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 leguaje de programación que ha tomado mucha fuerza en la comunidad científica. Por lo tanto, en esta sección explicaré los básicos de Julia, desde su instalación hasta análisis de regresiones multivariadas.

2.1. Instalación

Al momento de escritura de esta tesis la versión de Julia disponible es la v1.6.3 hasta el 23 de Septiembre del 2021. Para descargar Julia, hay que entrar al siguiente link: https://julialang.org/downloads/.

2.1.1. Windows

Para 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 de Windows, 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, eliges 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.

2.1.2. Mac

Vale: No voy a abarcar tanto MAC porque no tengo Mac jaja

Para el sistema operativo de Mac, solo hay una versión disponible de descarga. Selecciona la opción de 64-bit. Una vez descargado, mueve el archivo de Julia al folder de "Aplicaciones". Para verificar que está bien descargado, haz click en el ícono de Julia y debe salir una terminal como la siguiente:

Vale: Puedo copiar una imagen que no sea mía, sino de una página web? referenciendo obvio

2.2. Atom

Atom es un editor de texto de programación. Su meta es *no dificultar su uso ni posibilidad de modificación* (??, ATO). Busca ser lo suficientemente fácil para que cualquier persona, ya sea principiante o experto en programación lo pueda utilizar. Por lo tanto, es una buena herramienta para escribir, editar y documentar el trabajo que uno hace en Julia.

Vale: No estoy segura de como arreglar

2.2.1. Instalación en Windows

Por supuesto, el primer paso es instalar el programa. Para hacerlo, hay que ir a la página https://atom.io/ y hacer clic en el botón de Download.

Una vez descargado el archivo ejecutable solo hay que darle clic. Este proceso puede tardar un par de minutos. Sin embargo, una vez que termine el programa se ejecutará de manera automática. La pantalla se inicio se ve de la siguiente manera:

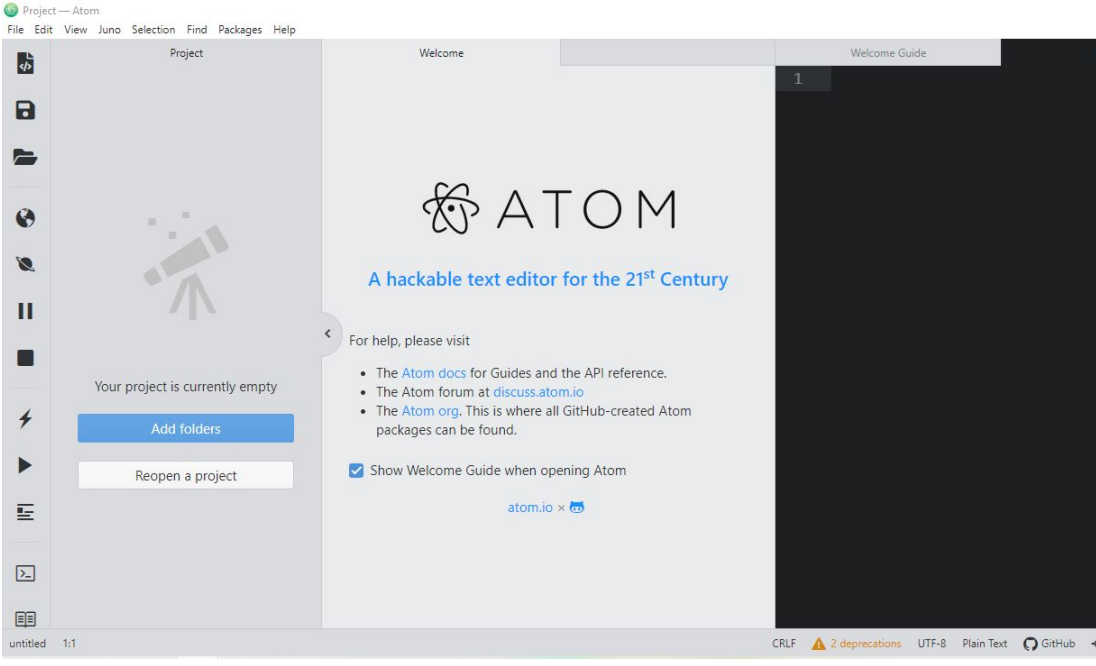


Figura 2.1. Ventana principal de Atom

El usuario puede modificar la pantalla de Atom con diferentes colores y temas siguiendo las instrucciones que vienen en la página https://flight-manual.atom.io/getting-started/sections/atom-basics/.

Sin embargo, esa información no es relevante para esta tesis por lo que se va a omitir.

2.2.2. Conexión con Julia

Ahora es momento de enlazar Julia con Atom. Para hacer esto, es necesario instalar el paquete Juno.

1. Abre Atom y selecciona File seguido de Settings.
2. En el panel izquierdo selecciona la opcion de Install y en el buscador de Install Packages escribe *uber-juno*.
3. Asegurate de que sea el paquete generado por JunoLab y haz click en el botón de Install.

Una vez instalado Juno, va a aparecer una opción llamada Juno en la parte superior de opciones entre View y Selection. Para correr Julia dentro de Atom debes seleccionar Juno y después Open REPL. Aparecerá una nueva ventana en la parte inferior de la pantalla que te dirá que oprimas enter para iniciar una sesión de Julia. Después de hacerlo, Julia iniciará y podrás usarlo de manera normal.
Una ventaja de Atom es utilizarlo por su función principal: ser un editor de texto de programación. Es una forma fácil y rápida de crear código de Julia y el usuario puede decidir si correrlo dentro de Atom o directo en Julia. Para crear un nuevo archivo, hay que seleccionar la opción de File en la parte superior izquierda y después la opción de New File. Para que Atom reconozca ese archivo como texto del lenguaje Julia hay que guardarlo con el nombre deseado agregando `.jl` al final del nombre.

2.3. Símbolo del sistema

Otra opción (mi favorita) es correr Julia desde el símbolo del sistema (también conocido como *Command Prompt* o cmd). De ahora en adelante, se usarán por igual los términos símbolo del sistema y cmd. Para facilitar el uso y trabajo en Julia, 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 las que Julia es un lenguaje de programación muy rápido es que tiene la capacidad para multihilo (*multithreading* en inglés). Esto significa que puede correr diferentes tareas de manera simultánea en varios hilos. 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.

Juno automáticamente pone la misma cantidad de hilos que la cantidad de núcleos de procesadores disponibles (Bezanson et al., 2019). Sin embargo, esto se puede modificar entrando a File > Settings > Packages. Después, buscar el paquete *julia-client* y seleccionar los Settings del paquete. Finalmente, dentro del apartado de Julia Options buscar Number of Threads y poner el número deseado.

Por otro lado, si se está usando el símbolo de sistema el camino es diferente. Antes de ejecutar julia, es necesario modificar la cantidad de hilos que se van a utilizar. En Windows, esto se hace escribiendo `set JULIA_NUM_THREADS=4` (??, Jul) 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. Jupyter

Otra forma de correr Julia es utilizando Jupyther. Jupyter es un programa que exite para desarrollar software de manera pública en decenas de lenguajes de programación . Una de las ventajas de utilizar Jupyter para esta tesis es que se pueden utilizar los tres lenguajes de programación (R, Python y Julia) en un mismo software.
La manera más fácil para obtener Jupyter es instalanda Anaconda. Anaconda es ua 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 esta página https://docs.anaconda.com/anaconda/install/windows/ y seguir las instrucciones de instalación. La instalación de este programa puede tomar unos cuantos minutos.
Una vez instalado Anaconda, basta con ejecutarlo y después seleccionar Launch debajo del ícono de Jupyter Notebook. Se abrirá una ventana en tu navegador como la siguiente:

Vale: Viene de la página oficial de Jupyter que todavía no tengo referencia

Para poder trabajar con Julia en Jupyter, hay que instalar el paquete `IJulia` como en las instrucciones

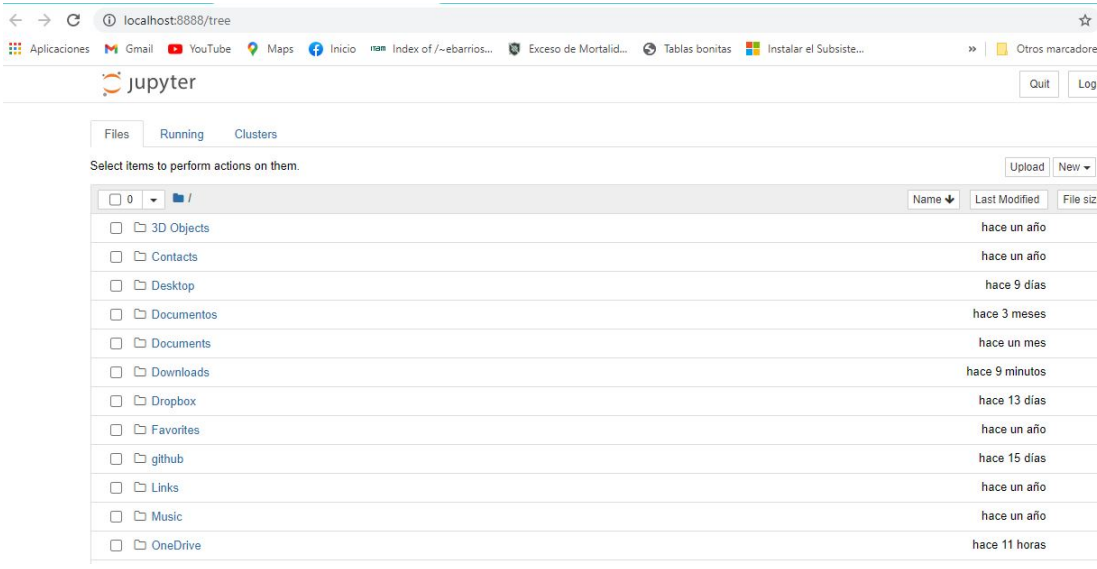


Figura 2.2. Ventana inicial de Jupyter

Vale: No me gustaría agregar en esta parte la instalación de paquetes pero si se tiene que hacer pues ni modo

Para confirmar que la instalación está bien hecha hay que abrir Jupyter, seleccionar New y debe aparecer la opción de Julia 1.6.3.

2.5. Básicos de Julia

Como ya se mencionó, Julia busca ser un lenguaje de programación sencillo e intuitivo. Por lo tanto, su sintáxis 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 viene del manual oficial de Julia (??, Jul).

2.5.1. Operaciones básicas

La siguiente tabla 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)`

La siguiente tabla muestra los operadores booleanos en Julia.

$!x$	negación
$x \;\&\&\; y$	operador de circuito corto *and*
$x \,\|\|\, y$	operador de circuito corto *or*

Vale: Se ven horribles los & & pero ncon verbatim no funciona

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, como no utilizaran en esta tesis, se omitirá mayor explicacion.

2.5.2. *Strings* (secuencias de caracteres)

Además de números, Julia puede asignar characteres a variables. Esto se hace utilizando las comillas dobles. Por ejemplo,

```
string = "Esta tesis es genial"
```

> "Esta tesis es genial"

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 :. Continuando con el ejemplo anterior,

Vale: Como pongo esto que se vean los resultados?

```
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"
"licenciada"

julia> nexo = "en"
"en"

julia> carrera = "matematicas aplicadas"
"matematicas aplicadas"

julia> espacio = " "
" "

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

2.5.3. Funciones

En Julia, una función es un objeto que asigna una tupla de argumentos a un valor de retorno (??, Jul). La sintáxis básica para definir funciones en Julia es:

```
julia> function f(x,y)
           x + y
       end
f (generic function with 1 method)
```

Figura 2.3. Sintáxis básica de una función en Julia.

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.

2.5.4. 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
```

```
A = [1, 9, 9, 7]
```

da como resultado un vector de 4 elementos de tipo Int64.

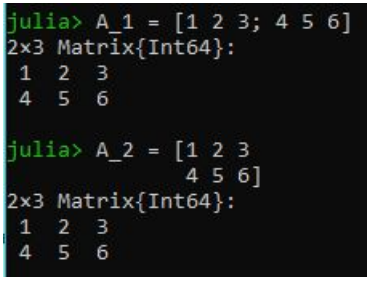
Si quisiera definir un vector renglón, se hace exactamente igual solamente omitiendo el uso de las comas. Sin embargo, es importante señalar que ahora Julia tomó el objeto A como una matriz, no como un vector.

Vale: Me interesa mucho ver lo de los tipos

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.




```
A_1 = [1 2 3; 4 5 6]
A_2 = [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]`.

Vale: No estoy segura de si poner las operaciones básicas de matrices (con sus definiciones, claro) ni la concatenación de matrices

2.5.5. *DataFrames*

Ya que conocemos los arreglos básicos que se pueden manejar en Julia es momento de conocer y manejar los *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 que uno crea en Julia o que viene de alguna base de datos externa.

Instalación del paquete

Similar a otros lenguajes de programación, cuando se usa un paquete por primera vez hay que instalarlo. En las ocasiones siguientes donde se quiera volver a usar el paquete basta con llamarlo con la palabra `using`. Para instalar el paquete, hay que poner las siguientes líneas de código.

Vale: Lo malo de esta versión es que no encuentro como enumerarlo

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

```
using Pkg
Pkg.add("DataFrames")
using DataFrames
```
(2.1)

La primera línea de comando le está diciendo a Julia que use el paquete llamado `Pkg`. Este paquete viene por default en la instalación de Julia. La segunda línea le está indicando a Julia que agregue el paquete llamado `DataFrames`. Una vez instalado el paquete de DataFrames, hay que hacerle saber a Julia que lo queremos usar en esta ocasión. Para eso, utilizamos la tercera línea del código. En caso de que Julia no marque ningún error, el paquete está listo para utilizarse.

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. Para crear un dataframe, 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 llamada col del dataframe llamado df. De la misma manera, si se quisiera agregar una columna nueva basta con asignarle datos a `df.colNueva`. Por ejemplo, si quisieramos agregar una columna llamada `color` al dataframe del ejemplo anterior, el código sería el siguiente:

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

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

Como cualquier otro objeto en Julia, los dataframes tienen diferentes funciones como 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 se omitirá.

Importar datos en un dataframe

Como se mencionó anteriormente, 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`. Para instalarlo, hay que seguir los pasos como en la 2.1 cambiando `DataFrames` por `CSV`. Una vez instalado el programa, basta utilizar el comando `CSV.read` y la ruta de la ubicación del archivo para exportar los datos. Por ejemplo,

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

```
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
```

2.5.6. Regresiones

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

Una vez que ya podemos manejar relativamente grandes volúmenes de información en un dataframe en Julia, podemos empezar con la parte verdaderamente estadística. ¿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 darle algún sentido o explicación a los datos.

Regresión es un método que permite a los investigadores resumir como predicciones o valores promedio de un resultado varian 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. Como es de esperarse, Julia tiene un paquete llamado `GLM` que calcula modelos lineales. Para instalarlo y utilizarlo, hay que seguir las instrucciones que vienen en la tabla 2.1 con este paquete. Para ajustar un modelo lineal generalizado, hay que usar la función `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`:

Vale: Viene del manual de GLM que todavia no tengo referencia

Vale: Todavía no entiendo muy bien esto

Regresión lineal simple

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

$$y = a + bx + \epsilon \tag{2.2}$$

Vale: no entiendo muy bien a que se refiere esto, creo que está fuera del alcance de la tesis

Para este modelo, use los datos de (Gelman et al., 2021) sobre el modelo creado por Douglas Hibbs para predecir las elecciones de Estados Unidos basandose solamente en el crecimiento económico. Los datos se ven de la siguiente manera:

Vale: Como le hago para

```
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
```

El modelo que buscamos es que el voto sea resultado del crecimiento económico. Por lo tanto, el modelo programdo en Julia se ve de la siguiente manera:

```
julia> elections_lm = lm(@formula(vote ~ growth), elections)
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, LinearAlgebra.Chole
skyPivoted{Float64, Matrix{Float64}}}}, Matrix{Float64}}

vote ~ 1 + growth

Coefficients:
──────────────────────────────────────────────────────────────────────────
               Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%
──────────────────────────────────────────────────────────────────────────
(Intercept)  46.2476     1.62193   28.51    <1e-13   42.769     49.7263
growth        3.06053    0.696274   4.40    0.0006    1.56717    4.55389
──────────────────────────────────────────────────────────────────────────
```

Al resultado de la regresión es el modelo $y = 46.3 + 3.1x$.

Regresión lineal múltiple

Vale: Todo esto lo saque de (Gelman et al., 2021)

El modelo de regresión lineal clásico para múltiples predictores es

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

Vale: Solo para que quede claro, es lo mismo que viene en el libro de Gelman

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

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

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

En primer lugar, analicé un modelo que incluye una relación entre dos predictores. Para este modelo, usé los datos de (Gelman et al., 2021) sobre 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`).

La relación que intento modelar es si el coeficiente intelectual y la educación de las madres afectan los resultados de los 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

```
using DataFrames, GLM, CSV

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

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

Vale: OJO que el coeficiente de momiq sale en R como 1.1 y aquí como 0.97

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

Por lo tanto, para incluir la relación entre dos predictores, basta usar un asterisco entre ellos en la formula de la regresión.

Vale: Me acabo de dar cuenta de eso del paquete CSV. Cambio todo para que ahora sea CSVFiles?? O solo lo menciono??

Como segundo ejemplo, analicé un modelo que tiene una variable categórica para mostrar su aplicación en Julia. A diferencia de otros lenguajes, en Julia no hay que especificar si el predictor es una variable categórica. Sin embargo, si la base de datos tiene valores faltantes (*missing values*) es mejor utilizar el paquete llamado CSVFiles. La instalación y uso del paquete es igual que como se muestra en las instrucciones 2.1.

Tomando en cuenta lo anterior, utilicé datos de una encuesta hecha a 1816 personas que buscaba predecir sus ganancias anuales tomando en cuenta aspectos como su altura, peso, sexo, etnicidad, educación, educación de los padres, etc. Para este ejemplo en específico, los predictores son la altura centrada en el promedio (`cHeight`), el sexo (`male`) y la etnicidad (`ethnicity`). La variable respuesta es el peso (`weight`). La razón por la que se centró la altura es para mejorar la comprensión del modelo. El código en Julia para este modelo es el siguiente:

```
using DataFrames, GLM, CSVFiles

earnings_data = DataFrame(load("C:/Users/Valeria/Documents/ITAM/Tesi
earnings_data.cHeight = earnings_data.height .- 66

fm = @formula(weight ~ cHeight + male + ethnicity)
earnings_lm = lm(fm, earnings_data)
```

El resultado es la regresión

Vale: Otra vez, los resultados son levemente diferentes que en el libro

Vale: Se ve bien así en tabla, no?

	Coeficiente	Error Estándar
`cHeight`	154.33	2.23
`male`	3.85	0.25
`ethnicity: Hispanic`	-6.15	3.56
`ethnicity: Other`	-12.26	5.18
`ethnicity: White`	-5.19	2.27

Gráfica de resultados

Histograma

Gráfica cuantil-cuantil

Análisis de residuales

Tabla de análisis de varianza

2.5.7. Paquete Distributions

Para aprender como funcionan las distribuciones en Julia. El manual del paquete viene aquí https://juliastats.org/Distributions.jl/v0.14/index.html

Capítulo 3

Ajuste de polinomios

3.1. El problema

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

$$y = X\beta. \tag{3.1}$$

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 $k+1$.

3.2. 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.

3.3. Planteamiento del problema

De esta forma, el vector y de la ecuación 3.1 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 3.1 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
```

3.4. Métodos

3.4.1. *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.

3.4.2. 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).*

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 3.1.

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).

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

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).*

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

Teorema 3.1. *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)

Vale: literal el teorema viene de Datta 2010

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

```
### 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 3.1 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 3.1, 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 3.1 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.

3.4.3. 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)

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

Teorema 3.2. *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

$$A = U\Sigma V^* \tag{3.2}$$

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)

La ecuación 3.2 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 3.3. *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)

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.

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

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

Recordando que buscamos obtener β que satisfaga la ecuación 3.1 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 3.1 podemos multiplicar por X^T para obtener

$$X^T X\beta = X^T y,\ y \in Col(V). \tag{3.3}$$

La ecuación 3.3 siempre da un sistema determinado (balanceado) López-Bonilla et al. (2018). Ahora bien, multiplicando 3.3 por $(X^T X)^\dagger$ y usando las propiedades de la matriz pseudo inversa podemos obtener

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

.

Por lo tanto, la pseudo inversa de Moore Penrose da la solución de mínimos cuadrados de 3.1 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*.

3.4.4. *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 2.1.

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.

3.5. 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 sometí 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. 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()
```

```r
`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.

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

Todos los métodos funcionan bien calculando el ajuste hasta llegar al polinomio de grado 5. Cuando calculamos el polinomio de grado 6, el método `fit` del paquete `GLM` llamado `LinearFit` comienza a fallar.

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.

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

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

3.6. Eigenvalores y valores singulares

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

3.7. 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).*

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

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

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 3.1 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.

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

Teorema 3.4. *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)

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:

$$Cond(A) = \frac{\sigma_{máx}}{\sigma_{mín}} \tag{3.4}$$

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

Antes de calcular el número de condición de nuestra matriz X de 3.1, 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).*

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 3.4. 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 3.1 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 4

R

Porque juntar Julia con R. Usar documentación oficial: https://cran.r-project.org/web/packages/JuliaCall/JuliaCall.pdf. También en esta sección incluir muchos ejemplos prácticos

4.1. Paquete JuliaCall

Como instalarlo en R

4.1.1. install_julia

4.1.2. julia_setup

De este no estoy segura, tengo que checar en las cosas de Rodrigo

4.1.3. Núcleos y threads

4.1.4. JuliaObject

4.1.5. julia_assign

4.1.6. julia_eval

4.1.7. julia_package

4.2. Ejemplo

Capítulo 5

Computadoras virtuales

5.1. Amazon

Como hacer una computadora virtual en Amazon que me haga todo esto. El tutorial oficial de como instalarlo en Amazon está aquí: https://d1.awsstatic.com/whitepapers/julia-on-sagemaker.pdf?did=wp_card&trk=wp_card

5.2. Docker

También se podría usar Docker pero te quita espacio en tu computadora y puede crashear como la mía. Para Windows como que no está muy bonito el asunto pero al parecer para Linux sí. También podría ser opción idk

Apéndice A

Extras

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

Bibliografía

Atom flight manual. https://flight-manual.atom.io/. Accessed: 2021-10-06.

Julia 1.6 documentation. https://docs.julialang.org/en/v1/. Accessed: 2021-10-07.

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

Bezanson, J., J. Nash, and K. Pamnany (2019, Jul). Announcing composable multi-threaded parallelism in julia.

Carrone, F., M. Nicolini, and H. Obst Demaestri (2021). *Data Science in Julia for Hackers*. https://datasciencejuliahackers.com/(visited 06-10-2021).

Datta, B. N. (2010). *Numerical linear algebra and applications*, Volume 116. Siam.

Garcia, S. R. and R. A. Horn (2017). *A second course in linear algebra*. Cambridge University Press.

Gelman, A., J. Hill, and A. Vehtari (2021). *Regression and other Stories*. Cambridge University Press.

López-Bonilla, J., R. López-Vázquez, and S. Vidal-Beltrán (2018, Jun). Moore-penrose's inverse and solutions of linear systems. *World Scientific News*.

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

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.