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

Ciudad de México 2021

«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 (~~tesis~~)**", otorgo de manera gratuita y permanente al Instituto Tecnológico Autónomo de México y a la Biblioteca Raúl Bailléres 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

Índice de algoritmos

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

Historia de Julia como programa, historia de R. También hay que decir que esto no es un manual de Julia ni de R ni de Python.

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 R o Python. Es una combinación de sintaxis simple con alto rendimiento computacional. Su slogan es ”Julia se ve como Python, se siente como Lisp, corre como Fortran”(Carrone et al., 2021). 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, posteriomente, 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 da 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 y usualmente las instrucciones a seguir vienen en las mismas páginas que Windows.

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. Después, cuando entiendas lo básico y empieces a generar programas que requieran mayor nivel computacional comiences a utilizar 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 porque tiene la capacidad para multihilo (*multithreading* en inglés). Esto significa que puede correr diferentes tareas de manera simultánea en varios hilos. Explicado de la manera 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 se van 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 podemos ejecutar Julia.

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

2.4. Básicos de Julia

Como ya se mencionó, 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 sería `x = 2` donde denotamos a x el valor de 2. Toda la información sobre la sintaxis que utiliza Julia la obtuve del manual oficial de Julia (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 ÷ y`	división de enteros	x/y truncado a un entero
`x \ y`	división inversa	equivalente a dividir `y / x`
`x ∧ y`	potencia	eleva `x` a la potencia `y`
`x% y`	residuo	equivalente a `rem(x,y)`
`!x`	negación	realiza lo contrario de `x`
`x & & y`	*and* lógico	verifica si `x` y `y` se cumplen
`x || y`	*or* 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 omitiré dar una mayor explicación.

2.4.2. *Strings* (secuencias de caracteres)

Además de números, Julia puede asignar characteres a variables. Esto se hace utilizando las comillas dobles. Similar a otros lenguajes de programación, podemos accesar 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> nexo = "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 (**?**). La sintáxis 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 quisieramos tener una función a la que le des dos números y te regrese el número mayor, la función sería la siguiente:

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

Para llamar a la función bastaría 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, se hace exactamente igual solamente omitiendo 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 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 requieren 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 programaría el código `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 como descargarlos y agregarlos. Antes de buscar 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 utilizar 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 (en este caso llamada 'col') dataframe llamado df. De la misma manera, si se quisiera agregar una columna nueva basta con asignarle datos a `df.colNueva`. Por ejemplo, si quisieras 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 datos por renglones, modificar y eliminar datos, 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 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)
```

Vale: Tengo que arreglar esto sí o sí

```
julia> df = CSV.read("C:/Users/Valeria/Documents/ITAM/Tesis/Julia con R/ejemplo_csv.csv", DataFrame)
5×6 DataFrame
 Row │ id     nombre      color    deporte    lugar_residencia  estatura
     │ Int64  String15    String7  String15   String31          Float64
─────┼──────────────────────────────────────────────────────────────────
   1 │     1  Valeria     morado   atletismo  Nuevo Leon             1.7
   2 │     2  Paula       azul     hiking     Estado de Mexico       1.75
   3 │     3  Maria Jose  verde    atletismo  Ciudad de Mexico       1.63
   4 │     4  Sofia       negro    funcional  Oaxaca                 1.66
   5 │     5  Monica      rojo     baile      Veracruz               1.58
```

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

Vale: Buen link de ayuda https://www.machinelearningplus.com/linear-regression-in-julia/

¿Cuál es el punto de tener una muestra de tamaño significativo si no sabemos 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. (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 del paquete Douglas Bates and contributors (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 predictores de las formula
- `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). Los datos fueron recabados 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: Como le hago para incluir 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)
16×5 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 que da Gelman.

Regresión lineal múltiple

El caso general de la sección anterior se conoce como regresión lineal múltiple. La diferencia es que en este caso hay múltiples predictores que cumplen ciertos criterios. Gelman et al. (2021) define este tipo de regresión como

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

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

$$y_i = X_i\beta + \epsilon_i, \text{ para } i = 1, \ldots, n \tag{2.1}$$

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 relación 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 hay relación significativa entre la educación y el coeficiente de las madres con los resultados de exámenes de los niños. 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 co

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

$$kid\_score = -11.48 + 51.26 * mom\_hs + 0.97 * mom\_iq$$
$$-0.48 * mom\_hs * mom\_iq + \epsilon$$

Otro de los aspectos que hay que resaltar en este ejemplo es que para incluir la relación entre dos predictores basta usar un asterisco entre ellos al momento de definir la formula 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. La listas son una estructuras de datos por lo que se usan para almacenar varios elementos en una sola variable. Las listas se crear usando paréntesis cuadrados `[]`. Por ejemplo si quisiera hacer una lista de animales en el zoológico haría `animales = ["zebra", "león", "jirafa", .elefante"]`. Para accesar 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 Foundation (Foundation).

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 diferentes 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).

Vale: Arreglar esto

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` NumPy (2022).

- np.array([lista]): Crea un arreglo con los valores de la lista.
- np.insert(arr, obj, values): Inserta los valores (values) en el arreglo arr antes de los indices 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 autómatico 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 `reg` 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 conjeto de herramientas rápidas y eficientes en cuanto a la memoria Foundation (Foundation). 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" ?? (jup).

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 prerequisitos 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 **?** 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 convencía, por lo que decidieron desarrollar uno propio".

R nació de la necesidad de hacer la transición de usuario a desarrollador. Los creadores buscaron crear un lenguaje que podría usarse para hacer un análisis de datos de manera interactiva y para escribir programas más largos.

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

4.0.1. lm

Vale: Falta cita de labs e inf de Barrios

Vale: Falta referencia de https://book and-overview-of-r.html

Vale: Referencia:ht project.org/a

Capítulo 5

Ajuste de polinomios

5.1. El problema

El problema que voy a abordar en esta sección es el mismo que abordaron Morgenstern and Morales (2015) en el artículo mencionado en las referencias. Sin embargo, la diferencia es que yo utilicé Julia, R y Python mientras que ellos compararon R, Excel, Stata, SPSS, SAS y Matlab.

Supongamos que tenemos un conjunto de datos con solamente dos variables x, y. Buscamos ajustarlos a un polinomio de grado k. Es decir, buscamos 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 mejores coeficientes que cumplan la ecuación anterior. Una manera más compacta de ver el problema es de forma matricial

$$y = X\beta. \tag{5.1}$$

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

En esta sección explicare como ajustar un conjunto de datos a un polinomio de grado k. Los datos que usare son proporcionados por el Instituto Nacional de Standards y Tecnología (NIST por sus siglas en inglés). Para este problema, seleccioné el conjunto llamado filip que se encuentra en https://www.itl.nist.gov/div898/strd/lls/data/LINKS/DATA/Filip.dat.
Estos datos contienen 82 pares ordenados (x_i, y_i). La siguiente imagen muestra las primeras 10 observaciones de los datos.

y	x
0.8116	-6.860121
0.9072	-4.32413
0.9052	-4.358625
0.9039	-4.358427
0.8053	-6.955852
0.8377	-6.661145
0.8667	-6.355463
0.8809	-6.118102
0.7975	-7.115148
0.8162	-6.815309

Vale: no sé si agregar una gráfica rápida de los datos porque a simple vista no parecen un polinomio de grado 10

Seleccioné este conjunto de datos porque además de proporcionar la información para el ajuste, también dan la respuesta al vector β con alta precisión en sus dígitos.

5.3. Planteamiento del problema

De esta forma, el vector y de la ecuación 5.1 es de tamaño 82 y corresponde a la columna y del conjunto de datos. Podríamos definir la matriz X de la siguiente manera

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

Representar la matriz X de esta forma tiene una 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 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.

En Julia, el código para cargar los datos 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 creé una función que tiene como argumento una variable k que representa la potencia del polinomio que quiero ajustar.

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

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

    #Defino 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,
    # elevo cada elemento a la potencia correspondiente
    for i = 1:k
        X[:, i + 1] = x.^i
    end
    return X
end
```

5.4. Métodos

5.4.1. *GLM*

Dado que el problema es ajustar una regresión lineal, el primer paquete que se viene a la mente por utilizar es GLM, ya que se sus siglas se traducen a 'Modelos Lineales Generalizados'.

El manual de este paquete se puede encontrar en https://juliastats.org/GLM.jl/v0.11/#Methods-applied-to-fitted-models-1.

Vale: No estoy segura si dejar esto o no

Para ajustar un modelo lineal generalizado, hay que utilizar la función `lm(formula, data)` donde

- formula: Corresponde a la fórmula del ajuste con los nombres de las columnas de los datos
- data: Debe ser un dataframe con los datos por ajustar. Los datos pueden contener valores NA.

En nuestro caso, queremos 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 = lm(@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 viene programada en la documentación del paquete StatsModels en el apartado de Extending @formula syntax que se encuentra en la dirección https://juliastats.org/StatsModels.jl/stable/internals/.

Vale: No sé si mejor poner la referencia?? creo que si

Este método no funcionó. Dado que los datos de NIST vienen con la respuesta correcta, fue claro observar que los resultados con el paquete GLM no ajustaron de manera precisa el polinomio.

5.4.2. Descomposición QR versión económica

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).*

Una de las aplicaciones de la descomposición QR es dar solución a problemas de mínimos cuadrados. Por lo tanto, es el segundo método que utilizamos para obtener los valores β de 5.1.

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

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

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).*

En nuestro problema las dimensiones $m \times n$ de la matriz X son 82×11. Además, 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 $R(X)$. 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 calada 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

$$AP = QR$$

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). La factorización QR con columnas pivoteadas siempre existe debido al siguiente teorema.

Vale: si se dice asi? le dicen version economica en el articulo pero no encuentro la traduccion

Teorema 5.1. *Sea A una matriz de $m \times n$ con rango(A) $= r \leq min(m, n)$. Entonces, existe una matriz de permutación P de $n \times n$ y una matriz ortogonal Q de dimensiones $m \times m$ tal que*

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

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

Para aplicar este método en Julia, hay que usar el siguiente código

Vale: literal el teorema viene de Datta 2010

```
### Con QR Pivoted
F = qr(X, Val(true))
Q = F.Q
P = F.P
R = F.R
```

Ya tenemos el código que calcula las matrices P, Q, R de la descomposición QR con columnas pivoteadas. Para resolver nuestro problema original 5.1 y obtener los valores de los elementos de β hay que hacer un poco de álgebra.

Recordemos que el problema es obtener β de la ecuación $y = X\beta$. También, por el teorema 5.1, sabemos que X siempre tiene descomposición QR con columnas pivoteadas. Es decir, $XP = QR$. Por otro lado, como P es matriz de permutación por lo que

$$\exists z \text{ tal que } Pz = \beta.$$

Por lo tanto, ya tenemos una expresión para β que podemos 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

$$y = XPz = QRz \longrightarrow y = QRz$$

Por lo tanto, el primer paso es resolver la ecuación

$$y = QRz$$

Para finalmente obtener β calculando

$$\beta = Pz$$

En Julia, esto se programa de la siguiente manera

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

Este método tampoco funcionó. Podemos empezar a consider que los datos son tan sensibles que la propagación del error es tal que no permite un buen ajuste del polinomio. Intentaremos con otros dos métodos.

5.4.3. Descomposición de valores singulares

La tercer manera en la que intenté resolver 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 $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$ los eigenvalores positivos en orden decreciente de $(A^*A)^{1/2}$. Los valores singulares de A son*

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

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

Los valores singulares tienen muchas apliciones. Sin embargo, para resolver ecuaciones lineales son usados para obtener la descomposición de valores singulares (DVS).

Teorema 5.2. *Sea $A \in M_{m\times n}(F)$ diferente de cero y sea $r = rango(A)$. Sean $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$ los valores singulares positivos de A y definamos*

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

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

$$A = U\Sigma V^* \tag{5.2}$$

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 lleva por nombre descomposición en valores singulares (DVS).

Es importante observar que las matrices U y V son matrices unitarias. Es decir,

$$UU^*u = u,\ \forall u \in Col(U)$$

$$VV^*v = v,\ \forall v \in Col(V)$$

Pseudoinversa de Moore-Penrose

Ya que conocemos la descomposición de valores singulares, podemos avanzar y definir la descomposición de valores singulares para la pseudoinversa de Moore Penrose.

Teorema 5.3. *Sea* A *una matriz de* $m \times n$ *de rango* r *con una descomposición en valores singulares de* $A = U\Sigma V^*$ *y valores singulares diferentes de cero* $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$*. Sea* $\Sigma^\dagger$ *una matriz de* $n \times m$ *definida como*

$$\Sigma^\dagger_{ij} = \begin{cases} \dfrac{1}{\sigma_i} \text{ si } i = j \leq r \\ 0 \text{ en otro caso.} \end{cases}$$

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

Podemos ver que en realidad lo único que cambia al calcular la pseudoinversa es la matriz Σ. Sin embargo, esta nueva matriz $A^\dagger$ tiene propiedades interesantes.

- $(A^TA)^\dagger A^T = A^\dagger$
- $(AA^T)^\dagger A = (A^\dagger)^T$
- $(A^TA)^\dagger(A^TA) = A^\dagger A = VV^T$

Recordando que buscamos obtener β que satisfaga la ecuación 5.1 podemos ver que el sistema lineal está sobredeterminado. Esto se puede verificar por las dimensiones de la matriz $X_{n\times m} = X_{82\times 11}$. Como $m < n$, sabemos que hay más ecuaciones que variables desconocidas.

Vale: Como sabemos que y esta en el espacio de columnas de V?

De la ecuación 5.1 podemos multiplicar por X^T para obtener

$$X^TX\beta = X^Ty,\ y \in Col(V). \tag{5.3}$$

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

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

.

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 de la siguiente manera:

```
# # # Inversa de Moore Penrose
N = pinv(X)
aux = ones(k + 1)
x_MP = N*y
```

Este método tampoco funcionó. Por lo tanto, investigue más a fondo los paquetes de Julia hasta encontrar el paquete *Polynomials*.

5.4.4. *Polynomials*

Polynomials es un paquete que proporciona aritmética básica, integración, diferenciación, evaluación y hallar raíces para polinomios univariados ?? (pol). Para poder usar el paquete primero hay que descargarlo usando el código 2.4.5.

El paquete *Polynomials* tiene su propia función `fit` que 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 corresponde al grado que buscamos que sea el polinomio (en este caso, grado 10).

A diferencia de los otros método utilizados para este problema, la función `fit` usa el metódo Gauss-Newton para resolver sistemas de ecuaciones no lineales. Sin embargo, en este caso tenemos un problema lineal. Esta fue la primera razón por la que este paquete no fue mi primera opción para resolver el problema.

Vale: No sé si tengo que explicar este método

La segunda razón es que a diferencia de la función `lm` del paquete *GLM*, 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. Sin embargo, tengo que agregarlo a esta sección de la tesis, ya que es el único método que funcionó.

El código en Julia se ve de la siguiente manera:

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

Este método fue el único de los cuatro que funcionó para el polinomio de grado 10. La desventaja de este método es que la función solamente arroja los coeficientes β. Si quisieramos ampliar el análisis y observar, por ejemplo, el valor p de algún regresor tendríamos que buscar otra manera de obtenerlo.

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 dan la respuesta correcta. Por tanto, para probar que los métodos estén programados de la manera correcta los somentí a una serie de pruebas.

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

La segunda prueba consiste en comparar los resultados con R. De manera análoga a Julia, en R también use los datos filip para ajustar un polinomio de grado k de $k = 1, 2, \ldots, 10$. Para esto, use la función `lm(formula, data)` donde los datos siempre son los mismos y la fórmula depende del grado del polinomio. 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 ganado muchos premios por sus aportaciones a la estadística. Sin duda lo más relevante en este caso es que es uno de los contribuidores principales en el desarrollo de R. El código que utilicé para resolver este problema es el mismo que Ripley hizo público.

La tercera prueba fue medir el tiempo que tomaba a R ejecutar la función `lm` con los parametros especificados. El 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)

# Guardo 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)
```

No voy a mostrar todas las tablas con los resultados ya que son muchas. Me voy a limitar a las tablas que considero tienen los resultados más relevantes. Para el polinomio de grado 1, todos los métodos obtienen los mismos resultados.

Todos los métodos funcionan bien calculando el ajuste hasta llegar

Polinomio grado 1

	PolFit	QRPivot	MoorePenrose	LinearFit	R
b0	1.0592655	1.0592655	1.0592655	1.0592655	1.0592655
b1	0.0340946	0.0340946	0.0340946	0.0340946	0.0340946

al polinomio de grado 5. Cuando calculamos el polinomio de grado 6, el método `fit` del paquete `GLM` llamado `LinearFit` comienza a fallar.

Polinomio grado 6

	PolFit	QRPivot	MoorePenrose	LinearFit	R
b0	-18.0975496	-18.0975496	-18.0975496	1.9043149	-18.0975496
b1	-22.2966441	-22.2966441	-22.2966441	0.0000000	-22.2966441
b2	-10.5769427	-10.5769427	-10.5769427	-0.4810935	-10.5769427
b3	-2.5981095	-2.5981095	-2.5981095	-0.2185587	-2.5981095
b4	-0.3486584	-0.3486584	-0.3486584	-0.0403353	-0.3486584
b5	-0.0242444	-0.0242444	-0.0242444	-0.0033915	-0.0242444
b6	-0.0006834	-0.0006834	-0.0006834	-0.0001075	-0.0006834

A partir del polinomio de grado 6, `LinearFit` comienza a fallar y comienza a dar resultados muy érroneos. Todos los métodos arrojan resultados correctos hasta que llegamos al polinomio de grado 10.

Finalmente, podemos ver la tabla de tiempos que le tomo a cada método. Las columnas representan los métodos usados mientras que los renglones el grado de polinomio que se ajustó.

Vale: Las tablas están por donde quieren, mejor copio los resultados?

Polinomio grado 10

	PolFit	QRPivot	MoorePenrose	LinearFit	R
b0	-1467.4896771	9.0134262	8.4430457	0.0000000	-1467.4895489
b1	-2772.1797121	1.6525458	1.3649863	0.0000000	-2772.1794685
b2	-2316.3711835	-5.7676064	-5.3507625	0.0000000	-2316.3709786
b3	-1127.9739915	-3.8636656	-3.3419108	0.0000000	-1127.9738909
b4	-354.4782499	-0.6703657	-0.4064616	0.0000000	-354.4782180
b5	-75.1242053	0.1806044	0.2577266	0.0000000	-75.1241984
b6	-10.8753186	0.1055234	0.1197715	0.0036864	-10.8753176
b7	-1.0622150	0.0214449	0.0231409	0.0019173	-1.0622149
b8	-0.0670191	0.0022775	0.0024040	0.0003759	-0.0670191
b9	-0.0024678	0.0001262	0.0001316	0.0000328	-0.0024678
b10	-0.0000403	0.0000029	0.0000030	0.0000011	-0.0000403

5.6. Eigenvalores y valores singulares

Vale: No estoy segura de que poner dado que no sale la igualdad basica entre eigenvalores y valores singulares

5.7. Número de condición y precisión de la solución

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

Como vimos en la sección de Evaluación de los métodos, los métodos sí están bien programados. Dejando de lado las funciones programadas por default en Julia, el método de factorización QR y descomposición de valores singulares arrojaron buenos resultados hasta los polinomios de grado 9. Esto nos puede llevar a pensar que en realidad, los datos

Tiempos de ejecución de los métodos

	PolFit	QRPivot	MoorePenrose	LinearFit	R
k_1	0.0000335	0.0000415	0.0000388	0.2541768	0.0049980
k_2	0.0000391	0.0001694	0.0000466	0.2781452	0.0009820
k_3	0.0000392	0.0000470	0.0000514	0.2525240	0.0009890
k_4	0.0000653	0.0000779	0.0000594	0.2771920	0.0009999
k_5	0.0000574	0.0000534	0.0000846	0.3059746	0.0010290
k_6	0.0003417	0.0000562	0.0001066	0.2566553	0.0020020
k_7	0.0000630	0.0000566	0.0000651	0.2702990	0.0010321
k_8	0.0000621	0.0000618	0.0000692	0.2647034	0.0020008
k_9	0.0000665	0.0000959	0.0000856	0.2526137	0.0020521
k_10	0.0000799	0.0037128	0.0001023	0.2622232	0.0029919

en sí son muy susceptibles a cambios. Es decir, cualquier cambio en la matriz X o en el vector y resultara en un ajuste de los coeficientes β poco preciso. Esta cualidad también 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 nuestro problema 5.1 tenemos tres casos:

- El vector y tiene impurezas mientras que la matriz X es exacta.
- La matriz X tiene impurezas mientras que el vector y es exacto
- Ambos, el vector y y la matriz X tiene impurezas.

En nuestro caso, nos vamos a enfocar en el tercer caso ya que no tenemos razón para pensar que solamente una columna de los datos originales tiene impurezas mientras que la otra no.

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*

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

(Datta, 2010, p. 65)

El teorema anterior nos está diciendo 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 pertubaciones de A y las pertubaciones de b. Además, el teorema nos dice que aunque las perturbaciones de A y b son pequeñas, puede haber un cambio grande en la solución si el número de condición es grande. Por lo tanto, $Cond(A)$ juega un papel crucial en la sensibilidad de la solución Datta (2010).

El número de condición tiene varias propiedades pero en la que nos queremos centrar es en la siguiente:

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

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 nuestra matriz X de 5.1, vamos a ver una última definición.

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

Ahora vamos a calcular el número de condición. Este calculo lo hice en Julia y en R usando ambos, la función que ya viene programada en cada lenguaje y usando 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 $numcond_1 = 1.7679692504686805e15$ y $numcond_2 = 1.7679692504686795e15$. 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 que por 9 sabemos que nuestro problema está mal condicionado.

Vale: Tengo duda, entonces alguien me podría decir que para que los uso

Capítulo 6

Regresión Lineal Múltiple

6.1. El problema

En esta sección veremos como hacer una regresión lineal múltiple usando Julia. El modelo de regresión lineal clásico es

Vale: No se porque no sale el beta 0

$$y_i = \beta_i X_{i1} + \cdots + \beta_k X_{ik} + \epsilon_i, \text{ para } i = 1, \ldots, n \text{ (Gelman et al., 2021, p. 146)} \tag{6.1}$$

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.

Vale: Esto es de las notas del profesor Barrios

De manera matricial, la ecuación 6.1 se puede escribir como

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

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

6.2. Los datos

Tomando en cuenta que los modelos estadísticos se utilizan como un reflejo de la realidad decidí usa los datos del Censo de Población y Vivienda 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 estoy utilizando 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 dio resultado a la obtención de información de más de 15 millones de personas.

Vale: agregar referencia de la página del INEGI

En este caso, elegí un tema que es de interés para todos los adultos: los ingresos. Más específicamente, quiero 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 ajusté es

$$ingresos \sim horas_{trabajadas} + sexo + edad + escolaridad + entidad_{residencia} + posicion_{laboral} + alfabetismo + aguinaldo + vacaciones + servicio_{medico} \tag{6.2}$$

6.3. Planteamiento del problema

El Censo es el conjunto de cuestionarios que se le aplican a distintas personas en México. Por eso, antes de comenzar a ajustar los datos, hay que hacer un filtro. Para hacer este filtro y saber que significa cada variable así como sus posibles respuestas, descargué el diccionario del cuestionario 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. El diccionario es de suma importancia ya que proporciona el significado de los códigos que usaron para pasar las respuestas de una hoja de papel a una base de datos.

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

La cantidad de datos con la que estoy trabajando es enorme por lo que en primer lugar seleccione solamente las columnas que necesito. Después, puse los siguientes filtros

1.

2. Obtuve solamente las personas tienen un trabajo remunerado. Es decir, no consideré las personas que se ocupan de las labores del hogar, son jubiladas o pensionadas, son estudiantes o tienen alguna incapacidad que les impide trabajar.

3. Descarté a las personas que viven y trabajan fuera de la República Mexicana.

4. Obtuve solamente a las personas que especificaron horas trabajadas e ingreso ganado.

5. Además, para las variables que representan los regresores quité a todas las personas que respondieron a alguna de esas preguntas con No específicado.

Finalmente, ya que la base se reduce considerablemente y todo el proceso dura alredodr de 20 minutos en efectuarse guardo la nueva base de datos filtrada para en un futuro solamente leerla y utilizarla. Al final, 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)

# Las columnas que estoy seleccionando para el ajuste
col_sel = ["ID_PERSONA","ENT","SEXO", "EDAD", "NIVACAD", "ALFABET",
            "INGTRMEN","HORTRA", "CONACT", "SITTRA", "ENT_PAIS_TRAB",
        "AGUINALDO", "VACACIONES", "SERVICIO_MEDICO", "UTILIDADES", "I
# Selecciono solo las columnas del ajuste
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)), sl

# # # FILTRO 2
personas_filt = subset(personas_filt, :ENT_PAIS_TRAB => ByRow(<(33)), sl
```

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

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

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

# # # FILTRO 4
function diferente_a(dataframe, columna, condicion)
    dataframe = subset(dataframe, columna => ByRow(!=(condicion)), skipm
    return dataframe
end

categorias_9 = ["SEXO", "AGUINALDO", "VACACIONES", "SERVICIO_MEDICO",
    "INCAP_SUELDO", "SAR_AFORE", "CREDITO_VIVIENDA", "ALFABET", "SITTR

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_MEDIC
    "ALFABET", "NIVACAD", "ENT_PAIS_TRAB", "ENT", "SITTRA"]

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

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

$$ingresos \sim horas_{trabajadas} + sexo + edad + escolaridad + entidad_{residencia}$$

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

$$ingresos \sim horas_{trabajadas} + sexo + edad + escolaridad + entidad_{residencia} + posicion_l$$

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 funció 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 laa 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_MEDIC

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

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

```
fit10(2500000, "2500mil")
```

6.5. Resultados

No tengo idea de como poner los resultados. Lit, ni idea.

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 todos 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

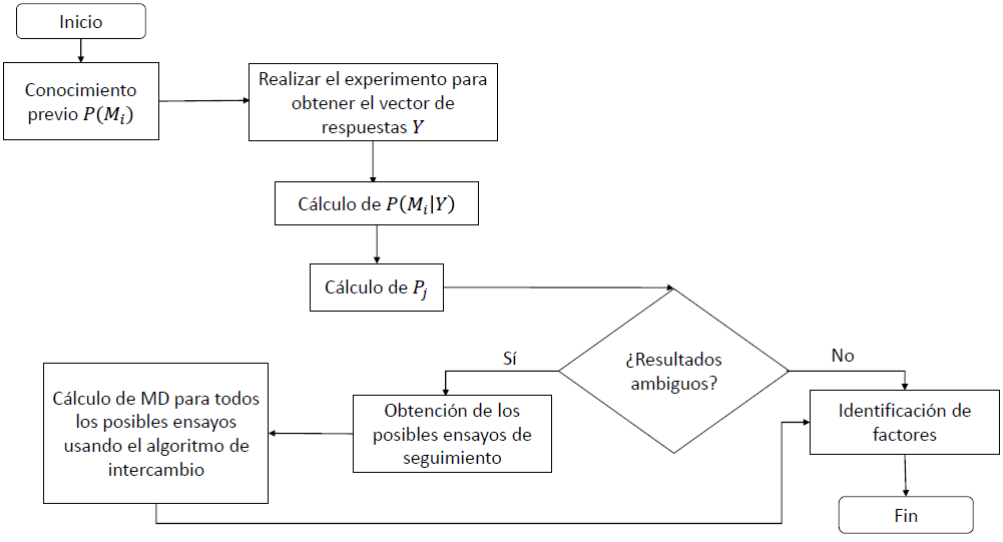

Figura 7.1. Diagrama de metodología descrita por Meyer et al. (1996)

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 $\pi,\ 0 < \pi < 1$ 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

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

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

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

y

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

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

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

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 con 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,

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

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

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

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

donde $I(p_i, p_j) = \int p_i ln(\frac{p_i}{p_j})$ 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 $\hat{y}(x) = f(x)'\beta$. 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 pseudocódido de la función `MDr` y el algoritmo de intercambio están en los apéndices de Noyola (2022).

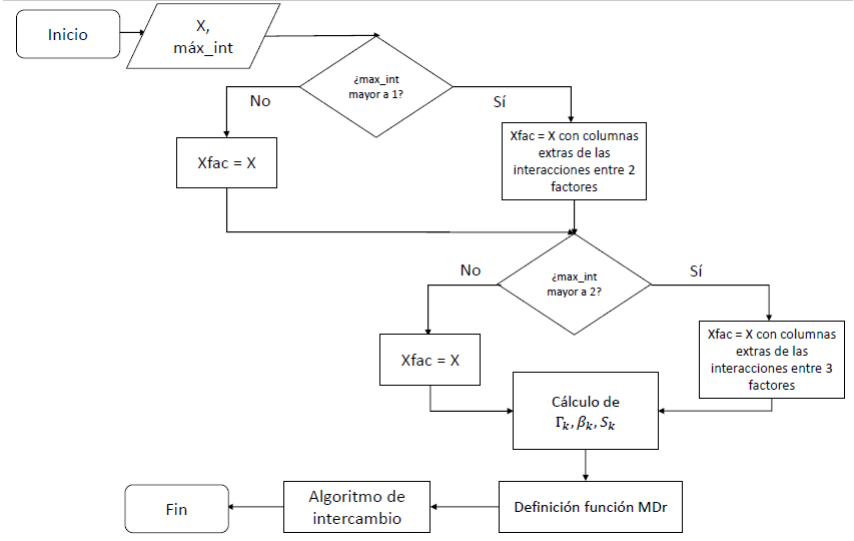

Figura 7.2. Diagrama de la función MDopt

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

Tabla 7.2. 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 probilidades , pero puedo decir que se calculan usando medias posteriores como estimadores de los efectos principales y sus interacciones.

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

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 varia 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_RO <- Sys.time()
e3_RO <- 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_RO

### 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.

Vale: Incluir aquí que falta el tiempo del setup

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.

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.

Apéndice A

Extras

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

Bibliografía

Polynomials.jl. https://juliamath.github.io/Polynomials.jl/stable/. Accessed: 2021-11-04.

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*@(1), 57–71.

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.

Douglas Bates, Simon Kornblith, A. N. M. B.-V. M. K. B. and contributors (2022, Enero). JuliaStats/GLM.jl: v1.6.1.

Foundation, P. S. Python 3.10.2 documentation. https://docs.python.org/3/index.html. Accesado: 2021-03-15.

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.

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*@(4), 303–313.

Mitchell, T. J. (1974). An algorithm for the construction of "d-optimal.^{ex}perimental designs. *Technometrics*@(2), 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. D. and S. C. Hicks (2021). Reproducible research: A retrospective. *Annual Review of Public Health 42, 79–93.*

Spence, L. E., A. J. Insel, and S. H. Friedberg (2000). Elementary linear algebra*. Prentice Hall.*

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.