Previa I

Autores: Nestor Marmolejo
Agustín Salazar

IS&C, Universidad Tecnológica de Pereira, Pereira, Colombia
Correo-e: nestor.marmolejo@utp.edu.co & agustin.salaza@utp.edu.co

***Resumen*— Este artículo tiene como objetivo implementar las funciones de ciertas librerías de Python por medio de un programa que tiene como fin, mostrar el comportamiento de las muertes por covid-19 en Colombia y predecir por medio de machine learning la semana en la que se espera un número específico de muertes. Explicando dicho programa de una manera detallada para un mejor entendimiento del lector.**

Palabras clave*— *Funciones, Covid-19, Colombia, Python, Machine Learning.

***Abstract*— This article aims to implement the functions of certain Python libraries through a program that aims to show the behavior of deaths from covid-19 in Colombia and predict Through machine learning the week in which a specific number of deaths is expected. Explaining said program in a detailed way for a better understanding of the reader.**

***Key Words*— Functions, Covid-19, Colombia, Python, Machine Learning.**

I. INTRODUCCIÓN

Para analizar las muertes por Covid-19 en Colombia, se implementa un software que permite visualizar por medio de gráficos de dispersión la línea de tendencia o ajuste polinomial que tienen los datos que contienen el valor de muertes diarias en dicho país. Esta muestra se extrajo directamente de la página oficial del Ministerio de Salud de Colombia, donde aparecen dos (2) columnas con datos. La primera contiene el número identificador o el número ascendente del día, es decir, día uno (1), día dos (2), etc. Hasta el día ciento ochenta (180). Dichos datos son actuales, el primer día representa el día veintiuno de marzo del dos mil veinte (21-Marzo-2020), donde el Gobierno Nacional dio a conocer a los ciudadanos de dicho país el primer fallecimiento por el Covid-19 y va hasta el día veinte de septiembre del mismo año (20-Septiembre-2020). El programa recopila estos datos y comprueba primero, por medio de unas funciones que los datos no tengan valores erróneos, es decir, que no existan datos que contengan una gran diferencia con el número de muertes del día anterior al día actual e igualmente en sentido contrario.
Se presentan una serie de gráficos, esto con el objetivo de ilustrar mejor lo que se está estudiando. Por último, el programa por medio de Machine Learning puede dictaminar en que semana se puede evidenciar un número especifico de muertes. Para obtener dicho resultado, es necesario que se entrene el programa, esto quiere decir que hay que tomar una pequeña muestra de la muestra total para indagar, estudiar y concluir sobre los mismos, para realizar este proceso, se calculan una serie de valores y datos que permiten que el programa se entrene y pueda dictaminar lo que se mencionó previamente.

II. METODOLOGÍA

Se importaron las siguientes librerías:

```
import scipy as sp
import matplotlib.pyplot as plt
import numpy as np
import os
```

Junto con ellas, se importaron las carpetas `'DATA'` y `'CHAR'` en el directorio del programa

- DATA_DIR: es el directorio de los datos
- CHART_DIR: es el directorio de los gráficos generados.

Se importa la librería `'utils'`:

```
from utils import DATA_DIR, CHART_DIR
```

Se eliminan las advertencias por el uso de funciones en el futuro, utilizando la función seterr de la librería numpy:

```
np.seterr(all='ignore')
```

Se importan los datos con lo que se trabajarán en el programa, son sacados de la página oficial del Ministerio de Salud de Colombia y representan las muertes diarias por el Covid-19 desde el primer día que se dio a conocer la primera muerte por el virus hasta la actualidad. El documento que contiene dichos datos se llama: MuertesCovidColombiaB.tsv

```
data = np.genfromtxt(os.path.join(DATA_DIR,
"MuertesCovidColombiaB.tsv"), delimiter="\t")
```

Se establece el tipo de dato y se pone un float de 64 bits para realizar operaciones grandes en un futuro.

```
data = np.array(data, dtype=np.float64)
```

Se definen los colores y los tipos de líneas que serán utilizados en las gráficas de dispersión.

```
colors = ['g', 'k', 'b', 'm', 'r']
linestyles = ['-', '-.', '--', ':', '-']
```

Se crea el vector 'x' y el vector 'y', que corresponden a la primera y segunda columna (respectivamente) de la matriz data:

```
x = data[:, 0]
y = data[:, 1]
```

Al tener los valores separados de esta manera, se utiliza la función 'isnan' de la librería numpy para verificar que el número de entradas sean correctas.

```
print("Número de entradas incorrectas:", np.sum(np.isnan(y)))
```

Al comprobar que ambos vectores se encuentran sin entradas incorrectas, se da inicio a la graficación definiendo los modelos con su respectivo ajuste polinomial.

Para simplificar el trabajo, se crea una función para graficar los diferentes tipos de polinomios que pueden definir una línea de dispersión.

```
def plot_models(x, y, models, fname, mx=None, ymax=None, xmin=None):
```

Se crea una nueva figura para ilustrar y visualizar mejor las dispersiones, o activa una existente.

```
    plt.figure(num=None, figsize=(12, 6))
```

Se borra el espacio de la figura por si existe algún tipo de basura o figura incorrecta.

```
    plt.clf()
```

Se realiza un gráfico de dispersión de 'y' frente a 'x' con tamaño de diez (10) puntos y colores de marcador para una mejor visualización.

```
    plt.scatter(x, y, s=10)
```

Se le asignan títulos a cada una de las gráficas para orientar mejor al visualizador de las gráficas.

```
    plt.title("Muertes por Covid-19 en Colombia")

    plt.xlabel("Semanas")

    plt.ylabel("Nro muertes")
```

Los primeros corchetes ([]) se refieren a los puntos en 'x', los siguientes corchetes ([]) se refieren a las etiquetas 'w * 7' es la agrupación de puntos por semana. El valor de 'w' permite ver estos puntos por semana desde 1 hasta 25, lo cual constituye las semanas analizadas.

```
    plt.xticks([w *7 for w in range(100)], ['%i' % w for w in range(100)])
```

Se pasa a evaluar el tipo de modelo, si no se define ningún valor para 'mx', este valor será calculado con la función 'linspace', la cual devuelve números espaciados uniformemente. En este caso, sobre el conjunto de valores x.

```
    if models:
        if mx is None:
            mx = np.linspace(0, x[-1], 1000)
```

La función 'zip()' toma elementos iterables (puede ser cero o más) los agrega en una tupla y los retorna.

Aquí se realiza un ciclo dependiendo del modelo, estilo de línea y color. Dibuja la gráfica de dispersión con su respectivo tipo de función polinomial

```
        for model, style, color in zip(models, linestyles, colors):
```

Dibuja la gráfica con el valor de 'x', el valor de la función evaluada en ese punto, el tipo de línea, el ancho de la misma y el color.

```
            plt.plot(mx, model(mx), linestyle=style, linewidth=2, c=color)
```

Esta función recibe el número identificador del modelo y la ubicación en la gráfica para este caso, dibuja en la parte superior izquierda la lista de los modelos visualizados.

```
        plt.legend(["d=%i" % m.order for m in models], loc="upper left")
```

Realiza el ajuste de escalamiento automático en el eje o ejes especificados.

```
    plt.autoscale(tight=True)
```

Se establecen los límites 'y' y 'x' de los ejes actuales.

```
    plt.ylim(ymin=0)
    if ymax:
        plt.ylim(ymax=ymax)
    if xmin:
        plt.xlim(xmin=xmin)
```

Se configuran las líneas de la cuadrícula.

```
    plt.grid(True, linestyle='-', color='0.75')
```

Se guarda la figura creada después de graficar los datos, esto para mayor comodidad.

```
    plt.savefig(fname)
```

La función 'os.path.join' une a uno o más componentes de ruta de forma inteligente. Este método concatena varios componentes de la ruta con exactamente un separador de directorio ("/") después de cada parte no vacía, excepto el último componente de la ruta. Si el último componente de la

ruta que se va a unir está vacío, se coloca un separador de directorio `("/")` al final.

Esta es la primera gráfica que se realiza, es decir, en esta gráfica se visualiza la dispersión de los datos que se tomaron en la muestra y se guarda en la carpeta `'CHART'` con el nombre `'1400_01_01.png'`.

```
plot_models(x, y, None, os.path.join(CHART_DIR,
"1400_01_01.png"))
```

Crea y dibuja los modelos de datos. La función **`'polyfit'`** de numpy, realiza un ajuste polinomial de mínimos cuadrados. Recibe como parámetros los vectores, el grado del polinomio y el valor full. En caso de ser falso, solo se devuelven los coeficientes. Cuando es verdadero, se devuelve la información de diagnóstico almacenada en las siguientes variables:

```
fp1, res1, rank1, sv1, rcond1 = np.polyfit(x,
y, 1, full=True)
```

La función `'poly1d'`, ayuda a definir una función polinomial. Facilita la aplicación de "operaciones naturales" en polinomios.

```
f1 = sp.poly1d(fp1)
```

Se definen funciones polinomiales de diferentes grados para graficar en la dispersión y poder visualizar qué polinomio se acerca más a la línea de dispersión.

```
f2 = sp.poly1d(fp2)
f3 = sp.poly1d(np.polyfit(x, y, 3))
f4 = sp.poly1d(np.polyfit(x, y, 4))
f5 = sp.poly1d(np.polyfit(x, y, 5))
f6 = sp.poly1d(np.polyfit(x, y, 6))
f10 = sp.poly1d(np.polyfit(x, y, 10))
f100 = sp.poly1d(np.polyfit(x, y, 100))
```

Se grafican los modelos, en cada modelo existen diferentes funciones polinomiales. La primera gráfica con una función de grado número cuatro (4). La segunda, la gráfica con una de grado número tres (3) y cinco (5). La última con una de grado número uno (1), dos (2), cuatro (4), diez (10) y cien (100); aunque para la función de grado número cien (100), toma automáticamente como si fuera una función de grado número sesenta y ocho (68).
Simultáneamente guarda estas figuras en la carpeta **`'CHART'`** y la titula con su respectivo nombre.

```
plot_models(x, y, [f4], os.path.join(CHART_DIR,
"1400_01_02.png"))
plot_models(x, y, [f3, f5],
os.path.join(CHART_DIR, "1400_01_03.png"))
plot_models(x, y, [f1, f2, f4, f10, f100],
os.path.join(CHART_DIR, "1400_01_04.png"))
```

Ajusta y dibuja un modelo utilizando el conocimiento del punto de inflexión, se escoge este punto ya que es cuando se puede ver un cambio brusco en la gráfica, es decir, donde cambia la concavidad de la misma.

```
inflexion = 13.5 * 7
```

`'xa'` es el número de días hasta la semana 13.5 (punto de inflexión) y **`'ya'`** es el número de muertes por día en `'xa'`.

```
xa = x[:int(inflexion)]
ya = y[:int(inflexion)]
```

`'xb'` es el número de días a partir de la semana 13.5 (punto de inflexión) y `'yb'` es el número de muertes por día en `'xb'`.

```
xb = x[int(inflexion):]
yb = y[int(inflexion):]
```

Se grafican dos parábolas (`'fa'` y `'fb'`), es decir, son de grado número dos (2).

```
fa = sp.poly1d(np.polyfit(xa, ya, 2))
fb = sp.poly1d(np.polyfit(xb, yb, 2))
```

Se presenta el modelo basado en el punto de inflexión y se aplica la función que se definió anteriormente. Como parámetros se le envían los vectores `'x'` y `'y'`, las funciones `'fa'` y `'fb'`, la dirección y nombre donde se va a guardar las gráficas.

```
plot_models(x, y, [fa, fb],
os.path.join(CHART_DIR, "1400_01_05.png"))
```

Se define la siguiente función que permite encontrar el error que puede tener una función polinomial.

```
def error(f, x, y):
    return np.sum((f(x) - y) ** 2)
```

Se extrapola de modo que se proyecten respuestas en un futuro (desde la semana veintiséis (26) en adelante).

```
plot_models(x, y, [f1, f2, f4, f10, f100],
os.path.join(CHART_DIR, "1400_01_06.png"),
    mx=np.linspace(0 * 7, 30 * 7, 100),
ymax=500, xmin=0 * 7)
```

Se entrenan los datos, pero esto se hará únicamente después del punto de inflexión.

Función de grado número uno (1).

```
fb1 = fb
```

Función de grado número dos (2).

```
fb2 = sp.poly1d(np.polyfit(xb, yb, 2))
```

Función de grado número tres (3).

```
fb3 = sp.poly1d(np.polyfit(xb, yb, 3))
```

Función de grado número diez (10).

```
fb10 = sp.poly1d(np.polyfit(xb, yb, 10))
```

Función de grado número cien (100).

```
fb100 = sp.poly1d(np.polyfit(xb, yb, 100))
```

Se procede a graficar después del punto de inflexión con los datos o valores que se encontraron anteriormente.

```
plot_models(x, y, [fb1, fb2, fb4, fb10, fb100],
os.path.join(CHART_DIR,       "1400_01_07.png"),
mx=np.linspace(0 * 7, 30 * 7, 100), ymax=500,
xmin=0 * 7)
```

Se inicia con el entrenamiento de los datos de prueba:

`Frac` es una pequeña fracción de los datos de la muestra, en este caso del treinta por ciento (30%). Esto se hace para optimizar y disminuir la complejidad de utilizar el cien (100%) de los datos y así garantizar una pronta respuesta.

```
frac = 0.3
```

Se guarda el valor exacto del treinta por ciento (30%) de los valores de `'xb'`.

```
split_idx = int(frac * len(xb))
print ("split_idx: ", split_idx)
```

Se guardan los índices de los valores barajados:

```
shuffled                                        =
sp.random.permutation(list(range(len(xb))))
print ("shuffled: ", shuffled)
```

Se ordenan los índices anteriores para comprobar que si están todos primeros cincuenta y cuatro (54) valores de `'xb'` ordenados:

```
test = sorted(shuffled[:split_idx])
#print ("test: ", test)
```

En esta variable se encuentran los primeros cincuenta y cuatro (54) valores de `'xb'` ordenados.

```
test = sorted(shuffled[:split_idx])
#print ("test: ", test)
```

En esta otra variable se encuentran depositados los últimos cincuenta y cuatro (54) valores de `'xb'` ordenados.

```
train = sorted(shuffled[split_idx:])
#print ("train: ", train)
```

Se crean las funciones de grado número uno (1) y dos (2) con los últimos cincuenta y cuatro (54) valores.

```
fbt1 = sp.poly1d(np.polyfit(xb[train],
yb[train], 1))
fbt2 = sp.poly1d(np.polyfit(xb[train],
yb[train], 2))
fbt4 = sp.poly1d(np.polyfit(xb[train],
yb[train], 4))
fbt10 = sp.poly1d(np.polyfit(xb[train],
yb[train], 10))
fbt100 = sp.poly1d(np.polyfit(xb[train],
yb[train], 100))
```

```
print("fbt2(x)= \n%s" % fbt2)
print("fbt2(x)-100= \n%s" % (fbt2-100))
```

Se dibuja la gráfica con las funciones de diferentes polinomios:

```
plot_models(x, y, [fbt1, fbt2, fbt4, fbt10,
fbt100], os.path.join(CHART_DIR,
"1400_01_08.png"), mx=np.linspace(0 * 7, 30 *
7, 100), ymax=500, xmin=0 * 7)
```

Se importa la función `'fsolve'` de la librería scipy.optimize la cual encuentra las raíces de una función.

```
from scipy.optimize import fsolve
#print(fbt2)
#print(fbt2 - 100)
```

Se hace una aproximación con la función `'fsolve'` restando a la función de grado número dos (2), cien (100) muertes/día. Se le manda un `'x0'` con el objetivo de disminuir los tiempos de respuesta. Se divide por siete (7) para encontrar el valor de la semana en la que se verá el valor anteriormente mencionado (cien (100)).

```
alcanzado_max = fsolve(fbt2 - 100, x0=200) /
(7)
print("\nLas cien (100) muertes por Covid-19 en
Colombia se esperan en la semana %f"
%alcanzado_max[0])
```

Todo el programa está disponible en la siguiente pagina web: https://repl.it/join/jkytuvtq-nestormarmolejo

III. ANÁLISIS Y RESULTADOS

A continuación, se presentan las gráficas mostradas en pantalla por el programa y guardadas en la carpeta CHART.

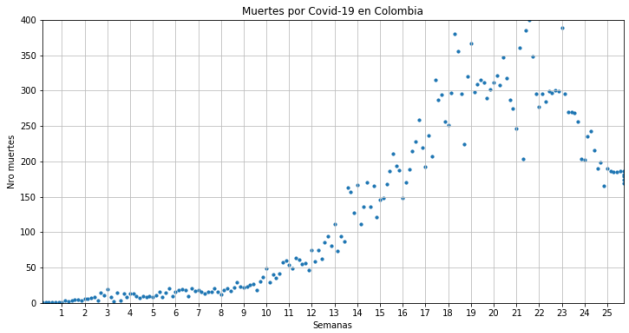

Figura 1. Gráfico de dispersión muertes/días.

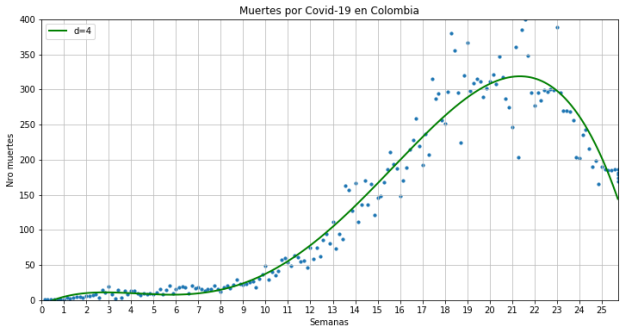

Figura 2. Gráfico con función polinomial de grado número cuatro (4).

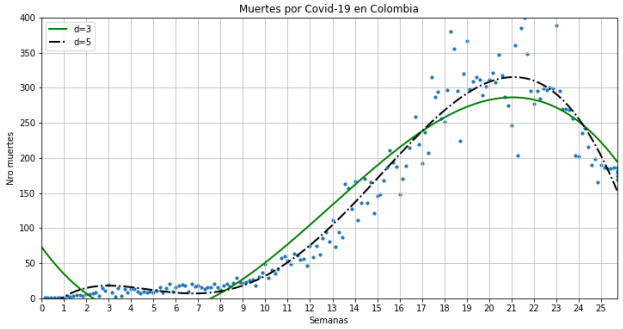

Figura 3. Gráfico con función polinomial de grado número tres (3) y cinco (5).

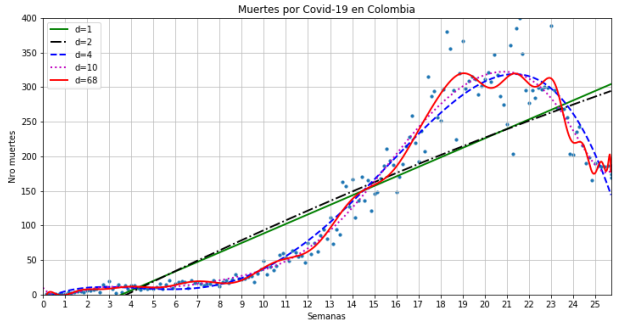

Figura 4. Gráfico con función polinomial de grado número uno (1), dos (2), cuatro (4), diez (10) y sesenta y ocho (68).

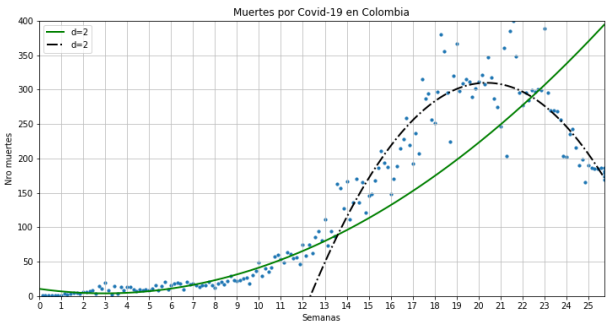

Figura 5. Gráfico con funciones polinomiales de grado número dos (2) antes y después del punto de inflexión.

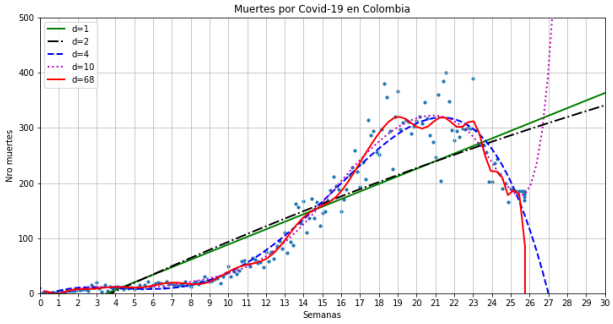

Figura 6. Gráfico con función polinomial de grado número uno (1), dos (2), cuatro (4), diez (10) y sesenta y ocho (68) con inclusión de semana veintiséis (26) a la treinta (30).

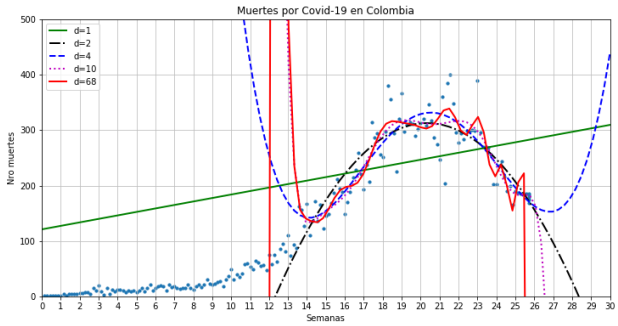

Figura 7. Gráfico con función polinomial de grado número uno (1), dos (2), cuatro (4), diez (10) y sesenta y ocho (68) con inclusión de la semana veintiséis (26) a la treinta (30) a partir del punto de inflexión.

IV. CONCLUSIONES

- ✓ Con los gráficos anteriores se puede determinar que el pico de las muertes por Covid-19 se vivió en la semana número veintiuno (21).
- ✓ Las muertes están teniendo un declive desde la semana número veintitrés (23), esto indica que cada vez el número de muertes esperadas será menor.
- ✓ Los diferentes tipos de funciones polinomiales permiten realizar augurios certeros.
- ✓ La utilización de funciones polinomiales de grados muy elevados genera errores al momento de realizar un presagio, por lo que es mejor utilizar funciones de grados menores para obtener respuestas más optimas.

V. BIOGRAFIAS

Salazar Agustín. Nació en Pereira, Colombia en 1998. Recibió el título de bachiller técnico en el año 2015 del Instituto Técnico Superior de Pereira. Actualmente se encuentra cursando su pregrado Ingeniería en Sistemas y Computación en la Universidad Tecnológica de Pereira.

Marmolejo Nestor. Nacido en Pereira-Risaralda Colombia el 27 de agosto de 1999, bachiller del Instituto Técnico Superior, Técnico en implementación y mantenimiento de equipos electrónicos industriales y estudiante de ingeniería en sistemas y computación en la Universidad Tecnológica de Pereira.