

INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO



Una tesis extendida ($\overline{\text{tesis}}$)

TESIS

QUE PARA OBTENER EL TÍTULO DE

LICENCIADO 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 “**Una tesis extendida ($\overline{\text{tesis}}$)**”, 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.

EL SERCH

Índice general

1. Introducción	1
2. Julia	3
2.1. Reproducibilidad	3
2.2. Instalación	4
2.3. Símbolo del sistema	5
2.3.1. <i>Multithreading</i>	5
2.4. Básicos de Julia	6
2.4.1. Operaciones básicas	6
2.4.2. <i>Strings</i> (secuencias de caracteres)	8
2.4.3. Funciones	8
2.4.4. Vectores y Matrices	9
2.4.5. Instalación de un paquete	11
2.4.6. <i>DataFrames</i>	12
2.4.7. Regresiones	14
3. Python	19
3.1. Listas	19
3.2. Paquetes	20
3.2.1. NumPy	21

3.2.2.	pandas	22
3.2.3.	os	22
3.2.4.	scikit-learn	23
3.2.5.	itertools	23
3.3.	Jupyter	24
3.3.1.	Julia	24
3.3.2.	R	25
4.	R	26
4.1.	lm	27
5.	Ajuste de polinomios	29
5.1.	El problema	29
5.2.	Los datos	30
5.3.	Planteamiento del problema	31
5.4.	Métodos para solucionar el problema	33
5.4.1.	<i>GLM</i>	33
5.4.2.	Descomposición QR versión económica	34
5.4.3.	Descomposición de valores singulares	37
5.4.4.	<i>Polynomials</i>	40
5.5.	Evaluación de los métodos	42
5.6.	Número de condición y precisión de la solución	48
5.7.	Opinión de la autora	51
6.	Regresión Lineal Múltiple	53
6.1.	El problema	53
6.2.	Los datos	54
6.3.	Planteamiento del problema	55
6.4.	Regresiones	58
6.4.1.	Observaciones	60

6.5. Resultados	62
7. MDopt	63
7.1. Metodología completa	64
7.2. Criterio MD	65
7.3. Algoritmo de intercambio	69
7.4. Función MDopt	72
7.5. Comparación entre lenguajes	73
7.6. Ejemplos y resultados	77
7.6.1. Ejemplo 1 - Proceso de moldeo por inyección . .	77
7.6.2. Ejemplo 2	86
8. Conclusiones	91
A. Extras	93

Índice de algoritmos

Índice de tablas

2.1. Operaciones básicas en Julia	7
7.1. Datos para el ejemplo 1	78
7.2. Modelos con la probabilidad posterior más alta para el ejemplo 1	78
7.3. Ejemplo 1, Colapsado en los factores A, C, E y H	80
7.4. Resultados para el ejemplo 1	85
7.5. Datos para el ejemplo 2	86
7.6. Resultados para el ejemplo 2	90

Índice de figuras

5.1. Conjunto de datos para el ejercicio	30
5.2. Resultados del polinomio grado 5	46
5.3. Resultados del polinomio grado 6	46
5.4. Resultados del polinomio grado 10	47
5.5. Tiempos de ejecución para cada método	48
7.1. Diagrama de metodología descrita por Meyer et al. (1996)	66
7.2. Diagrama de la función MDopt	73

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” Las palabras de la chilena Pamela Bustamante englobal la razón de ser de esta tesis.

Vale: Aquí sí vale escribir en primera persona?

Vale: Ref:
@pambus
<https://julia>

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 que estaba fundamentado en estadística bayesiana y requería de una gran cantidad de simulaciones. No tarde mucho tiempo en darme cuenta de que los recursos de ayuda para entender Julia son escasos y la mayoría están en inglés.

En ese tiempo aprendí mucho, pero fue a costa de muchos errores y tiempo invertido en cosas sencillas, pero que yo no sabía. Me surgieron preguntas como “¿Por qué Julia y no Python o R?”, “¿En verdad nadie

más ha tenido el mismo problema que yo?”, “¿Por qué no hay más ayuda sobre este tema?” Esta tesis es la respuesta.

¿Por qué Julia? Ya existen otros lenguajes de programación que tienen más usuarios y ayuda disponible. Para mí, la única forma en que valga la pena hacer el cambio a Julia es que sea mejor que lo que ya conozco. Por lo tanto, esta tesis es la comparación de Julia con R y Python en tres ejercicios distintos.

Es importante mencionar que este trabajo no es un manual de Julia ni de ningún otro lenguaje. Se da una breve introducción de Julia cuya intención es permitir que el lector navegue fácilmente por el código presentado. Además, se asume que el lector ya está familiarizado con Python y R por lo que solamente se presentan conceptos claves.

Después, se explican los tres ejercicios que pusieron a prueba los lenguajes. El primero toma datos del Instituto Nacional de Estándares y Tecnología y su objetivo es medir la precisión numérica al hacer un ajuste de un polinomio de grado 10. El segundo toma los datos del CENSO 2020 para ilustrar el manejo y análisis de una gran cantidad de datos. El tercero y último sale del camino del análisis de datos y presenta un algoritmo que se utiliza en la discriminación de modelos en diseños de experimentos. En este ejercicio, los cálculos son más intensivos por lo que busca medir la capacidad de cómputo de los lenguajes.

Vale: Como termino? 'Gracias y espero que lo disfruten? jajaja'

Capítulo 2

Julia

Julia es un lenguaje de programación gratis cuyo propósito general es ser tan rápido como C, pero manteniendo la facilidad de lenguaje de Ro 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. \(2021\)](#). Esta combinación de características hace que Julia sea un lenguaje de programación que ha tomado mucha fuerza en la comunidad científica. Ya que no es un lenguaje muy conocido, en esta sección explicaré como instalar Julia en una computadora con sistema Windows y algunos de los básicos del lenguaje.

2.1. Reproducibilidad

Antes de empezar, es necesario enfatizar que esta tesis es completamente reproducible. En Peng y Hicks definen que “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 poco ambigua, sí resaltan que la reproducibilidad es un medio para revisar y, posteriormente, confiar en el análisis de otros.

Por lo tanto, para este trabajo decidí publicar el código que utilicé en GitHub. Además, hago hincapié en las fuentes de los datos. A pesar de que esta tesis no es un manual de Julia, Python o R considero de suma importancia publicar mi trabajo para que les sirva a mis compañeros como referencia o punto de partida.

2.2. Instalación

Este trabajo está hecho y escrito en Windows, por lo que explicaré la instalación de Julia y todas las demás aplicaciones en este sistema operativo. Sin embargo, sé que la instalación en Mac y Linux es muy similar.

Al momento de la escritura y publicación de esta tesis la versión de Julia disponible es la v1.6.3. El primer paso ara descargar Julia, es entrar al link <https://julialang.org/downloads/>.

En esta versión de Julia, las opciones disponibles de descarga son un instalador de 64-bits o uno de 32-bits. Para saber el tipo de Sistema que tiene tu ordenador debes seleccionar el botón de **Start**, después Configuración > Sistema > Acerca de. En esta opción puedes ver el tipo de sistema que tiene tu computadora. Con esta información puedes elegir el instalador para tu computadora. Es importante seleccionar el **installer** y no el **portable**. Una vez descargado, seleccionas el archivo .exe y sigues los pasos de instalación.

2.3. Símbolo del sistema

Una vez instalado puedes correr Julia desde el símbolo de sistema o desde otro programa como Atom, Visual Studio Code o Jupyter Notebook. Una de las ventajas de utilizar Julia desde el símbolo de sistema (también conocido como *Command Prompt* o *cmd*) es que puedes controlar algunos parámetros del lenguaje. Mi sugerencia es que comiences a usar Julia directo desde el ícono que se genera automáticamente en la descarga.

Posteriormente, cuando entiendas lo básico y empieces a generar programas que requieran mayor nivel computacional entonces puedes usar el *cmd* para correr Julia. Para facilitar esto, te recomiendo agregar Julia a un *PATH*. Las instrucciones para hacerlo en Windows 10 están en la página <https://julialang.org/downloads/platform/#windows>.

2.3.1. *Multithreading*

Una de las razones por la que Julia tiene más velocidad que otros lenguajes es por su capacidad para multihilo (*multithreading* en inglés). Esto significa que puede correr diferentes tareas de manera simultánea en varios hilos. Explicado de forma más simple, la meta de los autores de Julia fue hacer un lenguaje de programación con un rendimiento tan alto que pudiera hacer varias cosas a la vez. Debido a que uno de los objetivos de esta tesis es medir la eficiencia y velocidad de Julia, es crucial conocer la característica del *multithreading* y como utilizarla.

Si estás usando Julia por medio del *cmd* es necesario modificar la cantidad de hilos que vas a utilizar antes de ejecutar Julia. En Windows, esto se hace escribiendo `set Julia_NUM_THREADS=4` (Bezanson et al., 2014). Si estás trabajando con otro sistema operativo, este link te puede ayudar a cambiar la cantidad de hilos

<https://docs.julialang.org/en/v1/manual/multi-threading/>. En este ejemplo, se cambiaron los hilos a 4, pero se puede poner cualquier número. Sin embargo, se recomienda que el número no exceda de la cantidad de procesadores físicos de la computadora. Después de hacer esta modificación, ahora sí puedes ejecutar Julia con la cantidad de hilos modificada.

Si estás usando Julia en algún editor de texto o programa externo la modificación del número de hilos se hace de forma diferente. Cada programa tiene su manera de hacerlo y usualmente las instrucciones viene en el manual del mismo.

Para observar que el cambio se ejecutó de manera correcta (en cualquier opción) basta con correr el comando `Threads.nthreads()` y observar que la respuesta sea el número deseado.

2.4. Básicos de Julia

Como ya mencioné, Julia busca ser un lenguaje de programación sencillo e intuitivo. Por lo tanto, su sintaxis es bastante sencilla. La asignación de variables se hace con un signo de igualdad `=`. El ejemplo más sencillo de esto es ejecutar `x = 2` donde se asigna a `x` el valor de 2. Toda la información sobre la sintaxis que utiliza Julia la obtuve de su manual oficial [Bezanson et al. \(2014\)](#).

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, para cada operación binaria existe su correspondiente operación punto (*dot operation* en inglés). Estas funciones están definidas para efectuarse elemento por elemento en vectores y matrices. Para llamarse basta agregar un punto antes del operador binario. Por ejemplo, `[1 9 9 7] .^ 2` eleva cada uno de los elementos del vector al cuadrado.

También es importante mencionar que Julia maneja los números imaginarios utilizando el sufijo `im`. Sin embargo, no los utilice en este trabajo así que omitiré dar una mayor explicación.

2.4.2. *Strings* (secuencias de caracteres)

Además de números, Julia puede asignar caracteres a variables. Esto se hace utilizando las comillas dobles. Similar a otros lenguajes de programación, podemos 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 genial"
julia> string[6]
't': ASCII/Unicode U+0074 (category Ll: Letter, lowercase)

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

Además, Julia también cuenta con la opción de concatenación de múltiples strings. Esto se hace utilizando un asterisco * para separar 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
"licencia 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

```
function f(x, y)
    x + y
end
```

Además, puedes agregar la palabra **return** para que la función regrese un valor. Por ejemplo, si quisiera tener una función a la que le doy dos números y regrese el número mayor, los comandos serian de la forma:

```
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 .

2.4.4. Vectores y Matrices

Definición 1. *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 para definir un vector columna se hace uso de los corchetes cuadrados `[]` y comas. 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. Es muy importante aprender a identificar como Julia lee los objetos ya que cada objeto tiene características y funciones diferentes.

Si quisiera definir un vector renglón, haría exactamente lo mismo excepto que omitiría el uso de las comas. Sin embargo, es importante señalar que ahora Julia tomó el objeto A como una matriz, no como un vector.

Definición 2. *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 única diferencia de que en lugar de punto y coma se cambia de renglón. Esta opción puede ser un poco tediosa ya que requiere que las columnas estén alineadas. Sin

embargo, es una forma más visual de ver las matrices. En la siguiente tabla se pueden ver 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 `A_2[2, 2]`.

Es importante mencionar que como muchos otros lenguajes, Julia ya tiene programadas las operaciones básicas de las matrices en el paquete `Linear Algebra`. El catálogo de funciones es bastante extenso para incluirlo en este trabajo, pero lo puedes encontrar en <https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/>.

2.4.5. Instalación de un paquete

Para hacer cualquier otra operación fuera de lo básico que ya mencioné, Julia hace uso de paquetes. Los paquetes son similar a las librerías en R. La lista completa de paquetes registrados en Julia se encuentra en <https://juliapackages.com/>.

Lo primero que hay que saber sobre estos paquetes es su instalación y uso. Antes de instalar cualquier paquete primero hay que usar el paquete

`Pkg` que ya viene por default cuando descargas Julia. Después, hay que usar el comando `Pkg.add` para agregar el paquete nuevo. Finalmente, llamas al paquete con el comando `using`. A continuación está un resumen de lo anterior.

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

Después, cada vez que vayas a usar un paquete basta con llamar al paquete con el comando `using`. En las siguientes secciones explico y ejemplifico 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 de alguna fuente externa.

Crear un dataframe

Aunque de manera general los dataframes se utilicen para manejar grandes cantidades de información exportada de otros formatos, es importante saber como se crea y manipula un dataframe desde cero en Julia. Hacerlo es simple. Primero, hay que escribir 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 lo mismo con la cantidad de columnas que se requieran. Por ejemplo, para hacer un dataframe con las claves únicas y nombres de cinco mujeres el código sería el siguiente:

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

Es importante nombrar las columnas del dataframe ya que de esta forma basta con escribir `df.col` para referirnos a la columna 'col' del dataframe 'df'. De la misma manera, si quisiera agregar una columna nueva basta con asignarle datos a `df.colNueva`. Por ejemplo, si quisiera agregar una columna llamada `color` al dataframe del ejemplo anterior, el código sería el siguiente:

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

Como cualquier otro objeto en Julia, los dataframes tienen diferentes funciones. Es posible seleccionar un subgrupo de datos, agregar información, modificar y eliminar columnas y renglones, etc. Sin embargo, eso no es de relevancia para esta tesis por lo que lo omitiré.

Importar datos en un dataframe

Como ya mencioné, los dataframes son utilizados para contener grandes cantidades de información. Usualmente, esta información no es generada en Julia por lo que hay importarla. Una de las formas más rápidas es usando el paquete `CSV`. Recuerda que para usarlo por primera vez hay que seguir los pasos descritos en la sección [2.4.5](#).

Una vez instalado el programa, basta utilizar el comando `CSV.read` y la ruta de la ubicación del archivo para exportar los datos.

```
df = CSV.read("C:/Users/Valeria/Documents/ejemplo.csv", DataFrame)
```

```
julia> df = CSV.read("C:/Users/Valeria/Documents/ITAM/Tesis/Julia con R/ejemplo_csv.csv", DataFrame)
5x6 DataFrame

```

Row	id Int64	nombre String15	color String7	deporte String15	lugar_residencia String31	estatura 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

Vale: Tengo que arreglar esto sí o sí

Los dataframes se pueden manejar de diferentes maneras. En este trabajo utilicé algunas de ellas, pero en caso de que tengas más dudas puedes consultar el manual oficial del paquete en <https://dataframes.juliadata.org/stable/>.

2.4.7. Regresiones

¿Cuál es el punto de tener una muestra de tamaño significativo si no sé analizarla? Una de las maravillas que nos regala la estadística es el uso de regresiones para intentar encontrar una explicación a los datos. 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. \(2021\)](#)

En pocas palabras, una regresión es una fórmula que intenta explicar como una variable depende otras. En Julia, esto se puede hacer con ayuda del paquete GLM, ya que facilita el cálculo de modelos lineales. Como todos los paquetes, primero hay que instalarlo usando los pasos en [2.4.5](#).

En este paquete la función principal se llama `glm`. En el manual oficial [Bates et al. \(2022\)](#) está descrita la manera en que se pueden

generar modelos más avanzados. La función principal es `glm(formula, data, family, link)` donde

- **formula:** usa los nombres de las columnas del dataframe de datos para referirse a las variables predictoras.
- **data:** el dataframe que contenga los datos de los predictores de la fórmula.
- **family:** podemos elegir entre `Bernoulli()`, `Binomial()`, `Gamma()`, `Normal()`, `Poisson()` o `NegativeBinomial()`
- **link:** se usa para especificar la función liga o *link function*. La lista de posibles opciones está en el manual oficial de GLM.

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 use los datos de [Gelman et al. \(2021\)](#). Dicha 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:

Vale:
Arreglar
esto


```
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   growth   vote   inc_party_candidate   other_candidate
Int64 Float64 Float64 String15 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
```

En este modelo busco que el voto sea resultado del crecimiento económico. El código para hacer esto en Julia es

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

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

Regresión lineal múltiple

La regresión lineal simple es el caso específico de la regresión lineal múltiple. La diferencia es que en el segundo caso hay múltiples predictores que cumplen 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 $n \times k$ con renglón X_i .

Para ejemplificar este tipo de modelo use un ejemplo que consta de dos predictores y la interacción entre ellos. De nuevo utilicé los datos 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`).

Busco 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

Vale: Como pongo el directorio de donde está mi base?

```
using DataFrames, GLM, CSV
```

```
data_kid = CSV.read("C:/Users/Valeria/Documents/ITAM/Tesis/Julia con
```

```
fm = @formula(kid_score ~ mom_hs + mom_iq + mom_hs*mom_iq)
```

```
kidscore_lm = lm(fm, data_kid)
```

Lo cual da como resultado el modelo

$$kid_score = -11.48 + 51.26 * mom_hs + 0.97 * mom_iq \\ -0.48 * mom_hs * mom_iq + \epsilon$$

Uno de los aspectos que es necesario resaltar en este ejemplo es que para incluir la relación entre dos predictores basta usar 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 hay que hacerle cambios a la base de datos en sí. Si **Julia** no reconoce estas columnas como categóricas entonces hay que cambiar su tipo en el dataframe. Abordo este problema más a fondo en el capítulo [6.4](#) . Por otro lado, puedes 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 por lo es más propenso a tener errores.

Capítulo 3

Python

“Python es un lenguaje de programación que te permite trabajar rápido e integrar sistemas más eficientemente” es la primera frase que se lee en la página oficial de Python <https://www.python.org/>. Guido van Rossum comenzó a crear el lenguaje a finales de los ochentas, pero lo hizo público hasta 1991. Empresas importantes como Youtube y Google han elogiado Python por su rapidez y constante desarrollo. Es un lenguaje más antiguo que Julia y con mucha mayor popularidad. Consecuentemente, hay muchos videos, artículos, blogs y libros sobre su uso y desarrollo. Decidí incluirlo en esta tesis ya que considero es un excelente punto de comparación con Julia no solo en rapidez sino también en la sencillez y facilidad de programación. En este capítulo voy a explicar los paquetes principales y la interfaz que utilice.

Vale: Lo puedo dejar así?

3.1. Listas

“Una *lista* es una colección de elementos en un orden particular” Matthes (2019). Las listas son el objeto principal y más básico de

Python. Las listas son una estructuras de datos por lo que se usan para almacenar varios elementos en una sola variable. Las listas se crean usando paréntesis cuadrados []. Por ejemplo si quisiera hacer una lista de animales en el zoológico haría `animales = ["zebra", "león", "jirafa", "elefante"]`. Para acceder a un elemento de la lista hay que usar los paréntesis cuadrados. Por ejemplo, `animales[1]` me regresaría "león". Uno de los aspectos más importantes es que a diferencia de R y Julia, las listas comienzan a numerar sus elementos desde el cero.

En esta tesis utilice las listas como estructura de datos ya que están ordenadas, pueden ser cambiadas y permiten valores duplicados. Por lo tanto, son fáciles y eficientes para trabajar. Como cada estructura, las listas tienen sus propios métodos que vienen en listas y explicados en la documentación de Python [Python-Software-Foundation \(2022\)](#).

3.2. Paquetes

No es sorpresa que Python tenga muchos paquetes para hacer todo tipo de programación. Una característica que me gusta de este lenguaje es que para usar cualquier instrucción de un paquete tienes que primero 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`. En este caso, `np` es el apodo del paquete NumPy. Si quisiera usar la función `array` del paquete NumPy tendría que poner `np.array`. Esto podría parecer tedioso pero lo considero una ventaja ya que siempre sabes el paquete que estás usando.

3.2.1. NumPy

NumPy es el paquete fundamental para computación científica en Python ya que proporciona los objetos de matriz multidimensional. Hay varias diferencias entre matrices NumPy y secuencias del Python estándar. Algunas de ellas son que los arreglos de NumPy tienen dimensiones fijas en su creación que no se pueden cambiar; sus elementos deben ser del mismo tipo de dato; facilitan operaciones matemáticas en grandes cantidades de datos; y, finalmente, una gran parte de la comunidad que utiliza Python también usa arreglos de NumPy [NumPy \(2022\)](#).

En esta tesis use NumPy para crear y manipular arreglos y hacer un ajuste polinomial de mínimos cuadrados. A continuación está la lista completa de comandos que utilice con su explicación. A pesar de que ya maneje los comandos, la información viene del paquete oficial de [NumPy \(2022\)](#).

Vale:
Arreglar
esto

- `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 de los índices obj.
- `np.arange`: Crea un arreglo con valores espaciados uniformemente desde start hasta el número antes de stop.
- `np.transpose(a)`: Transpone el objeto a.
- `np.concatenate(a_1, a_2, \dots)`: Une la secuencia de arreglos en uno 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 `a`.
- `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 primordial y básico de Python ya que se enfoca en la manipulación y análisis de datos. Sus funciones se enfocan en el uso eficiente de dataframes, leer y escribir datos, agrupación y unión de varios conjuntos de datos, entre otros [y el Equipo de Desarrollo de Pandas \(2022\)](#). En esta tesis utilice los siguientes comandos de pandas.

- `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 variables categóricas en variables indicadoras o dummie.

3.2.3. os

Otro paquete que utilice en este trabajo fue `os` ya que proporciona una manera de usar la funcionalidad dependiente del sistema operativo. En otras palabras es el paquete que permite hacer la conexión entre Python y los archivos de una computadora. Los

comandos de este paquete que utilice son dos. El primero fue `os.chdir(path)` que permite seleccionar el directorio en el que estoy trabajando. 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 hacer *machine learning* o aprendizaje automático en Python. También es conocido como **sklearn** y proporciona herramientas simples y eficientes para la predicción en análisis de datos. Sus herramientas hacen clasificación, regresión, *clustering* o agrupamiento, reducción de dimensiones y selección de modelos.

Para este trabajo utilice la parte de regresiones lineales del paquete. El usuario puede importar el paquete completo usando `import sklearn` o solo la parte de modelos lineales con el comando `from sklearn import linear_model`.

Con el paquete cargado, `regr = linear_model.LinearRegression()` guarda en `regr` que busco ajustar un modelo linear definido como la ecuación 2.1 usando el método de mínimos cuadrados. Después, `model = regr.fit(x, y)` calcula los coeficientes β .

3.2.5. itertools

Itertools es un módulo que implementa un conjunto de herramientas rápidas y eficientes en cuanto a la memoria [Python-Software-Foundation \(2022\)](#). Algunas de las herramientas que tiene este módulo se pueden recrear sin la necesidad del mismo, pero la ventaja de utilizar **itertools** es la velocidad en la que las genera. En

este trabajo yo utilice `itertools.combinations()` para crear las combinaciones de posibles factores activos del problema del capítulo 7.

Vale:
verificar
que sí sea
el capítulo
de MDopt

3.3. Jupyter

“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” [Jupyter \(Jupyter\)](#).

La manera más fácil para obtener Jupyter es instalando Anaconda. Anaconda es una interfaz gráfica que permite manejar y administrar aplicaciones, paquetes, ambientes y canales sin necesidad de usar comandos en el `cmd`. Para instalar Anaconda en Windows hay que ir a la página <https://docs.anaconda.com/anaconda/install/windows/> y seguir las instrucciones de instalación. Esto puede tomar unos cuantos minutos.

La versatilidad de Jupyter en los tres lenguajes es la razón principal por la que decidí usarlo en esta tesis. Poder usar los tres lenguajes en un mismo software me permitió tener una mejor organización y permitió una traducción entre lenguajes más sencilla.

Uno de los requisitos para instalar Jupyter es tener `Python`. Por lo tanto, este lenguaje que ya viene sin necesidad de ninguna otra instalación. El caso de R y Julia no es igual. En las siguientes secciones explico como instalarlos.

3.3.1. Julia

El primer paso es haber instalado Julia en la computadora. Después, es necesario correr los comandos `using Pkg; Pkg.add('IJulia')`. Es decir, es necesario instalar el paquete `IJulia`. Esto solo se tiene que

hacer una vez. Para confirmar que la instalación está bien hecha hay que abrir Jupyter, seleccionar **New** y debe aparecer la opción de Julia 1.6.3 (o la versión de Julia que esté instalada en la computadora).

3.3.2. R

Hay varias maneras de instalar R en Jupyter, pero la manera que viene en el manual de Anaconda [Inc \(2022\)](#) es la que voy a exponer.

1. Abrir el Navegador de Anaconda (no confundir con el Jupyter Notebook).
2. Selecciona **Environments** y después la opción de **Create** ubicada en la esquina inferior izquierda.
3. Aparecerá una ventana donde puedes nombrar el **Environment** como prefieras. Lo importante es seleccionar la versión de **Python** que tengas y también seleccionar la casilla al lado de R. Después pulsar la opción de **Create**.
4. Para usar el ambiente que acabas de crear en Jupyter debes seleccionar la flecha derecha al lado del nombre del ambiente que creaste en el paso anterior. Entre las opciones seleccionar la opción de **Open with Jupyter Notebook**.
5. Por último, selecciona el botón de **New** y después R para crear un archivo que trabaje con R.

Capítulo 4

R

“Ross Ihaka y Robert Gentleman, del departamento de Estadística de Auckland University, en Nueva Zelanda, estaban interesados en 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 que hubiera una transición de usuario a desarrollador. Los creadores buscaron crear un lenguaje que podría usarse para hacer un análisis de datos de manera interactiva y, además, para escribir programas más largos [Peng \(2015\)](#).

Uno de los puntos más fuertes a favor 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 [Team \(2022\)](#).

R se considera como uno de los lenguajes más sencillos para comenzar a aprender a trabajar métodos estadísticos en la computadora. Es un lenguaje muy sencillo de entender y perdona muchas especificaciones que Julia y Python no. Además, es uno de los más conocidos a nivel mundial

por lo que hay muchos libros, artículos y páginas web que abordan casi cualquier tema que se le relacione. Las razones anteriores son el motivo por el cual se decidió incluir dicho lenguaje en este trabajo.

Además, por ser el lenguaje más común y su inclusión en los temarios de las universidades, se parte de que el lector ya tiene los conocimientos básicos para entender esta tesis. Asimismo, la propia versatilidad de las funciones del lenguaje permitieron que los ejercicios de esta tesis funcionaran con pocas funciones. En el último ejercicio se utiliza un paquete ya programado, mientras que los primeros dos se enfocan en la función `lm` explicada a continuación.

4.1. `lm`

`lm` es una función usada para ajustar modelos lineales. Su fórmula es `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 puede ajustar modelos muy simples o más avanzados, dependiendo de los parámetros que se utilicen.

El caso más sencillo es el modelo de regresión lineal ya que el parámetro de fórmula se ve de la manera

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

Es decir, se pone la variable de respuesta y seguido de una virgullita y después los n predictores que se estén utilizando.

El caso de las regresiones polinomiales tiene una sintaxis un poco más complicada. La fórmula comienza como en el caso anterior, con la variable y y la virgullita. Sin embargo, para elevar el regresor x a la potencia k se debe escribir dentro de `I(...)`. Por ejemplo, si se quisiera

ajustar un conjunto de datos a un modelo $y \sim x^2$ el comando es

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

El comando `I(...)` se usa 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, R entiende que se debe tomar x^2 como una variable y no como la interacción de segundo orden de x .

Por otro lado, uno de los parámetros que tiene la fórmula de `lm` es `tol`, la tolerancia del ajuste. Cada método de cómputo estadístico tiene su tolerancia default, pero en ocasiones se puede modificar. En el ejercicio que se presenta en el capítulo 5 dicha tolerancia tuvo que ser modificada para lograr el cálculo correcto. Usualmente este parámetro no necesita ser cambiado, pero es útil tomarlo en cuenta para las ocasiones donde los datos son extremadamente sensibles y se busca un ajuste preciso.

En los siguientes capítulos se ponen a prueba los tres lenguajes ya descritos (R, Julia y Python) en tres ejercicios diferentes. El primero es el ajuste de un modelo lineal de grado diez con datos extremadamente sensibles. En este ejercicio se mide la precisión 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 muchos cálculos intensivos.

Capítulo 5

Ajuste de polinomios

5.1. El problema

El problema que se aborda en esta sección es el mismo que presentaron [Morgenstern and Morales \(2015\)](#) en su artículo publicado en la revista de Laberintos e Infinitos. La diferencia es que en esta tesis se utiliza Julia, R y Python mientras que ellos compararon R, Excel, Stata, SPSS, SAS y Matlab. El problema es el siguiente.

Supongamos que tenemos un conjunto de datos con solamente dos variables x , y . El reto es ajustar la información a un polinomio de grado k . Es decir, se busca ajustar los datos al modelo

$$y = \sum_{j=0}^k \beta_j x^j + \epsilon$$

donde j es el grado de la variable x .

El problema consiste en encontrar los coeficientes que mejor cumplan la ecuación anterior. Una manera más compacta y simple de plantear el problema es de forma matricial

$$y = X\beta. \quad (5.1)$$

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

Los datos que se usan son proporcionados por el Instituto Nacional de Standards y Tecnología (*NIST* por sus siglas en inglés). Dentro de sus múltiples conjuntos de datos, se selecciono el llamado **filip** que se encuentra en <https://www.itl.nist.gov/div898/strd/lls/data/LINKS/DATA/Filip.dat>. Los datos constan de 82 pares ordenados (x_i, y_i) cuya gráfica ?? se muestra a continuación.

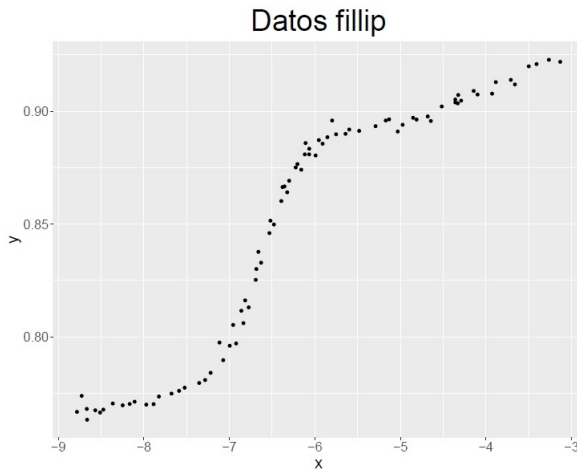


Figura 5.1. Conjunto de datos para el ejercicio

Se selecciono este conjunto de datos porque además de proporcionar la información necesaria para el ajuste, también dan la respuesta al

vector β con alta precisión en sus dígitos. Por lo tanto, es posible verificar la precisión del resultado de los coeficientes β .

5.3. Planteamiento del problema

De esta forma, ya es posible aterrizar la ecuación 5.1 al conjunto de datos. El vector y es de tamaño 82 y corresponde a la columna del mismo nombre en el conjunto `fillip`. Por otro lado, la matriz X se define como

$$X = \begin{pmatrix} 1 & x_{1,1} & x_{1,2} & \dots & x_{1,10} \\ 1 & x_{2,1} & x_{2,2} & \dots & x_{2,10} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & x_{81,1} & x_{81,2} & \dots & x_{81,10} \\ 1 & x_{82,1} & x_{82,2} & \dots & x_{82,10} \end{pmatrix}$$

Representar la matriz X de esta forma tiene una gran ventaja. Cada elemento puede ser visto como $x_{i,j}$ donde el renglón i representa la observación i de los datos. Por otro lado, 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 los datos elevado a la 5 potencia. Sin embargo, es importante reconocer que el elemento $x_{34,5}$ realmente está en la columna número 6 de la matriz. El pequeño cambio de notación es solamente para no perder de vista la potencia de las observaciones.

Por último, el vector β de la ecuación 5.1 es de dimensión 11 y es la incógnita del problema.

A lo largo del capítulo se explica la teoría y su aplicación en Julia. En primer lugar es necesario cargar los datos. El código que se utilizó es el siguiente


```

using CSV, DataFrames, Polynomials

filip = CSV.read("filip_data.csv", DataFrame)

x = filip.x
y = filip.y
k = 10 #grado del polinomio
n = length(x) # número de observaciones

```

Por otro lado, para generar la matriz X se creo una función que tiene como argumento la variable k que representa el grado del polinomio que se quiere ajustar. Asimismo, k especifica el número de columnas de la matriz.

```

function generar_X(k) # k es la potencia del polinomio

    n = size(filip, 1) #numero de renglones

    # Inicialización de una matriz vacía
    X = Array{Float64}(undef, n, k + 1)
    # Sabemos que 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

5.4. Métodos para solucionar el problema

5.4.1. *GLM*

Dado que el problema es ajustar una regresión lineal, el primer paquete que se piensa en utilizar es GLM ya que sus siglas se traducen a “Modelos Lineales Generalizados”. En el capítulo 2.4.7 se da una explicación más detallada de su función principal, `lm`.

En este ejercicio se busca ajustar un polinomio de grado 10 a los datos guardados con el nombre de `filip`. Por lo tanto, el código en Julia es

```
x_fit = glm(@formula(y ~ 1 + poly(x, 10)), filip)
```

donde `poly(x, 10)` es una función con sintaxis extendida que se utiliza específicamente para regresión polinomial. Esta función está descrita en la documentación del paquete `StatsModels` elaborado por [Language \(2021\)](#).

En este ejercicio, los parámetros de `family` y `link` se pueden omitir ya que el ejercicio requiere el modelo más simple.

Los resultados para todos los métodos se encuentran en la sección 5.5. Es claro que para este método (GLM en las tablas de resultados 5.2, 5.3, 5.4) los cálculos no arrojaron un resultado correcto. Dado que el instituto NIST proporciona la respuesta fue claro observar que la estimación de los coeficientes no fue precisa.

5.4.2. Descomposición QR versión económica

Uno de los métodos para solucionar problemas de mínimos cuadrados es usar la descomposición QR. Por lo tanto, es el segundo método que utilice para obtener los valores β de 5.1.

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

Sin embargo, en este problema no es posible utilizar la factorización QR usual ya que las dimensiones de la matriz $X_{n \times m} = X_{82 \times 11}$. Por lo tanto, X tiene rango $r = 10 < n$ por lo que la matriz R de la descomposición QR es singular. Como consecuencia, no se puede generar una base ortonormal de $R(X)$. A continuación se presenta la definición de una base ortonormal.

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, el proceso de factorización QR se puede modificar usando una matriz de permutación para generar una base ortonormal.

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

La idea del método QR modificado es generar una matriz de permutación P tal que

$$AP = QR, \text{ donde } R = \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix}$$

En este caso, si tomamos 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). Además, la factorización QR versión económica siempre existe debido al siguiente teorema de (Datta, 2010, p. 532) .

Teorema 5.1. 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.

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

```
### Con QR versión económica
using LinearAlgebra
F = qr(X, Val{true})
Q = F.Q
```

$$P = F \cdot P$$

$$R = F \cdot R$$

Para continuar resolviendo el problema original 5.1 y obtener los valores de los elementos de β es necesario hacer un poco de álgebra.

Por el teorema 5.1, sabemos que X siempre tiene descomposición QR versión económica. Es decir, $XP = QR$. Por otro lado, como P es matriz de permutación existe z tal que $Pz = \beta$.

Por lo tanto, ya hay una expresión para β que se puede sustituir en la ecuación 5.1 para obtener

$$y = X(Pz).$$

A la vez, sustituyendo en la fórmula de la descomposición QR

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

Uniendo las dos ecuaciones anteriores, obtenemos

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

Como tenemos los valores de y , Q y R , podemos resolver para obtener los valores de z y finalmente obtener β haciendo

$$\beta = Pz$$

En Julia, esto se programa de la siguiente manera

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

Este método tampoco funcionó. En las tablas de resultados 5.2, 5.3, 5.4, la columna QRvEcon muestran que el método parecía funcionar hasta llegar al polinomio de grado 10, donde falló. Con dos métodos fallidos es factible empezar a considerar que los datos son tan sensibles que la propagación del error es tal que no permite un buen ajuste del polinomio. Sin embargo, se continuó buscando la solución usando otros métodos.

5.4.3. Descomposición de valores singulares

La tercer manera en la que se intento solucionar este problema fue usando la descomposición de valores singulares para obtener la matriz pseudoinversa de Moore-Penrose.

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)

Los valores singulares tienen muchas aplicaciones. Una de ella es obtener la descomposición de valores singulares (DVS) para resolver ecuaciones lineales.

Teorema 5.2. 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

$$A = U\Sigma V^* \quad (5.2)$$

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 5.2 con las características del teorema anterior es la definición de la descomposición en valores singulares (DVS).

Es importante observar que 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

Ahora bien, ya que se explicó la descomposición de valores singulares se puede definir su uso en la pseudoinversa de Moore Penrose.

Teorema 5.3. *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).

Con la ecuación anterior es claro que lo único que cambia al calcular la pseudoinversa es la matriz Σ . Sin embargo, esta nueva matriz A^\dagger tiene propiedades interesantes como

- $(A^T A)^\dagger A^T = A^\dagger$
- $(A A^T)^\dagger A = (A^\dagger)^T$
- $(A^T A)^\dagger (A^T A) = A^\dagger A = V V^T$

Es importante volver a mencionar las dimensiones de la matriz $X_{n \times m} = X_{82 \times 11}$. Como $m < n$, sabemos que hay más ecuaciones que variables desconocidas. Por lo tanto, el sistema lineal está sobredeterminado.

De la ecuación 5.1 se puede multiplicar por X^T para obtener

$$X^T X \beta = X^T y, \quad y \in \text{Col}(V). \quad (5.3)$$

La ecuación 5.3 siempre da un sistema determinado (balanceado) López-Bonilla et al. (2018). Ahora bien, multiplicando 5.3 por $(X^T X)^\dagger$ y usando las propiedades de la matriz pseudoinversa que se mencionaron anteriormente podemos obtener

$$\begin{aligned}
(X^T X)^\dagger X^T X \beta &= (X^T X)^\dagger X^T y \\
\iff X^\dagger X \beta &= X^\dagger y \\
\iff V V^T \beta &= X^\dagger y \\
\beta &= X^\dagger y
\end{aligned}$$

Por lo tanto, la pseudo inversa de Moore Penrose da la solución de mínimos cuadrados de 5.1 (López-Bonilla et al., 2018).

En Julia, este método se puede programar en tres líneas de forma muy sencilla.

```

# # # Inversa de Moore Penrose
N = pinv(X)
aux = ones(k + 1)
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. Al igual que el método anterior, los cálculos parecían prometedores hasta llegar al polinomio de grado 10. Por lo tanto, se continuó indagando más en los paquetes de Julia hasta encontrar *Polynomials*.

5.4.4. *Polynomials*

Polynomials es un paquete que proporciona aritmética básica, integración, diferenciación, evaluación y hallar raíces para polinomios univariados JuliaMath (2021). Para poder usar el paquete primero hay que instalarlo usando las ya mencionadas instrucciones 2.4.5.

El paquete *Polynomials* tiene una 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 variables como entrada. Las primeras dos entradas son las correspondientes a x y y de los datos a utilizar (en este caso, los datos `filip`). La tercera entrada, `deg`, corresponde al grado que busco sea el polinomio, en este caso, grado 10).

A diferencia de los otros método que se utilizaron para este problema, la función `fit` usa el método Gauss-Newton para resolver sistemas de ecuaciones no lineales. Sin embargo, en este caso el problema es lineal. Esta fue la principal razón por la que este paquete no fue considerado al principio para resolver el problema.

La segunda razón es que a diferencia de la función `glm` del paquete con el mismo nombre, la función `fit` solamente aporta los coeficientes del ajuste del polinomio. Es decir, no da como resultado el error estandar, ni el valor p de la estimación.

El código en Julia es sumamente sencillo:

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

Si se quisiera ampliar el análisis y observar, por ejemplo, el valor p de algún predictor se tendría que buscar otra manera de obtenerlo.

A pesar de que se podría pensar que la función deja mucho que desear, 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.

5.5. Evaluación de los métodos

Una cuestión válida es preguntarse si tal vez lo que está mal es la implementación de los algoritmos y, debido a esto, no solucionan el problema de manera correcta. Por tanto, para probar que los métodos estén programados de la manera correcta fueron sometidos a una serie de pruebas.

La primera prueba consistió en que, usando los datos `filip`, cada método ajustaba un polinomio de grado k de $k = 1, 2, \dots, 10$. Al final, para cada polinomio de grado k se tenían cuatro resultados de ajuste (uno por cada método).

La segunda prueba consistió en comparar los resultados con R y Python. En ambos lenguajes se usaron los mismos datos `filip` y se calcularon todos los polinomios de grado k de $k = 1, 2, \dots, 10$.

En R se utilizó la función `lm(formula, data)` explicada a mayor detalle en la sección 4.1. Es importante mencionar que este problema ya ha sido abordado por otros usuarios y resuelto por Brian Ripley. Ripley es un matemático británico que ha escrito muchos libros sobre programación y ha sido galardonado en múltiples ocasiones por sus aportaciones a la estadística. Sin duda, uno de sus mayores logros es la constante e importante aportación al desarrollo de R. Por lo tanto, el código que se utilizó para resolver este problema es el mismo que Ripley hizo público.

Por otro lado, en Python se usó la función `polyfit` de NumPy explicado con más profundidad en la sección 3.2.1.

Finalmente, la tercera prueba fue medir el tiempo que tomaba a ambos lenguajes ejecutar sus respectivas funciones con los parámetros especificados. En R código es el siguiente

```
# Para polinomio de grado = 1
```

```

start <- Sys.time()
lm_1 <- lm(y ~ x, data = data, x = TRUE)
end <- Sys.time()

`resultados_grado_1`$R <- lm_1$coefficients
row.names(`resultados_grado_1`) <- c("b0", "b1")
X_1 <- lm_1$x

time_vec <- c(end - start)

# 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)

  # Ejecutamos el modelo
  start <- Sys.time()
  lm.plus <- lm(form, data = data, x = TRUE)
  end <- Sys.time()
  time <- end - start
  time_vec <- c(time_vec, time)

  # Guardamos el df correspondiente a un auxiliar
  resultados_aux <- get(paste("resultados_grado_", i))

```

```

# para unirle los coeficientes
resultados_aux$R <- lm.plus$coefficients

nombres <- c("b0")
# Para el nombre de los renglones
for (k in 1:i){
  nombres <- c(nombres, paste0("b", k))
}
row.names(resultados_aux) <- nombres

#Finalmente, hago el df final
assign(paste("resultados_grado_", i), resultados_aux)

assign(paste("X_", i), lm.plus$x)

```

En Python se definió una función que calculara los coeficientes β , guardara los resultados en un dataframe y calculara el tiempo de ejecución.

```

def polynomial_fit(grado_pol):
    start_time = time.time()
    # Ojo que regresa el coeficiente de mayor potencia primero
    python_fit = np.polyfit(x, y, deg = grado_pol)

    # Lo movemos solo para que esté en el
    # mismo orden que los demás métodos
    python_fit = np.flipud(python_fit)

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

```

```

    # 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 tiempo

# Hacemos un df vacio para guardar los tiempos
column_names = ['Grado', 'Tiempos']
tiempo_df = pd.DataFrame(columns = column_names)

# Calculamos todos los ajustes
for grado in range(1, 11):
    time_grado = polynomial_fit(grado)
    time_grado = {'Grado': grado, 'Tiempos': time_grado}
    tiempo_df = tiempo_df.append(time_grado, ignore_index = True)

```

No se mostraran todas las tablas con los resultados de todos los ajustes ya que es innecesario. Las tablas que sí se muestran son las que se considera tienen los resultados más relevantes.

Todos los métodos obtienen los resultados correctos en los ajustes de los polinomios de grado uno al quinto. La tabla 5.2 es evidencia de ello.

Los problemas comienzan cuando se calcula el polinomio de grado 6. El primer método utilizado, GLM, comienza a fallar como se puede

	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

observar en la tabla 5.3. Esto es de especial interés ya que, en teoría, este paquete está hecho para calcular el ajuste 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	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, todos los métodos arrojan resultados correctos hasta el polinomio de grado 9. Cuando se busca calcular el polinomio de grado 10, solamente las columnas Polynomials, R y Python muestran los resultados correctos.

Finalmente, se puede ver la tabla 5.5 que corresponde a los tiempos que le tomo a cada método hacer los cálculos. Cada columna corresponde

	GLM	QRvEcon	MoorePenrose	Polynomials	R	Python
b0	0.000000e+00	9.013426e+00	8.443046e+00	-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
b2	0.000000e+00	-5.767606e+00	-5.350763e+00	-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
b4	0.000000e+00	-6.703657e-01	-4.064616e-01	-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
b6	3.686442e-03	1.055234e-01	1.197715e-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
b8	3.758850e-04	2.277483e-03	2.403994e-03	-6.701912e-02	-6.701911e-02	-6.701911e-02
b9	3.281913e-05	1.262264e-04	1.316188e-04	-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

Figura 5.4. Resultados del polinomio grado 10

al método utilizado mientras que los reglones representan el grado del polinomio.

De los métodos programados en Julia, el más rápido es el hecho con el paquete `Polynomials`. `MoorePenrose` y `QRvEcon` no tardan mucho más, pero `GLM` es el que más tiempo toma. Aunque un tercio de segundo no sea mucho tiempo es mucho más del que le toma a los otros métodos. Como era de esperarse, los procedimientos hechos en `R` y `Python` toman muy poco tiempo.

Estos resultados dan pie a preguntarse la razón por la que la mitad de los métodos falla justo al hacer el cálculo del polinomio de grado 10, más no de los anteriores. En la siguiente sección se indaga más en este tema.

	GLM	QRvEcon	MoorePenrose	Polynomials	R	Python
k_1	0.3796052	0.0000533	0.0000453	4.56e-05	0.0484938622 secs	0.068188
k_2	0.3091849	0.0000520	0.0048597	4.00e-05	0.0019960403 secs	0.001999
k_3	0.3122847	0.0000536	0.0000745	4.31e-05	0.0013589859 secs	0.000544
k_4	0.3006190	0.0000850	0.0000556	6.36e-05	0.0010089874 secs	0.000000
k_5	0.3340590	0.0000590	0.0000651	5.47e-05	0.0020928383 secs	0.000000
k_6	0.3071109	0.0000665	0.0000736	6.48e-05	0.0009071827 secs	0.001147
k_7	0.3073423	0.0000606	0.0000677	5.98e-05	0.0009999275 secs	0.000000
k_8	0.3023170	0.0000695	0.0000910	8.88e-05	0.0019991398 secs	0.002000
k_9	0.3044160	0.0000743	0.0000749	7.81e-05	0.0017678738 secs	0.000000
k_10	0.2995928	0.0019696	0.0000935	8.97e-05	0.0021109581 secs	0.000000

Figura 5.5. Tiempos de ejecución para cada método

5.6. Número de condición y precisión de la solución

Como se vio en la sección 5.5, no queda duda de que los métodos sí están bien programados. Dejando de lado las funciones programadas en los paquetes de Julia, el método de factorización QR y descomposición de valores singulares arrojaron buenos resultados hasta los polinomios de grado 9. Esto puede dar pie a pensar que, en realidad, los datos en sí son muy susceptibles a cambios.

En otras palabras, cualquier cambio en la matriz X o en el vector y resulta en un ajuste de los coeficientes β poco preciso. Esta característica se conoce como que los datos tienen impurezas. 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 y tiene impurezas mientras que la matriz X es exacta.

2. La matriz X tiene impurezas mientras que el vector y es exacto.
3. Ambos, el vector y y la matriz X tiene impurezas.

En este caso, el enfoque es el tercer caso ya que no hay razón para pensar que solamente una columna de los datos originales tiene impurezas mientras que la otra no.

El número de condición es un valor que ayuda a determinar la sensibilidad de un sistema lineal.

Definición 8. El número $\|A\| \|A^{-1}\|$ se llama el número de condición de A y se denota $Cond(A)$ (Datta, 2010, p. 62).

Además, el número de condición da una referencia en que tan grandes son los cambios en un sistema. El siguiente teorema de (Datta, 2010, p. 65) es un ejemplo de esto.

Teorema 5.4. Supongamos que queremos resolver el sistema $Ax = b$. Supongamos que A es no singular, $b \neq 0$, y $\|\Delta A\| < \frac{1}{\|A^{-1}\|}$. Entonces

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

El teorema anterior explica que los cambios en la solución x son menor 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).

Asimismo, el número de condición tiene varias propiedades pero la relevante para este ejercicio es la siguiente:

$$\text{Cond}(A) = \frac{\sigma_{\max}}{\sigma_{\min}} \quad (5.4)$$

donde σ_{\max} y σ_{\min} son, respectivamente, el valor singular más grande y más pequeño de A .

Antes de calcular el número de condición de la matriz X del problema 5.1 es necesario ver una última definición.

Definición 9. *El sistema $Ax = b$ está mal condicionado si el $\text{Cond}(A)$ es grande (por ejemplo, $10^5, 10^8, 10^{10}$, etc). En otro caso, está bien condicionado (Datta, 2010, p. 68).*

Ahora bien, es momento de el número de condición. Este calculo se hizo en Julia y en R a manera de verificación. En ambos se uso la función que ya viene programada en cada lenguaje y la fórmula 5.4. En Julia, el código es

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

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

Los resultados son $\text{numCond}_1 = 1.7679692504686805e15$ y $\text{numCond}_2 = 1.7679692504686795e15$. Por otro lado, en R el código es

```
# con función 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, en ambos lenguajes cualquier método confirma que el número de condición de la matriz X de 5.1 es bastante grande. Por lo tanto, por la definición 9 se puede decir que el problema está mal condicionado. Esto podría causar muchas preguntas al lector, incluyendo si hubiera sido mejor utilizar otros datos o ajustar un polinomio de grado menor.

Es importante recordar dos cosas. La primera es que los datos vienen del Instituto Nacional de Standards y Tecnología (NIST) lo cual los hace diseñados específicamente para dar problemas. Consecuentemente, el segundo punto es recordar que esta sección de la tesis evalúa la precisión numérica de los lenguajes. Por lo tanto, no debe ser una sorpresa que el problema está mal condicionado y algunos métodos fallen. Al contrario, los resultados muestran cuales procedimientos son numéricamente más precisos y rápidos.

5.7. Opinión de la autora

Este ejercicio fue el primero que hice para la tesis y fue el más demandante mentalmente. El reto para mí, como autora, fue buscar cuatro formas diferentes de resolver un problema que, por momentos, pareció imposible.

Como usuaria de Julia quedo muy insatisfecha con los resultados, especialmente con los paquetes `GLM` y `Polynomials`. Probablemente hay maneras numéricamente mejores para programar la descomposición QR de una matriz y el cálculo de la pseudo inversa de Moore Penrose.

Sin embargo, los paquetes deben ser una herramienta para evitar hacer los cálculos que vienen directo del álgebra. El paquete `GLM` me decepcionó desde un inicio. Debió haber sido el primero en funcionar y por el contrario, fue el primero que falló.

Por otro lado, aunque el paquete `Polynomials` da la respuesta correcta en un tiempo muy corto, solamente da los resultados de los coeficientes. Este paquete se enfoca en todo lo relacionado con polinomios por lo que uno no debería esperar que haga un ajuste muy completo. Sin embargo, es el único que logró el resultado correcto.

R y Python no decepcionan ni sorprenden. Ambos son lenguajes que llevan mucho más tiempo siendo desarrollados por lo que la verdadera sorpresa sería que no funcionaran. Aun así, en el caso de R hay que saber tratar las variables de manera especial mientras que Python da los resultados muy fácil y con pocas líneas de código.

En conclusión, este ejercicio fue retador para mí y para Julia. En el futuro, si tuviera la opción de decidir en que lenguaje hacer el ajuste de un modelo lineal sin dudas elegiría a Python.

Capítulo 6

Regresión Lineal Múltiple

Vale: De verdad que no sé que título ponerle

6.1. El problema

En esta sección se explica como hacer una regresión lineal múltiple usando Julia. El modelo de regresión lineal clásico definido en (Gelman et al., 2021, p. 146) es

$$y_i = \beta_0 + \beta_1 X_{i1} + \cdots + \beta_k X_{ik} + \epsilon_i, \text{ para } i = 1, \dots, n \quad (6.1)$$

donde los errores son independientes y siguen una distribución normal con media 0 y desviación estándar σ . En este caso, y_i se refiere al nivel i -ésimo del regresor; x_{ij} es el j -ésimo regresor al i -ésimo nivel; y β_j es el coeficiente del j -ésimo regresor.

De manera matricial, la ecuación 6.1 se puede escribir como

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

donde X es una matriz de $n \times k$ donde su i -ésimo renglón es X_i .

Vale: Esto es de las notas del profesor Barrios

6.2. Los datos

Tomando en cuenta que los modelos estadísticos se utilizan como un reflejo de la realidad se tomo la decisión de usar los datos del Censo de Población y Vivienda (CENSO) 2020. Los datos son publicados por el Instituto Nacional de Estadística y Geografía (INEGI) y se pueden encontrar en la página <https://www.inegi.org.mx/programas/ccpv/2020/default.html>.

El propósito del Censo 2020 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 ??.

Los resultados del Censo que se utilizan vienen de las respuestas al llamado 'Cuestionario Ampliado' cuyas 103 preguntas resultan en alrededor de 200 variables de estudio. Asimismo, el Censo fue aplicado a 4 millones de viviendas a lo largo de la República Mexicana que resultó en la recaudación de información de más de 15 millones de personas.

En este caso, se eligió un tema que se considera, es de interés para todos los adultos: los ingresos. Más específicamente, se busca usar la RLM para ver como afectan diferentes variables al ingreso de cada persona. Usualmente, es trabajo del estadístico construir un modelo desde cero usando una combinación de lógica, referencias y experiencia. En este caso, el modelo final que se ajustó es

Vale: Se ve horrible, ayuda

Vale:

Arreglar
esta
referencia

$$\text{ingresos} \sim \text{horas}_{\text{trabajadas}} + \text{sexo} + \text{edad} + \text{escolaridad} + \text{entidad}_{\text{residencia}} + \\ \text{posicion}_{\text{laboral}} + \text{alfabetismo} + \text{aguinaldo} + \text{vacaciones} + \text{servicio}_{\text{medico}} \\ (6.2)$$

6.3. Planteamiento del problema

El CENSO no es solo un cuestionario. Es el conjunto de una mezcla de ellos que se le aplican a distintas personas en México dependiendo de factores como su edad, situación laboral y educación. Por esta razón, antes de comenzar a ajustar los datos, se debe filtrar la información.

Esto puede sonar sencillo, pero en realidad es el resultado de entender la estructura de las encuestas y como se relacionan entre ellas. Para hacerlo es necesario descargar el diccionario del diccionario ampliado que se encuentra en <https://www.inegi.org.mx/programas/ccpv/2020/default.html#Microdatos> dentro del apartado Documentación de la base de datos.

La información que da el diccionario son códigos, nomenclaturas y significados cuya función es entender como se paso de tener respuestas en hojas de papel a tenerlas en una base de datos.

La información de los resultados del Censo está dividida en tres partes: Viviendas, Personas y Migrantes. En esta ocasión, se utilizó solamente la base de datos correspondientes a Personas ya que contiene toda la información necesaria.

La cantidad de datos con la que se está trabajando es enorme por lo que el primer paso es seleccionar las columnas necesarias y desechar el resto. Después, 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, son 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 obtuvieron a las personas que especificaron horas trabajadas e ingreso ganado.
4. Cada regresor viene 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 todos los filtros anterior la cantidad de datos con los que se trabaja se reduce considerablemente. Los datos pasan de ser cerca de 15 millones a poco más de 3.5 millones. Sin embargo, todo el proceso dura alrededor de 20 minutos para ejecutarse por lo que se recomienda guardar la base de datos filtrada para evitar hacer estos comandos constantemente. En Julia, el código queda de la siguiente manera.

```
using CSV, DataFrames, StatsBase, GLM, Random, CategoricalArrays
```

```
# Equivalente a set.seed
```

```
Random.seed!(99)
```

```
# Leer la base de datos (toma alrededor de 4 minutos en cargar)
```

```
personas = CSV.read("Personas00.csv", DataFrame)
```

```

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

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

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

# # # FILTRO 2
personas_filt = subset(personas_filt,
                        :ENT_PAIS TRAB => ByRow(<(33)), skipmissing = true)

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

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

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

# # # FILTRO 4

```

```

function diferente_a(dataframe, columna, condicion)
    dataframe = subset(dataframe, columna => ByRow(!(condicion)),
        skipmissing = true)
    return dataframe
end

categorias_9 = ["SEXO", "AGUINALDO", "VACACIONES", "SERVICIO_MEDICO",
    "UTILIDADES", "INCAP_SUELDO", "SAR_AFORE", "CREDITO_VIVIENDA",
    "ALFABET", "SITTRA"]

categorias_99 = ["NIVACAD"]

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

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

# Finalmente, guardo el nuevo dataframe
CSV.write("personas_filtradas.csv", personas_filt)

```

6.4. Regresiones

Vale: No estoy segura de como ponerle de nombre a esta seccion

Lo primero que debemos notar es que la mayoría de las variables que use en mi regresión son categóricas. En este caso, Julia identifica la mayoría de la columnas como tipo `Int64` en vez de `Categorical`. Por

tanto, hay que transformar los datos que lo necesiten.

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

# Vector con todas las categorias
vector_categorias = ["SEXO", "AGUINALDO", "VACACIONES", "SERVICIO_MEDIO",
                     "ALFABET", "NIVACAD", "ENT_PAIS_TRAB", "ENT", "SITTRA"]

transform!(data, names(data, vector_categorias) .=> categorical, rename=)
```

Es muy importante no saltarse este paso ya que de lo contrario, la regresión no estará bien hecha.

Teniendo siempre en mente que el objetivo de esta tesis es probar los límites de Julia, tome la ecuación 6.2 y le quité de algunas variables. Se podría pensar como que tomé diferentes subconjuntos de variables y los puse en la regresión para ver si cambiaba la precisión del ajuste. La variable de respuesta siempre es la misma en todos los *sub-ajustes*.

Para el primer *sub-ajuste* tome las primeras 5 variables de la ecuación 6.2 y lo llamé fit5 (ya que tiene 5 regresores). Es decir, la ecuación fit5 es

$$\text{ingresos} \sim \text{horas}_{\text{trabajadas}} + \text{sexo} + \text{edad} + \text{escolaridad} + \text{entidad}_{\text{residencia}}$$

El segundo *sub-ajuste* llamado fit6 tiene los mismos 5 regresores que fit5 más uno extra, la posición laboral. Por eso, la ecuación fit6 es

$$\text{ingresos} \sim \text{horas}_{\text{trabajadas}} + \text{sexo} + \text{edad} + \text{escolaridad} + \text{entidad}_{\text{residencia}} + \text{posicion}$$

Es importante notar que las ecuación `fit5` y `fit6` siguen el mismo orden que 6.2. Esto no es una coincidencia. El orden de los regresores en la ecuación 6.2 está pensado precisamente para que cada variable sumada se agregue al conjunto de variables anterior y creé una nueva ecuación `fit`.

6.4.1. Observaciones

Saemos que no es lo mismo hacer un ajuste con 5 observaciones a hacer uno con 5 millones de observaciones. Hay diferencias en tiempo, precisión y credibilidad. Por tanto, continuando con el objetivo principal, cada una de las ecuaciones `fit` la probe con 500, 5 mil, 50 mil, 500 mil y 2.5 millones de observaciones. Es decir, use cada una de las ecuaciones `fit` para ajustar un modelo con las 5 cantidades de observaciones antes mencionadas. Es importante señalar que las observaciones se seleccionan al azar usando el comando `sample`. Una vez seleccionadas, guardaba el dataframe generado para usar exactamente los mismos datos en R y Python.

Finalmente, para hacer que todo funcione más rápido, hice una función para cada `fit`. Las funciones para cada `fit` son casi iguales a excepción de la fórmula que se necesita para el ajuste y el nombre con el que guardo los resultados. Fue importante para mí hacer una función para cada ecuación `fit` ya que creo esencial que la fórmula que se usa en el ajuste esté puesta de manera explícita.

Ya que las funciones y su aplicación son muy similares, solamente mostraré las funciones para `fit5` y para `fit10` como ejemplo.

El código para `fit5` es el siguiente.

```
### FIT BASE ###  
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)

# Hacemos el fit
sample_fit = lm(@formula(INGTRMEN ~ HORTRA + SEXO + EDAD + NIVACAD +
aux = "res_"
nombre_completo = aux*nombre_completo
CSV.write(nombre_completo, coeftable(sample_fit))
end

# Aplicamos la función para las observaciones
fit5(500, "500")
fit5(5000, "5mil")
fit5(50000, "50mil")
fit5(500000, "500mil")
fit5(2500000, "2500mil")

```

El código para fit10 es el siguiente.

```

### FIT 10###
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)

# Hacemos el fit
sample_fit = lm(@formula(INGTRMEN ~ HORTRA + SEXO + EDAD + NIVACAD +
                        + SITTRA + ALFABET + AGUINALDO + VACACIONES + SERVICIO_MEDIO))

aux = "res_"
nombre_completo = aux*nombre_completo
CSV.write(nombre_completo, coeftable(sample_fit))
end

# Fit 10: Fit 5 + SITTRA + ALFABET + AGUINALDO + VACACIONES + SERVICIO_MEDIO
fit10(500, "500")
fit10(5000, "5mil")
fit10(50000, "50mil")
fit10(500000, "500mil")
fit10(2500000, "2500mil")

```

6.5. Resultados

Falta poner las tablas de resultados, pero se ven muy feas.

Capítulo 7

MDopt

Una de las razones por las que decidí estudiar matemáticas aplicadas es justamente por la parte de 'aplicadas'. Cuando en los cursos de estadística comencé a aprender sobre modelos que pueden describir información real, mi sorpresa y emoción fueron auténticas. Sin embargo, en esas clases también aprendí que hay muchos modelos posibles para un conjunto de datos y no hay una sola manera de elegir el *mejor* modelo. Por lo tanto, para darle continuidad al tema de modelos lineales decidí abordar el problema de discriminación de modelos usando el método con el criterio MD, *Model Discrimination* propuesto por Box y Hill.

En primer lugar, hay que discriminar entre las variables o factores que realmente afectan la variable de respuesta de las que no. Para esto se puede utilizar el método bayesiano descrito por Ana Patricia Vela [Noyola \(2022\)](#). Hay ocasiones donde utilizando este método es muy fácil determinar si un factor afecta o no la variable de respuesta. Sin embargo, hay ocasiones donde los resultados son ambiguos y pareciera que hay varios modelos que describen los datos. Por lo tanto, la estrategia que

se usa es agregar ensayos adicionales específicos que, una vez agregados a los ensayos originales, darán una idea más clara sobre cuál es el mejor modelo.

Uno de estos métodos es el que utiliza el criterio MD, de *Model Discrimination*. La idea de este criterio es elegir ensayos que permitan la máxima discriminación posible entre los modelos probables Meyer et al. (1996).

En primer lugar, expongo un resumen del método completo descrito por Meyer et al. (1996) para la discriminación de modelos. Posteriormente, expongo la explicación detallada y matemática del criterio MD y el algoritmo de intercambio. Finalmente, hago la comparación entre los tres lenguajes con dos ejemplos.

7.1. Metodología completa

El proceso de identificación de los factores activos de un modelo no es sencillo ya que incluye pasos que usan estadística bayesiana. La explicación detallada está fuera del alcance de este trabajo, pero se puede encontrar en Meyer et al. (1996). Sin embargo, daré un breve resumen para tener el contexto donde se usan el criterio MD y el algoritmo de intercambio. Los detalles matemáticos están explicados en las siguientes secciones.

Supongamos que voy a realizar un experimento. Para esto, tengo que hacer un diseño con los factores que voy a incluir, sus niveles y los ensayos que voy a hacer. Como experta en el experimento debo tener una idea de cuáles modelos son los más probables a describir el experimento. Este conocimiento a priori se le denomina $P(M_i)$.

Después, ya que realicé el experimento tengo los niveles que use para cada factor y la respuesta obtenida Y . Con esta nueva información

puedo calcular la probabilidad de que cada modelo M_i sea el correcto. Es decir, calculo $P(M_i|Y)$. Además, puedo calcular P_j , la probabilidad de que el factor j esté activo. Las probabilidades más altas me indican cuáles son los posibles factores activos.

Si hay una clara diferencia entre las probabilidades P_j de los factores entonces puedo separar los activos de los no activos. En cambio, si no hay una clara diferencia entre probabilidades P_j ? explica que se deben hacer más ensayos para discriminar entre los posibles modelos.

En la elección de esos ensayos es donde entra el criterio MD ya que es un número que indica cuales ensayos vale la pena repetir para obtener la mayor diferencia en el vector respuesta Y . Entre mayor sea la diferencia en Y , más fácil se puede discriminar entre modelos. El algoritmo de intercambio se usa para calcular el criterio MD para todas las posibles combinaciones de ensayos de forma efectiva. El diagrama 7.1 muestra la explicación anterior.

7.2. Criterio MD

Lo siguiente es la explicación matemática que viene en el artículo de Meyer et al. (1996).

Supongamos que tenemos un experimento factorial fraccionario con k factores. Sea Y el vector de respuestas de tamaño $n \times 1$. El modelo que mejor describe a Y depende de cuales factores están activos además de que el análisis debe considerar 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, asumimos un modelo lineal normal usual $Y \sim N(X_i\beta_i, \sigma^2 I)$. La matriz X_i es la matriz de regresión para M_i e incluye los efectos principales para

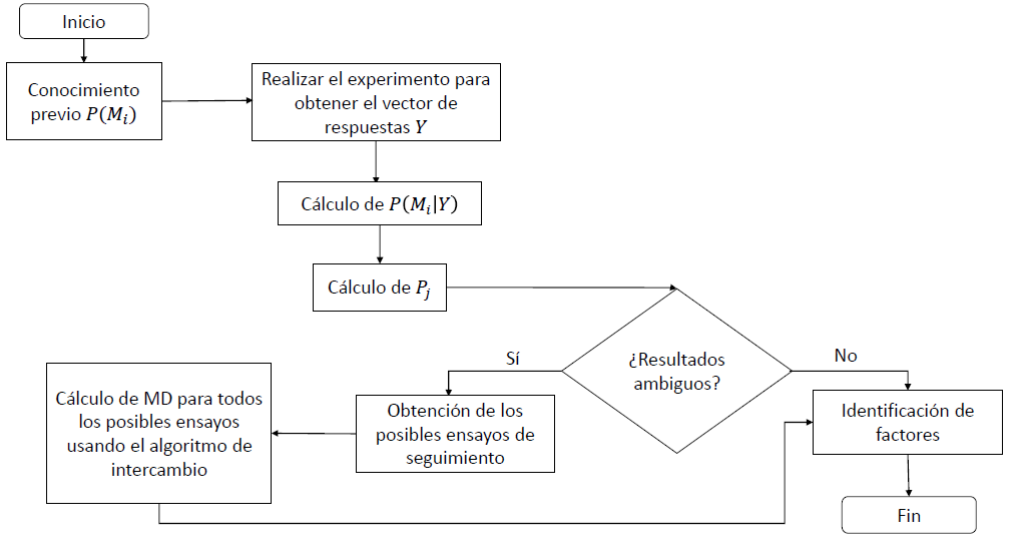


Figura 7.1. Diagrama de metodología descrita por Meyer et al. (1996)

cada factor activo y sus interacciones hasta cualquier orden deseado. Sea t_i el número de efectos (sin incluir el término constante) en M_i . Denotemos a M_0 como el modelo sin factores activos.

Ahora asignaremos distribuciones a priori no informativas al término constante β_0 y el error la desviación estándar σ que serán comunes para todos los modelos. Entonces, $p(\beta_0, \sigma) \propto \frac{1}{\sigma}$. El resto de coeficientes β_i tienen distribuciones normales a priori con media 0 y desviación estándar $\gamma\sigma$.

Finalmente, hay que agregar probabilidades a priori a cada uno de los modelos posibles. La regla de Pareto, o *sparsity of effects principle* dice que cuando hay varias variables, el sistema es más probable que esté dominado por los efectos principales e interacciones de orden bajo

Montgomery (2017). En otras palabras, buscamos pocos factores que sean los principales y que la combinación entre ellos sea de orden bajo. Por lo tanto, podemos asumir que existe una probabilidad π , $0 < \pi < 1$ que cualquier factor esté activo. Además, asumimos que la creencia a priori de que un factor esté activo es independiente de las creencias de los demás factores. Entonces, la probabilidad a priori del modelo M_i es $P(M_i) = \pi_i^f (1 - \pi)^{k-f_i}$.

Una vez observado el vector de datos Y , podemos actualizar las distribuciones de los parámetros para cada modelo y la probabilidad de que cada modelo sea válido. La probabilidad posterior de que M_i sea el modelo correcto es

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

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

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

y

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

Es importante notar que las probabilidades $P(M_i|\mathbf{Y})$ pueden ser sumadas sobre todos los modelos que incluyan al factor j para calcular la probabilidad posterior P_j de que el factor j está activo,

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

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

Si utilizara el análisis bayesiano, el experimento claramente sugeriría un modelo M_i cuando la probabilidad posterior $P(M_i|\mathbf{Y})$ es cercano a 1. Sin embargo, las conclusiones son ambiguas cuando hay varios modelos con probabilidades cercanas a 1.

El diseño MD propuesto por Box y Hill en 1967 [Box and Hill \(1967\)](#) tiene la siguiente forma. Sea \mathbf{Y}^* el vector de datos obtenidos de los ensayos adicionales y sea $P(\mathbf{Y}^*|M_i, \mathbf{Y})$ la densidad predictiva de \mathbf{Y}^* dados los datos iniciales \mathbf{Y} y el modelo M_i . Entonces,

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

Vale:
Aqui tengo
duda
porque
Meyer es
el que los
cita

Sea p_i la densidad predictiva para una nueva observación condicionada a las observaciones originales \mathbf{Y} and sea M_i el modelo correcto. Entonces, el diseño de criterio es

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

donde $I(p_i, p_j) = \int p_i \ln\left(\frac{p_i}{p_j}\right)$ es la información de Kullback-Leibler y mide la información media para discriminar a favor de M_i contra M_j cuando M_i es el verdadero modelo. Además, la proporción $\frac{p_i}{p_j}$ puede verse como la probabilidad en favor de M_i contra M_j dados los datos de los experimentos extras.

Entre mayor sea el valor de MD para un diseño, mejor ya que el diseño que maximice MD puede ser referido como el diseño *MD-óptimo*.

La intuición detrás de la fórmula del criterio MD puede ser más sencilla de entender si consideramos el ejemplo donde solo tenemos dos

modelos posibles, M_1 y M_2 . 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, buscamos un diseño que calcule una probabilidad alta a favor de M_1 si este es el modelo correcto; pero además que calcule lo mismo para M_2 si es el modelo correcto. En otras palabras, buscamos el valor de MD que señale el diseño correcto.

7.3. Algoritmo de intercambio

En su artículo, [Mitchell \(1974\)](#) explica un algoritmo llamado 'DETMAX' para la construcción de diseños experimentales 'D-óptimos'. Lo siguiente es un resumen de dicho artículo.

Consideremos el modelo lineal usual $y = X\beta + \epsilon$ donde y es un vector de observaciones de tamaño $n \times 1$, X es una matriz de $n \times k$ de constantes, β es el vector $k \times 1$ de coeficientes para ser estimados y ϵ es un vector de $n \times 1$ de variables aleatorias independientes e idénticamente distribuidas con una media 0 y una 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 depende en el modelo. Sea p el número de variables independientes y χ la región donde es factible realizar experimentos.

El estimador de mínimos cuadrados de β es $\hat{\beta} = (X'X)^{-1}X'y$, y la matriz de covarianza de $\hat{\beta}$ es $(X'X)^{-1}\sigma^2$. En cualquier punto $x \in \chi$, el valor estimado de la 'verdadera' respuesta es $\hat{y}(x) = f(x)'\hat{\beta}$. 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 $v(x) = f(x)'(X'X)^{-1}f(x)\sigma^2$. En este caso, σ^2 puede ser tomada como 1 sin pérdida de generalidad.

Uno de los diseños más populares para construir diseños óptimos es el de maximizar $|X'X|$ llamado diseño 'D-óptimos'. El propósito del

artículo de Mitchell es presentar un nuevo algoritmo llamado DETMAX para maximizar el determinante de la matriz $X'X$.

En primera instancia, el algoritmo fue creado para intercambiar puntos de diseño de la siguiente manera. Empezando con un diseño elegido al azar de n ensayos, el diseño original de n ensayos se puede mejorar

Vale: Se lee muy confuso, son puntos o ensayos?

Vale:
Agregar
que fue la
referencia
5

1. Sumando un ensayo número $n + 1$ elegido para que se alcance el incremento máximo posible de $|X'X|$. Después,
2. Quitando el ensayo en el diseño resultante que resulte en la menor disminución en $|X'X|$.

Estos dos pasos se llegan primero sumando 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.

Vale: Aquí quiero agregar un pequeño diagrama

Ahora bien, para tener mayor flexibilidad, este algoritmo básico fue modificado para permitir el reemplazamiento de más de un punto del diseño original en cada iteración. El requerimiento de que un diseño con $n + 1$ puntos sea regresado inmediatamente a un diseño con n puntos se relajo. Al algoritmo ahora se le permite hacer una 'excursión' donde se pueden construir diseños de varios tamaños que eventualmente regresan a un diseño de tamaño n .

Si no hay mejora en el determinante todos los diseños construidos en la excursión son eliminados y puestos en 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 punto 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 , quitar un punto si D no está en F y agregar un punto de lo contrario.
2. Si el número de puntos en D es menor que n , agregar un punto si D no está en F y quitar un punto de otra manera.

Para determinar si algún D está o no en F , solo se examina el determinante de $|X'X|$. A pesar de que esto no es un prueba de fuego (ya que dos diseños diferentes pueden tener el mismo determinante) parece ser una buena manera en probar equivalencias en poco tiempo.

Cada vez se vuelve más y más difícil tener un mejor diseño por lo que las excursiones se pueden alejar mucho de un nivel de n puntos. Para parar el algoritmo, Mitchell propone pone límites en el mínimo y máximo número de puntos permitidos en la construcción de un diseño durante una excursión los cuales recomienda poner estén en $n \pm 6$.

Mitchell adopta el enfoque de Dykstra en donde los puntos de diseño son seleccionados de una lista previamente especificada de candidatos. Esto trae facilidad de programación y el poder de excluir puntos que no son deseados o posibles.

Vale:
Agregar
referencia

Debido a la existencia de muchos diseños que son óptimos solo localmente, lo mejor es hacer varias intentos independientes en la solución. En cada intento, DETMAX empieza con un diseño completamente nuevo cuyos puntos son seleccionados aleatoriamente de una lista de candidatos. Él determina que diez intentos usualmente son suficientes para llegar al diseño óptimo.

7.4. Función MDopt

Noyola (2022) uso la información anterior para crear la función MDopt en la librería BsMD2. Yo utilicé esa función como referencia para crear funciones del mismo nombre en Julia y Python.

Esta función toma como entrada varios parámetros, pero los más relevantes son la matriz **X** y el número de interacciones entre factores llamado `max_int`. Si el diseño tiene interacciones entre dos factores, la función crea una matriz **Xfac** con ellas. Lo mismo sucede si el diseño tiene interacciones entre tres factores.

Después, la función hace el cálculo de $\Gamma_k, \beta_k, \delta_k$ con las fórmulas 7.2, 7.1 y 7.3 respectivamente. Posteriormente, se define otra función (dentro de MDopt) llamada MDr que calcula el número MD con la fórmula . Finalmente, MDopt usa la idea del algoritmo de intercambio para calcular el criterio MD para múltiples combinaciones de ensayos.

Vale: aquí falta

Es importante mencionar que la función MDopt escrita por Noyola devuelve el dataframe de todos los ensayos para los que calculo el valor de MD, la matriz **X**, la matriz de puntos candidatos, el vector de probabilidades para cada modelo considerado y las matrices con los factores activos para cada modelo.

En cambio, las funciones que yo hice en Julia y Python solamente devuelven el dataframe de todos los ensayos realizados con su valor MD ya que eso es lo que necesito para mostrar los resultados. El diagrama muestra lo anterior de manera visual.

Vale: falta

El pseudocódigo de la función MDr y el algoritmo de intercambio están en los apéndices de Noyola (2022).

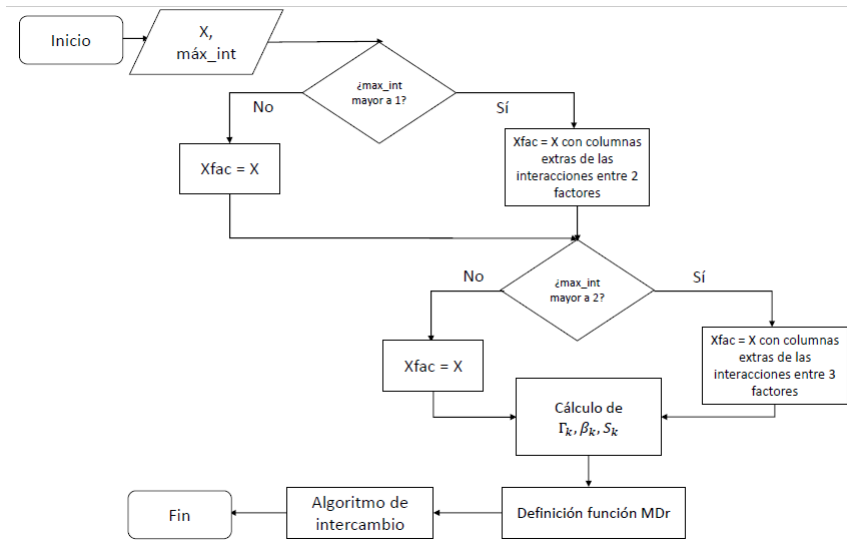


Figura 7.2. Diagrama de la función MDopt

7.5. Comparación entre lenguajes

Ya que la tesis es para probar la eficiencia y funcionalidad de Julia en comparación con R y Python use el mismo código en los tres lenguajes para ver cual de los tres hace los cálculos de MD de la manera más rápida y eficiente. En R utilice el paquete BsMD2 hecho por Ana Patricia Vela Noyola y el paquete BsMD elaborado por el profesor Ernesto Barrios. En Julia y Python utilice ese mismo código solo que adaptado a los diferentes lenguajes. Después, llamé a los códigos de Julia y Python en R para medir el tiempo que le toma a cada lenguaje ejecutar dos ejemplos distintos.

Es importante mencionar que fue más difícil traducir el código a Python ya que este lenguaje utiliza una enumeración diferente. Los objetos en Python comienzan a contarse desde el 0 mientras que en R

y Julia la numeración empieza en 1. Puede ser un poco confuso hacer la transición. Además, los resultados también tienen numeración diferente y, a pesar de que son exactamente iguales que en los otros lenguajes, hay que tener cuidado con su presentación para evitar confusiones.

No es extraño que en los paquetes, `JuliaCall` y `reticulate`, existan comandos que sirvan para efectuar las mismas tareas. Por lo tanto, la siguiente tabla es una lista de los comandos que utilicé en `JuliaCall` así como su simil en `reticulate`.

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

Para llamar Julia en R, utilice el paquete **JuliaCall**. Este paquete permite que Julia funcione dentro de R. Es decir, yo utilizo los objetos creados en R y los puedo trasladar a Julia para correr alguna función creada en Julia. El paquete es como un puente entre ambos lenguajes donde solamente hace la conexión más no los mezcla de ninguna otra forma.

En primer lugar, justo después de cargar la librería de **JuliaCall** es necesario usar el comando `julia_setup` y poner la dirección de la

carpeta donde está instalado Julia en tu computadora. Después, para cada ejemplo creo los objetos que necesito como entrada de la función. Posteriormente, utilizo el comando `julia_assign` para asignar los objetos creados en R a objetos nuevos en Julia. En caso de que la conversión de alguno de los objetos no sea la deseada, utilizo `julia_command` para hacer la conversión dentro de Julia. Además, debo tener la función que quiero utilizar en un documento con terminación `.jl` guardado en la carpeta de mi directorio de trabajo. Para agregar la función en R, utilizo el comando `julia_source` y dentro el nombre del documento. Finalmente, utilizo el comando `julia_eval` para correr la función que con los parámetros ya que cree.

Para llamar Python en R utilice el paquete `reticulate`. Este paquete funciona más como una extensión de R ya que puedes transitar fácilmente entre ambos lenguajes sin necesidad de muchos comandos. Mas bien, lo que se necesitan son prefijos como `.r` o `py$` para llamar los objetos de cada lenguaje.

Para utilizar la función que escribí en Python, lo primero que hice (después de llamar al paquete, claro está) es guardarla en un archivo con terminación `.py` y guardarlo en la carpeta del directorio de trabajo. Después, utilice el comando `source_python` para llamar el archivo. Con solamente llamar el archivo se cargan en R todas las funciones dentro de él. En este caso, yo solamente tenía una función pero en caso de tener varias, solo es necesario cargar el archivo una vez. Después, debo utilizar el comando `r_to_py` para convertir todos los objetos de R en objetos de Python. Una de las ventajas de este paquete es que crea el objeto de Python en el ambiente de R y si usas RStudio, puedes ver el objeto creado en la parte de **Environment** de tu pantalla. Para Julia esto no pasa lo cual puede llegar a ser confuso.

Posteriormente, si uno de los parámetros de la función es un entero

te recomiendo que también los conviertas en un objeto de Python ya que en ocasiones el paquete los convierte automáticamente en objetos de tipo `Float` cuando son enteros y la función puede fallar. Finalmente, puedes llamar a tu función de Python en R sin ningún comando adicional. Lo único que necesitas es usar el nombre de la función tal cual la usaste en Python y agregarle los parámetros que ya creaste.

7.6. Ejemplos y resultados

7.6.1. Ejemplo 1 - Proceso de moldeo por inyección

El primer ejemplo que utilice fue mencionado por [Meyer et al. \(1996\)](#) quien lo tomo de un artículo escrito por Box, Hunter y Hunter en 1978. El experimento consiste en estudiar los efectos de ocho factores en el encojimiento de un proceso de moldeo por inyección. El plan experimental fue un 2^{8-4} factorial fraccionado con generadores $I = ABDH = ACEH = BCFH = ABCG$. Los datos para este ejemplo están en la tabla [7.1](#).

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 7.1. Datos para el ejemplo 1

No es el objetivo de esta tesis explicar el análisis previo que se hace en este tipo de experimentos, pero sí es importante destacar que se calcula la probabilidad posterior de los modelos. En este análisis se ve que los posibles modelos son los que se muestran en la tabla ?.

Vale:
explicar
más esto

Modelo	Factores	Probabilidad 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 7.2. Modelos con la probabilidad posterior más alta para el ejemplo 1

Además, calculando las probabilidades posteriores P_j mencionadas en 7.4, los factores A , C , E , y H tienen una probabilidad posterior de 0.764 mientras que los demás factores tienen una probabilidad de 0. Por lo tanto, los factores A , C , E , y H son los que parecieran ser activos. Dado el análisis previo, el problema original con un diseño de 2^{8-4} paso a convertirse en un modelo con diseño 2^{4-1} por la reducción de factores. Con los ensayos que tenemos no es posible distinguir entre los cinco posibles modelos por lo que se necesitan ensayos adicionales para aclarar cuales son los factores activos.

La tabla 7.3 muestra las predicciones de las respuestas para todas las combinaciones de factores A , C , E , y H . El propósito de esta tesis no es indagar mucho en el cálculo de estas probabilidades , pero puedo

Vale: o sí?

Candidato	Ensayo	A	C	E	H	Y	1	2	3	4	
1	14, 16	-1	-1	-1	-1	21.9, 20.3	21.08	21.08	21.08	21.08	21.08
2	1, 3	-1	-1	1	1	14.0, 15.0	14.58	14.58	14.58	14.58	14.58
3	5, 7	-1	1	-1	1	27.6, 27.4	27.38	27.38	27.38	27.38	27.38
4	10, 12	-1	1	1	-1	17.1, 17.5	17.34	17.34	17.34	17.34	17.34
5	2, 4	1	-1	-1	1	16.8, 15.4	16.16	16.16	16.16	16.16	16.16
6	13, 15	1	-1	1	-1	15.9, 16.7	16.35	16.35	16.35	16.35	16.35
7	9, 11	1	1	-1	-1	22.3, 21.5	21.87	21.87	21.87	21.87	21.87
8	6, 8	1	1	1	1	24.0, 22.6	23.25	23.25	23.25	23.25	23.25
9		-1	-1	-1	1		21.08	14.58	27.38	16.16	16.16
10		-1	-1	1	-1		14.58	21.08	17.34	16.35	16.35
11		-1	1	-1	-1		27.38	17.34	21.08	21.87	21.87
12		-1	1	1	1		17.34	27.38	14.58	23.25	23.25
13		1	-1	-1	-1		16.16	16.35	21.87	21.08	21.08
14		1	-1	1	1		16.35	16.16	23.25	14.58	14.58
15		1	1	-1	1		21.87	23.25	16.16	27.38	27.38
16		1	1	1	-1		23.25	21.87	16.35	17.34	17.34

Tabla 7.3. Ejemplo 1, Colapsado en los factores A, C, E y H

Considero importante explicar a detalle la tabla 7.3. Los datos de primeros ocho candidatos son los mismos datos de la tabla 7.1, pero mostrando únicamente los factores A , C , E , y H . No es una sorpresa que la respuesta Y sea similar por candidato ya que los factores que creemos están activos se mantuvieron en los mismos niveles.

Por otro lado, los siguiente ocho candidatos son todas las posibles combinaciones para los cuatro modelos con mayor probabilidad. Tomemos, por ejemplo, el modelo que dice que los factores activos son A , C , E . Si ignoramos la columna H de la tabla 7.3, los 8 ensayos muestran todas las posibles combinaciones que puede tener un

experimento con estos tres factores. Lo mismo pasa con los otros tres modelos con tres factores cada uno. Para el modelo final con cuatro factores activos, la tabla completa es todas las posibles combinaciones.

Además, es importante notar que para los primeros ocho candidatos la respuesta de los modelos es similar. Mientras, para los siguientes ocho ésta varía mucho más. La similitud en las respuestas de los primeros ocho candidatos refuerza la idea de que es muy complicado distinguir entre los cinco posibles modelos. La diferencia en los siguientes ocho candidatos ayudará a que ahora sí sea posible distinguir entre los posibles modelos.

Como se menciono anteriormente, no es posible realizar todos los ensayos posibles así que tendremos que elegir. En este caso, buscamos generar un diseño de seguimiento de cuatro ensayos usando el criterio MD. Hay 3,876 posibles diseños que podemos generar de los 16 candidatos de la tabla 7.3. Se podría generar un código que calcule el valor MD para cada uno de esos diseños. Sin embargo, es mejor utilizar el algoritmo de intercambio ya que genera un diseño al azar de puntos candidatos y después los modifica hasta que un criterio de convergencia se satisface.

El código para el cálculo del criterio MD y el algoritmo de intercambio para R, Julia y Python es el siguiente.

```
## R paquete Patricia
library(BsMD2)
setwd("~/ITAM/Tesis/Julia con R/Code/MD-optimality")

# matriz de diseño 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,-1),
nrow=16,dimnames=list(1:16,c("blk", "f1", "f2", "f3", "f4")))
)

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)
Sys.time() - t

# # # R paquete original
library(BsMD)

s2 <- c(0.5815, 0.5815, 0.5815, 0.5815, 0.4412)

t_R0 <- Sys.time()
e3_R0 <- BsMD::MD(X = X, y = y, nFac = 4, nBlk = 1,

```

```

mInt = 3, g = 2, nMod =
p = p_mod, s2 = s2,
nf = c(3, 3, 3, 3, 4),
facs = fac_mod, nFDes = 4, Xcand = Xcand, mIter = 20,
nStart = 25, top = 10)
Sys.time() - t_R0

# # # Julia con R
library(JuliaCall)
julia_setup(JULIA_HOME = "C:/Users/Valeria/AppData/Local/Program

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)")

t_J <- 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)")
Sys.time() - t_J

```

```

# # # 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)

t_P <- 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)

```

`Sys.time() - t_P`

Es importante notar que para R utilice el paquete elaborado por Patricia así como el paquete original **BsMD** elaborado por el profesor Ernesto Barrios. En los tres lenguajes los resultados fueron los mismos y se muestran en la tabla 7.4

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 7.4. Resultados para el ejemplo 1

En cuanto a tiempo, al paquete de Patricia le tomo 5.618191 segundos en hacer el cálculo; al paquete **BsMD** del profesor Barrios le tomó 0.03919315; Julia se tardó 34.85702 segundos; y a Python 51.05128 segundos.

Es importante mencionar que en el caso de Julia el tiempo va disminuyendo las ocasiones consecutivas que corres el código inclusive cambiando los parámetros de la función. La segunda ocasión solo le tomo 9 segundos y la tercera 5 segundos. Esto es útil cuando se esté corrigiendo la función o simplemente se quiera correr varias veces para distintos diseños.

Vale:
Incluir
aquí que
falta el
tiempo del
setup

7.6.2. Ejemplo 2

En el ejemplo anterior, no había forma de replicar el experimento con los ensayos adicionales en las mismas condiciones en las que fue efectuado. El objetivo de este ejemplo, que también es tomado de Meyer [Meyer et al. \(1996\)](#), es evaluar la efectividad del criterio MD para generar datos que puedan identificar cuales son los factores activos.

Meyer utiliza datos de un experimento de reactor hecho por Box et al en 1978. Ese experimento es de tipo 2^5 factorial lo que significa que hay 32 ensayos del mismo. Este ejemplo Meyer busca probar la efectividad de su diseño tomando solamente 8 de los 32 ensayos originales y encontrando de manera correcta los factores que están activos. La idea es tener un diseño de seguimiento que pueda tomar los ensayos adicionales necesarios del experimento original. Los ocho ensayos elegidos están en el tabla [7.5](#).

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 7.5. Datos para el ejemplo 2

En análisis bayesiano previo para los datos de la figura [7.5](#) no muestra de manera clara que algún factor esté activo. Por lo tanto, un diseño de cuatro ensayos fue creado para encontrar el mejor

subconjunto de 4 de los 32 posibles candidatos de cinco factores en dos niveles.

El código para generar los resultados en los tres lenguajes es el siguiente.

```
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]))
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)
Sys.time() - t

library(JuliaCall)
julia_setup(JULIA_HOME = "C:/Users/Valeria/AppData/
```


Local/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 = .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)")
```

```
t_J <- 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)")
Sys.time() - t_J

```

```

# # # 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)

t_P <- 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)
Sys.time() - t_P

```

Igual que en el ejemplo anterior, para **R** utilice ambos paquetes **BsMD** y **BsMD2**. En los tres lenguajes los resultados fueron exactamente los mismo y se muestran en la tabla 7.6.

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 7.6. Resultados para el ejemplo 2

En cuanto a tiempo, al paquete **BsMD2** le tomo 9.573741 minutos obtener los resultados; el paquete **BsMD** hizo el cálculo en 0.4537661 segundos; Julia se tardó 50.54355 segundos; y, finalmente a Python le tomó 11.058 minutos.

Es importante mencionar que este ejemplo es el más pesado computacionalmente que voy a mostrar en esta tesis. No es sorpresa que el paquete **BsMD** sea el más rápido, ya que utiliza Fortran para hacer sus cálculos. Lo que más sorprende es que Julia sea el lenguaje que quede en segundo lugar con semejante ventaja. En este caso, Julia es mínimo 10 veces más rápido que sus competidores. Incluso usando Python desde otra plataforma Julia es más rápido. Por lo tanto, este ejemplo termina demostrando la capacidad computacional que tiene Julia para este tipo de algoritmos.

Capítulo 8

Conclusiones

El objetivo de este trabajo fue hacer una comparación entre Julia, R y Python. En primer lugar y dado que Julia es un lenguaje todavía muy desconocido se dió una breve introducción a los básicos de Julia y lo indispensable para entender esta tesis. Se asumió que el lector ya conoce R y Python por lo que se expusieron de manera más superficial.

La segunda parte de la tesis fueron los ejercicios que, se piensa, aportan conocimiento sobre el mejor uso de los tres lenguajes. En el primer ejercicio se tomaron datos de NIST para hacer el ajuste de un polinomio. Los datos son sensibles y el problema presentó un reto numéricamente. Se realizaron cuatro intentos de resolución en Julia de los cuales solo uno proporcionó el resultado correcto. La solución en R y, sobretodo, en Python fue más rápida y sencilla.

Vale: Falta decir un poco más, siento

El segundo ejercicio fue ajustar una regresión lineal a datos obtenidos con el CENSO 2020. Si bien Julia pudo hacer los filtros de manera adecuada la sintaxis de los comandos es un poco extraña y no da pie al fácil entendimiento de la instrucción. El objetivo fue

comparar y evaluar el manejo de datos en los tres lenguajes. Los tres lenguajes mostraron hacer un análisis rápido y certero para todas las pruebas que se hicieron.

El tercer ejercicio fue el más retador teóricamente y en práctica. Un diseño de experimentos se hace para obtener información sobre los factores que afectan o no un resultado. El algoritmo que se utilizó fue el propuesto por Meyer para elegir los ensayos extras necesarios en caso de que los primeros no dieran suficiente información sobre el modelo que describe el experimento. La fórmula para el valor MD es compleja y se tiene que hacer hasta que el algoritmo termine (alrededor de 70 veces). Por lo tanto, este ejercicio fue una prueba de rapidez de cálculos intensivos y es el que más resalta las habilidades y funciones de Julia. El mismo algoritmo fue hecho en los tres lenguajes, pero Julia fue el más rápido de todos.

En mi opinión, Julia no busca ser la copia de R o Python en análisis y manejo de datos (el dichoso data science). Julia está hecho para ser un lenguaje que realiza simulaciones y cálculos complejos con un alto rendimiento. Julia tiene mucho que ofrecer, pero un análisis de datos fácil y intuitivo no es una de esas cosas. Su uso va mucho más allá y se enfoca en ser el mejor lenguaje para hacer estadística bayesiana, análisis epistemológicos y análisis de sistemas dinámicos.

Habiendo dicho lo anterior, este trabajo se puede extender de muchas maneras. La primera que se viene a la mente es enfocarse en otras áreas de las matemáticas y seguir con la comparación.

Vale: Se me fue la otra que queria decirrrrrrrrrrrrrrr

Apéndice A

Extras

No se olviden cambiar toda la información. Los quiero un chingo

Bibliografía

- Barrios, E. (2010). R: Un lenguaje para análisis de datos y graficación. *Laberintos e Infinitos*, 35–41.
- Bates, D., S. Kornblith, A. Noack, M. Bouchet-Valat, M. Krabbe, and contributors (2022, Enero). JuliaStats/GLM.jl: v1.6.1.
- Bezanson, J., S. Karpinski, V. Shah, A. Edelman, et al. (2014). Julia language documentation. *The Julia Manual*. <http://docs.julialang.org/en/release-0.2/manual>, 1–261.
- Box, G. E. and W. J. Hill (1967). Discrimination among mechanistic models. *Technometrics* 9.
- Carrone, F., M. Nicolini, and H. Obst Demaestri (2021). Data Science in Julia for Hackers. <https://datasciencejuliahackers.com/> (visited 06-10-2021).
- Datta, B. N. (2010). Numerical linear algebra and applications, *Volume 116*. Siam.
- Garcia, S. R. and R. A. Horn (2017). A second course in linear algebra. Cambridge University Press.

- Gelman, A., J. Hill, and A. Vehtari (2021). Regression and other Stories. Cambridge University Press.
- Inc, A. (2022). Using the r programming language in jupyter notebook. <https://docs.anaconda.com/anaconda/navigator/tutorials/r-lang/>.
- JuliaMath (2021). Polynomials.jl. <https://juliamath.github.io/Polynomials.jl/stable/>. Accessed: 2021-11-04.
- Jupyter, P. Jupyter. <https://jupyter.org/>.
- Language, J. P. (2021). Statsmodel.jl. <https://juliastats.org/StatsModels.jl/stable/>. Accessed: 2021-10-12.
- López-Bonilla, J., R. López-Vázquez, and S. Vidal-Beltrán (2018, Jun). 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.
- Meyer, R. D., D. M. Steinberg, and G. Box (1996). Follow-up designs to resolve confounding in multifactor experiments. *Technometrics* 38, 303–313.
- Mitchell, T. J. (1974). An algorithm for the construction of "d-optimal" experimental designs. *Technometrics* 16, 203–10.
- Montgomery, D. C. (2017). Design and analysis of experiments. John wiley & sons.
- Morgenstern, I. and J. L. Morales (2015). Regresión lineal. aritmética inexacta y algoritmos numéricos estables. *Laberintos e Infinitos*, 25–34.

- Noyola, A. P. V. (2022). *Discriminación de factores en experimentos factoriales*.
- NumPy, L. C. (2022, Enero). *NumPy User Guide: Release 1.22.0*.
- Peng, R. (2015). R Programming for Data Science.
- Peng, R. D. and S. C. Hicks (2021). *Reproducible research: A retrospective*. Annual Review of Public Health 42, 79–93.
- Python-Software-Foundation (2022). Python 3.10.2 documentation. <https://docs.python.org/3/index.html>. Accesado: 2022-03-15.
- Spence, L. E., A. J. Insel, and S. H. Friedberg (2000). *Elementary linear algebra*. Prentice Hall.
- Team, R. C. (2022). What is R?
- y el Equipo de Desarrollo de Pandas, W. M. (2022, Febrero). pandas: powerful Python data analysis toolkit.

Una tesis extendida (\overline{tesis}), escrito por Valeria
Aurora Pérez Chávez, se terminó de imprimir
de madrugada,
con mucha cafeína en las venas
y ojeras en la cara.