de manera rápida, flexible y sencilla. Sus funciones se enfocan en el uso eficiente de dataframes, lectura y escritura de datos, agrupación y unión de conjuntos de datos, entre otros (McKinney and the Pandas Development Team, 2022). En esta tesis se utilizaron los siguientes comandos de este paquete.

- `pd.read_csv(filepath)`: Lee un archivo `csv` y lo convierte a dataframe.
- `pd.DataFrame(data)`: Crea un objeto de tipo dataframe con los datos `data`.
- `pd.get_dummies(data)`: Convierte las variables categóricas `data` en variables indicadoras o dummie.

3.2.3. os

Otro paquete utilizado en este trabajo fue **os** ya que proporciona una manera portátil de usar la funcionalidad dependiente del sistema operativo (Python Software Foundation, 2022). Este es el paquete que permite hacer la conexión entre **Python** y los archivos de una computadora. Los comandos que se utilizaron son dos. El primero fue `os.chdir(path)` que permite seleccionar el directorio en el que se está trabajando. Mientras que el segundo fue `os.listdir(path)` que proporciona una lista de archivos en el `path` dado.

3.2.4. scikit-learn

scikit-learn es un paquete creado para *machine learning* o aprendizaje de máquina en **Python**. También es conocido como

sklearn y proporciona herramientas simples y eficientes para la predicción en análisis de datos. Por ejemplo, clasificación, regresión, agrupamiento o conglomerado y reducción de dimensiones en modelos ([Python Software Foundation, 2022](#)). En este trabajo se utilizó para hacer regresiones lineales. El usuario puede importar el paquete de dos maneras.

```
import sklearn
from sklearn import linear_model

regr = linear_model.LinearRegression()
model = regr.fit(x, y)
```

La primera línea de instrucción importa el paquete **sklearn** completo, mientras que la segunda solo toma la parte de modelos lineales. Con el paquete cargado, la tercera línea de código se encarga de guardar en la variable **regr** que lo que se busca es ajustar un modelo lineal definido como en la ecuación 2.1. Finalmente, el último comando del código calcula los coeficientes β usando el método de mínimos cuadrados.

3.2.5. **itertools**

“itertools es un módulo que estandariza un conjunto importante de herramientas rápidas y eficientes de memoria que son útiles en sí mismas o en combinación”, [Python Software Foundation \(2022\)](#). Este paquete tiene ciertas funciones implementadas que se pueden recrear sin la necesidad del mismo. Sin embargo, la ventaja de utilizar **itertools** es la velocidad en la que las genera.

En este trabajo se utilizó **itertools.combinations()** para crear las combinaciones de posibles factores activos del problema de

discriminación de modelos presentado en el capítulo 7. Todos los paquetes anteriores se utilizaron mediante la interfaz gráfica Jupyter Notebook que se presenta a continuación.

3.3. Jupyter Notebook

“Jupyter Notebook es la aplicación web original para crear y compartir documentos computacionales. Es un programa que existe para desarrollar software de manera pública en decenas de lenguajes de programación incluyendo R, Python y Julia”, [Project Jupyter \(2022\)](#).

Una manera sencilla de obtener Jupyter Notebook es instalando Anaconda, una interfaz gráfica que permite manejar y administrar aplicaciones, paquetes, ambientes y canales sin necesidad de usar comandos en el sistema operativo. Para instalar Anaconda en Windows se debe ir a la página <https://docs.anaconda.com/anaconda/install/windows/> y seguir las instrucciones de instalación. Esto puede tomar unos minutos. Al terminar, la ejecución de Jupyter Notebook abrirá una ventana del explorador que se verá de manera similar a la imagen 3.1.

Jupyter Notebook tiene la ventaja de facilitar el uso indistinto de los tres lenguajes utilizados en este trabajo. Uno de los prerequisites para instalarlo es la instalación previa de Python. Por lo tanto, este es el lenguaje que ya viene en la interfaz. La figura 3.2 muestra, a manera de ejemplo, un archivo que utiliza Python en Jupyter Notebook. En la esquina superior derecha se puede verificar el lenguaje que se está utilizando.

El caso para R y Julia es diferente por lo que se expondrá su implementación en las siguientes secciones.

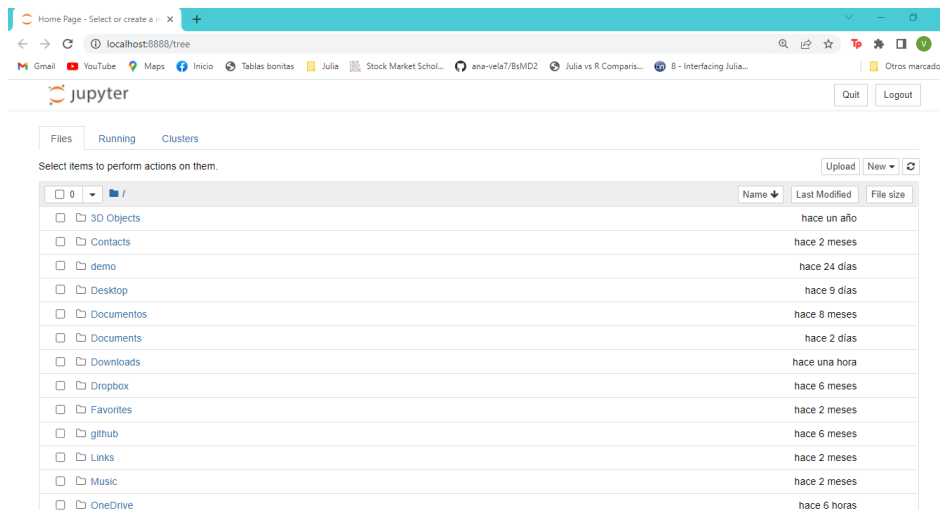


Figura 3.1. Pantalla principal de Jupyter Notebook

3.3.1. Julia

El primer paso para utilizar Julia en la interfaz Jupyter Notebook es tener instalado el lenguaje en la computadora. Posteriormente, se debe instalar el paquete `Julia` usando los pasos descritos en la sección 2.4.5. Esto solo se tiene que hacer una vez. Para confirmar que la instalación esté bien hecha se debe abrir Jupyter Notebook, seleccionar **New** y debe aparecer la opción de Julia 1.6.3 (o la versión de Julia que esté instalada en la computadora). El archivo nuevo generado con Julia se ve similar a la figura 3.3

3.3.2. R

Existen varias maneras de instalar R en Jupyter, pero se expondrán los pasos descritos en el manual de Anaconda, [Anaconda Inc \(2022\)](#).

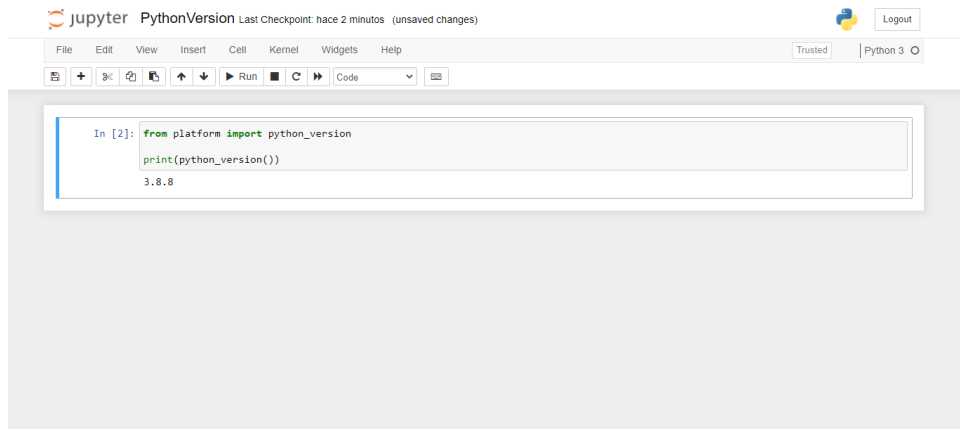
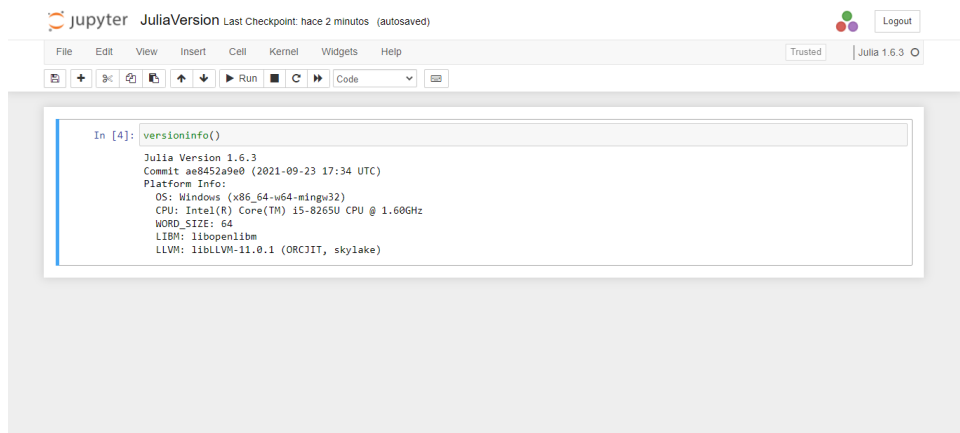


Figura 3.2. Archivo generado con Python en Jupyter Notebook

1. Abrir el Navegador de Anaconda (no confundir con el de Jupyter Notebook).
2. Seleccionar **Environments** y después la opción de **Create** ubicada en la esquina inferior izquierda.
3. Aparecerá una ventana donde permite al usuario nombrar el **Environment** como prefiera. Se debe seleccionar la versión de Python que se tenga y seleccionar la casilla del lado izquierdo de R. Después, se debe pulsar la opción de **Create**.
4. Para usar el ambiente que se acaba de crear en Jupyter Notebook se selecciona la flecha de lado derecho del nombre del ambiente nuevo. Entre las opciones seleccionar la opción de **Open with Jupyter Notebook** como se muestra en la figura 3.4
5. Por último, se debe seleccionar el botón de **New** y después R para



The screenshot shows a Jupyter Notebook interface with the title bar 'Jupyter JuliaVersion Last Checkpoint: hace 2 minutos (autosaved)'. The top menu bar includes 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. Below the menu is a toolbar with icons for file operations, cell navigation, and execution. The main area displays a code cell with the input 'In [4]: versioninfo()' and its output. The output provides detailed information about the Julia environment, including the version, commit hash, platform, OS, CPU, word size, and linked libraries.

```
In [4]: versioninfo()

Julia Version 1.6.3
Commit ae8452a9e0 (2021-09-23 17:34 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-11.0.1 (ORCJIT, skylake)
```

Figura 3.3. Archivo generado con Julia en Jupyter

crear un archivo que trabaje con R.

En la figura 3.5 se muestra el ejemplo de un archivo generado con R en Jupyter Notebook. En el siguiente capítulo se exponen las funciones utilizadas en R.

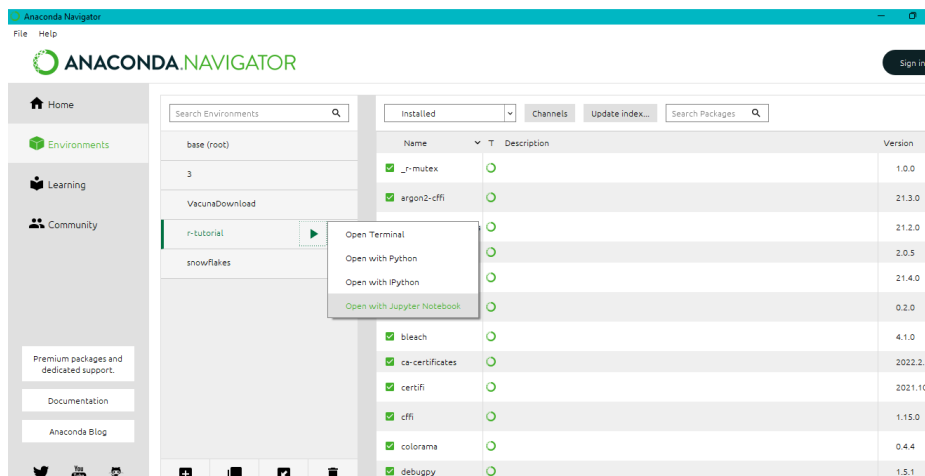


Figura 3.4. Ejecución de R desde el navegador de Anaconda

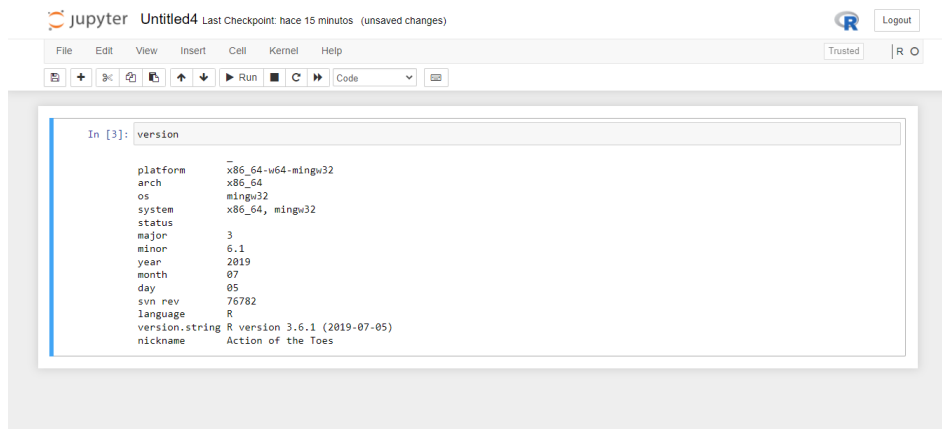


Figura 3.5. Archivo generado con R en Jupyter Notebook

Capítulo 4

R

“Ross Ihaka y Robert Gentleman, del departamento de Estadística de Auckland University en Nueva Zelanda estaban interesados en el cómputo estadístico y reconocieron la necesidad de un mejor ambiente de cálculo del que tenían. Ninguno de los productos comerciales les convenía, por lo que decidieron desarrollar uno propio”, [Barrios \(2010\)](#).

R nació de la necesidad de tener una transición de usuario a desarrollador. [Peng \(2015\)](#) explica que los creadores buscaron desarrollar un lenguaje que podría usarse para hacer análisis de datos de manera interactiva y, que además, fuera capaz de escribir programas más largos. Una de las principales cualidades de R es la facilidad para crear gráficos bien diseñados y con calidad de publicación que pueden incluir símbolos matemáticos y fórmulas en caso de ser necesarios ([R Core Team, 2022](#)).

R se considera como uno de los lenguajes más sencillos para aprender a trabajar métodos estadísticos computacionales. La sencillez de su sintaxis permite que incluso un usuario nuevo navegue de manera fluida por el código. Es conocido como uno de los lenguajes

más usados para análisis y graficación. Su popularidad en la comunidad científica ha impulsado el desarrollo del lenguaje y la creación de todo tipo de materiales de apoyo. Las razones anteriores son el motivo por el cual se decidió incluir el lenguaje en esta tesis.

Debido a la popularidad ya mencionada, en este trabajo se parte de la premisa de que el lector ya cuenta con los conocimientos básicos para entender el código que se presenta. El avanzado desarrollo del lenguaje permitió que los ejercicios de este trabajo se hicieran con pocas funciones. En el último proyecto se utiliza un paquete ya programado, mientras que los primeros dos se enfocan en la función `lm` explicada a continuación.

4.1. Función `lm`

La función `lm` es usada para analizar modelos lineales. La manera de llamarla es con el comando `lm(formula, data, subset, weights, na.action, method = 'qr', model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE, contrasts = NULL, offset, ...)`. Con esto se puede observar que la función cuenta con muchos argumentos que la hacen muy versátil. De hecho, en el capítulo 5 es necesario modificar el argumento `tol` para poder resolver el problema de manera exitosa.

El modelo de regresión lineal multivariada debe tener como el argumento de `formula` una ecuación de la forma

$$y \sim x_1 + x_2 + \dots + x_k$$

Es decir, se indica la variable de respuesta y seguido de una virgulilla y después los n predictores que se estén utilizando. Si se tratara de escribir la `formula` como ecuación, la virgulilla juega el papel de un

signo de igualdad.

La función `lm` también permite analizar modelos con variables cuyo grado es mayor a uno. En este caso, se debe utilizar la función llamada interpretación inhibida, mejor conocida como `I(...)`. El argumento `formula` comienza como en el caso anterior, con la variable y y la virgulilla. Después, se eleva el regresor x a la potencia k dentro de la función `I(...)`. Por ejemplo, si se quisiera ajustar el modelo $y = \beta_0 + \beta_1 x + \beta_2 x^2$ el comando es

```
y ~ x + I(x^2)
```

La función `I(...)` se utiliza para cambiar la clase de un objeto para indicar que el objeto debería ser tratado de la forma 'como si fuera'. Esta instrucción se usa especialmente para los operadores especiales de fórmula como es el caso de \wedge . En el ejemplo anterior, la función de interpretación inhibida da la instrucción de tomar x^2 como una variable y no como la interacción de segundo orden de x .

Por otro lado, uno de los argumentos que tiene la función `lm` es `tol`, la tolerancia del ajuste. En este caso, la tolerancia determina si las columnas de una matriz son linealmente independientes o no. Cuando se tiene una matriz con valores muy pequeños (como es el caso del ejercicio presentado en el capítulo 5) se necesita la tolerancia para determinar cuando un valor se considera como cero. En dicho ejercicio el argumento `tol` tuvo que ser modificado para lograr el resultado correcto. Usualmente este parámetro no necesita ser cambiado, pero es útil tomarlo en cuenta para ocasiones donde los datos son extremadamente sensibles y se busca un ajuste preciso.

La segunda parte de la tesis comienza en el siguiente capítulo. En esta parte se exponen tres ejercicios diferentes en los tres lenguajes descritos en la primera parte (R, Julia y Python). El primer ejercicio es

el ajuste de un modelo lineal de grado diez con datos extremadamente sensibles. En este ejercicio se mide la precisión numérica de los cálculos de los tres lenguajes. El segundo ejercicio es el ajuste de modelos lineales de distintos órdenes usando grandes cantidades de datos. El objetivo fue ilustrar y comparar el manejo y análisis de datos. Finalmente, el tercer ejercicio es sobre la discriminación de modelos en diseños de experimentos. El punto de comparación fue la rapidez en la que los lenguajes hacen una gran cantidad de cálculos intensivos.

Capítulo 5

Ajuste de polinomios

En los capítulos anteriores se presentaron tres lenguajes de programación, Julia, R y Python, con la intención de utilizarlos para la solución de tres proyectos. En este capítulo se presenta el primero de ellos. El problema fue diseñado y publicado por el National Institute of Standards and Technology (NIST) cuya misión incluye proveer soluciones que garanticen el estándar de medición. El ejercicio busca medir la precisión numérica en los cálculos que ejecutan los lenguajes de programación. NIST provee un conjunto de datos, un problema y su solución con alta precisión numérica en sus dígitos. La tarea del usuario es desarrollar una solución cuya precisión se acerque lo más posible a la presentada por el instituto.

En este capítulo se toma un problema propuesto por NIST donde la tarea es ajustar un polinomio de grado diez a un conjunto de datos. La solución se programó en los tres lenguajes ya mencionados y se compara la exactitud de la respuesta, el tiempo de ejecución y la facilidad de programación. La importancia de este proyecto se centra en la precisión numérica. NIST establece los estándares de referencia de las mediciones.

Por lo tanto, si los cálculos logran alcanzar un alto grado de precisión numérica se puede confiar en la robustez del método de solución.

5.1. El problema

Suponga que se tiene un conjunto de datos con solo dos variables x , y . El problema propuesto es ajustar la información a un polinomio de grado p . Es decir, se busca ajustar los datos al modelo

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \dots \beta_{p-1} x^{p-1} + \beta_p x^p \quad (5.1)$$

El ejercicio consiste en calcular los coeficientes que *mejor cumplan* la ecuación anterior. Una manera de hacer el cálculo es con el método de mínimos cuadrados.

5.1.1. Método de Mínimos Cuadrados

El objetivo del método de mínimos cuadrados es encontrar una función que mejor se aproxime a los datos. Esto es, suponga que se tiene un conjunto de datos donde se tiene una variable de respuesta y y x_1, x_2, \dots, x_p regresores. Suponga que se quiere ajustar los datos al modelo

$$y_i = \beta_0 + \beta_1 x_{i1} + \dots + \beta_p x_{ip} + \epsilon_i \text{ donde } i = 1, \dots, n$$

La ecuación anterior representa una curva de orden p . El método de mínimos cuadrados busca minimizar la suma de cuadrados entre la curva y los datos. Es decir,

$$\min_{\beta} \sum_{i=1}^n \epsilon_i^2 = \min_{\beta_0, \dots, \beta_p} \left(\sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_{i1} - \dots - \beta_p x_{ip})^2 \right)$$

donde y es un vector de tamaño n , X es una matriz de tamaño $n \times (k+1)$ y β es un vector de tamaño $k+1$.

5.2. Los datos

El conjunto de datos que se utilizan para trabajar este problema son proporcionados por el Instituto Nacional de Estándares y Tecnología (NIST, por sus siglas en inglés). Su departamento de estadística ofrece varios servicios, entre ellos generar datos ligados a problemas cuya solución suponga un reto numérico. Para este ejercicio se escogió el conjunto de datos llamado `fillip` cuyo problema es ajustar un polinomio de grado 10. El cálculo de los coeficientes del ajuste presentado por NIST contiene hasta 15 dígitos decimales cuyo objetivo es que el usuario pueda valorar la precisión de su ajuste.

Los datos constan de 82 pares ordenados (x_i, y_i) y se pueden encontrar en <https://www.itl.nist.gov/div898/strd/lls/data/LINKS/DATA/Filip.dat>. Su gráfica se muestra en la figura 5.1.

5.3. Planteamiento del problema

Con la información ya presentada, se puede aterrizar la ecuación 5.1 al problema propuesto. La ecuación queda de la forma

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \dots + \beta_9 x_i^9 + \beta_{10} x_i^{10}$$

donde $i = 1, 2, \dots, 82$.

El vector y es de tamaño 82 y corresponde a la columna del mismo nombre en los datos `fillip`. La incógnita del problema es el vector

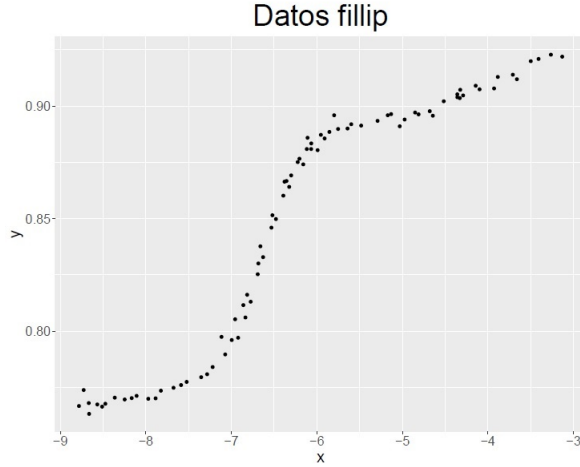


Figura 5.1. Conjunto de datos fillip para el ajuste del polinomio

de tamaño 11 conformado por los coeficientes β . Los regresores x_i^j se obtienen de los datos `fillip`, específicamente de la columna `x`. De forma matricial, el problema se puede representar como

$$y = X\beta. \quad (5.2)$$

o, equivalentemente,

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{81} \\ y_{82} \end{bmatrix} = \begin{pmatrix} 1 & x_{1,1} & x_{1,2} & \dots & x_{1,10} \\ 1 & x_{2,1} & x_{2,2} & \dots & x_{2,10} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & x_{81,1} & x_{81,2} & \dots & x_{81,10} \\ 1 & x_{82,1} & x_{82,2} & \dots & x_{82,10} \end{pmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_9 \\ \beta_{10} \end{bmatrix} \quad (5.3)$$

Representar la matriz X de esta manera tiene una ventaja particular. Cada elemento puede ser considerado como $x_{i,j}$, donde el

renglón i representa la observación i de los datos. Asimismo, la columna j representa la potencia a la que está elevada la observación i . Por ejemplo, el elemento $x_{34,5}$ es la observación 34 de la columna \mathbf{x} de los datos elevado a la 5 potencia. Sin embargo, el elemento $x_{34,5}$ realmente está en la columna número 6 de la matriz. El objetivo principal de esta notación es no perder de vista la potencia de las observaciones.

Hasta ahora se ha planteado el problema propuesto como cualquier ejercicio de ajuste de polinomios. La siguiente sección presenta las condiciones para que la solución de un problema se considere sensible. Asimismo, se presenta la pauta para considerar un problema como mal o bien condicionado.

5.4. Número de condición y precisión de la solución

Considere el sistema lineal presentado de manera matricial como

$$Ax = b. \quad (5.4)$$

Note la similitud de la ecuación anterior con la ecuación 5.2. En esencia presentan el mismo problema con variables llamadas de manera diferente. En esta sección se utiliza la notación presentada en la ecuación 5.4 por congruencia con las definiciones y teoremas presentados.

Se considera que los datos tienen impurezas cuando cualquier cambio en la matriz A o en el vector b resulta en un ajuste de los coeficientes x poco preciso. El caso contrario, donde los métodos dan resultados precisos se conoce a los datos como exactos (Datta, 2010).

En general, para el problema 5.1 se tienen tres casos posibles:

1. El vector b tiene impurezas, mientras que la matriz A es exacta.
2. La matriz A tiene impurezas, mientras que el vector b es exacto.
3. Ambos, el vector b y la matriz A tiene impurezas.

Este ejercicio se enfoca en el tercer caso, ya que no hay razones para asumir que solo una columna de los datos es la causante de las impurezas. En este caso se considera que el sistema lineal es sensible, ya que un pequeño cambio en el vector y o en la matriz A generan una variación importante en la solución del mismo. La sensibilidad de un sistema lineal se puede determinar con el número de condición.

Definición 1. El número $\| A \| \| A^{-1} \|$ se llama el número de condición de A y se denota $Cond(A)$ (Datta, 2010, p. 62).

Asimismo, el número de condición establece una relación en la magnitud en los cambios en un sistema como se muestra en el siguiente teorema presentado.

Teorema 5.1. (Datta, 2010, p. 65) Suponga que se busca resolver el sistema lineal $Ax = b$. Suponga también que A es no singular, $b \neq 0$, y $\| \Delta A \| < \frac{1}{\| A^{-1} \|}$. Entonces

$$\frac{\| \delta x \|}{\| x \|} \leq \left(\frac{Cond(A)}{1 - Cond(A) \frac{\| \Delta A \|}{\| A \|}} \right) \left(\frac{\| \Delta A \|}{\| A \|} + \frac{\| \delta b \|}{\| b \|} \right).$$

La importancia del número de condición se ejemplifica en el teorema anterior. Los cambios en la solución x son menores o iguales a una constante determinada por el número de condición multiplicada por la suma de las perturbaciones de A y las perturbaciones de b . Además, el teorema establece que, aunque las perturbaciones de A y b

sean pequeñas, puede haber un cambio grande en la solución si el número de condición es grande. Por lo tanto, $Cond(A)$ juega un papel crucial en la sensibilidad de la solución (Datta, 2010).

El número de condición presenta una lista de propiedades. La más relevante para este ejercicio es la definición del número de condición como cociente de valores singulares.

$$Cond(A) = \frac{\sigma_{max}}{\sigma_{min}} \quad (5.5)$$

donde σ_{max} y σ_{min} son, respectivamente, el valor singular más grande y más pequeño de A . La condición de un sistema lineal está ligado al número de condición con la siguiente definición.

Definición 2. *El sistema $Ax = b$ está mal condicionado si el $Cond(A)$ es grande (por ejemplo, $10^5, 10^8, 10^{10}$, etc). En otro caso, está bien condicionado (Datta, 2010, p. 68).*

El número de condición se calculó en Julia y se verificó en R. En ambos lenguajes se calculó de dos maneras distintas. La primera fue utilizando la función ya programada en cada lenguaje, mientras que la segunda fue utilizando la fórmula 5.5. En Julia, el código es

```
# Con función de Julia
julia> numCond_1 = cond(X_10)

# Usando propiedad de valores singulares
julia> sing_values = svd(X_10).S
julia> sing_values = sort(sing_values)
julia> numCond_2 = sing_values[length(sing_values)] /
sing_values[1]
```

Los resultados son $numCond_1 = 1.7679692504686805e15$ y $numCond_2 = 1.7679692504686795e15$. Por otro lado, en R el código es

```
# con funcion de R
numCond_R1 <- cond(X)

# Usando propiedad de valores singulares
S.svd <- svd(X)
S.d <- S.svd$d
S.d <- sort(S.d, decreasing = TRUE)
numCond_R2 <- S.d[1] / S.d[length(S.d)]
```

Los resultados son $numCond_{R1} = numCond_{R2} = 1.767962e15$. En conclusión, ambos lenguajes confirman que el número de condición de la matriz X del problema 5.2 es bastante grande. Por lo tanto, por la definición 2 se puede decir que el problema está mal condicionado. Esto podría ocasionar muchas preguntas al lector, incluyendo si hubiera sido mejor utilizar otros datos o ajustar un polinomio de grado menor.

Se deben recordar dos cuestiones. La primera es que los datos fueron generados en el Instituto Nacional de Estándares y Tecnología. Los datos y el problema propuestos fueron diseñados para presentar un alto nivel de dificultad de cálculo. El segundo punto es recordar que el objetivo de esta sección del trabajo es evaluar la precisión numérica de los lenguajes. Por lo tanto, no debe ser una sorpresa que el problema esté mal condicionado. Se espera encontrarse con dificultades de programación. De esta forma, cuando se encuentre un método de solución que obtenga los resultados similares a los presentados por NIST, se puede confiar en la precisión numérica.

Sin embargo, la pregunta sobre la eficiencia de los métodos en polinomios de orden menor se consideró un excelente punto de

comparación entre soluciones. Por lo tanto, se decidió extender el problema propuesto y ajustar los datos a polinomios de grado k , donde $k = 1, 2, \dots, 10$. En total, se calcularon 10 ajustes para cada uno de los 6 métodos presentados en las siguientes secciones. En Julia se desarrollaron 4 métodos de solución mientras que en R y Python solo fue necesario implementar uno en cada lenguaje.

5.5. Solución usando Julia

Morgenstern and Morales (2015) abordaron este problema con el objetivo de mostrar y comparar la precisión numérica de los lenguajes R, Excel, Stata, SPSS, SAS y Matlab. El artículo presenta cuatro métodos para calcular las soluciones a los coeficientes β de la ecuación 5.2. El propósito de esta sección es mostrar cuatro métodos de solución al problema propuesto en Julia. Dos de los métodos son tomados del artículo presentado por Morgenstern y Morales, mientras que en los otros dos se toman de funciones programadas en paquetes de Julia.

En primer lugar se debe leer el archivo donde se encuentran los datos `filip`. Además, se deben inicializar variables que contengan el grado del polinomio que se busca ajustar y el total de observaciones.

```
julia> using CSV, DataFrames
julia> filip = CSV.read("filip_data.csv", DataFrame)
julia> x = filip.x
julia> y = filip.y
julia> k = 10 #grado del polinomio
julia> n = length(x) # número de observaciones
```

En segundo lugar, se desarrolló una función que generara la matriz X definida en 5.3. Esta función se nombró `generar_X(k)` y toma como

argumento k , la potencia a la que se busca ajustar el polinomio.

```
julia> function generar_X(k) # k es la potencia del polinomio

    n = size(filip, 1) # número de renglones
    # Inicialización de una matriz vacía
    X = Array{Float64}(undef, n, k + 1)
    # La primera columna siempre es
    # un vector de unos
    X[:, 1] = ones(n)

    # Para el resto de la columnas,
    # se eleva cada elemento a la potencia correspondiente
    for i = 1:k
        X[:, i + 1] = x.^i
    end
    return X
end
```

Hasta este punto se presentó el código en común necesario para desarrollar los cuatro métodos de solución propuestos. A continuación se muestra la teoría y el código de cada método.

5.5.1. *GLM*

Ya que el problema es ajustar un modelo de regresión lineal, el primer acercamiento propuesto es utilizar el paquete **GLM** ya que sus siglas se traducen a “Modelos Lineales Generalizados”. Su función principal es `lm` de la que se da una explicación detallada en la sección 2.4.7.

En este ejercicio, el argumento `formula` se presenta usando la función llamada `poly`. Similar a la función `I(...)` de R, el objetivo de `poly`

es construir un objeto que tenga las propiedades de un polinomio. La construcción y definición de `poly` se encuentra en la documentación del paquete `StatsModels` elaborado por [Julia Programming Language \(2021\)](#).

Por otro lado, al argumento `data` se asignan los datos `fillip` ya leídos. El resto de los argumentos se omiten ya que su opción default es la necesaria en este ejercicio. Por lo tanto, el código de este método en Julia es

```
julia> using GLM
julia> x_fit = lm(@formula(y ~ 1 + poly(x, 10)), filip)
```

Los resultados para todos los métodos se encuentran en la sección 5.8. Es claro que para este método (GLM en las tablas de resultados 5.2, 5.3, 5.4) los cálculos no arrojan un resultado correcto. Dado que NIST proporciona la respuesta, fue claro observar que la estimación de los coeficientes no fue precisa.

5.5.2. Factorización QR versión económica

El segundo método que se empleó para solucionar el problema propuesto es el que usa la factorización QR versión económica. En primer lugar se presenta la descomposición QR y posteriormente se muestra su versión económica.

Definición 3. *La factorización QR de una matriz A de dimensiones $m \times n$ es el producto de una matriz Q de tamaño $m \times n$ con columnas ortogonales y una matriz R cuadrada y triangular superior ([Garcia and Horn, 2017](#), p. 191).*

En el problema propuesto por NIST no es posible utilizar la factorización QR definida anteriormente. Las dimensiones de la matriz

X son $n \times m = 82 \times 11$. Además, por construcción la primera columna de la matriz es un vector de unos por lo que su rango es $r = 10 < 11 = n$. Entonces, la matriz R de la descomposición QR es singular por lo que no se puede generar una base ortonormal de $R(X)$.

Definición 4. *Una secuencia de vectores u_1, u_2, \dots (finita o infinita) en un espacio de producto interno es ortonormal si*

$$\langle u_i, u_j \rangle = \delta_{ij} \text{ para toda } i, j$$

Una secuencia ortonormal de vectores es un sistema ortonormal (Garcia and Horn, 2017, p. 147).

Definición 5. *Una base ortonormal para un espacio de producto interno finito es una base que es un sistema ortonormal (Garcia and Horn, 2017, p. 149).*

Sin embargo, afortunadamente el proceso de factorización QR se puede modificar usando una matriz de permutación que genere una base ortonormal.

Definición 6. *Una matriz P es una matriz de permutación si exactamente una entrada en cada renglón y en cada columna es 1 y el resto de las entradas son 0 (Garcia and Horn, 2017, p. 183).*

La idea de la factorización QR versión económica es generar una matriz de permutación P tal que

$$AP = QR,$$

donde

$$R = \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix}$$

Si se define r como el rango de X entonces R_{11} es de dimensión $r \times r$ triangular superior y Q es ortogonal. Las primeras r columnas de Q forman una base ortonormal de $R(X)$ (Datta, 2010). Esta variación de la factorización QR siempre existe como lo expone el siguiente teorema.

Teorema 5.2. *Sea A una matriz de $m \times n$ con $\text{rango}(A) = r \leq \min(m, n)$. Entonces, existe una matriz de permutación P de $n \times n$ y una matriz ortogonal Q de dimensiones $m \times m$ tal que*

$$Q^T AP = \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix}$$

donde R_{11} es una matriz triangular superior de tamaño $r \times r$ con entradas en la diagonal diferentes de cero (Datta, 2010, p. 532).

El paquete `LinearAlgebra` en Julia tiene la función `qr` que permite obtener la descomposición QR versión económica.

Con QR versión económica

```
julia> using LinearAlgebra
julia> F = qr(X, Val{true})
julia> Q = F.Q
julia> P = F.P
julia> R = F.R
```

Ya que se obtuvo la factorización QR versión económica se necesita más teoría algebraica para resolver el problema original 5.2. Las características de la matriz X permiten que el teorema 5.2 se cumpla. Es decir, $XP = QR$. Por otro lado, como P es matriz de permutación existe z tal que $Pz = \beta$. Por lo tanto, existe una expresión en la que β se puede sustituir en la ecuación 5.2 para obtener

$$y = X(Pz).$$

Asimismo, se sustituye en la fórmula de la factorización QR

$$(XP)z = (QR)z.$$

Si se unen las dos ecuaciones anteriores, se obtiene

$$\begin{aligned}y &= XPz = QRz \\ \implies y &= QRz\end{aligned}$$

Anteriormente se calcularon las matrices Q y R mientras que el vector y se obtiene del conjunto de datos originales. Por lo que la ecuación anterior se puede resolver para obtener los valores de z . Finalmente, se obtiene β con la expresión

$$\beta = Pz$$

En Julia, esta solución se programa de la siguiente manera

```
# 1. Resolver QRz = y
julia> z = Q\R \ y
# 2. Resolver beta = Pz
julia> x_QR = P*z
```

Este método también fracasó. En las tablas de resultados 5.2, 5.3, 5.4, las columnas QRvEcon muestran que el método parecía funcionar hasta llegar al polinomio de grado 10, donde falló. Se continuó buscando la solución usando la descomposición de valores singulares.

5.5.3. Descomposición de valores singulares

La tercera forma en la que se intentó solucionar el problema propuesto fue usando la descomposición de valores singulares para obtener la matriz pseudoinversa de Moore-Penrose. En primer lugar se

muestra la definición de valores singulares. Posteriormente, se presenta su descomposición así como su aplicación en la definición de la matriz pseudoinversa.

Definición 7. Sea A una matriz de $m \times n$ y sea $q = \min\{m, n\}$. Si el rango de $A = r \geq 1$, sean $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$ los eigenvalores positivos en orden decreciente de $(A^*A)^{1/2}$. Los valores singulares de A son

$$\sigma_1, \sigma_2, \dots, \sigma_r \text{ y } \sigma_{r+1} = \sigma_{r+2} = \dots = \sigma_q = 0.$$

Si $A = 0$, entonces los valores singulares de A son $\sigma_1 = \sigma_2 = \dots = \sigma_q = 0$. Los valores singulares de $A \in M_n$ son los eigenvalores de $(A^*A)^{1/2}$ que son los mismos eigenvalores de $(AA^*)^{1/2}$ (*Garcia and Horn, 2017, p. 420*)

Teorema 5.3. Sea $A \in M_{m \times n}(F)$ diferente de cero y sea $r = \text{rango}(A)$. Sean $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$ los valores singulares positivos de A y definamos

$$\Sigma_r = \begin{pmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r \end{pmatrix} \in M_r(R).$$

Entonces, existen matrices unitarias $U \in M_m(F)$ y $V \in M_n(F)$ tales que

$$A = U\Sigma V^* \tag{5.6}$$

donde

$$\Sigma = \begin{pmatrix} \Sigma_r & 0_{r \times (n-r)} \\ 0_{(m-r) \times r} & 0_{(m-r) \times (n-r)} \end{pmatrix} \in M_{m \times n}(R)$$

tiene las mismas dimensiones que A . Si $m = n$, entonces $U, V \in M_n(F)$ y $\Sigma = \Sigma_r \oplus 0_{n-r}$ (*Garcia and Horn, 2017, p. 421*).

La ecuación 5.6 con las características del teorema anterior es la definición de la descomposición en valores singulares (DVS). Además, las matrices U y V son matrices unitarias. Es decir,

$$UU^*u = u, \quad \forall u \in \text{Col}(U)$$

$$VV^*v = v, \quad \forall v \in \text{Col}(V)$$

Pseudoinversa de Moore-Penrose

En la descomposición de valores singulares se puede modificar la matriz Σ para obtener la pseudoinversa de Moore Penrose A^\dagger como se muestra en el siguiente teorema.

Teorema 5.4. *Sea A una matriz de dimensiones $m \times n$ de rango r con una descomposición en valores singulares de $A = U\Sigma V^*$ y valores singulares diferentes de cero $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r$. Sea Σ^\dagger una matriz de $n \times m$ definida como*

$$\Sigma_{ij}^\dagger = \begin{cases} \frac{1}{\sigma_i} & \text{si } i = j \leq r \\ 0 & \text{en otro caso.} \end{cases}$$

Entonces $A^\dagger = V\Sigma^\dagger U^$ y esta es la descomposición de valores singulares de A^\dagger (Spence et al., 2000, p. 414).*

La matriz A^\dagger tiene las siguientes propiedades:

1. $(A^T A)^\dagger A^T = A^\dagger$
2. $(A A^T)^\dagger A = (A^\dagger)^T$
3. $(A^T A)^\dagger (A^T A) = A^\dagger A = V V^T$

En este punto es conveniente recordar las dimensiones de la matriz $X_{n \times m} = X_{82 \times 11}$. Ya que $m < n$, hay más ecuaciones que variables desconocidas. Por lo tanto, el sistema lineal está sobredeterminado. [López-Bonilla et al. \(2018\)](#) establece que si la ecuación 5.1 se multiplica por X^T se obtiene el sistema determinado (balanceado)

$$X^T X \beta = X^T y, \quad y \in \text{Col}(V). \quad (5.7)$$

Ahora bien, si se multiplica 5.7 por $(X^T X)^\dagger$ y se usan las propiedades de la matriz pseudoinversa mencionadas anteriormente se pueden obtener los siguientes resultados.

$$(X^T X)^\dagger X^T X \beta = (X^T X)^\dagger X^T y$$

$$\text{por la propiedad 1} \quad \Longleftrightarrow \quad X^\dagger X \beta = X^\dagger y$$

$$\text{por la propiedad 3} \quad \Longleftrightarrow \quad V V^T \beta = X^\dagger y$$

$$V \text{ es matriz unitaria} \quad \Longleftrightarrow \quad \beta = X^\dagger y$$

Por lo tanto, la pseudo inversa de Moore Penrose es la solución de mínimos cuadrados del problema 5.2 ([López-Bonilla et al., 2018](#)). En Julia, este método se puede programar en las dos líneas siguientes.

```
## Inversa de Moore Penrose
julia> N = pinv(X)
julia> x_MP = N*y
```

Este método tampoco funcionó. Los resultados de este método corresponden a la columna `MoorePenrose` de las tablas 5.2, 5.3, 5.4. Se continuó indagando en los paquetes de Julia hasta encontrar *Polynomials*.

5.5.4. *Polynomials*

Polynomials es un paquete que proporciona aritmética básica, integración, diferenciación, evaluación y hallar raíces para polinomios univariados (JuliaMath, 2021). Las instrucciones para instalarlo se encuentran en la sección 2.4.5.

El paquete *Polynomials* contiene la función llamada `fit` que ajusta un polinomio de grado `deg` a `x` y `y` usando interpolación polinomial o aproximación por mínimos cuadrados (JuliaMath, 2021). La función toma tres argumentos como entrada. Los primeros dos corresponden a `x` y `y` de los datos a utilizar (en este caso, los datos `filip`). El tercer argumento, `deg`, compete al grado que se busca sea el polinomio.

Los métodos de solución propuestos anteriormente utilizan el procedimiento de mínimos cuadrados ya que el problema es un sistema de ecuaciones lineales. De acuerdo a la documentación del paquete, la función `fit` utiliza el método de Gauss-Newton que se emplea para resolver sistemas de ecuaciones no lineales. El cambio en metodología causó que el paquete no fuera considerado en primera instancia como opción para la resolución del problema. Adicionalmente, a diferencia de la función `lm` del paquete GLM, la función `fit` del paquete *Polynomials* solamente aporta los coeficientes del ajuste del polinomio. Es decir, dicha función no calcula los estimadores del ajuste. Por otra parte, el código en Julia es sumamente sencillo:

```
julia> using Polynomials
julia> x_pol = Polynomials.fit(x, y, 10)
```

A pesar de que la función `fit` decepciona al no calcular los estimadores, es necesario agregarlo a esta sección de la tesis, ya que es el único método que funcionó. Las tablas de resultados 5.2, 5.3, 5.4

muestra que este es el único método que, en conjunto con R y Python da los resultados correctos. Las siguientes secciones muestran la solución en dichos lenguajes.

5.6. Solución usando R

Los problemas presentados por NIST tienen cierto grado de fama dentro de la comunidad científica. El problema propuesto en este capítulo no es la excepción. Ejemplo de ello es la solución que presentó Brian Ripley en el 2006. Ripley es un estadístico británico famoso por ser el autor de diversos libros de estadística y programación. Sus aportaciones son tales que, en diversas ocasiones, ha sido galardonado por universidades de sumo prestigio como es la Universidad de Cambridge. Tal vez su logro más conocido es haber sido parte del equipo creador de R llamado *The R Core team*.

Sin duda, Ripley es un gigante de la estadística en R por lo que en esta sección se utiliza la solución que él mismo propuso e hizo pública para el problema 5.2. Dicha solución utiliza la función `lm(formula, data)` cuya explicación detallada se encuentra en la sección 4.1. Dado que el problema propuesto presenta un alto grado de sensibilidad, el argumento `tol` de la función `lm` debe ser modificado. El siguiente código muestra el código en R para el ajuste de los polinomios de grado k , donde $k = 1, 2, \dots, 10$.

```
# Para polinomio de grado = 1
time_vec <- c(1)

for (i in runs){
  start <- Sys.time()
  lm_1 <- lm(y ~ x, data = data, x = TRUE)
```

```

    end <- Sys.time()
    time_vec <- c(time_vec, end - start)
  }

  tiempos_r[1, ] <- time_vec

  # Para polinomios de grado > 1
  for (i in 2:10){
    #Hacemos el modelo
    model <- paste("y~x", paste("+I(x^", 2:i, ")") ,
    sep='', collapse=''))

    # Lo convertimos en formula
    form <- formula(model)
    time_vec <- c(i)

    #Ejecutamos el modelo
    for (j in runs){
      start <- Sys.time()
      lm.plus <- lm(form, data = data,
      x = TRUE, tol = 1e-10)
      end <- Sys.time()
      time_vec <- c(time_vec, end - start)
    }

    tiempos_r[i, ] <- time_vec
  }

```

5.7. Solución usando Python

Igual que en R, en Python se buscó solucionar el problema usando paquetes ya desarrollados. Para esto, se utilizó la función `polyfit` del paquete `NumPy`. Como se explica en la sección 3.2.1, el objetivo de la función es ajustar un polinomio de orden específico a un conjunto de datos. Para ejecutar `polyfit` en el ajuste de los diez polinomios se desarrolló una función en Python. Dicha función se nombró `polynomial_fit` y sus objetivos son calcular las aproximaciones de los coeficientes β , guardar los resultados en un dataframe y medir el tiempo de ejecución. El código se muestra a continuación.

```
def polynomial_fit(grado_pol):
    tiempos_list = []
    for i in runs:
        start_time = time.time()

        # Regresa el coeficiente para la potencia
        # mayor primero
        python_fit = np.polyfit(x,
                                y, deg = grado_pol)

        # Medimos el tiempo
        tiempo = time.time() - start_time
        tiempos_list.append(tiempo)

        # Lo movemos para que este igual que en
        # los otros programas
        python_fit = np.flipud(python_fit)

        # Guardamos los coeficientes en un dataframe
```

```

        resultado=pd.DataFrame(python_fit)
# Cambiamos el nombre de la columna
        resultado.columns = ['Python']
        nombre_archivo = "res_python-gr" +
        str(grado_pol) + ".csv"
        resultado.to_csv(nombre_archivo)

    return tiempos_list

# Hagamos un df vacio para guardar los tiempos
    names = []
    for i in runs:
        names.append("Tiempos_" + str(i))

    names.insert(0, 'Grado')
    tiempo_df = pd.DataFrame(columns = names)

# Calculemos todos los ajustes
    for grado in range(1, 11):
        res_dic = {'Grado': grado}
        for i in runs:
            time_grado = []
            time_grado = polynomial_fit(grado)
            res_dic['Tiempos_' + str(i)] =
                time_grado[i - 1]
    tiempo_df = tiempo_df.append(res_dic ,
        ignore_index = True)

    tiempo_df.to_csv("tiempos_NIST_Python.csv")

```


En la siguiente sección se presentan los resultados, con sus tiempos de ejecución, de todos los métodos en los tres lenguajes. Asimismo, se presenta la experiencia de usuario en la resolución de este problema.

5.8. Resultados y Conclusiones

El objetivo del problema propuesto es medir la precisión numérica. Por lo tanto, en esta sección se presentan los resultados de los 6 métodos sin redondear los decimales. NIST no presenta la estimación de coeficientes para los polinomios de grado $1, 2, \dots, 9$. Sin embargo, se considera que el ajuste para polinomios de orden menor requiere menos recursos computacionales. Por lo tanto, si las estimaciones de tres o más métodos coinciden, los cálculos se pueden considerar correctos.

Se decidió omitir las tablas con los resultados de todos los ajustes ya que se consideró innecesario. No obstante, las tablas presentadas contienen los resultados de mayor interés para el objetivo de este proyecto. En caso de requerir el resultado de un ajuste que no se presenta en esta sección, se puede consultar la carpeta NIST del repositorio de GitHub de este trabajo: https://github.com/valperez/Tesis_Julia/tree/main/Tesis%20Julia%20con%20R/Code/NIST.

Ahora bien, se debe mencionar que todos los métodos obtienen los mismos resultados en los ajustes de los polinomios de orden 1 al 5. La tabla 5.2 es evidencia de ello.

A partir de este punto, el método GLM comienza a presentar fallas. Los resultados del ajuste del polinomio de grado 6 difieren de los calculados con el resto de los métodos como se puede observar en la tabla 5.3. Esto es de especial interés ya que, en teoría, este paquete fue creado para ajustar a modelos lineales. Este método no se recupera con los polinomios de mayor grado y termina fallando rotundamente.

	GLM	QRvEcon	MoorePenrose	Polynomials	R	Python
b0	4.3006543682	4.3006538792	4.3006538792	4.3006538792	4.3006538792	4.3006538792
b1	2.9237731063	2.9237726501	2.9237726501	2.9237726501	2.9237726501	2.9237726501
b2	0.9589166858	0.9589165208	0.9589165208	0.9589165208	0.9589165208	0.9589165208
b3	0.1481183596	0.1481183306	0.1481183306	0.1481183306	0.1481183306	0.1481183306
b4	0.0106383672	0.0106383648	0.0106383648	0.0106383648	0.0106383648	0.0106383648
b5	0.0002825197	0.0002825197	0.0002825197	0.0002825197	0.0002825197	0.0002825197

Figura 5.2. Resultados del polinomio grado 5

	GLM	QRvEcon	MoorePenrose	Polynomials	R	Python
b0	1.9043148726	-1.809755e+01	-1.809755e+01	-1.809755e+01	-1.809755e+01	-1.809755e+01
b1	0.0000000000	-2.229664e+01	-2.229664e+01	-2.229664e+01	-2.229664e+01	-2.229664e+01
b2	-0.4810934568	-1.057694e+01	-1.057694e+01	-1.057694e+01	-1.057694e+01	-1.057694e+01
b3	-0.2185586558	-2.598110e+00	-2.598110e+00	-2.598110e+00	-2.598110e+00	-2.598110e+00
b4	-0.0403353177	-3.486584e-01	-3.486584e-01	-3.486584e-01	-3.486584e-01	-3.486584e-01
b5	-0.0033914863	-2.424444e-02	-2.424444e-02	-2.424444e-02	-2.424444e-02	-2.424444e-02
b6	-0.0001074643	-6.834185e-04	-6.834185e-04	-6.834185e-04	-6.834185e-04	-6.834185e-04

Figura 5.3. Resultados del polinomio grado 6

En cambio, el resto de los métodos obtienen resultados similares en los ajustes hasta el polinomio de grado 9. No obstante, cuando se busca calcular el polinomio de grado 10, solamente las columnas **Polynomials**, **R** y **Python** de la tabla 5.4 muestran los resultados correctos.

En cuanto a los tiempos de ejecución, la tabla 5.5 presenta una comparación entre los métodos desarrollados. Las columnas corresponden al método utilizado mientras que los reglones representan el grado del polinomio.

De los métodos programados en Julia, el más rápido es el correspondiente al paquete **Polynomials**. **MoorePenrose** y **QRvEcon** no tardan mucho más, pero sí es notorio el salto que se da en el método

	GLM	QRvEcon	MoorePenrose	Polynomials	R	Python	NIST
b0	0.000000e+00	9.013426e+00	8.443046e+00	-1.467490e+03	-1.467490e+03	-1.467490e+03	-1.467490e+03
b1	0.000000e+00	1.652546e+00	1.364986e+00	-2.772180e+03	-2.772179e+03	-2.772179e+03	-2.772180e+03
b2	0.000000e+00	-5.767606e+00	-5.350763e+00	-2.316371e+03	-2.316371e+03	-2.316371e+03	-2.316371e+03
b3	0.000000e+00	-3.863666e+00	-3.341911e+00	-1.127974e+03	-1.127974e+03	-1.127974e+03	-1.127974e+03
b4	0.000000e+00	-6.703657e-01	-4.064616e-01	-3.544782e+02	-3.544782e+02	-3.544782e+02	-3.544782e+02
b5	0.000000e+00	1.806044e-01	2.577266e-01	-7.512421e+01	-7.512420e+01	-7.512420e+01	-7.512420e+01
b6	3.686442e-03	1.055234e-01	1.197715e-01	-1.087532e+01	-1.087532e+01	-1.087532e+01	-1.087532e+01
b7	1.917312e-03	2.144494e-02	2.314088e-02	-1.062215e+00	-1.062215e+00	-1.062215e+00	-1.062215e+00
b8	3.758850e-04	2.277483e-03	2.403994e-03	-6.701912e-02	-6.701911e-02	-6.701911e-02	-6.701912e-02
b9	3.281913e-05	1.262264e-04	1.316188e-04	-2.467811e-03	-2.467811e-03	-2.467811e-03	-2.467811e-03
b10	1.074670e-06	2.889643e-06	2.990001e-06	-4.029625e-05	-4.029625e-05	-4.029625e-05	-4.029625e-05

Figura 5.4. Resultados del polinomio grado 10

GLM siendo aproximadamente 1,000 veces más lento. A pesar de que un quinto de segundo no represente mucho tiempo, sí es mucho más del que le toma a los otros métodos. Los resultados de ejecución de R son levemente más lentos que los métodos más rápidos de Julia, pero Python demuestra un gran poder computacional al ser el lenguaje más rápido de la triada.

En cuanto a la experiencia de usuario, este proyecto representó un reto algebraico mayor al resto. La tarea de encontrar cuatro formas diferentes de solución a un problema fue un desafío. Como usuaria de los tres lenguajes presentados, quedo insatisfecha con los resultados de Julia, especialmente con los paquetes GLM y Polynomials. Los paquetes deben ser una herramienta que presenten un algoritmo optimizado de la teoría algebraica. El fracaso tan temprano del paquete GLM fue una sorpresa. Por otro lado, a pesar de que el paquete Polynomials obtiene una respuesta con alta precisión numérica, se siente la falta del cálculo de los estimadores. Esto no es una sorpresa ya que el enfoque del paquete

	Grado	GLM	QRvEcon	MoorePenrose	Polynomials	R	Python
1	1	0.1362870	0.00012086	0.00008646	0.00009836	0.001312351	0.013895273
2	2	0.1648300	0.00024928	0.00023900	0.00019864	0.001726389	0.000851154
3	3	0.1863261	0.00014762	0.00121714	0.00016612	0.002147818	0.000000000
4	4	0.1473972	0.00009532	0.00011170	0.00010084	0.002681637	0.000000000
5	5	0.1752443	0.00011164	0.00014202	0.00013140	0.002812576	0.000000000
6	6	0.2322631	0.00013628	0.00017562	0.00015960	0.002726793	0.000206709
7	7	0.2126248	0.00019364	0.00022930	0.00018628	0.002858210	0.000199223
8	8	0.1976703	0.00019236	0.00021262	0.00017024	0.007611227	0.002474499
9	9	0.2257354	0.00014516	0.00016850	0.00014092	0.004818010	0.000000000
10	10	0.1642656	0.00016096	0.00019044	0.00016220	0.004371214	0.000000000

Figura 5.5. Tiempos de ejecución para cada método

es toda la teoría relacionada con polinomios, no con ajustes de modelos lineales.

R y Python no decepcionan ni sorprenden. Ambos son lenguajes que llevan más tiempo siendo desarrollados por lo que la verdadera sorpresa sería que no funcionaran. Aún así, en el caso de R es necesario el conocimiento del argumento que representa la tolerancia del ajuste. En cambio, en Python se obtienen los resultados de manera sencilla y con pocas líneas de código. Por lo tanto, por la sencillez y rapidez del lenguaje, Python sería mi candidato principal para hacer el ajuste de un modelo lineal.

En el siguiente capítulo se retoma el enfoque en análisis de datos para desarrollar el ajuste de un modelo lineal usando una extensa cantidad de datos.

Capítulo 6

Modelos de Regresión Lineal

“Una actividad importante en estadística es la creación de modelos estadísticos que, se espera, reflejen aspectos importantes del objeto de estudio con algún grado de realismo. En particular, el objetivo del análisis de regresión es construir modelos matemáticos que describan o expliquen relaciones que pueden existir entre variables”, [Seber and Lee \(2003\)](#).

El uso de la regresión como método para mostrar la relación entre dos o más variables se remonta al siglo XIX. En 1875 fue Sir Francis Galton quien, usando semillas de guisantes, hizo el primer análisis de regresión. Galton distribuyó paquetes de dichas semillas a sus amigos para que ellos las sembraran y le regresaran el resultado. Los paquetes tenían casi el mismo peso, siendo *casi* la palabra clave, ya que existía una variación pequeña de peso. Así, Galton pudo establecer una relación entre los pesos de las semillas *madre* contra las semillas *hija*, [Stanton \(2001\)](#).

Ciertamente, el análisis de regresión se ha desarrollado de acuerdo a las demandas del tiempo. Hoy en día es posible diseñar una metodología que recabe información sobre los habitantes de un país sin la necesidad de entrevistar a todos y cada uno de ellos. Se toma una muestra significativa que busca obtener conclusiones sobre las características socioeconómicas y culturales de la población. Sin embargo, cuando el país estudiado tiene cerca de 130 millones de habitantes (como es el caso de México), la muestra contiene información de 15 millones de personas. La demanda actual necesita el constante desarrollo de programas con la capacidad de leer, analizar y guardar extensas cantidades de información.

Los modelos de regresión lineal múltiple son de los más usados en la estadística como una primera aproximación a análisis más complejos. El propósito de este capítulo es mostrar el uso de la regresión lineal múltiple en Julia, R y Python con un gran número de datos obtenidos del Censo de Población y Vivienda 2020.

6.1. El modelo

En capítulos anteriores se habló de la regresión lineal múltiple cuyo modelo se vuelve a considerar en este capítulo. El modelo se define como

$$\begin{aligned} y_i &= \beta_0 + \beta_1 X_{i1} + \cdots + \beta_k X_{ik} + \epsilon_i, \text{ para} \\ i &= 1, \dots, n \\ k &= 1, \dots, p \text{ (Gelman et al., 2021, p. 146)} \end{aligned} \tag{6.1}$$

donde ϵ_i representa el error del modelo que en el caso más elemental se supone sigue una distribución normal con media cero y desviación estándar σ . En este caso, y_i se refiere a la respuesta al i -ésimo de los

regresores; x_{ik} es el k -ésimo regresor al i -ésimo nivel; β_0 y β_k son los coeficientes del modelo.

Es útil entender el modelo con un ejemplo. Imagine que trabaja en un laboratorio donde se estudia el crecimiento de girasoles. Se tienen 50 plantas que se exponen a diferentes cantidades de agua, luz solar y tierra con abono. Se busca relacionar el crecimiento de los girasoles con cambios en los factores ambientales. Por lo tanto, se tiene una tabla donde en la primera columna se documenta el crecimiento de cada planta medido en centímetros. Las siguientes tres columnas registran las cantidades de agua, luz solar y tierra a la que se expone cada girasol. Por tanto, cada renglón representa uno de los 50 girasoles del experimento.

La variable respuesta y_i se refiere al crecimiento del girasol i -ésimo. Como hay 50 plantas, $n = 50$. Por otro lado, los regresores son el agua, la luz solar y la tierra con abono. Sea el agua el primer regresor, $x_{9,1}$ representa el nivel de agua del girasol 9. Lo mismo sucede con la luz y tierra. En este caso, $k = 3$ ya que hay tres regresores. Finalmente, β_k es el efecto que tienen los regresores en el crecimiento de los girasoles.

De manera matricial, Gelman define la expresión 6.1 como

$$y_i = X_i\beta + \epsilon_i, \text{ para } i = 1, \dots, n \text{ (Gelman et al., 2021, p. 146)}$$

donde X_i es el i -ésimo renglón de la matriz X de dimensión $n \times k$. Adicionalmente, se pide que la matriz X sea de rango completo cuyas razones no se discuten en este trabajo.

6.2. Los datos

Siguiendo con la idea de [Seber and Lee \(2003\)](#) se decidió tomar como ejemplo para este proyecto la relación entre el ingreso del mexicano con factores que le influyen. La información se obtuvo del Censo de Población y Vivienda (Censo) 2020 publicado por el Instituto Nacional de Estadística y Geografía (INEGI) que se encuentra en la página <https://www.inegi.org.mx/programas/ccpv/2020/default.html>. En México, el Censo se captura cada 10 años y se busca tener una muestra de todo el territorio nacional.

De acuerdo al [INEGI \(2020\)](#), el objetivo del Censo “es producir información sobre el volumen, la estructura y la distribución espacial de la población, así como de sus principales características demográficas, socioeconómicas y culturales; además de obtener la cuenta de las viviendas y sus características tales como los materiales de construcción, servicios y equipamiento, entre otros”.

Recabar la información anterior es una labor demandante que se divide en dos tipos de cuestionarios llamados “básico” y “ampliado”. En el cuestionario ampliado las preguntas incluyen especificaciones sobre los residentes del territorio nacional, las viviendas particulares y los migrantes internacionales. Mientras que el básico busca información general centrada en el acceso a servicios públicos en el hogar y la escolaridad de sus residentes.

En este trabajo los resultados que se utilizan provienen del cuestionario ampliado cuyas 103 preguntas resultan en cerca de 200 variables de estudio. Más aún, el Censo fue aplicado a 4 millones de viviendas a lo largo de la República Mexicana que resultó en la obtención de información de más de 15 millones de personas.

En este ejercicio se eligió un tema de interés personal: los ingresos.

Más específicamente, se usa la regresión lineal múltiple para observar el efecto de diferentes factores al ingreso de cada persona. Usualmente, los modelos se basan en una mezcla de teoría, lógica, experiencia y referencias. En este caso, el modelo propuesto es

$$\begin{aligned}
 y = & \beta_0 + \beta_1 * \text{horas}_{trabajadas} + \beta_2 * \text{sexo} + \\
 & \beta_3 * \text{edad} + \beta_4 * \text{escolaridad} + \beta_5 * \text{entidad}_{trabajo} + \\
 & \beta_6 * \text{posicion}_{laboral} + \beta_7 * \text{alfabetismo} + \beta_8 * \text{aguinaldo} + \\
 & \beta_9 * \text{vacaciones} + \beta_{10} * \text{servicio}_{medico}
 \end{aligned} \tag{6.2}$$

6.3. Planteamiento del problema

El Censo mexicano es un estudio extensivo sobre la vida de sus habitantes. El área de interés determina el filtro de información y la selección de columnas. Es necesario entender la estructura de las encuestas y la relación entre ellas. Para ello, el INEGI también proporciona el diccionario ampliado que se encuentra en la página [https:](https://www.inegi.org.mx/programas/ccpv/2020/default.html#Microdatos)

[//www.inegi.org.mx/programas/ccpv/2020/default.html#Microdatos](https://www.inegi.org.mx/programas/ccpv/2020/default.html#Microdatos) dentro del apartado Documentación de la base de datos. La información que aporta el diccionario es clave ya que expone los códigos y las nomenclaturas que permiten entender como se paso de tener respuestas en hojas de papel a tenerlas en una base de datos.

Los resultados del Censo se dividen en tres partes: Viviendas, Personas y Migrantes. Para este trabajo, se utilizó la base de datos correspondientes a Personas ya que contiene la información necesaria para llevar a cabo el ajuste del modelo de ingresos propuesto en 6.2. El volumen de datos con el que se trabajo es amplio por lo que lo primero

fue seleccionar las variables en la base de datos que representan a los regresores. Posteriormente, se agregaron los siguientes filtros:

1. Se seleccionaron solamente las personas que tienen un trabajo remunerado. Es decir, no se consideró a las personas que se ocupan de las labores del hogar, son jubiladas o pensionadas, estudiantes o tienen alguna incapacidad que les impida tener un sueldo.
2. Se descartó a las personas que viven y trabajan fuera de la República Mexicana.
3. Se seleccionaron a las personas que especificaron horas trabajadas e ingreso ganado.
4. Cada regresor es resultado de una pregunta hecha y tiene una variable asignada. Si el entrevistado decide no responder a alguna pregunta se marca la respuesta como **No especificado**. En este caso, se eliminaron a las personas que no respondieron alguna de las preguntas que corresponden a los regresores.

Con los filtros anteriores la cantidad de datos con la que se trabaja se reduce de 15 a 3.5 millones. Este proceso toma alrededor de 20 minutos en ejecutarse y, posteriormente, se guardó la nueva base de datos. El filtrado de la información se ejecutó en Julia y se guardó para utilizarla posteriormente en R y Python. El código de lectura de la base de datos y los comandos utilizados para la selección de información son los siguientes. A pesar de ser un código largo, está debidamente documentado.

```
### Inicio de código para filtrado de datos  
julia> using CSV, DataFrames, StatsBase,
```

GLM, Random, CategoricalArrays

```
# Equivalente a set.seed de R
julia> Random.seed!(99)

# Lectura la base de datos
# (toma alrededor de 4 minutos en cargar)
julia> personas = CSV.read("Personas00.csv",
    DataFrame)

# Lista con columnas necesarias para el ajuste
julia> col_sel = ["ID_PERSONA", "SEXO", "EDAD",
    "NIVACAD", "ALFABET", "INGTRMEN", "HORTRA",
    "SITTRA", "ENT_PAIS TRAB", "AGUINALDO",
    "VACACIONES", "SERVICIO_MEDICO", "CONACT",
    "ENT"]

# Se seleccionan de la base de datos
julia> personas_filt = personas[:, col_sel]

# # # FILTRO 1
julia> cond_act = [10, 13, 14, 15, 16, 17,
    18, 19, 20]
julia> personas_filt = subset(personas_filt,
    :CONACT => ByRow(in(cond_act)),
    skipmissing = true)

# # # FILTRO 2
julia> personas_filt = subset(personas_filt,
```

```

        :ENT_PAIS_TRAB => ByRow(<(33)),
        skipmissing = true)

julia> personas_filt = subset(personas_filt,
        :ENT => ByRow(<(33)), skipmissing = true)

# # # FILTRO 3
julia> personas_filt = subset(personas_filt,
        :HORTRA => ByRow(!=(999)),
        skipmissing = true)

julia> personas_filt = subset(personas_filt,
        :INGTRMEN => ByRow(!=(999999)),
        skipmissing = true)

# # # FILTRO 4
julia> function diferente_a(dataframe, columna,
        condicion)
        dataframe = subset(dataframe,
        columna => ByRow(!=(condicion)),
        skipmissing = true)

        return dataframe
end

julia> categorias_9 = ["SEXO", "AGUINALDO",
        "VACACIONES", "SERVICIO_MEDICO",
        "ALFABET", "SITTRA"]

```

```

julia> categorias_99 = ["NIVACAD"]

julia> for i = 1:length(categorias_9)
    personas_filt = diferente_a(personas_filt,
    categorias_9[i], 9)
end

julia> for i = 1:length(categorias_99)
    personas_filt = diferente_a(personas_filt,
    categorias_99[i], 99)
end

# Finalmente, se guarda el nuevo dataframe
julia> CSV.write("personas_filtradas.csv",
    personas_filt)

```

6.4. Factores y Sub-ajustes

Lo primero que se debe verificar es que la lectura de datos haya clasificado las variables de manera correcta. En el modelo presentado en [6.2](#), la mayoría de las variables del ajuste son categorías o factores, pero Julia las lee como `Int64`. Por ejemplo, la variable `NIVACAD` responde a la pregunta sobre el último año o grado aprobado por el entrevistado. Hay 15 respuestas que corresponden a todos los niveles académicos posibles, desde no haber recibido ningún tipo de educación formal hasta haber obtenido un doctorado. Cada respuesta se identifica con un número que va del 0 al 14. Por lo tanto, en el modelo propuesto [6.2](#), el regresor `NIVACAD` es de tipo categórico y tiene 15 niveles. Ya que las respuestas son número, el comando `CSV.read` de Julia identifica incorrectamente la

columna como tipo `Int64`. Esto sucede con todos los factores del modelo compilados en la lista `categorias` del código mostrado a continuación. En el siguiente código se muestra la manera en la que se convirtieron las variables a factores.

```
julia> using DataFrames
julia> data = CSV.read("personas_filtradas.csv",
DataFrame)

# Vector con todas las categorias
julia> categorias = ["SEXO", "AGUINALDO",
"VACACIONES", "SERVICIO_MEDICO", "ALFABET",
"NIVACAD", "ENT_PAIS_TRAB", "ENT", "SITTRA"]

julia> transform!(data,
names(data, vector_categorias) .=> categorical,
renamecols=false)
```

Si se omitiera el paso anterior el resultado del ajuste no tendría sentido ya que se considerarían a los regresores de tipo categórico como regresores continuos. Por tanto, no proporcionarían el efecto de cada categoría en la variable de respuesta.

Uno de los objetivos de este trabajo es ilustrar las capacidades de los tres lenguajes de programación por lo que se tomó el modelo 6.2 y se retiraron algunas variables. Se puede pensar como que se tomaron diferentes subconjuntos de variables y, con ellas, se hizo el ajuste de las regresiones para notar si había cambios en la precisión del ajuste. La variable de respuesta es la misma en todos los ahora denominados *sub-ajustes*.

El primer *sub-ajuste* se definió con los primeros 5 regresores del

modelo 6.2 y se nombró como **fit5** (por la cantidad de regresores). Es decir, el modelo **fit5** es

$$y = \beta_0 + \beta_1 * horas_{trabajadas} + \beta_2 * sexo + \beta_3 * edad + \beta_4 * escolaridad + \beta_5 * entidad_{trabajo}$$

El segundo *sub-ajuste* llamado **fit6** tiene los 5 regresores incluidos en **fit5** y uno extra, la posición laboral. Por tanto, el modelo **fit6** queda de la siguiente manera:

$$y = \beta_0 + \beta_1 * horas_{trabajadas} + \beta_2 * sexo + \beta_3 * edad + \beta_4 * escolaridad + \beta_5 * entidad_{trabajo} + \beta_6 * posicion_{laboral}$$

Los sub-ajustes **fit5** y **fit6** siguen el mismo orden que 6.2. Esto no es una coincidencia. El orden de los regresores del modelo 6.2 está pensado precisamente para que cada variable sumada se agregue al conjunto de variables anterior y cree un nuevo modelo *fit*.

6.4.1. Código en Julia

Sería interesante ver la reacción de Galton si supiera los alcances a los que ha llegado su experimento de semillas. Él mismo estaría de acuerdo con que no es lo mismo hacer un ajuste con 5 observaciones a hacer uno con 5 millones de ellas. El código es el mismo, pero en el trabajo de máquina se observan cambios en tiempo de ejecución y precisión numérica. Ambos factores son las razones por las que se eligió este ejercicio. En esta sección se muestra como se midieron dichos cambios.

Cada uno de los modelos *fit* ya mencionados se ejecutaron con muestras de 500, 5 mil, 50 mil, 500 mil y 2.5 millones de observaciones.

Es decir, se usaron cada una de las expresiones `fit` para el ajuste de modelos con los volúmenes de información antes mencionados. La elección de observaciones se hizo al azar usando el comando `sample` en Julia. Una vez seleccionadas, se guardaba el dataframe generado para usar exactamente los mismos datos en R y Python. Guardar las muestras generadas tiene el propósito de poder utilizar la misma información en los ajustes y obtener así, un punto de comparación en la precisión del cálculo de los coeficientes.

La ejecución de lo anterior es repetitivo por lo que se buscó fuera hecho de la manera más rápida y eficiente posible. Se desarrolló una función para cada `fit` cuyos argumentos fueran la cantidad de observaciones que se utilizan y el nombre con el que se guarda el archivo. El nombre de las funciones coinciden con la cantidad de regresores que se están utilizando.

Inclusive con este acercamiento, se puede notar que crear una función para cada modelo resulta repetitivo. Se pudo haber desarrollado una sola función que también tomara como argumento la fórmula a utilizarse en el ajuste. Sin embargo, se decidió no hacerlo de esa forma ya que se consideró de suma importancia tener la fórmula escrita en cada función.

Debido a la similitud entre funciones y su ejecución se muestran solamente las correspondientes a `fit5` y `fit10` a manera de ejemplo. El código para la función `fit5` es el siguiente.

```
### FIT BASE ###  
julia> function fit5(cantidad_sample, nombre_facil)  
    nombre_fit = "fit5"  
  
    sample_rows = sample(1:nrow(data), cantidad_sample,  
                        replace=false)
```



```

df_sample = data[sample_rows, :]

nombre_completo = nombre_facil*"_"*nombre_fit*".csv"
# Guardamos el documentos para usarlo en R
CSV.write(nombre_completo, df_sample)

tiempos = []
local sample_fit
# Hacemos el fit
for i in runs
    sample_fit = @timed lm(@formula(INGTRMEN ~ HORTRA
    + SEXO + EDAD + NIVACAD + ENT_PAIS_TRAB),
    df_sample)
    push!(tiempos, sample_fit[2])
end

aux = "res_"
nombre_completo = aux*nombre_completo
CSV.write(nombre_completo, coeftable(sample_fit[1]))
return tiempos
end

# Fit 5
julia> tiempos_fit5 = DataFrame()
julia> tiempos_fit5[:, "500"] = fit5(500, "500")
julia> tiempos_fit5[:, "5mil"] = fit5(5000, "5mil")
julia> tiempos_fit5[:, "50mil"] = fit5(50000, "50mil")
julia> tiempos_fit5[:, "500mil"] = fit5(500000, "500mil")

```

```
julia> tiempos_fit5[!, "2500mil"] = fit5(2500000, "2500mil")
julia> CSV.write("tiemposJulia_Censo_fit5.csv", tiempos_fit5)
```

Por otro lado, el código para fit10 es el siguiente.

```
### FIT 10###
julia> function fit10(cantidad_sample, nombre_facil)
    nombre_fit = "fit10"

    sample_rows = sample(1:nrow(data), cantidad_sample,
        replace=false)

    df_sample = data[sample_rows, :]

    nombre_completo = nombre_facil*"_"*nombre_fit*".csv"
    # Guardamos el documentos para usarlo en R
    CSV.write(nombre_completo, df_sample)

    tiempos = []
    local sample_fit
    for i in runs
        # Hacemos el fit
        sample_fit = @timed lm(@formula(INGTRMEN ~ HORTRA
            + SEXO + EDAD + NIVACAD + ENT_PAIS_TRAB + SITTRA
            + ALFABET + AGUINALDO + VACACIONES
            + SERVICIO_MEDICO), df_sample)
        push!(tiempos, sample_fit[2])
    end

    aux = "res_"
```

```

        nombre_completo = aux*nombre_completo
        CSV.write(nombre_completo, coeftable(sample_fit[1]))
        return tiempos
    end

## Fit 10: Fit 5 + SITTRA + ALFABET + AGUINALDO +
## VACACIONES + SERVICIO_MEDICO
julia> tiempos_fit = DataFrame()
julia> tiempos_fit[!, "500"] = fit10(500, "500")
julia> tiempos_fit[!, "5mil"] = fit10(5000, "5mil")
julia> tiempos_fit[!, "50mil"] = fit10(50000, "50mil")
julia> tiempos_fit[!, "500mil"] = fit10(500000, "500mil")
julia> tiempos_fit[!, "2500mil"] = fit10(2500000, "2500mil")
julia> CSV.write("tiemposJulia_Censo_fit10.csv", tiempos_fit)

```

6.5. Resultados y Conclusiones

En cualquiera de los sub-ajustes, la mayoría de los regresores son de tipo categórico por lo cual tiene diferentes niveles. La variable NIVACAD comentada en la sección anterior es un ejemplo de ello. Otro ejemplo es la variable llamada ENT_PAIS_TRAB que corresponde a la entidad mexicana donde el entrevistado trabajó más recientemente. Ya que México se divide en 32 entidades federativas, cada estado representa un nivel del regresor. Si se cuentan los niveles de cada factor, al final el modelo con más variables, fit10, tiene un total de 54 regresores. La tabla 6.1 es una muestra resumida de los resultados obtenidos en Julia usando 2.5 millones de observaciones en el modelo fit10.

Nombre	Coficiente	Error estándar	t	$Pr(> t)$	95 % inf	95 % sup
Ordenada al origen	4133.26	125.42	32.95	$4.16e^{-238}$	3887.43	4379.09
SEXO:3	-1407.49	20.27	-69.42	0	-1447.23	-1367.76
EDAD	33.30	0.72	46.12	0	31.88	34.71
NIVACAD:1	208.16	209.00	0.99	0.32	-201.48	617.80
NIVACAD:2	287.16	68.59	4.18	$2.38e^{-05}$	152.72	421.60
NIVACAD:3	629.53	71.20	8.84	$9.42e^{-19}$	489.98	769.08
⋮	⋮	⋮	⋮	⋮	⋮	⋮
SITTRA:3	-692.44	32.10	-21.57	$3.30e^{-103}$	-755.35	-629.53
ALFABET:3	-544.43	66.47	-8.19	$2.59e^{-16}$	-674.70	-414.16
AGUINALDO:2	-459.48	34.30	-13.39	$6.51e^{-41}$	-526.71	-392.25
VACACIONES:4	-843.60	38.65	-21.83	$1.31e^{-105}$	-919.35	7678.85
SERVICIO MEDICO:6	-1110.25	34.21	-32.46	$4.91e^{-231}$	-1177.30	-1043.21

Tabla 6.1. Resultados para el modelo fit10 con 2.5 millones de observaciones en Julia

Como se observa en la tabla 6.1, Julia no solo da el cálculo de los coeficientes, también da indicadores como lo son el error estándar y el intervalo de confianza.

En R se utilizó la función `lm` para hacer los ajustes mientras que en Python se usó la función `LinearRegression()` del paquete `sklearn` explicado a detalle en la sección 3.2.4. El código completo en ambos lenguajes se encuentra en el apéndice A.1.

Los tres lenguajes hacen el ajuste de manera correcta y con el mismo nivel de precisión. Se notó que, para cada *sub-ajuste*, los tres volúmenes de información más pequeños obtenían un resultado de ajuste muy distinto. Sin embargo, para los ajustes donde se utilizaron 500 mil y 2.5 millones de observaciones el cálculo de coeficientes era

muy similar. En la tabla 6.2 se presenta una comparativa de los resultados obtenidos en los tres lenguajes usando el *sub-ajuste* fit10 con 2.5 millones de datos. A manera de ejemplo, se seleccionaron todos los regresores continuos y un nivel de cada regresor categórico. Asimismo, en la tabla se omitió redondear los resultados del ajuste para observar que el nivel de precisión. En este caso, la precisión numérica se mantuvo igual en los tres lenguajes.

Regresor	Julia	R	Python
Ordenada al origen	4044.00297	4044.00297	4044.00297
HORTRA	41.12933	41.12933	41.12933
SEXO:3	-1403.30415	-1403.30415	-1403.30415
EDAD	34.34650	34.34650	34.34650
NIVACAD:9	4276.80672	4276.80672	4276.80672
ENT_PAIS_TRAB:13	-671.72888	-671.72888	-671.72888
SITTRA:2	-668.43938	-668.43938	-668.43938
ALFABET:3	-534.68337	-534.68337	-534.68337
AGUINALDO:2	-483.42688	-483.42688	-483.42688
VACACIONES:4	-812.44030	-812.44030	-812.44030
SERVICIO_MEDICO: 6	-1118.54895	-1118.54895	-1118.54895

Tabla 6.2. Comparación de resultados para el modelo fit10 con 2.5 millones de observaciones en R, Julia y Python

Por otro lado, en cuanto a tiempo de ejecución, la tabla A.2 muestra el tiempo que le tomo a los tres lenguajes ejecutar cada ajuste. Los tres lenguajes presentan un aumento de tiempo a medida que aumenta el volumen de observaciones a ajustar. Asimismo, se presenta un aumento menos significativo en los tiempos de ejecución a medida que aumenta el número de regresores. Julia y Python presentan tiempos de ejecución muy similares en todos los ajustes donde se utilizaron 500 mil datos o

menos. **Python** mantiene su rapidez cuando se utilizan 2.5 millones de datos, mientras que **Julia** presenta un aumento en tiempos de ejecución a medida que aumentan la cantidad de regresores utilizados para el ajuste.

La diferencia en tiempo se hace más notoria si se considera la ejecución del código completo. El desempeño de **Julia** parece ser más lento en general ya que invierte una cantidad considerable de tiempo en guardar los resultados en un archivo de tipo **CSV**. Por otro lado, **Python** hace mucho más rápido en hacer la ejecución del código.

En cambio, **R** es el lenguaje más lento de la triada de lenguajes. Su tiempo mínimo de ejecución es de 25.170 segundos mientras que el tiempo máximo es de 42.471 segundos. Adicionalmente, el lenguaje presenta dificultades al manejar grandes cantidades de datos. Se presentó una sensación de entorpecimiento en la ejecución del código ya que los cálculos eran lentos y, en ocasiones, se presentó un mensaje que pedía reiniciar la sesión de **R**.

En suma, en este ejemplo el paquete **GLM** de **Julia** cumple su propósito de ajustar el modelo de manera correcta y rápida. Más aún, el paquete también proporciona los estimadores necesarios para un análisis completo de regresión. En cambio, si bien las funciones utilizadas en **R** y **Python** no presentan dicha tabla, sí existen los comandos para obtener los estimadores.

Los tres lenguajes **Julia**, **R** y **Python** proporcionan herramientas que permiten leer una base de datos y ajustar un modelo lineal de manera precisa. Además, los lenguajes ofrecen herramientas para calcular los estimadores clave para hacer un análisis de regresión. Las diferencias entre los lenguajes se presentan al considerar el tiempo de ejecución y el rendimiento. **Julia** y **Python** realizan el ajuste de manera rápida y fluida, sin saturaciones de memoria o falta de respuesta del lenguaje. En cambio, **R** presenta complicaciones en el manejo de bases de datos

extensas. En consecuencia, es el lenguaje más lento en realizar el ajuste y pide el constante reinicio de la sesión en la que se está trabajando.

Los obstáculos presentados con R dieron pie a la realización de que, hasta este punto, la ejecución de Julia nunca ha presentado fallas. Investigando sobre las capacidades de Julia se encontró en repetidas ocasiones que tiene un propósito muy claro: ser el lenguaje más eficiente con la mayor cantidad de aplicaciones posible. Por lo tanto, en el siguiente capítulo se cambia el enfoque de análisis de regresión para ilustrar un ejemplo de discriminación de modelos en diseños de experimentos. El objetivo del problema es mostrar la capacidad de los lenguajes de ejecutar una función que conlleva una gran cantidad de cálculos complejos.

Capítulo 7

Discriminación de modelos

“Los experimentos son realizados por científicos en prácticamente todos los campos de investigación, usualmente para descubrir algo sobre un determinado proceso o sistema. Se puede definir un experimento como una prueba o una serie de pruebas en la que se realizan cambios intencionados en las variables de entrada de un proceso o sistema para observar e identificar los cambios que se puedan observar en variable de salida”, [Montgomery \(2017\)](#).

Se debe llevar un registro de los cambios de las variables de entrada y salida con el objetivo de establecer relaciones y obtener conclusiones. La información recolectada sobre los cambios en las variables se debe analizar por medio de diversos métodos estadísticos. Montgomery establece que “usualmente es muy útil presentar los resultados de los experimentos en términos de un modelo empírico, esto es, una ecuación derivada de los datos que exprese la relación entre la respuesta y factores de diseño importantes”.

Los resultados que se obtienen de los experimentos representan un avance científico indispensable. No obstante, la realización de

experimentos tiene un obstáculo importante. [Box et al. \(2005\)](#) lo enuncia de manera precisa como “El conocimiento es poder. Es la clave para la innovación y los beneficios. Pero, la obtención de nuevo conocimiento puede ser complejo, prolongado y costoso”. El diseño de experimentos es el método que se encarga de determinar una manera eficiente de hacer experimentos donde se obtengan resultados concretos.

El objetivo del experimento es determinar los factores que tengan mayor influencia en los cambios de la respuesta. La elección de dichos factores resulta en posibles modelos diferentes. En ocasiones puede ocurrir que los resultados de un experimento sean ambiguos en la decisión de los factores que afectan la variable de respuesta. En estos casos puede resultar conveniente agregar algunos ensayos extras. [Meyer et al. \(1996\)](#) proponen el criterio MD, de Model Discrimination. Este criterio busca determinar que factores contribuyen a los cambios en la variable de respuesta.

En este capítulo se comenta un resumen del artículo publicado por [Meyer et al. \(1996\)](#). Asimismo, se exponen dos ejemplos donde se utiliza el criterio MD para mostrar su uso en la discriminación entre posibles modelos. El objetivo de este capítulo es exponer el poder de cómputo de los tres lenguajes en discriminación de modelos en el contexto de diseño de experimentos. El criterio MD se programó en Julia y Python mientras que en R se utilizaron los paquetes `BsMD` y `BsMD2` desarrollados por Ernesto Barrios y Patricia Vela respectivamente.

7.1. Preliminares y definiciones

Dado que el enfoque de esta tesis no es el diseño de experimentos, se presenta una lista de términos y sus definiciones obtenidos de [Lawson](#)

(2015). El objetivo es proporcionar los antecedentes necesarios para la comprensión de este capítulo.

1. Ensayos: es la acción donde el experimentador cambia al menos una de las variables estudiadas para observar el efecto de su acción.
2. Variable independiente: es una de las variables bajo estudio que se controla en un valor o nivel objetivo durante un experimento. El nivel se cambia de manera sistemática de ensayo a ensayo para determinar el efecto que tiene en la respuesta.
3. Variable dependiente (o variable respuesta denotada por Y): es la característica del experimento que se mide después de cada ensayo. La magnitud de la respuesta depende de los cambios en las variables independientes o factores y en los factores confundidos.
4. Factores confundidos: surgen cuando cada cambio que hace el experimentador para un factor, entre ensayos, está acoplado con un cambio idéntico en otro factor. En esta situación es imposible determinar que factor causa cualquier cambio observado en la respuesta o variable dependiente.
5. Efecto: es el cambio en la respuesta causado por el cambio en un factor o variable independiente. Después de que los ensayos en un diseño de experimentos son realizados, el efecto se puede estimar calculándolo de los datos de respuesta observados.
6. Diseño experimental: es una colección de experimentos o ensayos planeados con anticipación a la ejecución. Los ensayos seleccionados en un diseño de experimentos dependerán del propósito del diseño.

7. Diseño factorial. Lawson lo define como un experimento donde los ensayos consisten en todas las posibles combinaciones de los niveles de los factores estudiados. Por su parte, [Montgomery \(2017\)](#) expande la definición al exponer un ejemplo. Un diseño factorial es aquel en donde si hay a niveles del factor A y b niveles del factor B cada ensayo contiene todas las combinaciones ab de tratamientos. Son los diseños más eficientes para estudiar los efectos de dos o más factores.
8. Diseño factorial fraccionado: es una variación del diseño factorial básico donde solo se realizan un subconjunto de ensayos.
9. Factor activo: aquellos factores que tienen influencia real en la respuesta.
10. Factor no activo: aquellos factores que no tienen influencia real en la respuesta.

7.2. Metodología General

En esta sección se pretende explicar la metodología propuesta por [Meyer et al. \(1996\)](#) para la discriminación de modelos cuya teoría matemática se expone a detalle en las siguientes dos secciones. Algunos pasos de este procedimiento incluyen teoría de diseños de experimentos y teoría bayesiana. Esos temas están fuera del alcance de esta tesis. Sin embargo, se presentan referencias donde se puede indagar más sobre el tema.

Suponga que le interesa un fenómeno y decide hacer un experimento donde se recolecten datos para conocer más sobre el objeto de interés. La planeación de este tipo de experimentos se conoce como diseño estadístico de experimentos que [Montgomery \(2017\)](#)

define como “el proceso de planeación del experimento para que se recopilen datos que puedan ser analizados por métodos estadísticos que resulten en conclusiones objetivas y válidas.”

Montgomery también expone una serie de pautas para realizar un diseño de experimentos. El primer paso se menciona anteriormente y es distinguir un fenómeno al que se le pueda diseñar un experimento para conocer más sobre él. El segundo paso consiste en la elección de la variable de respuesta, los factores que influyen en ella y los niveles a los que se someten. Meyer propone el caso donde se toma una variable de respuesta Y , posiblemente afectada por j factores que tienen dos niveles cada uno. Las combinaciones de factores activos crean un modelo M_i para Y .

El tercer paso es la elección del diseño experimental que, en el caso de Meyer, la elección es un experimento factorial fraccionado donde se realiza solo un subconjunto de los ensayos posibles. En este punto, Meyer propone incluir conocimiento previo (también conocido como información *a priori*) del fenómeno. Se completa el diseño experimental al asignar probabilidades a priori a cada uno de los posibles modelos denominadas $P(M_i)$. Se busca proponer una relación lineal entre factores ya que resulta en la creación de modelos más simples.

El cuarto paso del diseño de experimentos es la realización del experimento. En este paso se registra el cambio en los niveles de cada factor junto con su efecto en la variable respuesta denominada Y . La información corregida con la realización del experimento se conoce como conocimiento posterior (o *a posteriori*) ya que se conoce después de haber realizado el experimento. En este punto Meyer utiliza teoría bayesiana para calcular la probabilidad $P(M_i|Y)$ de que cada modelo M_i sea el correcto. Para facilitar la comprensión de la probabilidad, se

invita a que lea $P(M_i|Y)$ en voz alta.

El quinto paso propuesto por Montgomery es el análisis estadístico de los datos. Meyer propone calcular la probabilidad de que cada factor j está activo denominada P_j . Entre mayor sea la probabilidad P_j , mayor es la posibilidad de que el factor j esté activo. Si existe una diferencia clara y prominente entre las probabilidades P_j entonces se puede hacer una separación entre factores activos y no activos. En el caso contrario, Meyer establece que se deben realizar más ensayos para discriminar entre los factores y, consecuentemente, entre modelos.

Es en la elección de esos ensayos donde se presenta el criterio MD. El criterio MD valora que ensayos son necesarios repetir para aclarar el efecto de los factores en la variable respuesta Y . Además, la propuesta en el artículo de Meyer utiliza un algoritmo de intercambio donde se calcula el valor MD para todas las combinaciones de ensayos posibles. Se selecciona el conjunto de ensayos con mayor MD para realizarlos nuevamente y obtener resultados concretos.

Finalmente, el último paso del diseño estadístico de experimentos es la obtención de conclusiones. En la metodología de Meyer, se llega a este paso cuando los ensayos extras hacen una clara distinción entre factores activos y no activos. Con esto, se cumple el objetivo de describir el fenómeno estudiado con el modelo más probable. El diagrama 7.1 muestra de manera visual la metodología enunciada.

7.3. Criterio MD

Esta sección presenta una explicación matemática detallada del criterio MD propuesto por Meyer et al. (1996). Sea Y el vector de respuestas de tamaño $n \times 1$ de un experimento factorial fraccionado con k factores. El modelo que mejor describa Y depende de los

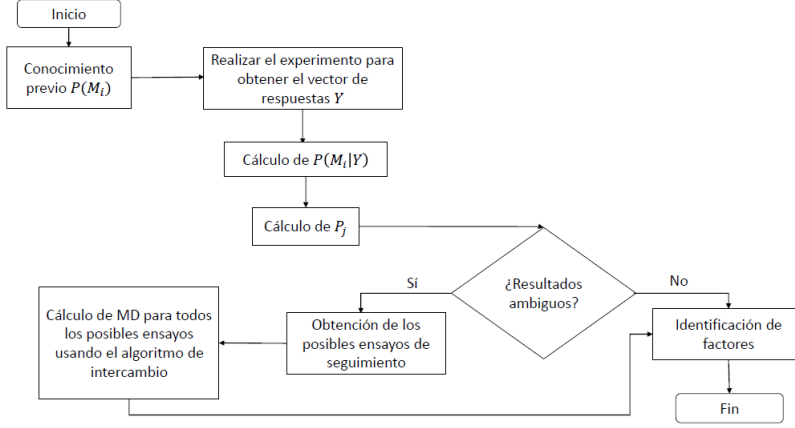


Figura 7.1. Diagrama de metodología descrita por Meyer et al. (1996)

factores activos en el experimento y en este análisis se consideran todas las posibles combinaciones de dichos factores. Sea M_i el modelo con una combinación particular de factores activos f_i donde $0 \leq f_i \leq k$. Condicionado a que M_i sea el modelo verdadero, se asume un modelo lineal normal, $Y \sim N(X_i\beta_i, \sigma^2 I)$.

La matriz X_i es la matriz de regresión para M_i e incluye los efectos principales para cada factor activo y su interacción hasta cualquier orden deseado. Sea t_i el número de efectos (sin incluir el término constante t_0) en M_i . Se denota además a M_0 como el modelo sin factores activos.

Posteriormente, se asignan distribuciones a priori no informativas al término constante β_0 y el error la desviación estándar σ . Ambos valores son comunes en todos los modelos. Por lo tanto, $p(\beta_0, \sigma) \propto \frac{1}{\sigma}$. Los coeficientes restantes β_i tienen distribuciones normales a priori independientes con media 0 y desviación estándar $\gamma\sigma$.

Se completa el modelo al asignar probabilidades previas a cada uno

de los posibles modelos. Otro supuesto es la regla de Pareto o *sparsity of effects principle* que es pensar que cuando hay varias variables, el sistema es más probable es el dominado por pocos efectos principales e interacciones de orden bajo [Montgomery \(2017\)](#). Se asume que existe la probabilidad π , $0 < \pi < 1$ de que cualquier factor esté activo. Se asume también que la información previa de que un factor esté activo no tiene efecto en la creencia de que otros factores estén activos. Por lo tanto, la probabilidad a priori del modelo M_i sigue una función de probabilidad $P(M_i) = \pi_i^f (1 - \pi)^{k-f_i}$.

Después, se realiza el experimento y se obtienen los resultados en el vector de datos Y . Con la nueva información se pueden actualizar las distribuciones de los parámetros de cada modelo así como la probabilidad de que los modelos sean correctos. En este punto se utiliza teoría de estadística bayesiana que va más allá de lo alcances de esta tesis. En caso de que se busque más información del tema se recomienda leer [Mendoza and Regueiro \(2011\)](#).

La probabilidad posterior de que el modelo M_i describa los datos de manera precisa es

$$P(M_i|Y) \propto \pi_i^f (1 - \pi)^{k-f_i} \gamma^{-t_i} |\Gamma_i + X_i' X_i|^{-1/2} S_i^{-(n-1)/2},$$

donde

$$\hat{\beta}_i = (\Gamma_i + X_i' X_i)^{-1} X_i' Y, \quad (7.1)$$

$$\Gamma_i = \frac{1}{\gamma^2} \begin{pmatrix} 0 & 0 \\ 0 & I_{t_i} \end{pmatrix} \quad (7.2)$$

y

$$S_i = (Y - X_i\hat{\beta}_i)'(Y - X_i\hat{\beta}_i) + \hat{\beta}_i' \Gamma_i \hat{\beta}_i. \quad (7.3)$$

Asimismo, se puede sumar el conjunto de $P(M_i|Y)$ donde el factor j esté incluido. El resultado es la probabilidad *a posteriori* P_j de que el factor j esté activo,

$$P_j = \sum_{M_i: \text{factor}_j \text{ activo}} P(M_i|Y) \quad (7.4)$$

El conjunto de probabilidades $\{P_j\}$ ofrece un resumen de la actividad de cada uno de los factores del experimento.

En este punto del análisis ya se tienen las probabilidades $P(M_i|Y)$ donde, si alguna es cerca a 1, señalaría el modelo que mejor describe los datos. Sin embargo, es común que los datos no identifiquen sin ambigüedad un modelo en particular. [Meyer et al. \(1996\)](#) desarrolla un método para identificar un conjunto de ensayos cuya realización aclara la actividad de los factores.

Meyer describe el diseño de discriminación de modelos (MD) propuesto por Box y Hill en 1967 de la siguiente manera. Sea Y^* el vector de datos correspondiente a los resultados de los nuevos ensayos. Sea $P(Y^*|M_i, Y)$ la densidad predictiva de Y^* dados los datos iniciales Y y el modelo M_i . Entonces, el cálculo del valor MD es

$$MD = \sum_{0 \leq i \neq j \leq m} P(M_i|Y)P(M_j|Y) \int_{-\infty}^{\infty} p(Y^*|M_i, Y) \times \ln\left(\frac{p(Y^*|M_i, Y)}{p(Y^*|M_j, Y)}\right) dY^*$$

Sea p_i la densidad predictiva para una nueva observación condicionada a las observaciones originales Y cuando M_i sea el modelo correcto. Entonces, el diseño del criterio es

$$MD = \sum_{0 \leq i \neq j \leq m} P(M_i|Y)P(M_j|Y)I(p_i, p_j) \quad (7.5)$$

donde $I(p_i, p_j) = \int p_i \ln\left(\frac{p_i}{p_j}\right)$ es la información de Kullback-Leibler que describe la información media para discriminar a favor de M_i contra M_j cuando M_i es el verdadero modelo. Adicionalmente, el cociente $\frac{p_i}{p_j}$ se puede entender como la probabilidad en favor de M_i contra M_j dada la información de los ensayos extras.

La intuición detrás de la fórmula del criterio MD puede resultar más sencilla de entender si se considera el ejemplo donde solo se tienen dos modelos posibles, M_1 y M_2 . El valor MD es proporcional a la suma del valor esperado de $\ln(p_1/p_2)$ dado M_1 y el valor condicional esperado de $\ln(p_2/p_1)$ dado M_2 . Entonces, se busca un diseño que calcule una probabilidad alta a favor de M_1 si este es el modelo correcto; pero que también calcule lo equivalente para M_2 si es el modelo correcto. Es decir, se busca el valor de MD que señale el diseño correcto.

Ahora, se simplifica la ecuación 7.5 condicionando sobre σ^2 e integrando sobre σ^2 en el último paso. Pero, primero se debe derivar la distribución predictiva de Y^* para cada modelo. Sean X_i y X_i^* las matrices de regresión para los ensayos iniciales y adicionales, respectivamente, cuando M_i es el modelo. Entonces,

$$Y^*|M_i, Y, \sigma^2 \sim N(\hat{Y}_i^*, \sigma^2 V_i^*)$$

donde $\hat{Y}_i^* = X_i^* \hat{\beta}_i$, $\hat{\beta}_i$ definido como la ecuación 7.1 y

$$V_i^* = I + X_i^* (\Gamma_i + X_i' X_i)^{-1} X_i'^*$$

El cociente de las densidades para dos modelos, M_i y M_j es entonces,

$$\ln \left(\frac{p(Y^*|M_i, Y, \sigma^2)}{p(Y^*|M_j, Y, \sigma^2)} \right) = \frac{1}{2} \ln \left(\frac{|V_j^*|}{|V_i^*|} \right) - \frac{1}{2\sigma^2} ((Y^* - \hat{Y}_j^*)' V_j^{*-1} (Y^* - \hat{Y}_j^*) - (Y^* - \hat{Y}_i^*)' V_i^{*-1} (Y^* - \hat{Y}_i^*))$$

Se integra con respecto a $p(Y^*|M_i, Y, \sigma^2)$ para obtener

$$\frac{1}{2} \ln\left(\frac{|V_j^*|}{|V_i^*|}\right) - \frac{1}{2\sigma^2} (n\sigma^2 - \sigma^2 \text{tr}(V_j^{*-1} V_i^*) - (\hat{Y}_i^* - \hat{Y}_j^*)' V_j^{*-1} (\hat{Y}_i^* - \hat{Y}_j^*))$$

Finalmente, se integra con respecto a $p(\sigma^2|Y, M_i)$ y se obtiene el criterio MD definido

$$MD = \frac{1}{2} \sum_{0 \leq i \neq j \leq m} P(M_i|Y) P(M_j|Y) \times (-n^* + \text{tr}(V_j^{*-1} V_i^*) + (n-1) \times ((\hat{Y}_i^* - \hat{Y}_j^*)' V_j^{*-1} (\hat{Y}_i^* - \hat{Y}_j^*)) / S_i) \quad (7.6)$$

donde S_i está definido como en la ecuación 7.3.

Se busca que el valor MD se maximice para un diseño y cuando esto sucede, el diseño se denomina *MD-óptimo*. En la siguiente sección se abordará el problema de obtener todas las combinaciones posibles de ensayos adicionales para elegir el diseño con mayor MD.

7.4. Algoritmo de intercambio

En esta sección se resume el artículo escrito por [Mitchell \(1974\)](#) donde propone el algoritmo llamado “DETMAX” usado para la construcción de diseños experimentales “D-óptimos”.

Considere el modelo de regresión lineal múltiple $y = X\beta + \epsilon$. Donde el vector de observaciones y es de tamaño $n \times 1$; X es la matriz de $n \times k$ de constantes; β es el vector $k \times 1$ de coeficientes por estimar; y ϵ es el vector de errores de tamaño $n \times 1$ de variables aleatorias independientes e idénticamente distribuidas con una media 0 y varianza desconocida σ^2 . El renglón i -ésimo de X es $f(x_i)'$ donde x_i es el i -ésimo punto de diseño y la función f está determinada por el modelo. Sea p el número

de variables independientes y χ la región donde es factible realizar el experimento.

Se utilizó el método de mínimos cuadrados para calcular los coeficientes β . El estimador de β es $\hat{\beta} = (X'X)^{-1}X'y$, y su matriz de covarianza es $(X'X)^{-1}\sigma^2$. En cualquier punto $x \in \chi$, el valor estimado de la “verdadera” respuesta es $\hat{y}(x) = f(x)'\beta$. Si el modelo es correcto, la esperanza de $\hat{y}(x)$ es la esperanza de la respuesta en el punto x . Es decir, el modelo predice correctamente y . La varianza de $\hat{y}(x)$ está dada por $v(x) = f(x)'(X'X)^{-1}f(x)\sigma^2$. Sin pérdida de generalidad, se puede considerar a σ^2 igual a 1.

Una de las maneras más populares de construir diseños óptimos es el llamado diseño “D-óptimos” donde se busca maximizar $|X'X|$. El propósito del artículo de Mitchell es presentar su versión de diseño D-óptimo llamado “DETMAX”.

7.4.1. Algoritmo básico

En 1970, Mitchell publicó un primer artículo donde introdujo su versión del algoritmo de intercambio. En esa ocasión estableció que se debe empezar con un diseño de n ensayos elegido al azar. Para mejorarlo, se agregan y eliminan puntos de acuerdo a las siguientes consignas:

1. Se suma un ensayo número $n + 1$ elegido para alcanzar el incremento máximo posible de $|X'X|$. Después,
2. Se elimina el ensayo en el diseño resultante que genere la menor disminución en $|X'X|$.

Estos dos pasos se realizan al primero sumar al diseño original el punto donde $v(x)$ sea máximo y después restando del diseño con $n + 1$ ensayos resultante el punto donde $v(x)$ es mínimo.

7.4.2. Incorporación de excursiones

Posteriormente, en su artículo [Mitchell \(1974\)](#) propone una versión generalizada del algoritmo anterior. El requerimiento de que el diseño con $n + 1$ puntos sea regresado inmediatamente a un diseño con n puntos se relajó. Ahora, al algoritmo se le permite hacer una “excursión” donde se pueden construir diseños de varios tamaños que eventualmente regresan a uno de tamaño n .

Si no hay un incremento en el determinante, todos los diseños contruidos en la excursión son eliminados y agregados a un conjunto de diseños fallidos llamado F . El conjunto F es usado después para guiar la siguiente excursión, la cual siempre empieza con el mejor diseño actual de n puntos.

Sea D el diseño actual en cualquier momento durante una excursión. Las reglas para continuar con la excursión son las siguientes:

1. Si el número de puntos en D es mayor que n , se elimina un punto si D no está en F y se agrega un punto de lo contrario.
2. Si el número de puntos en D es menor que n , se agrega un punto si D no está en F y se elimina un punto de otra manera.

Para determinar si algún diseño D está o no en F , se debe examinar el determinante $|X'X|$. A pesar de que esto no es un prueba definitiva (ya que dos diseños diferentes pueden tener el mismo determinante), Mitchell establece que no parece afectar el rendimiento del algoritmo y solo toma una fracción de tiempo en comparación a hacer la prueba exacta de equivalencia.

En cada iteración se vuelve más difícil obtener un mejor diseño ya que las excursiones se pueden alejar mucho de un nivel de n puntos. Para parar el algoritmo, Mitchell propone establecer límites en el mínimo y

máximo número de puntos permitidos en la construcción de un diseño durante una excursión. Su recomendación es establecerlo entre $n \pm 6$.

7.4.3. Puntos candidatos

Mitchell adopta el enfoque de [Dijkstra \(1971\)](#) donde los puntos de diseño son seleccionados de una lista previamente especificada de candidatos. El objetivo de esto es la facilidad de programación y el poder de exclusión de puntos no deseados o posibles.

Se puede presentar el caso donde los diseños sean óptimos localmente por lo tanto, se recomienda ejecutar el algoritmo repetidas veces para encontrar la solución. En cada ocasión, DETMAX empieza con un diseño completamente nuevo cuyos puntos seleccionados aleatoriamente de una lista de candidatos. De esta forma, se asegura que el diseño solución es el óptimo globalmente. Mitchell establece que diez intentos usualmente son suficientes para llegar al diseño óptimo.

7.5. Función MDopt

[Vela \(2022\)](#) utilizó la teoría sobre discriminación de modelos de Meyer así como el algoritmo de intercambio de Michell para crear el paquete BsMD2. El paquete se enfoca en la aplicación de la metodología general establecida por Meyer e incluye la función MDopt. La función MDopt calcula el criterio MD para los ensayos extras sugeridos para discriminar entre posibles modelos. En este ejercicio se utilizó la función creada por Vela para desarrollar una función MDopt análoga en Julia y Python. Esta sección resume el algoritmo utilizado para crear la función, mientras que en el siguiente apartado se pone en práctica con dos ejemplos.

Suponga que el diseño de experimentos ya está definido y ahora busca utilizar la función `MDopt` para calcular el valor MD de una serie de ensayos extras necesarios. El primer argumento de la función `MDopt` es el correspondiente a la matriz \mathbf{X} cuyas columnas representan los factores del experimento y los renglones a los ensayos. El diseño del experimento es factorial fraccionado por lo que los factores tienen dos niveles, en este caso representados como $+$ o $-$. Por lo tanto, la matriz \mathbf{X} está compuesta por 1 y -1 .

El segundo argumento de la función es llamado `max_int` y corresponde a la interacción máxima entre factores. La función utiliza la secuencia condicional `if` para construir la matriz \mathbf{Xfac} . Se comienza definiendo la matriz \mathbf{Xfac} como una copia de la matriz \mathbf{X} . Si `max_int` es mayor a 1 existen interacciones entre dos factores. Por lo tanto, se agregan columnas a \mathbf{Xfac} correspondientes a todas las posibles combinaciones de interacciones entre factores. La interacción entre una dupla de factores se señala con un 1. El proceso se repite para el caso donde `max_int` sea mayor a 2 para añadir las interacciones entre tres factores.

Posteriormente, la función calcula $\Gamma_k, \beta_k, \delta_k$ para cada modelo k usando las fórmulas 7.2, 7.1 y 7.3 respectivamente. Luego, se define otra función (dentro de `MDopt`) llamada `MDr` cuyo objetivo es calcular el valor MD para un conjunto de ensayos. Finalmente, la función `MDopt` implementa el algoritmo de intercambio para calcular el valor MD para distintas combinaciones de ensayos.

Al término de los cálculos, la función `MDopt` de Vela añade formato a distintos objetos cuyo objetivo está fuera del alcance de este trabajo. En cambio, la función `MDopt` desarrollada para este proyecto se limita a dar formato al dataframe que muestra los ensayos a los que se calculó el valor MD. El diagrama 7.2 presenta el desarrollo de la función `MDopt`

de manera visual.

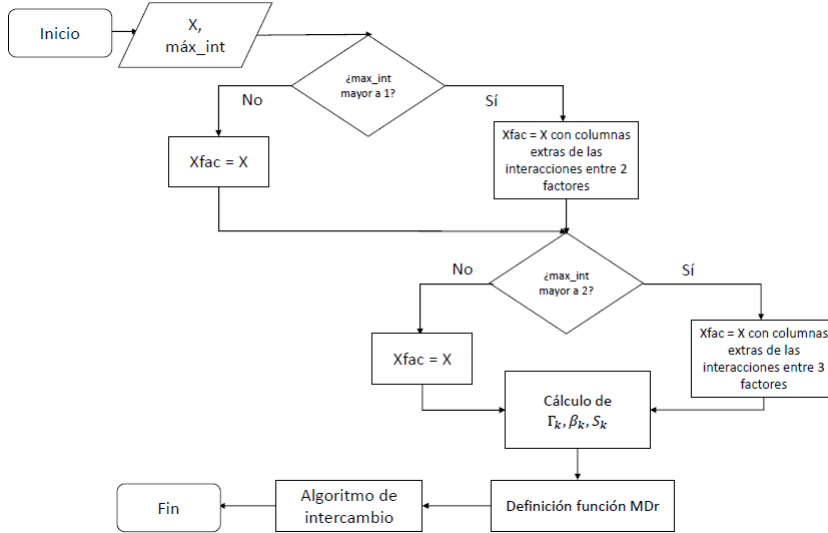


Figura 7.2. Diagrama de la función MDopt

El desarrollo detallado y el pseudocódigo de la función MDr y del algoritmo de intercambio se pueden encontrar en los de Apéndices del trabajo de Vela (2022).

7.6. Implementación de MDopt en los lenguajes

Este proyecto se eligió para ilustrar el poder de cálculo de Julia, R y Python ya que representa un reto de rapidez y complejidad de cálculo. Se utilizó el algoritmo presentado en la sección anterior para desarrollar la función MDopt en Julia y Python. Posteriormente, se utilizaron los paquetes JuliaCall y reticulate de R para crear un vínculo de R

con Julia y Python respectivamente. Así, se exportaron y ejecutaron las funciones `MDopt` a R. Se utilizaron dos ejemplos de experimentos que presenta Meyer et al. (1996) para mostrar y comparar la rapidez de ejecución de las tres funciones. En sus ambiente naturales, Python y Julia son más rápidos, pero para efecto de este trabajo se utilizó R como punto de comparación ya que es uno de los propósitos de esta tesis. En esta sección expone el paralelismo que se encontró al implementar lenguajes externos en R.

Para ejecutar lenguajes externos en R se requieren dos paquetes. El primero es `JuliaCall` y, como su nombre lo indica, se encarga de ejecutar comandos de Julia en R. Asimismo, el paquete `reticulate` cumple el mismo propósito con Python. Ambos paquetes fueron desarrollados con el objetivo de crear un enlace entre su lenguaje y R por lo que se encontraron similitudes entre ellos. La tabla 7.1 presenta los comandos análogos de los paquetes con especificaciones que se encontraron al trabajar con ellos.

JuliaCall	reticulate	Uso	Especificaciones
julia_setup	use_python	Es usado para especificar la dirección del programa (Julia o Python) dentro de la computadora	use_python no es necesario a menos que se tengan varias versiones de Python instaladas.
julia_source	source_python	Añaden a R las funciones que estén dentro de los archivos especificados.	Es necesario tener la terminación del archivo correcta.
julia_assign	r_to_py	Convierten los objetos de R en objetos del programa externo.	JuliaCall no agrega los objetos al ambiente de R, reticulate sí.
julia_eval y julia_command	repl_python	Ejecutan el lenguaje externo dentro de R.	Con repl_python, la consola de R se convierte en una de Python.

Tabla 7.1. Comandos análogos en los paquetes JuliaCall y reticulate

7.6.1. JuliaCall

El paquete JuliaCall permite el funcionamiento de Julia dentro de R. Una de las funciones del paquetes es ejecutar funciones creadas

en Julia usando como argumentos objetos creados en R. Si tuviera que describir el funcionamiento de este paquete sería que es como un puente entre ambos lenguajes. `JuliaCall` solamente conecta los lenguajes, más no los mezcla de ninguna otra forma. En esta sección se expondrá la manera en la que se trabajó `JuliaCall` en R.

Suponga que busca ejecutar, dentro de R, una función llamada `function(x, y)` creada en Julia. Se puede ejecutar de diferentes maneras, pero el siguiente es el procedimiento que se realizó:

1. Cargar el paquete `JuliaCall`.
2. Ejecutar el comando `julia_setup` que toma como argumento la dirección de instalación de Julia en el ordenador.
3. Crear los objetos de entrada de la función, `x`, `y`.
4. Utilizar el comando `julia_assign` para asignar los objetos `x`, `y` creados en R a objetos nuevos en Julia.
5. Verificar que la conversión de objetos haya sido ejecutada de manera correcta. En caso de que no sea así, utilizar el comando `julia_command` para realizar la conversión dentro de Julia.
6. La función `function` debe ser creada y guardada en un archivo de tipo `.jl` en Julia. El archivo solo debe contener la definición de la función `function` y los paquetes necesarios para su ejecución. Se recomienda guardar el archivo en el directorio de trabajo que se esté utilizando en R.
7. Utilizar `julia_source(archivo.jl)` para agregar `function` a R.
8. Ejecutar la instrucción `julia_eval` para correr la función con los argumentos `x`, `y` ya creados.

7.6.2. `reticulate`

El objetivo del paquete `reticulate` es hacer posible la ejecución de Python en R. La documentación describe que “el paquete `reticulate` provee herramientas para la interoperabilidad entre Python y R” [Ushey and Alaire \(2022\)](#). Los usuarios del paquete tienden a usar funciones de ambos lenguajes de manera entrelazada buscando obtener el código más eficiente posible. A diferencia de `JuliaCall`, mi experiencia con `reticulate` es que funciona como una extensión de R ya que permite navegar fácilmente entre ambos lenguajes sin la necesidad de comandos. Más bien, se necesitan los prefijos `.r` y `py$` para distinguir los objetos de cada lenguaje.

Análogo a la sección anterior, suponga que se busca ejecutar, dentro de R, una función llamada `functionPy(x,y)` creada en Python. El procedimiento que se siguió en este trabajo es el siguiente:

1. Cargar el paquete `reticulate`.
2. Crear los objetos de entrada de la función `x`, `y`.
3. La función `functionPy(x,y)` debe ser creada en Python y guardada en un archivo con terminación `.py`. El archivo puede tener varias funciones definidas.
4. Utilizar el comando `source_python(archivoPy.py)` para agregar la función `functionPy(x,y)` al ambiente de R.
5. Convertir los objetos `x`, `y` creados en R a objetos de Python con el comando `r_to_py`. A diferencia de `JuliaCall`, `reticulate` agrega los objetos y funciones de Python al ambiente de R. En caso de utilizar `RStudio`, se puede visualizar la colección de objetos, variables y funciones en la sección de `Environment`.

6. Se recomienda verificar que la conversión de objetos haya sido correcta. En ocasiones los objetos tipo `Int` en R se convierten a `Float` y la función `functionPy()` puede marcar un error.
7. Ejecutar la función `functionPY()` de manera usual. Es decir, no se necesita ningún comando adicional del paquete `reticulate`.

La siguiente sección muestra dos ejemplos utilizados para ejecutar las funciones `MDopt` de los paquetes `BsMD` y `BsMD2`, así como las creadas en Julia y Python.

7.7. Ejemplos y resultados

7.7.1. Ejemplo 1 - Proceso de moldeo por inyección

El primer ejemplo que se utilizó es tomado del artículo de [Meyer et al. \(1996\)](#), quien a su vez, lo tomó de otro artículo publicado en 1978 por Box, Hunter y Hunter. El ejemplo describe un experimento que busca estudiar los efectos de 8 factores en un proceso de moldeo por inyección. Los factores se representan con las letras mayúsculas A, B, C, D, E, F, G, H . El diseño experimental es un 2^{8-4} factorial fraccionado con generadores $I = ABDH = ACEH = BCFH = ABCG$. Este tipo de diseño está fuera de los alcances de este trabajo, pero se puede encontrar una explicación detallada en el capítulo 8.4 de [Montgomery \(2017\)](#). Los datos para el primer ejemplo se presentan en la tabla 7.2.

Ensayo	A	B	C	D	E	F	G	H	Y
1	-1	-1	-1	1	1	1	-1	1	14.0
2	1	-1	-1	-1	-1	1	1	1	16.8
3	-1	1	-1	-1	1	-1	1	1	15.0
4	1	1	-1	1	-1	-1	-1	1	15.4
5	-1	-1	1	1	-1	-1	1	1	27.6
6	1	-1	1	-1	1	-1	-1	1	24.0
7	-1	1	1	-1	-1	1	-1	1	27.4
8	1	1	1	1	1	1	1	1	22.6
9	1	1	1	-1	-1	-1	1	-1	22.3
10	-1	1	1	1	1	-1	-1	-1	17.1
11	1	-1	1	1	-1	1	-1	-1	21.5
12	-1	-1	1	-1	1	1	1	-1	17.5
13	1	1	-1	-1	1	1	-1	-1	15.9
14	-1	1	-1	1	-1	1	1	-1	21.9
15	1	-1	-1	1	1	-1	1	-1	16.7
16	-1	-1	-1	-1	-1	-1	-1	-1	20.3

Tabla 7.2. Datos para el ejemplo 1 - proceso de moldeo por inyección

Como se mencionó en la sección 7.2, uno de los pasos del análisis de este tipo de diseños es calcular $P(M_i|Y)$, la probabilidad posterior de cada modelo M_i . Los resultados se obtuvieron del artículo de Meyer, pero se pueden calcular con el paquete **BsMD2**. En la tabla 7.3 se observan 5 modelos diferentes que tienen la probabilidad más alta de incluir los factores activos del experimento.

Modelo	Factores	Probabiliad posterior
1	A,C,E	0.2356
2	A,C,H	0.2356
3	A,E,H	0.2356
4	C,E,H	0.2356
5	A,C,E,H	0.0566

Tabla 7.3. Modelos con la probabilidad posterior más alta para el ejemplo 1

Asimismo, calculando las probabilidades posteriores P_j definidas en 7.4 los posibles factores activos son A, C, E , y H . Estos factores presentan una probabilidad $P_j = 0.764$ mientras que la probabilidad del resto de factores es 0. Por lo tanto, de los 8 factores iniciales que, se creía, afectaban la variable respuesta Y se pueden eliminar los cuatro cuya $P_j = 0$. De esta manera, el problema original con un diseño 2^{8-4} se convierte en un diseño 2^{4-1} .

Los ensayos realizados no proporcionan suficiente información para distinguir entre los 5 posibles modelos, por lo que se necesitan ensayos adicionales. En la tabla 7.4 se muestra diferentes combinaciones de los niveles de los factores A, C, E, H y la predicción de su efecto en la variable respuesta Y .

Punto candidato						Predicciones del modelo				
	A	C	E	H	Y	1	2	3	4	5
1	-1	-1	-1	-1	21.9, 20.3	21.08	21.08	21.08	21.08	21.09
2	-1	-1	1	1	14.0, 15.0	14.58	14.58	14.58	14.58	14.54
3	-1	1	-1	1	27.6, 27.4	27.38	27.38	27.38	27.38	27.44
4	-1	1	1	-1	17.1, 17.5	17.34	17.34	17.34	17.34	17.32
5	1	-1	-1	1	16.8, 15.4	16.16	16.16	16.16	16.16	16.13
6	1	-1	1	-1	15.9, 16.7	16.35	16.35	16.35	16.35	16.33
7	1	1	-1	-1	22.3, 21.5	21.87	21.87	21.87	21.87	21.88
8	1	1	1	1	24.0, 22.6	23.25	23.25	23.25	23.25	23.27
9	-1	-1	-1	1		21.08	14.58	27.38	16.16	19.75
10	-1	-1	1	-1		14.58	21.08	17.34	16.35	19.75
11	-1	1	-1	-1		27.38	17.34	21.08	21.87	19.75
12	-1	1	1	1		17.34	27.38	14.58	23.25	19.75
13	1	-1	-1	-1		16.16	16.35	21.87	21.08	19.75
14	1	-1	1	1		16.35	16.16	23.25	14.58	19.75
15	1	1	-1	1		21.87	23.25	16.16	27.38	19.75
16	1	1	1	-1		23.25	21.87	16.35	17.34	19.75

Tabla 7.4. Ejemplo 1, Colapsado en los factores A, C, E y H

Los primeros 8 renglones de la tabla 7.4 muestran los 16 ensayos realizados en el experimento mostrados en la tabla 7.2. Cada renglón muestra una dupla de ensayos donde los factores A, C, E y H tuvieron los mismos niveles en el experimento original. Por lo tanto, se muestran dos valores en la respuesta Y .

El número máximo de factores activos en los cinco posibles modelos es cuatro y cada factor cuenta con dos niveles ($-$ o $+$). Las columnas A, C, E y H muestran las $2^4 = 16$ posibles combinaciones de factores activos. Las columnas correspondientes a las predicciones del modelo muestra la predicción de la variable de respuesta bajo cada uno de los

modelos presentados en la tabla 7.3. Por ejemplo, tome el modelo 1 que tiene como factores activos a A, C, E . En este modelo, la columna H no es necesaria ya que el factor H no está activo en el modelo 1. Tome el primer renglón, donde los tres factores se colocan en el nivel -1. La predicción de la variable Y bajo el modelo 1 es 21.08. El valor se repite en la predicción del modelo 1 para el punto candidato 9 ya que los factores A, C, E están en el nivel -1. Esta repetición ocurre para los renglones donde el nivel de los factores activos es el mismo.

La tabla 7.4 muestra los puntos de diseño candidatos a ser los ensayos extra del experimento. En este ejemplo se busca generar un diseño con cuatro ensayos extras cuya elección dependerá de su valor MD. Hay 3,876 posibles diseños que se pueden generar de los 16 puntos candidatos mostrados en la tabla 7.4. Un posible acercamiento es calcular el valor MD para cada uno de esos diseños. Sin embargo, una forma más eficiente es utilizar el algoritmo de intercambio presentado en la sección 7.4. El algoritmo genera un conjunto de puntos candidatos aleatorio, calcula el valor MD y agrega y elimina puntos para crear un conjunto de ensayos con el máximo valor MD posible. Lo anterior se repite hasta satisfacer los criterios de convergencia.

El código para el cálculo del criterio MD y el algoritmo de intercambio para R, Julia y Python se muestra a continuación.

```
# Para guardar los tiempos
tiempos_df<-data.frame(matrix(nrow = 5, ncol = 4))
colnames(tiempos_df) <- c("BsMD2" , "BsMD" ,
      " JuliaCall" , " reticulate")
row.names(tiempos_df) <- seq(1, 5)

runs <- seq(1, 5)
```



```

# # # R tesis Paty
library(BsMD2)
setwd("~/ITAM/Tesis/Julia_con_R/Code/MD-optimality")

#matriz de diseno inicial
X <- as.matrix(BM93e3[1:16, c(1,2,4,6,9)])
#vector de respuesta
y <- as.vector(BM93e3[1:16,10])
#probabilidad posterior de los 5 modelos
p_mod <- c(0.2356,0.2356,0.2356,0.2356,0.0566)

fac_mod <- matrix(c(2,1,1,1,1,3,3,2,2,2,4,4,3,4,
                    3,0,0,0,0,4), nrow=5,
                  dimnames=list(1:5,
                                c("f1", "f2", "f3", "f4")))

Xcand <- matrix(c(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                  -1,-1,-1,-1,1,1,1,1,-1,-1,-1,-1,1,1,1,1,
                  -1,-1,1,1,-1,-1,1,1,-1,-1,1,1,-1,-1,1,1,
                  -1,1,-1,1,-1,1,-1,1,-1,1,-1,1,-1,1,-1,1,
                  -1,1,1,-1,1,-1,-1,1,1,-1,-1,1,-1,1,1,-1),
                nrow=16, dimnames=list(1:16,
                                        c("blk", "f1", "f2", "f3", "f4")))
)

times <- c()
for (i in runs){
  t <- Sys.time()
  e3_R <- BsMD2::MDopt(X = X, y = y,

```

```

Xcand = Xcand, nMod = 5, p_mod = p_mod,
fac_mod = fac_mod, nStart = 25)
t_2 <- Sys.time()
t_final <- difftime(t_2, t, unit = "secs")

times <- c(times, t_final)
}
tiempos_df["BsMD2"] <- times

### R paquete original
library(BsMD)

s2 <- c(0.5815, 0.5815, 0.5815, 0.5815, 0.4412)
times <- c()
for (i in runs){
  t_RO <- Sys.time()
  e3_RO <- BsMD::MD(X = X, y = y, nFac = 4,
nBlk = 1, mInt = 3, g = 2, nMod = 5,
p = p_mod, s2 = s2, nf = c(3, 3, 3, 3, 4),
fac = fac_mod, nFDes = 4, Xcand = Xcand,
mIter = 20, nStart = 25, top = 10)
  t_2 <- Sys.time()
  t_final <- difftime(t_2, t_RO,
unit = "secs")
  times <- c(times, t_final)
}
tiempos_df["BsMD"] <- times

### Julia con R

```

```

library( JuliaCall)
julia_setup(JULIA_HOME =
"C:/Users/Valeria/AppData/Local/Programs/Julia -1.6.3/bin")

julia_source("MDopt.jl")
      # Conversiones para los tipos de Julia
X_J <- as.data.frame(X)
julia_assign("X_J", X_J)
julia_assign("y_J", y)
julia_assign("p_mod_J", p_mod)
julia_assign("fac_mod_J", fac_mod)
julia_command(" fac_mod_J=NamedArray( fac_mod_J)")
julia_eval(" fac_mod_J=Int64.( fac_mod_J)")
julia_assign("Xcand_J", Xcand)
julia_command("Xcand_J=NamedArray(Xcand_J)")
julia_eval("Xcand_J=Int64.(Xcand_J)")

times <- c()
for (i in runs){
      t <- Sys.time()
      julia_eval("MDopt(X=X_J,y=y_J,Xcand=Xcand_J,
nMod=5,p_mod=p_mod_J,fac_mod=fac_mod_J,
nFDes=4,max_int=3,g=2,Iter=20,
nStart=10,top=10)")
      t_2 <- Sys.time()
      t_final <- difftime(t_2, t, unit = "secs")

      times <- c(times, t_final)
}

```

```

tiempos_df["JuliaCall"] <- times

      # Python con R
library(reticulate)
source_python("MD_Python.py")

X_P <- as.data.frame(X)
Xcand_P <- as.data.frame(Xcand)
fac_mod_P <- as.data.frame(fac_mod)

X_P <- r_to_py(X_P)
y_P <- r_to_py(y)
Xcand_P <- r_to_py(Xcand_P)
p_mod_P <- r_to_py(p_mod)
fac_mod_P <- r_to_py(fac_mod_P)

nMod_P <- r_to_py(5L)
nFDes_P <- r_to_py(4L)
max_int_P <- r_to_py(3L)
g_P <- r_to_py(2L)
Iter_P <- r_to_py(20L)
nStart_P <- r_to_py(25L)
top_P <- r_to_py(10L)

times <- c()
for (i in runs){
  t <- Sys.time()
  MD_Python(X = X_P, y = y_P, Xcand = Xcand_P,
    nMod = nMod_P, p_mod = p_mod_P,

```

```

    fac_mod = fac_mod_P, nFDes = nFDes_P,
    max_int = max_int_P, g = g_P, Iter = Iter_P,
    nStart = nStart_P, top = top_P)
    t_2 <- Sys.time()
    t_final <- difftime(t_2, t, unit = "secs")

    times <- c(times, t_final)
}
tiempos_df["reticulate"] <- times

write.csv(tiempos_df, "tiempos_MD_ej3.csv")

```

Los resultados en los tres lenguajes fueron los mismos y se muestran en la tabla 7.5.

Diseño	Puntos de diseño	MD
1	9, 9, 12, 15	85.67
2	9, 11, 12, 15	83.63
3	9, 11, 12, 12	82.18
4	9, 12, 15, 16	77.05
5	9, 12, 13, 15	76.74
6	9, 10, 11, 12	76.23
7	2, 9, 12, 15	71.23
8	5, 9, 12, 15	70.75
9	2, 9, 12, 12	67.69
10	9, 10, 12, 16	66.58

Tabla 7.5. Resultados para el ejemplo 1

En este ejemplo, el paquete creado por Vela calculó la solución en un promedio de 4.06 segundos mientras que su antecesor, BsMD realizó

el cálculo en 0.03 segundos. A la función creada en `Julia` le tomó 3.07 segundos ejecutar el código. Sin embargo, se debe considerar que el tiempo que le toma al comando `julia.setup` es largo, cerca de 5 minutos. A pesar de que este comando solo se debe realizar una vez por cada sesión de `R` utilizada, el paquete `reticulate` carga casi al instante. No obstante, la función de `Python` realiza el cálculo en 15.32 segundos.

Una característica singular que presenta `Julia`, es que al ser un lenguaje *just in time*, el tiempo de ejecución va disminuye en gran medida después de la primera vez, incluso si se modifican los argumentos de la función `MDopt`. Esto ocurre independientemente del programa donde se utilice (en este caso, `Jupyter Notebook` o `R`).

7.7.2. Ejemplo 2

El segundo experimento presentado por [Meyer et al. \(1996\)](#) proviene de un diseño factorial original con 5 factores, del cual se obtuvieron 2^5 ensayos. Se extrajeron solamente $2^3 = 8$ ensayos como experimento original. Después, se obtuvo el diseño adicional dictado por el criterio MD y se utilizaron los datos del experimento completo para evaluar la eficiencia de este diseño adicional. Los ocho ensayos elegidos se presentan en la tabla [7.6](#).

Ensayo	A	B	C	D	E	Y
1	-1	-1	-1	1	1	44
2	1	-1	-1	-1	-1	53
3	-1	1	-1	-1	1	70
4	1	1	-1	1	-1	93
5	-1	-1	1	1	-1	66
6	1	-1	1	-1	1	55
7	-1	1	1	-1	-1	54
8	1	1	1	1	1	82

Tabla 7.6. Datos para el ejemplo 2

El análisis bayesiano previo de este ejemplo no muestra una distinción clara de factores activos y factores no activos. Se decidió obtener un diseño adicional de 4 ensayos para encontrar el mejor subconjunto de los 32 ensayos candidatos del diseño original. El código para generar los resultados en los tres lenguajes es el siguiente.

```
# Para guardar los tiempos
tiempos_df<-data.frame(matrix(nrow = 5, ncol = 4))
colnames(tiempos_df) <- c("BsMD2", "BsMD",
    "JuliaCall", "reticulate")
row.names(tiempos_df) <- seq(1, 5)

runs <- seq(1, 5)

### R tesis Paty
library(BsMD2)
setwd("~/ITAM/Tesis/Julia_con_R/Code/MD-optimality")
data(M96e2)
print(M96e2)
```

```

X <- as.matrix(cbind(blk = rep(-1, 8),
                      M96e2[c(25,2,19,12,13,22,7,32), 1:5]))
y <- M96e2[c(25,2,19,12,13,22,7,32), 6]

pp <- BsProb1(X = X[, 2:6], y = y, p = .25,
gamma = .4, max_int = 3, max_fac = 5, top = 32)

p <- pp@p_mod
facs <- pp@fac_mod
Xcand <- as.matrix(cbind(blk = rep(+1, 32),
M96e2[, 1:5]))

times <- c()
for (i in runs){
  t <- Sys.time()
  e4_R <- BsMD2::MDopt(X = X, y = y,
Xcand = Xcand, nMod = 32, p_mod = p,
fac_mod = facs, g = 0.4, Iter = 10,
nStart = 25, top = 5)
  t_2 <- Sys.time()
  t_final <- difftime(t_2, t, unit = "secs")

  times <- c(times, t_final)
}
tiempos_df["BsMD2"] <- times

### R package original
library(BsMD)

```



```

reactor8.BsProb <- BsProb(X = X, y = y, blk = 1,
mFac = 5, mInt = 3, p = 0.25, g = 0.40, ng = 1,
nMod = 32)

nf <- reactor8.BsProb$nf
s2 <- reactor8.BsProb$sigtop

times <- c()
for (i in runs){
  t_RO <- Sys.time()
  ej4_RO <- BsMD::MD(X = X, y = y, nFac = 5,
nBlk = 1, mInt = 3, g = 0.40, nMod = 32,
p = p, s2 = s2, nf = nf, facs = facs,
nFDes = 4, Xcand = Xcand, mIter = 20,
nStart = 25, top = 5)
  t_2 <- Sys.time()
  t_final <- difftime(t_2, t_RO,
unit = "secs")
  times <- c(times, t_final)
}
tiempos_df["BsMD"] <- times

library( JuliaCall)
julia_setup(JULIA_HOME = ~/Programs/Julia-1.6.3/bin")
#####
julia_source("MDopt.jl")
#####
X<-as.matrix(cbind(blk=rep(-1,8),
M96e2[c(25,2,19,12,13,22,7,32),1:5]))

```

```

y<-M96e2[c(25,2,19,12,13,22,7,32),_6]
#####
pp<-BsProb1(X=X[,_2:6],_y=_y,_p=_p*_25,
#####gamma=_4,_max_int=_3,_max_fac=_5,
#####top=_32)
#####
p<-pp@p_mod
facs<-pp@fac_mod
Xcand<-as.matrix(cbind(blk=_rep(+1,_32),
#####M96e2[,_1:5]))
#####
######_Conversiones_para_los_tipos_de_Julia
X<-as.data.frame(X)
julia_assign("X",_X)
julia_assign("y",_y)
julia_assign("p_mod",_p)
julia_assign("fac_mod",_facs)
julia_command("fac_mod=NamedArray(fac_mod)")
julia_eval("fac_mod=Int64.(fac_mod)")
julia_assign("Xcand",_Xcand)
julia_command("Xcand=NamedArray(Xcand)")
julia_eval("Xcand=Int64.(Xcand)")
#####
times<-c()
for_(i_in_runs){
#####t<-Sys.time()
#####julia_eval("MDopt(X=X,y=y,
Xcand=Xcand,nMod=32,p_mod=p_mod,
fac_mod=fac_mod,nFDes=4,max_int=3,

```

```

g=0.4,Iter=10,nStart=25,top=5)")
#####t_2<-Sys.time()
#####t_final<-difftime(t_2,t,unit="secs")
#####
#####times<-c(times,t_final)
}
tiempos_df["JuliaCall"]<-times
#####
########Python con R
library(reticulate)
source_python("MD_Python.py")
#####
X_P<-as.data.frame(X)
Xcand_P<-as.data.frame(Xcand)
fac_mod_P<-as.data.frame(facs)
#####
X_P<-r_to_py(X_P)
y_P<-r_to_py(y)
Xcand_P<-r_to_py(Xcand_P)
p_mod_P<-r_to_py(p)
fac_mod_P<-r_to_py(fac_mod_P)
nMod_P<-r_to_py(32L)
nFDes_P<-r_to_py(4L)
max_int_P<-r_to_py(3L)
g_P<-r_to_py(0.4)
Iter_P<-r_to_py(10L)
nStart_P<-r_to_py(25L)
top_P<-r_to_py(5L)
#####

```

```

times<-c()
for (i in runs){
  ##### t<-Sys.time()
  ##### MD_Python(X=X_P, y=y_P,
  ##### Xcand=Xcand_P, nMod=nMod_P,
  ##### p_mod=p_mod_P, fac_mod=fac_mod_P,
  ##### nFDes=nFDes_P, max_int=max_int_P,
  ##### g=g_P, Iter=Iter_P, nStart=nStart_P,
  ##### top=top_P)
  ##### t_2<-Sys.time()
  ##### t_final<-difftime(t_2,t, unit="secs")

  ##### times<-c(times, t_final)
}
tiempos_df["reticulate"]<-times
#####
##### write.csv(tiempos_df, "tiempos_MD_ej4.csv")
#####
#####
#####
#####

```

En los tres lenguajes los resultados fueron los mismos y se muestran en la tabla 7.7.

Diseño	Puntos de diseño	MD
1	4, 10, 11, 26	0.64
2	4, 10, 11, 28	0.63
3	4, 10, 12, 27	0.63
4	4, 10, 26, 27	0.63
5	4, 12, 26, 27	0.62

Tabla 7.7. Resultados para el ejemplo 2

Este ejemplo es el más pesado e intensivo computacionalmente que se mostrará en este documento. Los tiempos de ejecución lo muestran. El paquete **BsMD2** realizó el cálculo en 256.27 segundos; el paquete **BsMD** hizo el cálculo en 0.41 segundos; Julia se tardó 23.88 segundos; y, finalmente a Python le tomó 547.90 segundos.

El paquete **BsMD** sale de R para hacer los cálculos en **Fortran**, un lenguaje de programación de alto nivel adaptado al cálculo numérico y a la computación científica. La rapidez del paquete no está en la capacidad computacional de R sino en la de **Fortran**. Por tanto, no es una sorpresa que **BsMD** ejecute la función **MDopt** en menos de 1 segundo.

En segundo lugar está **Julia**, que muestra su capacidad computacional al realizar los cálculos en menos de 1 minuto. En este ejercicio, **Julia** muestra ser al menos 10 veces más rápido que **Python** y el paquete **BsMD2** en R. Asimismo, se ejecutaron las funciones **MDopt** en **Julia** y **Python** usando **Jupyter Notebook** para observar si existían cambios en la rapidez de los cálculos. En el caso de **Julia** la función se ejecutó en un tiempo muy similar al mostrado con **JuliaCall**. Sin embargo, utilizar la función **MDopt** en **Python** (usando **Jupyter Notebook**) es mucho más rápido que en R usando el paquete **reticulate**. En **Jupyter Notebook**, el cálculo de resultados se hizo en 2.87 minutos. A pesar de la mejora en el tiempo de ejecución, **Julia** permanece como el segundo lenguaje más rápido para ejecutar la función **MDopt**.

7.8. Conclusiones

El objetivo de este proyecto fue mostrar la capacidad computacional de los tres lenguajes en cálculos intensivos. Para esto, se presentó el

problema de determinar los mejores ensayos adicionales para distinguir entre factores activos y no activos en un experimento. [Meyer et al. \(1996\)](#) ofrece como solución el uso del criterio MD y el algoritmo de intercambio para discriminar entre los posibles modelos que describan el fenómeno del experimento.

Se tomó como referencia la función `MDopt` desarrollada por [Vela \(2022\)](#) para crear funciones con el mismo nombre y objetivo en `Python` y `Julia`. Después, se ejecutaron las funciones en `R` usando los paquetes `reticulate` y `JuliaCall` respectivamente. Asimismo, se utilizó el paquete `BsMD` desarrollado por Ernesto Barrios que realiza los cálculos en `Fortran`, pero presenta los resultados en `R`.

Se midieron los tiempos de ejecución para determinar el lenguaje que realiza los cálculos más rápidamente. El paquete más rápido fue `BsMD` seguido por `Julia`. El paquete `BsMD2` toma el tercer lugar mientras que la función desarrollada en `Python` fue la más tardía. Los resultados en los tiempos de ejecución muestran la complejidad del algoritmo y la capacidad de cada uno de los lenguajes.

Este proyecto realmente muestra el potencial de rendimiento que tiene `Julia`. Una de las razones de esa rapidez es que `Julia` pide definir el tipo de objeto con el que se va a trabajar. Esto hace que las funciones asignadas a los objetos funcionen de la manera más eficiente posible. Sin embargo, la definición e inicialización de objetos en `Julia` fue lo más complicado de este proyecto. Tuve que buscar como definir un vector de dataframes, un vector de vectores y un vector de matrices que guardaran los valores que iba necesitando en la función `MDopt`. En cambio, en `Python` todos los objetos se definieron como listas sin ninguna otra especificación.

Otra característica que presenta `Julia` es la necesidad de utilizar un paquete especial llamado `NamedArrays` para nombrar los renglones y

las columnas de los arreglos. Usar este paquete fue la única forma que encontré de referirme a un renglón o columna específica. Esta sería una mejora que propondría para los paquetes `LinearAlgebra` y `StatBase`.

Capítulo 8

Conclusiones

Uno de los objetivos de este trabajo de tesis es mostrar algunas de las capacidades de tres lenguajes de cómputo estadístico: Julia, R y Python. El documento se divide en dos partes. En la primera parte, se hace una breve descripción de cada lenguaje utilizado sin pretender ser un manual. En el capítulo 2 de Julia, por ser el lenguaje más novedoso, se mencionan los pasos iniciales desde la instalación hasta la definición de objetos básicos. En los capítulos 3 y 4, de Python y R respectivamente, se asume que el lector está más familiarizado con estos lenguajes por lo que se presentan directamente los paquetes utilizados en este trabajo.

En la segunda parte del trabajo se presentan tres distintos proyectos para mostrar los desempeños de los lenguajes. El primer proyecto es un problema retador numéricamente. Se trabajó con un problema propuesto por NIST donde se busca hacer el ajuste de un polinomio de grado 10 con 82 observaciones. De manera intencional, el sistema lineal asociado está mal condicionado lo que genera dificultades al calcular la solución. Así, el objetivo de este ejercicio es medir la precisión numérica presentada por los lenguajes.

En **Julia** se realizaron cuatro intentos de solución al problema donde dos de ellos utilizan teoría algebraica mientras los restantes emplean funciones de paquetes ya programados. De las cuatro manera de solución, solamente una obtuvo los resultados correctos. En **R** se utilizó la solución sugerida por Brian Ripley donde se emplea la función `lm` con un cambio en el argumento de tolerancia. Por último, en **Python** se utilizó la función `polyfit` del paquete `NumPy` con la cual el cálculo de la solución fue preciso sin necesidad de ningún ajuste a la función.

El segundo proyecto consiste en el ajuste de modelos de regresión lineal múltiple a una base de datos extensa. Los datos utilizados se tomaron del Censo 2020 realizado por el INEGI. Para propósito de desempeño, y no de análisis, se crearon 5 modelos que buscaban relacionar los ingresos de una persona con variables que pudieran tener un efecto en la suma remunerada. Se comenzó con un modelo con 5 regresores y se fue agregando 1 regresor en la creación del modelo nuevo. Además, se tomaron 5 volúmenes de datos diferentes, comenzando con una muestra de 500 observaciones y aumentando la cantidad hasta llegar a 2.5 millones de datos. En total, para este proyecto se construyeron 25 ajustes diferentes.

El objetivo de este ejemplo fue observar el manejo de grandes cantidades de datos en cada uno de los lenguajes. **Julia** presenta la misma rapidez que **Python** en el cálculo del ajuste. Sin embargo, la rapidez de **Python** al generar archivos de resultados sobrepasa de gran manera a **Julia**. Por otro lado, **R** presenta un manejo de datos lento y entorpecido por el constante reinicio de sesión. Sin embargo, los tres lenguajes presentaron el cálculo de los resultados de manera correcta y precisa.

El tercer ejemplo pertenece a la rama del diseño de experimentos.

Se tomó el artículo publicado por Meyer et al. (1996) donde se presentan los pasos para hacer un estudio de los resultados de un experimento. La metodología comienza con un análisis bayesiano que busca elegir los factores que afectan la variable de respuesta. La particularidad crucial de este ejemplo es que no hay distinción clara en los factores activos y no activos del experimento. Por lo tanto, Meyer propone el criterio MD, de *Model Discrimination*. El criterio busca seleccionar algunas ensayos extras necesarios para determinar que factores afectan a la variable de respuesta y poder así, elegir el mejor modelo que describa al experimento.

El proyecto presenta una alta complejidad e intensidad en cálculos. En R, se utilizó el paquete BsMD2 creado por Patricia Vela que aborda el problema con la función MDopt que hace el cálculo del criterio MD. Esta función se programó en Julia y Python para posteriormente, unir los tres lenguajes en R. Es decir, se programó la función MDopt en Julia y en Python. Después, se utilizaron los paquetes JuliaCall y reticulate para importar dichas funciones en R. De esta forma, se obtuvo una comparación en los tiempos de ejecución donde Julia muestra toda su capacidad computacional y logra ejecutar el algoritmo más rápido que los otros lenguajes.

En mi opinión, Julia no busca ser la copia de R o Python en análisis y manejo de datos. Julia está hecho para ser un lenguaje que realiza simulaciones y cálculos complejos con un alto rendimiento. Julia tiene mucho que ofrecer incluyendo un análisis de datos fácil e intuitivo. Sin embargo, su uso va más allá y sus aplicaciones incluyen modelos de estadística bayesiana, análisis epistemológicos y análisis de sistemas dinámicos.

Una extensión de esta tesis podría ser la estandarización de las pruebas realizadas para su ejecución en otros lenguajes. De esta

manera, se podría obtener una comparación con más lenguajes y determinar que lenguaje es el más rápido. Otro posible camino de mejora es probar los proyectos presentados en este trabajo en diferentes arquitecturas. Es decir, cambiar de procesadores, marca del ordenador en el que se trabaja, e incluso de sistema operativo. Sería interesante observar el cambio en los tiempos de ejecución y en la aplicación de las pruebas. Finalmente, se podría ampliar este trabajo eligiendo y utilizando distintos paquetes que también se enfoquen en modelos de regresión y diseño de experimentos.

Apéndice A

Código

A.1. Código del modelo de regresión lineal que utiliza el Censo

A.1.1. R

Debido a la similitud de los *sub-ajustes*, solo se presenta el código correspondiente a los modelos fit5 y fit10. El código completo se presenta en el repositorio de GitHub https://github.com/valperez/Tesis_Julia/tree/main/Tesis%20Julia%20con%20R/Code/Censo.

```
### Inicio de codigo donde se calculan los  
### resultados en R y se comparan con Julia  
# Paquetes  
library(dplyr)  
library(stringr)  
library(knitr)  
library(tidyverse)
```

```

# Lectura de dos tipos de archivos: la informacion para los
# ajustes y los resultados de lo ajustes en Julia
fnames <- list.files(pattern = "*.csv")
temp <- lapply(fnames, read_csv)

# Edicion de nombres
n = length(fnames)
for (i in 1:n){
    fnames[i] <- str_replace(fnames[i],
    ".csv", "")
}

# Asignacion de archivos a objetos
n = length(temp)
for (i in 1:n){
    assign(fnames[i], temp[[i]])
}

# Edicion de dataframes de resultados
for (i in (n/2 + 1):n){
    aux <- get(fnames[i])
    aux <- aux %>%
    select(Name, `Coef.`) %>%
    rename(Julia = `Coef.`)
    assign(fnames[i], aux)
}

# Desarrollo de funcion que hace el ajuste en R y
# une dichos resultados con los obtenidos en Julia

```

```

union_resultados <- function(vec_strings , fm, runs){
  for (i in 1:(k/2)){
    aux <- get(vec_strings[i])
    tiempos <- c()
    for (i in 1:runs){
      t1 <- Sys.time()
      ajuste <- lm(fm, aux)
      t2 <- Sys.time()
      dif <- difftime(t2, t1, units = "secs")
      tiempos <- c(tiempos, dif)
    }

    tiempos_df <- as.data.frame(tiempos)
    coeficientes <- ajuste$coefficients
    remove(aux)
    #Para vaciar un poco de la memoria
    res_aux <- get(vec_strings[(k/2) + i])
    res_aux <- res_aux %>%
    mutate(R = coeficientes)

    resultados <- list("coef" = res_aux,
      "tiempo" = tiempos_df)
    return (resultados)
  }
}

n_runs <- 5
n_fits <- 5

```

```

# Fit 5 (base)
fm_5 <- formula(INGTRMEN ~ HORTRA + as.factor(SEXO) + EDAD + as
as.factor(ENT_PAIS_TRAB))
fit5_strings <- fnames[str_detect(fnames, "fit5")]

k <- length(fit5_strings)

fit5 <- data.frame(matrix(NA, nrow = n_runs,
ncol = n_fits))

for (i in (k/2 +1):k){
  results <- union_resultados(fit5_strings, fm_5,
runs = n_runs)
  assign(fit5_strings[i], results$coef)
  fit5[, i - k/2] <- results$tiempo
}

fit5_strings <- str_remove(fit5_strings, "_fit5")
nombres <- fit5_strings[1:(k/2)]
colnames(fit5) <- nombres

# Fit 10
fm_10 = formula(INGTRMEN ~ as.factor(SEXO) +
EDAD + as.factor(NIVACAD) + HORTRA +
as.factor(ENT_PAIS_TRAB) + as.factor(SITTRA) +
as.factor(ALFABET) + as.factor(AGUINALDO) +
as.factor(VACACIONES) + as.factor(SERVICIO_MEDICO))

fit10_strings <- fnames[str_detect(fnames, "fit10")]

```

```

k <- length(fit10_strings)

fit10 <- data.frame(matrix(NA, nrow = n_runs,
                           ncol = n_fits))

for (i in (k/2 + 1):k){
  results <- union_resultados(fit10_strings,
                              fm_10, runs = n_runs)
  assign(fit10_strings[i], results$coef)
  fit10[, i - k/2] <- results$tiempo
}

fit10_strings <- str_remove(fit10_strings, "_fit10")
nombres <- fit10_strings[1:(k/2)]
colnames(fit10) <- nombres

```

A.1.2. Python

```

### Inicio de codigo donde se calculan los
### resultados del ajuste en Python
import pandas as pd
import numpy as np
import os
import time
import random
from sklearn.linear_model import LinearRegression
from sklearn import linear_model
from os.path import isfile, join

```



```

import time

    # Lectura archivos
allfiles = [f for f in os.listdir('~\\Censo') if
             isfile(join('~\\Censo', f))]
letras = ['2', '5']
filt_files = [idx for idx in allfiles if idx[0] == letras[0]
              or idx[0] == letras[1]]

runs = range(1, 6)

### FIT BASE ###
def fit5(nombre_archivo):
    fit_data = pd.read_csv(nombre_archivo)
    # Vector con todas las categorias
    vector_categorias = ["SEXO", "NIVACAD",
                          "ENT_PAIS TRAB"]

    # Y las transformamos
    for columna in vector_categorias:
        fit_data[columna] =
            fit_data[columna].astype('category')

    x = fit_data[['HORTRA', 'SEXO', 'EDAD',
                  'NIVACAD', 'ENT_PAIS TRAB']]
    y = fit_data[['INGTRMEN']]

    # Los hacemos dummies
    x = pd.get_dummies(data = x, drop_first = True)

```

```

# Hacemos la regresion
    regr = linear_model.LinearRegression()

    tiempos_list = []
    for i in runs:
        start = time.time()
# Ojo que regresa el coeficiente para la potencia mayor primera
        model = regr.fit(x, y)
# Medimos el tiempo
        tiempo = time.time() - start
        tiempos_list.append(tiempo)

# Guardamos los intercepts
        intercepts = model.intercept_
# Guardamos los coeficientes de los resultados
        coeficientes = model.coef_
        coeficientes = np.transpose(coeficientes)

        intercepts = pd.DataFrame(intercepts)
        coeficientes = pd.DataFrame(coeficientes)
# Los unimos
        resultados = np.vstack([intercepts, coeficientes])
# Los guardamos
        nombre = 'Python_' + nombre_archivo
        resultados = pd.DataFrame(resultados)
        resultados.to_csv(r"~Python\\" + nombre)

    return tiempos_list

```

```

# Ahora hacemos la regresion con fit5
fit5_files = [s for s in filt_files if "fit5" in s]

# Hagamos un df vacio para guardar los tiempos
names = [names.replace('_fit5.csv', '') for
          names in fit5_files]

res_dic = {}
for i in runs:
    res_dic[names[i - 1]] = fit5(fit5_files[i - 1])

tiempo_df = pd.DataFrame.from_dict(res_dic)

tiempo_df.to_csv(r'~timePython-Censo_fit5.csv',
                 index=False)

### Fit 10 ###
def fit10(nombre_archivo):
    fit_data = pd.read_csv(nombre_archivo)
# Vector con todas las categorias
    vector_categorias = ["SEXO", "NIVACAD",
                          "ENT_PAIS TRAB", "SITTRA", "ALFABET",
                          "AGUINALDO", "VACACIONES",
                          "SERVICIO_MEDICO"]

# Y las transformamos
    for columna in vector_categorias:

```

```

fit_data[columna] =
    fit_data[columna].astype('category')

x = fit_data[['HORTRA', 'SEXO', 'EDAD',
              'NIVACAD', 'ENT_PAIS_TRAB', 'SITTRA', 'ALFABET',
              'AGUINALDO', 'VACACIONES', 'SERVICIO_MEDICO']]
y = fit_data[['INGTRMEN']]

# Los hacemos dummies
x = pd.get_dummies(data = x, drop_first = True)

# Hacemos la regresion
regr = linear_model.LinearRegression()
start = time.time()
model = regr.fit(x, y)
end = time.time()
tiempo = end - start

# Guardamos los intercepts
intercepts = model.intercept_
# Guardamos los coeficientes de los resultados
coeficientes = model.coef_
coeficientes = np.transpose(coeficientes)

intercepts = pd.DataFrame(intercepts)
coeficientes = pd.DataFrame(coeficientes)

# Los unimos
resultados = np.vstack([intercepts, coeficientes])

```

```

# Los guardamos
    nombre = 'Python_' + nombre_archivo
    resultados = pd.DataFrame(resultados)
    resultados.to_csv(r"~\\Censo\\Python\\" + nombre)
return tiempo

# Ejecucion de la funcion
tiempos = list()
# Ahora hacemos la regresio con fit8
fit10_files = [s for s in filt_files if "fit10" in s]
for file in fit10_files:
    tiempos.append(fit10(file))
tiempos_df['fit10'] = tiempos

```

A.2. Tiempos de ejecución de los modelos de regresión lineal múltiple utilizados en el Censo

Volúmen de datos	fit5	fit6	fit7	fit8	fit9	fit10
Julia						
500	0.132	0.179	0.532	0.385	0.441	0.514
5 mil	0.006	0.009	0.196	0.010	0.186	0.283
50 mil	0.049	0.063	0.075	0.099	0.118	0.139
500 mil	0.505	0.668	0.797	0.959	1.072	1.179
2.5 millones	4.309	5.239	5.967	6.530	7.753	11.912
Python						
500	0.005	0.007	0.004	0.008	0.007	0.003
5 mil	0.011	0.013	0.013	0.011	0.014	0.011
50 mil	0.072	0.074	0.075	0.077	0.014	0.082
500 mil	0.849	0.954	0.951	0.978	0.078	0.998
2.5 millones	4.739	4.739	4.917	4.911	5.105	5.286
R						
500	25.170	29.947	27.221	31.436	34.280	34.677
5 mil	25.696	25.263	30.815	30.479	33.783	35.087
50 mil	27.704	29.673	27.350	30.134	37.695	34.308
500 mil	25.276	29.106	27.028	31.586	35.247	33.887
2.5 millones	30.227	33.199	32.107	32.359	34.347	42.471

**Tabla A.1. Tiempos de ejecución en segundos de Julia,
Python y R**

Bibliografía

- Anaconda Inc (2022). Using the R programming language in Jupyter Notebook. <https://docs.anaconda.com/anaconda/navigator/tutorials/r-lang/>.
- Barrios, E. (2010). R: Un lenguaje para análisis de datos y graficación. *Laberintos e Infinitos*, pages 35–41.
- Bates, D., Kornblith, S., Noack, A., Bouchet-Valat, M., Krabbe, M., and contributors (2022). JuliaStats/GLM.jl: v1.6.1. <https://juliastats.org/GLM.jl/v0.11/index.html>.
- Bezanson, J., Karpinski, S., Shah, V., Edelman, A., et al. (2014). Julia Language Documentation. *The Julia Manual*. <https://docs.julialang.org/en/v1.6/>, pages 1–261.
- Box, G. E., Hunter, W. H., Hunter, S., et al. (2005). *Statistics for Experimenters*. John Wiley and son.
- Carrone, F., Nicolini, M., and Obst Demaestri, H. (2021). *Data Science in Julia for Hackers*. <https://datasciencejuliahackers.com/> (Visitado el 06-10-2021).
- Datta, B. N. (2010). *Numerical linear algebra and applications*, volume 116. Siam.

- Dykstra, O. (1971). The augmentation of experimental data to maximize [X X]. *Technometrics*, 13(3):682–688.
- Garcia, S. R. and Horn, R. A. (2017). *A second course in linear algebra*. Cambridge University Press.
- Gelman, A., Hill, J., and Vehtari, A. (2021). *Regression and other Stories*. Cambridge University Press.
- INEGI (2020). Censo de Población y Vivienda 2020. <https://www.inegi.org.mx/programas/ccpv/2020/default.html>.
- Julia Programming Language (2021). Statsmodel.jl. <https://juliastats.org/StatsModels.jl/stable/>. Accessed: 2021-10-12.
- JuliaMath (2021). Polynomials.jl. <https://juliamath.github.io/Polynomials.jl/stable/>. Accessed: 2021-11-04.
- La Comunidad NumPy (2022). NumPy User Guide: Release 1.22.0. <https://numpy.org/doc/stable/numpy-user.pdf>.
- Lawson, J. (2015). *Design and Analysis of Experiments with R*, volume 115. CRC press.
- López-Bonilla, J., López-Vázquez, R., and Vidal-Beltrán, S. (2018). Moore-Penrose’s inverse and solutions of linear systems. *World Scientific News*.
- Matthes, E. (2019). *Python crash course: A hands-on, project-based introduction to programming*. No Starch Press.
- McKinney, W. and the Pandas Development Team (2022). pandas: powerful Python data analysis toolkit. <https://pandas.pydata.org/docs/pandas.pdf>.

- Mendoza, M. and Regueiro, P. (2011). Estadística bayesiana. http://allman.rhon.itam.mx/~lnieto/index_archivos/NotasBayesMR.pdf.
- Meyer, R. D., Steinberg, D. M., and Box, G. (1996). Follow-up designs to resolve confounding in multifactor experiments. *Technometrics*, 38(4):303–313.
- Mitchell, T. J. (1974). An Algorithm for the Construction of 'D-Optimal' Experimental Designs. *Technometrics*, 16(2):203–10.
- Montgomery, D. C. (2017). *Design and analysis of experiments*. John Wiley & sons.
- Morgenstern, I. and Morales, J. L. (2015). Regresión Lineal. Aritmética inexacta y algoritmos numéricos estables. *Laberintos e Infinitos*, page 25–34.
- Peng, R. (2015). *R Programming for Data Science*. <https://bookdown.org/rdpeng/rprogdatascience/>.
- Peng, R. D. and Hicks, S. C. (2021). Reproducible Research: A Retrospective. *Annual Review of Public Health*, 42:79–93.
- Project Jupyter (2022). Jupyter. <https://jupyter.org/>.
- Python Software Foundation (2022). Python 3.10.2 documentation. <https://docs.python.org/3/index.html>. Accesado: 2022-03-15.
- R Core Team (2022). What is R? <https://www.r-project.org/about.html>.
- Seber, G. A. and Lee, A. J. (2003). *Linear regression analysis*. John Wiley & Sons, 2nd edition.

- Spence, L. E., Insel, A. J., and Friedberg, S. H. (2000). *Elementary linear algebra*. Prentice Hall.
- Stanton, J. M. (2001). Galton, Pearson, and the peas: A brief history of linear regression for statistics instructors. *Journal of Statistics Education*, 9(3).
- The Julia Community (2022). Why We Use Julia, 10 Years Later. <https://julialang.org/blog/2022/02/10years/>. Accessed: 2022-03-28.
- Ushey, K. and Alaire, J. (2022). R Interface to Python. <https://rstudio.github.io/reticulate/#r-interface-to-python>.
- Vela, P. (2022). Discriminación de Factores en Experimentos Factoriales.

Cómputo estadístico: Julia, Python y R,
escrito por Valeria Aurora Pérez Chávez, se
terminó de imprimir de madrugada,
con mucha cafeína en las venas
y ojeras en la cara.