DATOS ESTRUCTURADOS

En este capítulo abordaremos el trabajo en ciencia de datos utilizando datos estructurados, aplicando la metodología de aprendizaje basado en proyectos (ABP) a través de varios casos de estudio. Comenzaremos con un Análisis Exploratorio de Datos (EDA) aplicado a un dataset de reporte de casos de COVID-19 en la ciudad de Bogotá (Colombia). Posteriormente, realizaremos tanto EDA como Ingeniería de Características (FE) a un dataset de deserción de clientes bancarios.

3.1. Caso de estudio 1: Análisis Exploratorio dataset COVID-19 de Bogotá

En el primer semestre de 2020, cuando inició la pandemia por COVID-19, la Alcaldía de Bogotá publicó de forma abierta los datos de reporte diario de casos positivos, con el fin de poner esta información a disposición de toda la comunidad. Este dataset hace parte del portal **Datos Abiertos Bogotá**, bajo el nombre *"Casos confirmados de COVID-19 en Bogotá D.C."*. La actualización más reciente se encuentra disponible en: https://n9.cl/5uyga

Con el dataset del **18 de abril de 2020**, se elaboró un video explicativo publicado en el canal de YouTube de Dora María Ballesteros, disponible en: https://youtu.be/kezqpilbKyA

Para este capítulo, utilizaremos el dataset del **19 de mayo de 2020**, sobre el cual se realizó un proceso de EDA en un **Jupyter Notebook** (para propósitos académicos, recomiendo Google Colaboratory).

El análisis se realizó en Python, iniciando con la importación de librerías:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
data = pd.read_csv('/content/osbcovid19_Mayo9.csv', encoding = "ISO-8859-1")
data_copy=data.copy()
```

En este caso:

- numpy se emplea para el manejo eficiente de arreglos.
- matplotlib se usa para la visualización de datos.
- pandas es la herramienta principal para la manipulación de dataframes.

Con read_csv() realizamos la lectura del dataset desde la ubicación donde fue almacenado.

3.1.1. Pre-visualización del dataset.

Ahora, podemos determinar el tamaño del dataset y realizar una primera visualización del contenido:

```
registro=len(data)
print(registro)
data.head(registro)
```

Este dataset contiene 8 columnas y 3824 filas. La figura 7 presenta una pre-visualización del dataset.

	Fecha de diagnóstico	Ciudad de residencia	Localidad de residencia	Edad	Sexo	Tipo de caso	Ubicación	Estado
0	10/04/2020	Bogotá	Usaquén	19	F	Importado	Casa	Recuperado
1	10/04/2020	Bogotá	Engativá	22	F	Importado	Casa	Recuperado
2	10/04/2020	Bogotá	Engativá	28	F	Importado	Casa	Recuperado
3	10/04/2020	Bogotá	Fontibón	36	F	Importado	Casa	Recuperado
4	10/04/2020	Bogotá	Kennedy	42	F	Importado	Casa	Recuperado
...	...	...	...	...	...	...	...	...
3819	7/05/2020	Bogotá	San Cristóbal	42	M	En estudio	Casa	Moderado
3820	7/05/2020	Bogotá	Teusaquillo	36	M	En estudio	Casa	Moderado
3821	7/05/2020	Bogotá	Engativá	56	F	En estudio	Hospital	Severo
3822	8/05/2020	Bogotá	Engativá	33	M	En estudio	Casa	Moderado
3823	8/05/2020	Bogotá	Sin Dato	41	M	En estudio	Hospital	Severo

3824 rows × 8 columns

Figura 7. Pre-visualización dataset COVID-19 Bogotá

3.1.2. Tipo de datos en el dataset

Continuamos con la exploración del dataset. Es importante no solo identificar cuántas columnas lo conforman, sino también conocer el **tipo de dato** asociado a cada una de ellas. Por ejemplo: ¿son variables numéricas o categóricas?

Para esto utilizamos el siguiente comando:

```
data.info()
```

El resultado se muestra en la Figura 8.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3824 entries, 0 to 3823
Data columns (total 8 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Fecha de diagnóstico    3823 non-null   object
 1   Ciudad de residencia    3824 non-null   object
 2   Localidad de residencia 3824 non-null   object
 3   Edad                    3824 non-null   int64
 4   Sexo                    3824 non-null   object
 5   Tipo de caso            3824 non-null   object
 6   Ubicación               3824 non-null   object
 7   Estado                  3824 non-null   object
dtypes: int64(1), object(7)
memory usage: 239.1+ KB
```

Figura 8. Tipo de datos del dataset COVID-19 Bogotá

Con esta información observamos que, de las ocho columnas disponibles, **solo una es numérica** (la variable *Edad*, de tipo entero). Las demás corresponden a **variables categóricas** (texto), incluida la **fecha de diagnóstico**, la cual inicialmente aparece como una cadena de caracteres y no como un tipo de dato temporal (datetime).

3.1.3. Distribución de los estados reportados del dataset

Posteriormente, podemos visualizar el estado en el que se encuentran los casos reportados de COVID-19 en el dataset. Para ello utilizamos:

```
from matplotlib import pyplot as plt
import seaborn as sns
data.groupby('Estado').size().plot(kind='barh', color=sns.palettes.mpl_palette('Dark2'))
plt.gca().spines[['top', 'right',]].set_visible(False)
```

Y obtenemos:

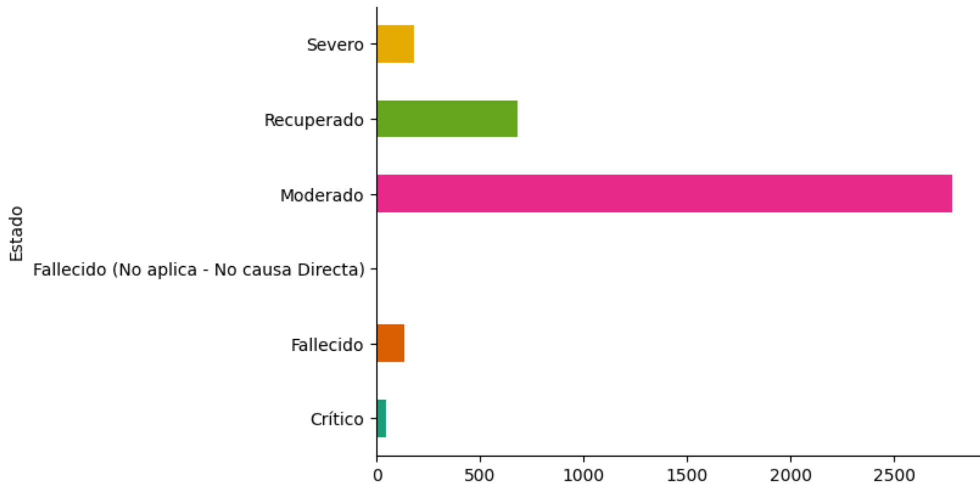

Figura 9. Distribución de los estados de los casos reportados en el dataset COVID-19 Bogotá

¿Por qué es importante este reporte?

Porque permite a las autoridades tomar decisiones basadas en la evolución del estado clínico de los casos. Para la fecha del dataset, la mayoría correspondía a casos moderados; sin embargo, con el paso del tiempo, la distribución cambió hacia casos severos y, lamentablemente, fallecidos.

Estas variaciones en las proporciones son fundamentales para activar alertas tempranas, ajustar políticas de salud pública y gestionar adecuadamente la capacidad hospitalaria.

3.1.4. Distribución de los estados reportados vs. Localidad

Vamos a realizar una gráfica que nos permita identificar el estado de los pacientes en cada una de las localidades de Bogotá. Utilizaremos una gráfica tipo **heatmap**:

```
from matplotlib import pyplot as plt
import seaborn as sns
import pandas as pd

plt.subplots(figsize=(7, 7))

df_2dhist = pd.DataFrame({
    x_label: grp['Estado'].value_counts()
    for x_label, grp in data.groupby('Localidad de residencia')
})
sns.heatmap(df_2dhist, cmap='crest')
plt.xlabel('Localidad de residencia')
_ = plt.ylabel('Estado')
```

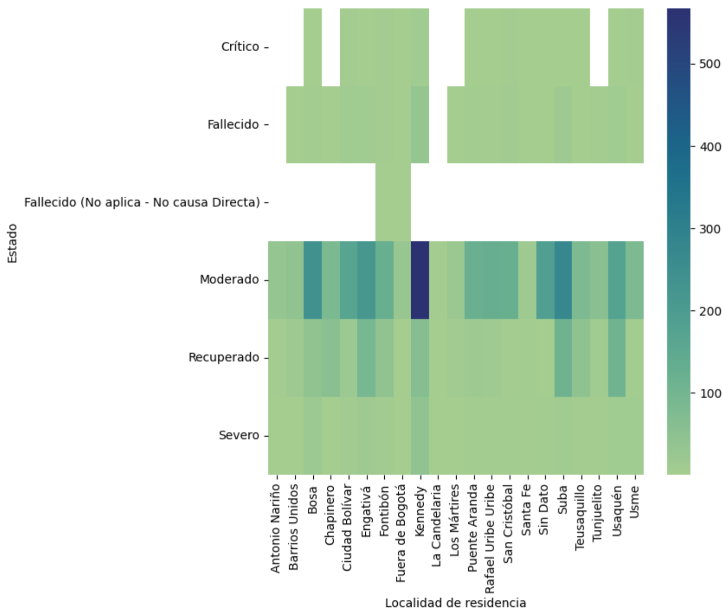

Figura 10. Distribución del estado vs. localidad de los casos reportados en el dataset COVID-19 Bogotá

¿Por qué es importante este reporte?

En su momento, esta información le permitió a la Alcaldesa tomar decisiones relacionadas con el "cierre" de localidades. Por ejemplo, para la fecha de este dataset, la localidad de Kennedy presentaba el mayor número de casos en estado Moderado, mientras que La Candelaria registraba la menor cantidad en ese mismo estado.

Si observamos el estado crítico, algunas localidades aún no tenían pacientes reportados en dicha condición.

Por esta razón, no era necesario cerrar todas las localidades de forma simultánea: se buscaba proteger la economía sin perder de vista el comportamiento real del impacto de la pandemia al interior de la ciudad.

3.1.5. Conversión de la fecha de diagnóstico

Como vimos previamente, la columna "Fecha de diagnóstico" es de tipo *object*, lo que indica que aún no se encuentra en un formato de fecha que nos permita trabajar con ella. Aunque los datasets con información temporal los abordaremos en el capítulo de Series de Tiempo, en este caso trabajaremos con esta columna sin convertirla en el índice del dataset, tratándola simplemente como un dato estructurado.

Queremos transformar esta columna, originalmente un texto, en un número que represente el consecutivo del año: 1 para el 1 de enero, 2 para el 2 de enero, y así sucesivamente. Para ello escribimos el siguiente código en Python:

```
data["Fecha de diagnóstico"] = pd.to_datetime(data["Fecha de diagnóstico"], dayfirst='True').dt.strftime("%Y%m%d") # quita / (ej. 20200410)
data['Fecha de diagnóstico'] = data['Fecha de diagnóstico'].astype('datetime64[ns]') # separa con - (ej. 2020-04-10)
data['Fecha de diagnóstico'] = data['Fecha de diagnóstico'].dt.dayofyear # convierte al día del año (ej. 101)
```

Este proceso se realiza en tres pasos: primero, convertimos el texto inicial a un formato de fecha indicando que el día aparece antes que el mes, y lo transformamos temporalmente a una cadena YYYYMMDD para eliminar caracteres separadores. Luego, reconvertimos esta cadena a un tipo .datetime64[ns] para que Python la interprete correctamente como fecha. Finalmente, extraemos el número de día correspondiente dentro del año mediante .dt.dayofyear, obteniendo un entero entre 1 y 365 (o 366 en años bisiestos). Este valor numérico facilita realizar análisis temporales sin necesidad de trabajar aún con series de tiempo completas.

Adicionalmente, vamos a renombrar la columna de Fecha de diagnóstico, dado que ahora corresponde es al "Día de diagnóstico"

```
data = data.rename(columns={'Fecha de diagnóstico':
'Día de diagnóstico'})
data.head(registro)
```

Y obtenemos:

	Día de diagnóstico	Ciudad de residencia	Localidad de residencia	Edad	Sexo	Tipo de caso	Ubicación	Estado
0	101.0	Bogotá	Usaquén	19	F	Importado	Casa	Recuperado
1	101.0	Bogotá	Engativá	22	F	Importado	Casa	Recuperado
2	101.0	Bogotá	Engativá	28	F	Importado	Casa	Recuperado
3	101.0	Bogotá	Fontibón	36	F	Importado	Casa	Recuperado
4	101.0	Bogotá	Kennedy	42	F	Importado	Casa	Recuperado
...	...	...	...	...	...	...	...	...
3819	128.0	Bogotá	San Cristóbal	42	M	En estudio	Casa	Moderado
3820	128.0	Bogotá	Teusaquillo	36	M	En estudio	Casa	Moderado
3821	128.0	Bogotá	Engativá	56	F	En estudio	Hospital	Severo
3822	129.0	Bogotá	Engativá	33	M	En estudio	Casa	Moderado
3823	129.0	Bogotá	Sin Dato	41	M	En estudio	Hospital	Severo

Figura 11. Pre-visualización del dataset posterior a la conversión de la fecha de diagnóstico

Al comparar la Figura 11 con la Figura 7, observamos que fechas como **10/04/2020** han sido reemplazadas por **101**, que corresponde al consecutivo del año. Contar ahora con esta columna en formato numérico nos permite realizar varios análisis exploratorios que antes no eran viables.

3.1.6. Distribución temporal de los estados

La Figura 9 nos mostraba la distribución de los estados, pero no nos aportaba información en relación a cómo había evolucionado a lo largo de los días.

Para ello, continuaremos con gráfica tipo *heatmap*, con el siguiente código:

```
from matplotlib import pyplot as plt
import seaborn as sns
import pandas as pd
plt.subplots(figsize=(8, 5))
df_2dhist = pd.DataFrame({
    x_label: grp['Fecha de diagnóstico'].value_counts()
    for x_label, grp in data.groupby('Estado')
})
sns.heatmap(df_2dhist, cmap='crest')
plt.xlabel('Ubicación')
_ = plt.ylabel('Dia de diagnóstico')
```

Este código genera un **mapa de calor** que muestra la frecuencia de aparición del **Día de diagnóstico** en cada uno de los estados reportados. Primero, se agrupan los datos por la columna *Estado* y, dentro de cada grupo, se calcula cuántas veces aparece cada valor de *Día de diagnóstico*. Con ello se construye un nuevo dataframe (df_2dhist), donde las columnas representan los estados y las filas los días del año. Finalmente, sns.heatmap() visualiza esta matriz utilizando la paleta *crest*, permitiendo identificar patrones temporales asociados a cada estado.

Obteniendo:

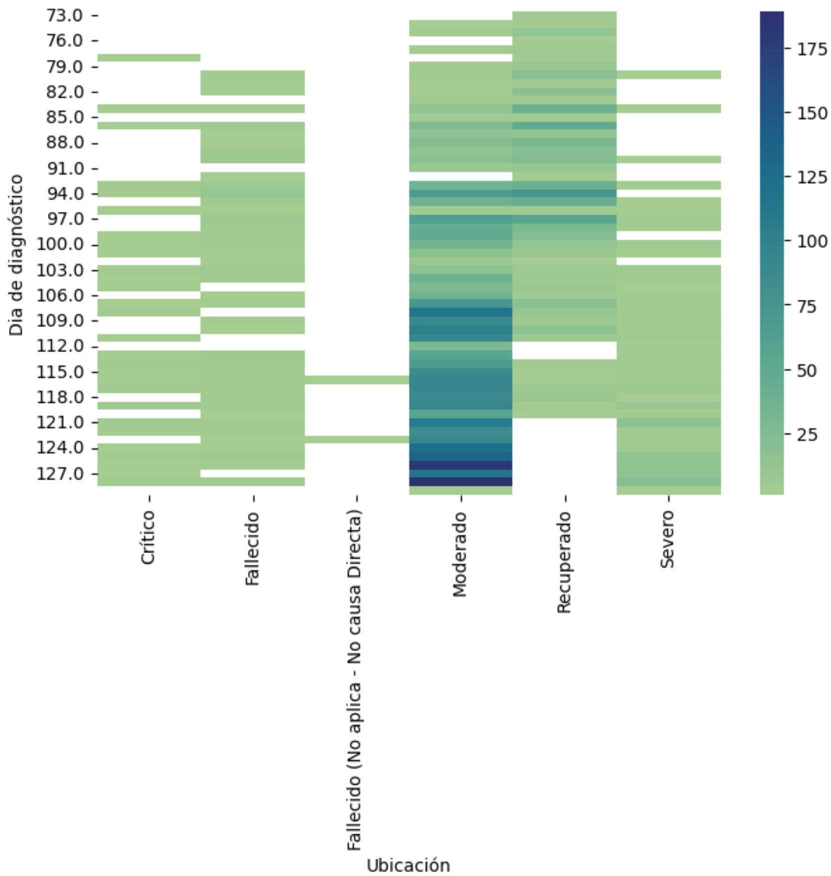

Figura 12. Distribución del estado vs. día de diagnóstico de los casos reportados en el dataset COVID-19 Bogotá

¿Por qué es importante este reporte?

Porque permite identificar cómo evolucionan los diferentes estados clínicos a lo largo del tiempo. Al observar la frecuencia del *Día de diagnóstico* para cada estado, es posible detectar patrones temporales, picos de contagio, momentos de mayor presión sobre el sistema de salud y periodos críticos que requieren intervención. Este tipo de visualización ayuda a comprender no solo cuántos casos hubo, sino **cuándo** ocurrieron y **cómo variaron entre estados**, ofreciendo información clave para la toma de decisiones en salud pública.

3.1.7. Casos de contagio por Género

Una de las preguntas que surgió al inicio de la pandemia era si los hombres o las mujeres eran más susceptibles al contagio. Para responderla, utilizamos la columna "Sexo" del dataset y determinamos cuántos registros corresponden a "M" y cuántos a "F".

El siguiente código calcula el número total de personas contagiadas por COVID-19 en Bogotá y desglosa esta cantidad por sexo, obteniendo tanto las cifras absolutas como el porcentaje que cada grupo representa dentro del total.

Posteriormente, se genera una gráfica de barras estética que compara visualmente el número de casos entre hombres y mujeres, incorporando colores diferenciados, etiquetas con valores y porcentajes, y un diseño limpio adecuado para el análisis exploratorio de datos.

```
cantidad_contagiados = len(data)
cantidad_contagiados_M = len(data[data["Sexo"] == "M"])
porc_M = cantidad_contagiados_M * 100 / cantidad_contagiados
cantidad_contagiados_F = len(data[data["Sexo"] == "F"])
porc_F = cantidad_contagiados_F * 100 / cantidad_contagiados

print('cantidad de contagiados por COVID en Bogotá:',
cantidad_contagiados)
print('cantidad de hombres contagiados por COVID:',
cantidad_contagiados_M,
    f'({porc_M:.2f}%)')
print('cantidad de mujeres contagiadas por COVID:',
cantidad_contagiados_F,
    f'({porc_F:.2f}%)')

plt.figure(figsize=(6,4))
labels = ['Hombres', 'Mujeres']
values = [cantidad_contagiados_M, cantidad_contagiados_F]
porcentajes = [porc_M, porc_F]

plt.figure(figsize=(6,5))
bars = plt.bar(labels, values,
```

```
                color=['#1aa6b7', '#a3bac3'],  # turquesa + gris azulado
                edgecolor='#444444')

plt.title('Casos confirmados de COVID-19 en Bogotá por
sexo', fontsize=12)
plt.ylabel('Número de casos', fontsize=16)

ax = plt.gca()
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

plt.grid(axis='y', linestyle='--', alpha=0.3)

for bar, pct in zip(bars, porcentajes):
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2,
             height,
             f'{int(height)}\n({pct:.1f}%)',
             ha='center', va='bottom',
             fontsize=9)

plt.tight_layout()
plt.show()
```

Y se obtiene la siguiente figura:

```
cantidad de contagiados por COVID en Bogotá: 3824
cantidad de hombres contagiados por COVID: 1931 (50.50%)
cantidad de mujeres contagiadas por COVID: 1893 (49.50%)
<Figure size 600x400 with 0 Axes>
```

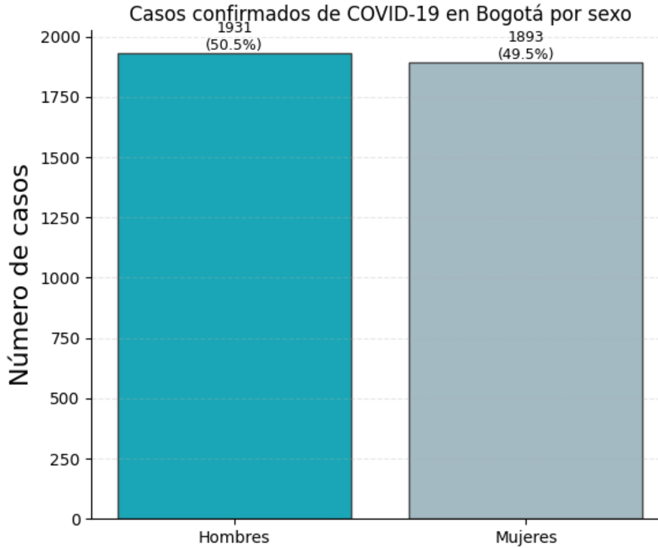

Figura 13. Distribución de casos reportados por género

De acuerdo con la figura anterior, se puede derrumbar el mito que los hombres se contagiaban mucho más que las mujeres.

3.1.8. Casos de contagio por Día de Diagnóstico

La siguiente inquietud que se generó tanto en la comunidad como en las autoridades de salud, era que tan rápido estaba creciendo la tasa de contagios. Es decir, a medida que pasaban los días, cuantos nuevos casos iban apareciendo. Para ello, utilizamos el siguiente código en Python:

```
# Número de días únicos reportados
num_dias = data['Día de diagnóstico'].nunique()

# Histograma estilo "elegante"
plt.figure(figsize=(10,4))
plt.hist(data['Día de diagnóstico'],
    bins=num_dias,
    color='#1aa6b7',       # Turquesa elegante
```

```
            edgecolor='#2e4756',   # Gris azul oscuro para delinear
            alpha=0.85)

    plt.xlabel('Fecha de diagnóstico', fontsize=10)
    plt.ylabel('Número de casos', fontsize=10)
    plt.title('Histograma de casos diarios de COVID-19 en
    Bogotá', fontsize=12)
    # Estilo visual más limpio
    ax = plt.gca()
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    plt.xticks(rotation=45)
    plt.grid(axis='y', linestyle='--', alpha=0.3)
    plt.tight_layout()
    plt.show()
```

Iniciamos calculando cuántos días únicos aparecen en la columna “Día de diagnóstico” con las instrucción nunique(), lo cual permite identificar cuántas fechas diferentes de reporte existen en el dataset. Con este valor, se genera un histograma en el que cada *bin* representa exactamente un día de la pandemia. Esto permite visualizar la distribución diaria de casos sin agrupar varios días en un mismo intervalo. Finalmente, se añaden etiquetas a los ejes, un título descriptivo y se rota el texto del eje horizontal para facilitar la lectura de las fechas, mostrando así un histograma claro y detallado del número de casos por día.

Obtenemos la siguiente gráfica:

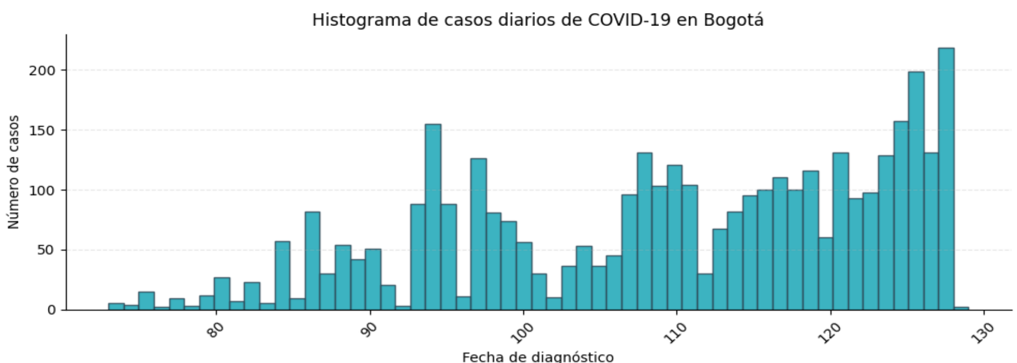

Figura 14. Histograma de casos diarios de COVID-19 en Bogotá

¿Por qué es importante este reporte?

El histograma permite observar de manera directa la evolución de los contagios diarios de COVID-19 en Bogotá, mostrando cómo se pasó de una fase inicial con pocos casos a un crecimiento progresivo y sostenido hacia el final del periodo analizado. Este tipo de visualización es fundamental en un EDA porque hace visibles patrones, picos y tendencias que no se aprecian al revisar la tabla de datos, facilitando así la identificación de momentos críticos, cambios en la dinámica de transmisión y posibles puntos de interés para un análisis más profundo o una toma de decisiones informada. Por ejemplo, al inicio de la pandemia (alrededor del día 70), la cantidad de casos reportados por día era muy baja (menor de 20); posteriormente, se observa un primer pico alrededor del día 95, seguido de una disminución en los contagios diarios (muy seguramente como resultado de las medidas de restricción y cierres por localidades). Sin embargo, hacia el día 120 del año (es decir, iniciando abril) se evidencia un aumento considerable en el número de contagios, superando los 100 casos diarios y marcando una fase de expansión más acelerada del virus.

3.1.9. Evolución acumulada de casos de COVID-19 en Bogotá

Como complemento a la gráfica anterior, podemos también dibujar el acumulado de casos, y así fácilmente observar cómo va cambiando la pendiente de casos acumulados, es decir, que tan rápido está creciendo en un momento dado frente al inicio de la pandemia.

Para ello, primero ordenamos la columna “Día de diagnóstico” de manera cronológica y luego agrupamos los registros para contabilizar cuántos casos corresponden a cada día. A continuación, aplicamos una suma acumulada para obtener el total progresivo de contagios a lo largo del tiempo. Finalmente, graficamos estos valores en una curva que permite visualizar cómo se incrementaron los casos de forma acumulada durante el periodo analizado.

```
# Ordenar por fecha
data = data.sort_values('Día de diagnóstico')

# Contar casos diarios
casos_por_dia = data.groupby('Día de diagnóstico').size()
```

```
# Calcular los contagios acumulados
acumulado = casos_por_dia.cumsum()

plt.figure(figsize=(10,4))
plt.plot(acumulado.index, acumulado.values,
        color='#1aa6b7', linewidth=2.5)

plt.xlabel('Día de diagnóstico', fontsize=10)
plt.ylabel('Contagios acumulados', fontsize=10)
plt.title('Evolución acumulada de casos de COVID-19 en Bogotá',
fontsize=12)

ax = plt.gca()
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

plt.grid(axis='y', linestyle='--', alpha=0.3)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

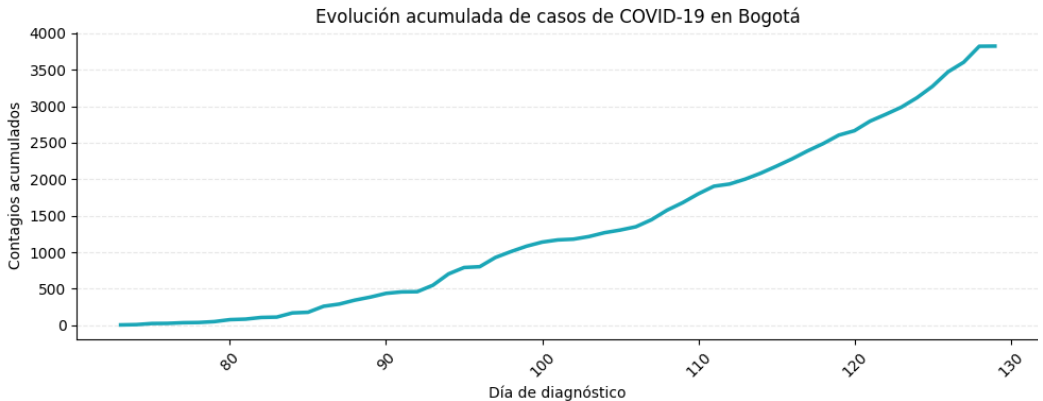

Figura 15. Evolución acumulada de casos de COVID-19 en Bogotá

3.1.10. Distribución de casos de COVID-19 por edad

Cuando hablamos de distribución, normalmente nos referimos a un histograma. Por ello, nuevamente utilizaremos este tipo de gráfica para nuestra figura, así:

```
# Asegurar que la columna Edad sea numérica
data['Edad'] = pd.to_numeric(data['Edad'], errors='coerce')

# Calcular número de bins: uno por cada edad en el rango del dataset
edad_min = int(data['Edad'].min())
edad_max = int(data['Edad'].max())
num_bins = edad_max - edad_min

plt.figure(figsize=(10,4))
plt.hist(
    data['Edad'].dropna(),
    bins=num_bins,
    color='#1aa6b7',      # turquesa
    edgecolor='#2e4756',  # gris oscuro
    alpha=0.85
)

plt.xlabel('Edad', fontsize=10)
plt.ylabel('Número de casos', fontsize=10)
plt.title('Distribución de casos de COVID-19 por edad', fontsize=12)

ax = plt.gca()
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

plt.grid(axis='y', linestyle='--', alpha=0.3)
plt.tight_layout()
plt.show()
```

Comenzamos verificando que los datos de la columna "Edad" sean de tipo numérico y, posteriormente, calculamos la edad mínima y máxima reportadas en el dataset. Con estos valores determinamos la cantidad de barras

del histograma, de modo que cada *bin* corresponda a una edad específica. La parte final del código se encarga de ajustar la estética del gráfico para obtener una visualización más clara y profesional, como se presenta en la Figura 16.

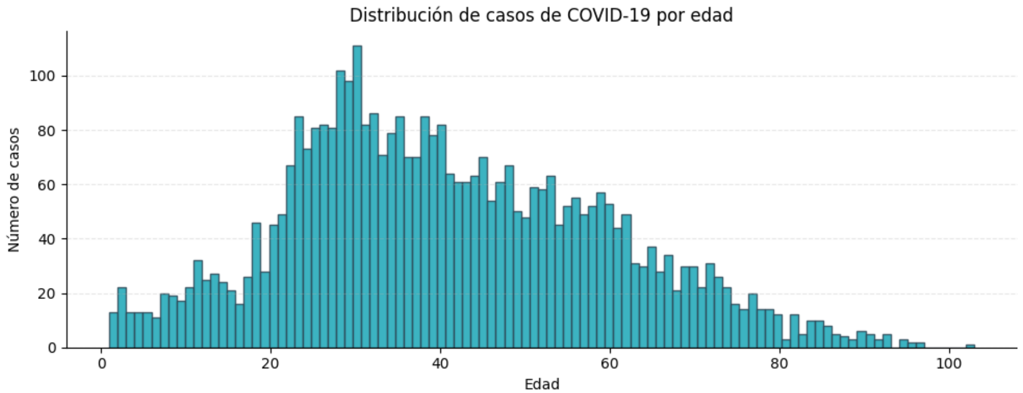

Figura 16. Distribución de casos de COVID-19 por edad

Como complemento a la información obtenida en la gráfica anterior, ahora podemos determinar el total de casos por rango de edad y su porcentaje respecto al total de casos reportados a la fecha. Para ello utilizamos un código que inicialmente define los *bins* para los rangos, por ejemplo: bins = [0, 20, 40, 60, 80, 100], y posteriormente agrupa los datos según estos intervalos utilizando la instrucción pd.cut.

Para nuestro caso:

- data['Edad'] es la columna original con valores numéricos.
- bins indica los límites de cada intervalo (0 a 20, 20 a 40, etc.).
- labels asigna un nombre a cada intervalo para facilitar su interpretación.
- right=False significa que los intervalos son cerrados por la izquierda y abiertos por la derecha; por ejemplo:
    - **0–20** incluye edades desde 0 hasta 19,
    - **20–40** incluye desde 20 hasta 39,y así sucesivamente.

El resultado es una nueva columna llamada **GrupoEdad**, donde cada persona queda clasificada en su rango correspondiente. Posteriormente, el código utiliza value_counts() para contar cuántos casos hay en cada intervalo y luego calcula su porcentaje respecto al total del dataset. Finalmente, estos resultados se organizan en un dataframe que muestra tanto el número de casos como el porcentaje asociado a cada grupo etario.

El código se presenta a continuación:

```
# Definir rangos de edad
bins = [0, 20, 40, 60, 80, 100]
labels = ['0-20', '20-40', '40-60', '60-80', '80-100']

# Crear una nueva columna con el grupo de edad
data['GrupoEdad'] = pd.cut(data['Edad'], bins=bins,
labels=labels, right=False)

# Calcular total de casos por grupo
casos_por_grupo = data['GrupoEdad'].value_counts().sort_index()

# Calcular porcentajes
porcentajes = (casos_por_grupo / len(data)) * 100

# Unir resultados en una sola tabla
resultado = pd.DataFrame({
    'Casos': casos_por_grupo,
    'Porcentaje (%)': porcentajes.round(2)
})
print(resultado)
```

Y se obtiene el siguiente resultado:

```
           Casos  Porcentaje (%)
GrupoEdad
0-20         408           10.67
20-40       1580           41.32
40-60       1171           30.62
60-80        566           14.80
80-100        98            2.56
```

Figura 17. Reporte casos por rango de edad y porcentaje

¿Por qué es importante este reporte?

Nos permite no solo identificar las edades con mayor número de contagios, sino también comprender las dinámicas de exposición asociadas a cada grupo etario. La mayor concentración de casos entre los **20 y 40 años** coincide con la población en edad laboral activa, lo que indica que el incremento de contagios en este grupo no se debe únicamente a la edad, sino a sus **mayores niveles de exposición**. Durante la pandemia, quienes debían desplazarse para trabajar, utilizar transporte público o desempeñar actividades presenciales estuvieron más expuestos al virus.

Para las autoridades de salud, esta distribución señala que las estrategias de control deben considerar **los patrones de movilidad y la actividad laboral**, priorizando medidas para trabajadores presenciales y sectores esenciales. En este sentido, la dinámica de contagio estuvo fuertemente influenciada por la exposición y no exclusivamente por la vulnerabilidad etaria.

3.1.11. Distribución de fallecidos por COVID-19 según edad

Continuando con nuestro EDA, es importante identificar cuántas personas han fallecido según su edad. Para ello, lo primero que hacemos es filtrar del dataset aquellas filas en las que la columna *Estado* corresponde a "Fallecido". Posteriormente, calculamos nuevamente el número de *bins* a partir de la edad mínima y máxima registradas en este grupo, y utilizamos una gráfica tipo histograma para visualizar la distribución de edades de las personas fallecidas.

```
# Filtrar y crear copia para evitar warnings
fallecidos = data[data['Estado'] == 'Fallecido'].copy()

# Calcular número de bins (no necesariamente es el mismo
del grupo completo - todos los estados)
edad_min = int(fallecidos['Edad'].min())
edad_max = int(fallecidos['Edad'].max())
num_bins_fallecidos = edad_max - edad_min

plt.figure(figsize=(10,4))
plt.hist(
    fallecidos['Edad'].dropna(),
```

```
        bins=num_bins_fallecidos,
        color='#1aa6b7',
        edgecolor='#2e4756',
        alpha=0.85
    )

    plt.xlabel('Edad', fontsize=10)
    plt.ylabel('Número de casos', fontsize=10)
    plt.title('Distribución de fallecidos por COVID-19 según
    edad', fontsize=12)

    ax = plt.gca()
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)

    plt.grid(axis='y', linestyle='--', alpha=0.3)
    plt.tight_layout()
    plt.show()
```

Obteniendo:

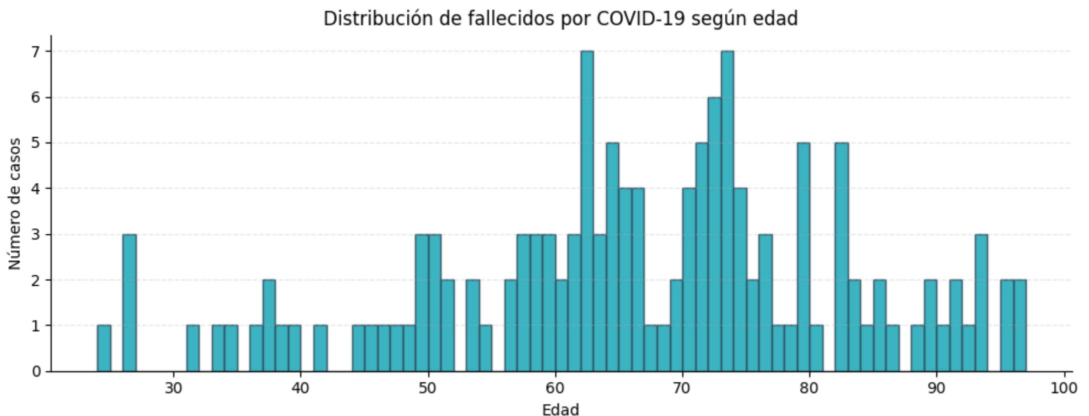

Figura 18. Distribución de fallecidos por COVID-19 según edad

Como complemento, calculamos la cantidad de fallecidos por grupo de

edad, y su porcentaje respecto al total de fallecidos, así:

```
# Clasificar en grupos de edad
fallecidos['GrupoEdad'] = pd.cut(
    fallecidos['Edad'],
    bins=bins,
    labels=labels,
    right=False
)

# Calcular cantidad de fallecidos por grupo
fallecidos_por_grupo = fallecidos['GrupoEdad'].value_counts().sort_
index()

# Calcular porcentaje respecto al total de fallecidos
porcentajes = (fallecidos_por_grupo / len(fallecidos)) * 100

# Crear tabla final
tabla_fallecidos = pd.DataFrame({
    'Fallecidos': fallecidos_por_grupo,
    'Porcentaje (%)': porcentajes.round(2)
})

print(tabla_fallecidos)
```

Obteniendo:

```
           Fallecidos  Porcentaje (%)
GrupoEdad
0-20                0            0.00
20-40              12            8.82
40-60              28           20.59
60-80              70           51.47
80-100             26           19.12
```

Figura 19. Distribución de fallecidos por COVID-19 según edad

Si comparamos los resultados de la Figura 19 con los de la Figura

17 nos damos cuenta que el mayor porcentaje de fallecidos se encuentra en el grupo de 60-80 años (51.47%), pese a que solamente representa el 14.8% de los casos reportados de COVID-19. Es decir, la probabilidad de muerte para ese grupo de edad es significativamente más alta que para cualquiera de los otros grupos.

3.1.12. Casos de contagio por Localidad, Fallecidos por Localidad, y Tasa de muerte por Localidad

Vamos ahora a realizar EDA por localidad. De alguna manera podemos “replicar” lo que hicimos con el dataset completo, pero ahora localidad por localidad. Iniciaremos por calcular la cantidad de contagiados por localidad, con el siguiente código:

```
casos_por_localidad = data['Localidad de residencia'].
value_counts().sort_values(ascending=True)

plt.figure(figsize=(10,6))
plt.barh(
    casos_por_localidad.index,
    casos_por_localidad.values,
    color='#1aa6b7',
    edgecolor='#2e4756',
    alpha=0.85
)
plt.xlabel('Número de casos', fontsize=10)
plt.ylabel('Localidad', fontsize=10)
plt.title('Número de casos de COVID-19 por localidad', fontsize=12)

ax = plt.gca()
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

plt.grid(axis='x', linestyle='--', alpha=0.3)
plt.tight_layout()
plt.show()
```

Cuando utilizamos **value_counts()**, podemos rápidamente realizar un

conteo para cada una de las opciones de la columna a la que estamos haciendo referencia. Este método es mucho más eficiente que realizar filtros, pues en ese caso tendríamos que escribir una línea de código por cada localidad.

Como resultado, obtenemos:

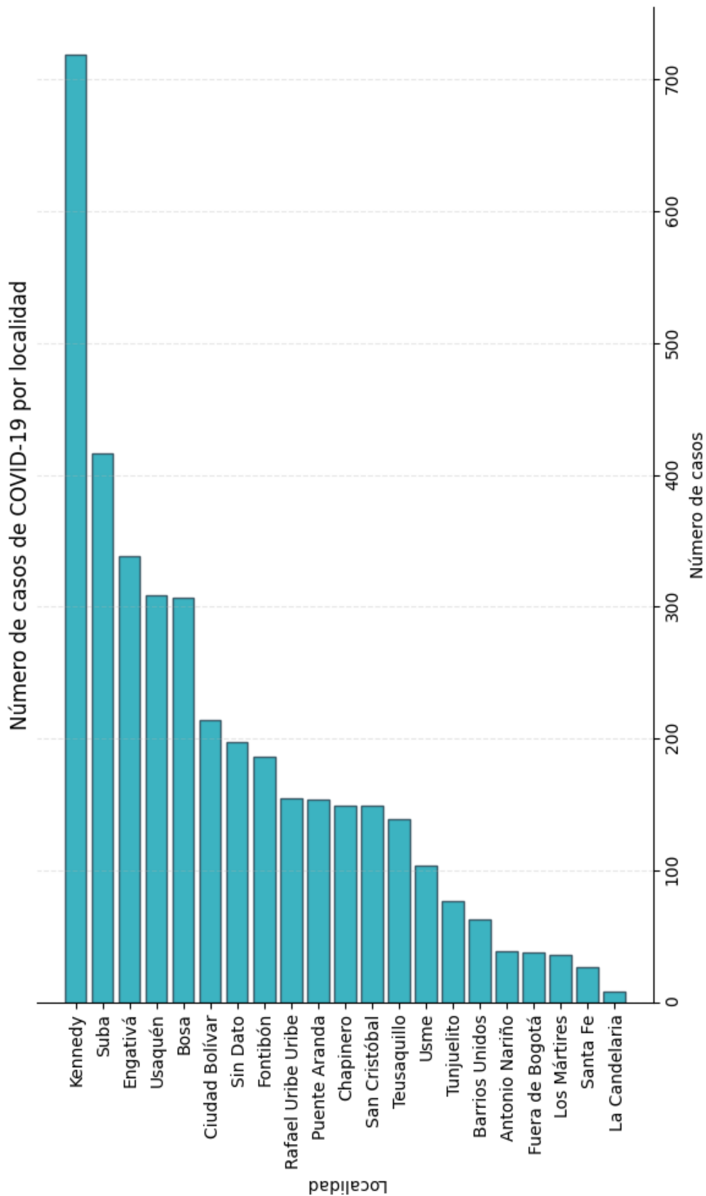

Figura 20. Casos de contagio por Localidad

Las localidades donde se concentra la mayor cantidad de contagios: **Kennedy**, seguida por **Suba** y **Engativá**. En el rango medio aparecen **Usaquén, Bosa, Ciudad Bolívar, Fontibón** y **San Cristóbal**, lo que evidencia una distribución amplia del virus en sectores con alta movilidad y actividad económica. Por su parte, localidades como **La Candelaria, Santa Fe, Los Mártires** y **Fuera de Bogotá** presentan un número significativamente menor de casos, lo cual puede estar relacionado con su menor densidad poblacional o con la naturaleza predominantemente administrativa o comercial de estas zonas.

Ahora, vamos a calcular cuántos fallecidos se tienen en cada una de las Localidades. Primero, realizamos un filtro para crear un subdataset correspondiente a "Fallecido" de la columna "Estado". Posteriormente, con **value_counts()** calculamos los fallecidos por Localidad.

```
fallecidos = data[data['Estado'] == 'Fallecido'].copy()

# Contar fallecidos por localidad
fallecidos_por_localidad = fallecidos['Localidad de residencia'].value_
counts().sort_values(ascending=True)

# Crear gráfica
plt.figure(figsize=(10,6))
plt.barh(
    fallecidos_por_localidad.index,
    fallecidos_por_localidad.values,
    color='#1aa6b7',       # turquesa del libro
    edgecolor='#2e4756',   # gris oscuro
    alpha=0.85
)

plt.xlabel('Número de fallecidos', fontsize=10)
plt.ylabel('Localidad', fontsize=10)
plt.title('Número de fallecidos por COVID-19
según localidad', fontsize=12)

ax = plt.gca()
ax.spines['top'].set_visible(False)
```

```
    ax.spines['right'].set_visible(False)

    plt.grid(axis='x', linestyle='--', alpha=0.3)
    plt.tight_layout()
    plt.show()
```

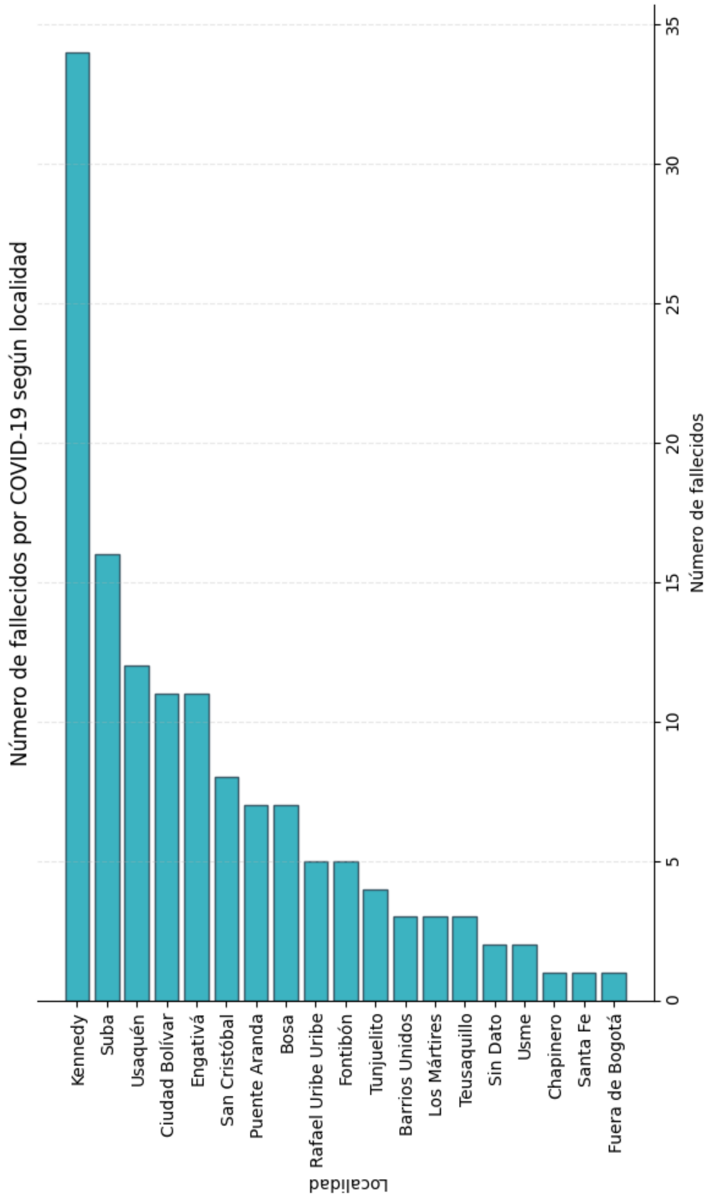

Figura 21. Número de fallecidos por COVID-19 según localidad

Localidades como Kennedy, Suba y Usaquén registran el mayor número de muertes, lo que puede relacionarse con su alta densidad poblacional, mayor volumen de movilidad diaria o características socioeconómicas que incrementan la exposición o dificultan el acceso oportuno a servicios de salud. En contraste, localidades como Santa Fe, Chapinero, Teusaquillo, Usme y Fuera de Bogotá presentan una mortalidad significativamente menor.

3.1.13. Tasa de muerte por Localidad

Finalizaremos nuestro EDA con la tasa de muerte por localidad. Esa tasa la calculamos de acuerdo con la siguiente ecuación matemática

$$Tasa\ de\ muerte\ por\ localidad = \frac{Número\ de\ fallecidos\ en\ la\ localidad}{Número\ total\ de\ contagios\ en\ la\ localidad} * 100$$

Utilizamos el siguiente código en Python:

```
# 1. Total de contagiados por localidad
contagiados_localidad = data['Localidad de residencia'].value_counts()

# 2. Total de fallecidos por localidad
fallecidos_localidad = data[data['Estado'] == 'Fallecido']['Localidad de residencia'].value_counts()

# 3. Unir ambas series en un solo dataframe
df_tasa = pd.DataFrame({
    'Contagiados': contagiados_localidad,
    'Fallecidos': fallecidos_localidad
})

# 4. Reemplazar NaN (localidades sin fallecidos) por 0
df_tasa = df_tasa.fillna(0)

# 5. Calcular la tasa de muerte
df_tasa['Tasa de muerte (%)'] = (df_tasa['Fallecidos']
/ df_tasa['Contagiados']) * 100
df_tasa['Tasa de muerte (%)'] = df_tasa['Tasa de muerte (%)'].round(2)
```

```
# 6. Ordenar de menor a mayor tasa
df_tasa = df_tasa.sort_values(by='Tasa de muerte (%)')

print(df_tasa)
```

Inicialmente, calculamos la **tasa de muerte por localidad**, combinando la información de contagiados y fallecidos reportados en el dataset. En primer lugar, se obtiene el total de contagiados por localidad utilizando value_counts(), que permite contar cuántos registros existen en cada categoría de la columna *Localidad de residencia*. Luego, se filtran los casos cuyo estado es “Fallecido” y se repite el proceso para obtener el número de fallecidos por localidad.

A continuación, ambas series se integran en un mismo *dataframe*, lo que facilita comparar el número total de contagiados con el número de fallecidos en cada zona. Dado que algunas localidades pueden no registrar fallecidos, se reemplazan los valores faltantes con cero mediante **fillna(0)** para evitar errores en los cálculos. Posteriormente, se calcula la tasa de muerte dividiendo el número de fallecidos entre el total de contagiados y multiplicando por 100, expresándola como porcentaje. Esta tasa se redondea a dos decimales para mejorar su presentación. Finalmente, el *dataframe* se ordena de menor a mayor tasa, lo que permite identificar rápidamente qué localidades presentan una mortalidad proporcional más alta.

Y obtenemos:

Localidad de residencia	Contagiados	Fallecidos	Tasa de muerte (%)
Antonio Nariño	39	0.0	0.00
La Candelaria	8	0.0	0.00
Chapinero	149	1.0	0.67
Sin Dato	197	2.0	1.02
Usme	104	2.0	1.92
Teusaquillo	139	3.0	2.16
Bosa	307	7.0	2.28
Fuera de Bogotá	38	1.0	2.63
Fontibón	186	5.0	2.69
Rafael Uribe Uribe	155	5.0	3.23
Engativá	338	11.0	3.25
Santa Fe	27	1.0	3.70
Suba	416	16.0	3.85
Usaquén	309	12.0	3.88
Puente Aranda	154	7.0	4.55
Kennedy	719	34.0	4.73
Barrios Unidos	63	3.0	4.76
Ciudad Bolívar	214	11.0	5.14
Tunjuelito	77	4.0	5.19
San Cristóbal	149	8.0	5.37
Los Mártires	36	3.0	8.33

Figura 22. Tasa de muerte por Localidad

¿Para qué sirve este reporte?

Este reporte permite identificar la **tasa de muerte por localidad**, un indicador mucho más informativo que observar únicamente el número de contagiados o de fallecidos por separado. Al relacionar ambos valores, es posible determinar en qué zonas de la ciudad el impacto del COVID-19 fue proporcionalmente más alto. Por ejemplo, localidades como **Los Mártires**, **San Cristóbal** y **Tunjuelito** presentan tasas de muerte más elevadas, a pesar de no ser las que registran más contagios. Esto sugiere posibles diferencias en acceso a servicios de salud, condiciones socioeconómicas, comorbilidades o tiempos de atención. En contraste, localidades con un alto número de contagios como **Kennedy**, **Suba** y **Engativá** muestran una tasa de mortalidad proporcionalmente menor. Este análisis permite a las autoridades orientar estrategias de intervención focalizada, asignación de recursos y políticas de prevención según la vulnerabilidad específica de cada territorio.

3.2. Caso de estudio 2: EDA y FE al Bank Churn dataset

Para este segundo caso de estudio, cambiaremos de dataset a una temática distinta. El dataset anterior correspondía al sector salud, y el de este caso al sector bancario. Aunque realizaremos algunas acciones de Análisis Exploratorio de Datos (EDA), el mayor énfasis de este caso estará en la Ingeniería de Características (Feature Engineering, FE).

La información de este dataset es:

- **RowNumber**: Número de fila dentro del dataset. No aporta información analítica; únicamente indica la posición del registro.
- **CustomerId**: Identificador único asignado a cada cliente. Sirve para distinguir un cliente de otro, pero no se utiliza como variable predictiva.
- **Surname**: Apellido del cliente. En la mayoría de modelos, este atributo no aporta información relevante y suele eliminarse para evitar sesgos.
- **CreditScore**: Puntaje crediticio del cliente. Un valor numérico que indica la solvencia o comportamiento esperado frente al crédito. Valores más altos representan mayor estabilidad financiera.
- **Geography**: País de residencia del cliente (por ejemplo: France, Spain, Germany). Es una variable categórica que permite analizar diferencias de comportamiento entre regiones.
- **Gender**: Género del cliente (Male / Female). Variable categórica usada en algunos análisis demográficos.
- **Age**: Edad del cliente. Es una de las variables más influyentes en el análisis de abandono bancario (churn).
- **Tenure**: Número de años que el cliente lleva vinculado al banco. Permite conocer la antigüedad y evaluar la fidelidad.
- **Balance**: Saldo actual de la cuenta del cliente. Un valor numérico que representa los fondos disponibles en el banco.
- **NumOfProducts**: Número de productos que el cliente tiene con el banco (por ejemplo, tarjetas, cuentas, créditos). Indica el nivel de relación comercial.

- **HasCrCard**: Indica si el cliente posee tarjeta de crédito.

  1 = Sí

  0 = No

- **IsActiveMember**: Indica si el cliente es considerado activo dentro del banco.

  1 = Activo

  0 = Inactivo

- **EstimatedSalary**: Salario estimado del cliente. Es una variable numérica que representa su nivel de ingresos aproximado.

- **Exited**: Variable objetivo del dataset. Indica si el cliente abandonó el banco (churn).

  1 = El cliente se fue

  0 = El cliente permaneció

Realizamos importe de librerías, y pre-visualización, como vimos en el capítulo 3.1. Nuestro dataset contiene 14 columnas y 10.002 filas.

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exite
0	1	15634602	Hargrave	619	France	Female	42.0	2	0.00	1	1.0	1.0	101348.88	[illegible]
1	2	15647311	Hill	608	Spain	Female	41.0	1	83807.86	1	0.0	1.0	112542.58	[illegible]
2	3	15619304	Onio	502	France	Female	42.0	8	159660.80	3	1.0	0.0	113931.57	[illegible]
3	4	15701354	Boni	699	France	Female	39.0	1	0.00	2	0.0	0.0	93826.63	[illegible]
4	5	15737888	Mitchell	850	Spain	Female	43.0	2	125510.82	1	NaN	1.0	79084.10	[illegible]
...	...	...	...	...	...	...	...	...	...	...	...	...	...	..
9997	9998	15584532	Liu	709	France	Female	36.0	7	0.00	1	0.0	1.0	42085.58	[illegible]
9998	9999	15682355	Sabbatini	772	Germany	Male	42.0	3	75075.31	2	1.0	0.0	92888.52	[illegible]
9999	9999	15682355	Sabbatini	772	Germany	Male	42.0	3	75075.31	2	1.0	0.0	92888.52	[illegible]
10000	10000	15628319	Walker	792	France	Female	28.0	4	130142.79	1	1.0	0.0	38190.78	[illegible]
10001	10000	15628319	Walker	792	France	Female	28.0	4	130142.79	1	1.0	0.0	38190.78	[illegible]

10002 rows × 14 columns

Figura 23. Pre-visualización dataset Bank churn

3.2.1. Análisis Exploratorio de Datos Bank churn dataset

Iniciamos con el tipo de dato de cada columna del dataset, con la siguiente instrucción:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10002 entries, 0 to 10001
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   RowNumber        10002 non-null  int64
 1   CustomerId       10002 non-null  int64
 2   Surname          10002 non-null  object
 3   CreditScore      10002 non-null  int64
 4   Geography        10001 non-null  object
 5   Gender           10002 non-null  object
 6   Age              10001 non-null  float64
 7   Tenure           10002 non-null  int64
 8   Balance          10002 non-null  float64
 9   NumOfProducts    10002 non-null  int64
 10  HasCrCard        10001 non-null  float64
 11  IsActiveMember   10001 non-null  float64
 12  EstimatedSalary  10002 non-null  float64
 13  Exited           10002 non-null  int64
dtypes: float64(5), int64(6), object(3)
memory usage: 1.1+ MB
```

Figura 24. Tipo de datos Bank churn dataset

Es recomendable iniciar la tarea de EDA con este paso, dado que nos permite identificar el tipo de datos de cada columna, así como la cantidad de datos faltantes en cada una de ellas. Cuando nuestro alcance como científico de datos supera el EDA y queremos realizar modelamiento, necesitamos que todas las columnas sean numéricas o booleanas, por lo que debemos aplicar ingeniería de características en las que no lo sean. Para este caso de estudio, tenemos tres columnas que no son numéricas.

Por otro lado, podemos deducir que existen columnas con datos faltantes, dado que no todas las columnas tienen los 10.002 valores, algunas tienen un dato faltante.

Como segundo paso de nuestro EDA vamos a conocer el comportamiento de la variable de salida, que corresponde a Exited. Queremos saber si nuestro dataset es balanceado o no, y cuantos casos corresponden a cada categoría. Este problema corresponde a una **clasificación binaria**.

Escribimos el siguiente código en Python.

```
from matplotlib import pyplot as plt
import seaborn as sns

# Agrupar y contar
counts = data.groupby('Exited').size()

# Crear gráfico
plt.figure(figsize=(6,4))
ax = counts.plot(kind='barh', color=sns.palettes.mpl_palette('Dark2'))
ax.spines[['top', 'right']].set_visible(False)

# Añadir las cantidades encima de cada barra
for i, value in enumerate(counts):
    plt.text(value + 1,        # posición x (ligeramente a la derecha del valor)
             i,                # posición y (alineada con la barra)
             str(value),       # texto que se muestra
             va='center',      # alineación vertical
             fontsize=10)

plt.xlabel('Número de casos')
plt.ylabel('Exited')
plt.title('Distribución de clientes según Exited')

plt.tight_layout()
plt.show()
```

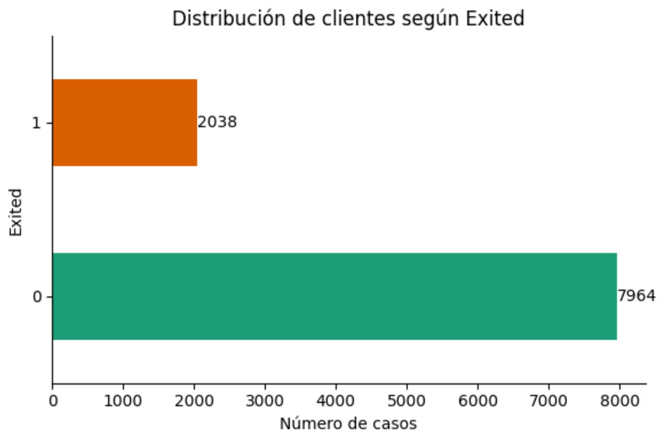

Figura 25. Distribución de casos Bank churn dataset

Y la proporción por clase

```
# PROPORCIÓN POR CLASE
data['Exited'].value_counts(normalize=True)
```

De acuerdo con la variable *Exited*, se observa que el 79,63 % de los clientes permanece activo, mientras que el 20,37 % ha cancelado su cuenta.

3.2.2. Transformación de columnas del dataset: de object a integer

Vamos a empezar con la columna "Gender". Inicialmente, verificamos la cantidad de opciones dentro de esta columna, así:

```
# CANTIDAD DE GENEROS DENTRO DEL DATASET
num_generos_distintos = data['Gender'].nunique()
print(f"Número de generos distintos: {num_generos_distintos}")
```

Y nos aparece que hay 2 géneros distintos. De la pre-visualización podemos observar que las dos opciones son: Female y Male.

Posteriormente, convertimos esta columna a entero realizando una asignación de "1" cuando es Male y de "0" cuando es Female. Adicionalmente contamos la cantidad de no nulos y lo visualizamos.

```
# CONVERTIR LA COLUMNA "GENDER" DE OBJECT A ENTERO
```

```
data['Gender'] = data['Gender'].apply(lambda x: 1 if x == 'Male' else 0)
no_nulos_genero = data['Gender'].count()
print(no_nulos_genero)
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	15634602	Hargrave	619	France	0	42.0	2	0.00	1	1.0	1.0	101348.88	1
1	2	15647311	Hill	608	Spain	0	41.0	1	83807.86	1	0.0	1.0	112542.58	0
2	3	15619304	Onio	502	France	0	42.0	8	159660.80	3	1.0	0.0	113931.57	1
3	4	15701354	Boni	699	France	0	39.0	1	0.00	2	0.0	0.0	93826.63	0
4	5	15737888	Mitchell	850	Spain	0	43.0	2	125510.82	1	NaN	1.0	79084.10	0
5	6	15574012	Chu	645	Spain	1	44.0	8	113755.78	2	1.0	0.0	149756.71	1
6	7	15592531	Bartlett	822	NaN	1	50.0	7	0.00	2	1.0	1.0	10062.80	0
7	8	15656148	Obinna	376	Germany	0	29.0	4	115046.74	4	1.0	0.0	119346.88	1
8	9	15792365	He	501	France	1	44.0	4	142051.07	2	0.0	NaN	74940.50	0
9	10	15592389	H?	684	France	1	NaN	2	134603.88	1	1.0	1.0	71725.73	0

Figura 26. Columna "Gender" transformada, dataset Bank churn.

Ahora, procedemos con la columna “Geograhphy”. De nuevo, lo primero que hacemos es revisar los valores al interior de la columna.

```
# CANTIDAD DE PAISES DENTRO DEL DATASET

num_paises_distintos = data['Geography'].nunique()
print(f"Número de paises distintos: {num_paises_distintos}")

# CANTIDAD DE VECES QUE APARECE CADA PAIS

vu_country_counts = data['Geography'].value_counts()
print(vu_country_counts)
```

Dado que no existe jerarquía entre los diferentes valores, la transformación adecuada para este caso corresponde a *one-hot ecoding*. Como son tres países diferentes, entonces se crearán tres nuevas columnas donde solamente “se enciende” una columna a la vez por registro.

La transformación y la pre-visualización la realizados con el siguiente código:

```
# CONVERTIR LA COLUMNA "GEOGRAPHY",
UTILIZANDO ONE-HOT ENCODING
data = pd.get_dummies(data, columns=['Geography'], prefix='Country')
# Mostrar el resultado
data.head(registro)
```

Geography
France
Spain
France
France
Spain
Spain
NaN
Germany
France
France

Country_France	Country_Germany	Country_Spain
True	False	False
False	False	True
True	False	False
True	False	False
False	False	True
False	False	True
False	False	False
False	True	False
True	False	False
True	False	False

Figura 27. Transformación columna “Geography”, dataset Bank churn.

En la parte izquierda de la Figura 27 apreciamos la columna "Geography" la cual ha sido eliminada y reemplazada por tres columnas, como aparece en la parte derecha de la misma figura. Por ejemplo, para *France* se obtiene el vector [True, False, False]; mientras que, para *Spain* se tendrá el valor [False, False, True].

Finalmente, procedemos con la columna "Surname". En este caso, la cantidad de opciones diferentes es bastante significante:

```
# CANTIDAD DE APELLIDOS DIFERENTES DEL DATASET

num_apellidos_distintos = data['Surname'].nunique()
print(f"Número de apellidos distintos: {num_apellidos_distintos}")
```

Número de apellidos distintos: 2932

A su vez, cada apellido se repite un número de veces que no siempre es el mismo (como es de esperarse):

```
# CANTIDAD DE VECES QUE APARECE CADA APELLIDO

vu_surname_counts = data['Surname'].value_counts()
print(vu_surname_counts)
```

```
Surname
Smith      32
Martin     29
Walker     29
Scott      29
Brown      26
          ..
Hull        1
Sturdee     1
Flannagan   1
Dwyer       1
Corby       1
Name: count, Length: 2932, dtype: int64
```

Teniendo en cuenta lo anterior, en lugar de aplicar técnicas tradicionales como *one-hot encoding*, que generarían una elevada dimensionalidad, emplearemos una estrategia de codificación basada en la variable objetivo (target encoding). Para ello, calculamos la **probabilidad condicional** de abandono del cliente dada cada categoría del apellido, es decir,

$$P(Exited = 1 \mid Surname)$$

y este valor se asigna posteriormente a cada registro del dataset como una nueva variable numérica.

Esta transformación permite conservar información estadísticamente relevante sobre el comportamiento de los clientes, reduciendo al mismo tiempo la complejidad del espacio de características. Sin embargo, es importante advertir que esta técnica debe aplicarse cuidadosamente dentro del conjunto de entrenamiento para evitar problemas de *data leakage*.

```
# 1. Calcular la probabilidad condicional P(Exited=1 | Surname)
apellido_churn_prob = data.groupby('Surname')['Exited'].mean()

# 2. Mapear esa probabilidad a cada fila del dataframe original
data['Surname_Exited_Prob'] = data['Surname'].map(apellido_churn_
prob)

# 3.  Re-ubicar la columna "Surname_Exited_Prob" en la cuarta
 posición
col = data.pop('Surname_Exited_Prob')
data.insert(3, 'Surname_Exited_Prob', col)
```

Surname	Surname_Exited_Prob
Hargrave	1.000000
Hill	0.117647
Onio	0.250000
Boni	0.214286
Mitchell	0.100000
Chu	0.136364
Bartlett	0.250000
Obinna	0.500000
He	0.277778
H?	0.052632

Figura 28. Creación de nueva columna a partir de probabilidad condicional de Surname.

Posteriormente eliminamos la columna “Surname” (dado que ahora nos quedaremos con “Surname_Exited_Prob), así:

```
data = data.drop(columns=[‘Surname’])
```

3.2.3. Imputación de datos al interior de una columna del dataframe

Hasta este punto, ya hemos transformado las tres columnas de tipo *object* del dataset. Las demás columnas son numéricas, de tipo int64 o float64. No obstante, aún existen datos faltantes en algunas de ellas, por lo que es necesario realizar un proceso de imputación.

Para identificar en qué columnas faltan datos, utilizamos:

```
data.isnull().mean()
```

RowNumber	0.0000
CustomerId	0.0000
CreditScore	0.0000
Gender	0.0000
Age	0.0001
Tenure	0.0000
Balance	0.0000
NumOfProducts	0.0000
HasCrCard	0.0001
IsActiveMember	0.0001
EstimatedSalary	0.0000
Exited	0.0000
Country_France	0.0000
Country_Germany	0.0000
Country_Spain	0.0000
Surname_Exited_Prob	0.0000

Figura 29. Creación de nueva columna a partir de probabilidad condicional de Surname.

De acuerdo con el resultado anterior, tres columnas requieren proceso de imputación:
"Age", "HasCrCard" y "IsActiveMember".

Imputación en la columna "Age"

Para esta variable numérica utilizamos como estrategia el **promedio (media)** de los demás valores para reemplazar el dato faltante:

```
# IMPUTAR EN COLUMNA AGE
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='mean')
```

```
# Aplicar imputación solo a 'col1'
data[['Age']] = imputer.fit_transform(data[['Age']])
```

Imputación en la columna "HasCrCard"

Esta columna solo posee dos valores posibles: **1** si el cliente tiene tarjeta de crédito, o **0** si no la tiene. Por lo tanto, no resulta adecuado utilizar una métrica como el promedio. En este caso se emplea como estrategia el **valor más frecuente (moda)**:

```
# Crear el imputador con estrategia de más frecuente
imputer = SimpleImputer(strategy='most_frequent')

# Aplicar imputación solo a 'col1'
data[['HasCrCard']] = imputer.fit_transform(data[['HasCrCard']])
```

Imputación en la columna "IsActiveMember"

Esta columna presenta un comportamiento similar a "HasCrCard", por lo que utilizamos la misma estrategia basada en la moda:

```
# Crear el imputador con estrategia de más frecuente
imputer = SimpleImputer(strategy='most_frequent')

# Aplicar imputación solo a 'col1'
data[['IsActiveMember']] = imputer.fit_transform(data[['IsActiveMember']])
```

3.2.4. Eliminar columnas innecesarias del dataset

Ya estamos muy próximos a poder realizar el modelamiento de nuestro dataset. Sin embargo, aún debemos ejecutar un paso previo: eliminar columnas innecesarias. Aunque las columnas **"RowNumber"** y **"CustomerId"** son numéricas, estas no aportan información útil al proceso de modelado, dado que la primera corresponde únicamente a una secuencia, y la segunda es el identificador único de cada cliente, sin valor predictivo.

Para ello, utilizamos el siguiente código:

```
data = data.drop(columns=["RowNumber", "CustomerId"])
```

De forma opcional, procederemos a reordenar las columnas del dataset. Como buena práctica, la variable de salida suele ubicarse en la primera columna o en la última columna del daraset. En este caso, la ubicaremos en la primera columna.

```
col = data.pop('Exited')
data.insert(0, 'Exited', col)
data.head(10)
```

	Exited	Surname_Exited_Prob	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Country_France	Country_Germany	Country_Spain
0	1	1.000000	619	0	42.000000	2	0.00	1	1.0	1.0	101348.88	True	False	False
1	0	0.117647	608	0	41.000000	1	83807.86	1	0.0	1.0	112542.58	False	False	True
2	1	0.250000	502	0	42.000000	8	159660.80	3	1.0	0.0	113931.57	True	False	False
3	0	0.214286	699	0	39.000000	1	0.00	2	0.0	0.0	93826.63	True	False	False
4	0	0.100000	850	0	43.000000	2	125510.82	1	1.0	1.0	79084.10	False	False	True
5	1	0.136364	645	1	44.000000	8	113755.78	2	1.0	0.0	149756.71	False	False	True
6	0	0.250000	822	1	50.000000	7	0.00	2	1.0	1.0	10062.80	False	False	False
7	1	0.500000	376	0	29.000000	4	115046.74	4	1.0	0.0	119346.88	False	True	False
8	0	0.277778	501	1	44.000000	4	142051.07	2	0.0	1.0	74940.50	True	False	False
9	0	0.052632	684	1	38.922311	2	134603.88	1	1.0	1.0	71725.73	True	False	False

Figura 30. Dataset Bank churn con ingeniería de características.

Por último, vamos a verificar que todas las columnas sean numéricas y/o booleanas, y que no existan datos faltantes, es decir que cada columna contenga 10.002 datos no-nulos:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10002 entries, 0 to 10001
Data columns (total 14 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Exited              10002 non-null  int64
 1   Surname_Exited_Prob 10002 non-null  float64
 2   CreditScore         10002 non-null  int64
 3   Gender              10002 non-null  int64
 4   Age                 10002 non-null  float64
 5   Tenure              10002 non-null  int64
 6   Balance             10002 non-null  float64
 7   NumOfProducts       10002 non-null  int64
 8   HasCrCard           10002 non-null  float64
 9   IsActiveMember      10002 non-null  float64
 10  EstimatedSalary     10002 non-null  float64
 11  Country_France      10002 non-null  bool
 12  Country_Germany     10002 non-null  bool
 13  Country_Spain       10002 non-null  bool
dtypes: bool(3), float64(6), int64(5)
memory usage: 889.0 KB
```

Figura 31. Información del dataset posterior a FE.

¡Ahora sí, nuestro dataset está listo para realizar modelamiento!

3.2.5. Modelamiento

Entramos a la fase final. Nuestro dataset de Banck Churn no está inicialmente listo para realizar un modelamiento. Tenía columnas de tipo objeto, datos faltantes en columnas numéricas, y algunas columnas que no aportaban información para el modelo. Hemos realizado tareas de limpieza de datos (ej. imputación) y conversión o transformación de variables. Nuestro dataset ya cumple con las condiciones para que puede ingresarse a un modelo de clasificación binario. Para nuestro caso, vamos a seleccionar un árbol de decisión, el cual el mismo identifica los “patrones” para la toma de decisión en relación a si el cliente seguirá activo y se retirará.

Inicialmente importamos las librerías de trabajo:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,
 classification_report
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree
```

Posteriormente, separamos los *features* y la salida, así:

```
# 🔹 1. Separar features (X) y variable objetivo (y)
X = data.drop(columns=[“Exited”])  # “Diagnostico” es la columna que
indica si es COVID o H1N1
y = data[“Exited”]
```

A continuación, separamos el *dataset* en dos particiones: entrenamiento y validación. Típicamente, se utilizan del 70% - 80% para entrenamiento, y el resto para validación (es decir, del 20% al 30%).

```
# 🔹 2. Dividir en entrenamiento (80%) y validación (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
 random_state=42)
```

Como tercer paso, creamos y entrenamos el árbol de decisión. Para este caso hemos definido un árbol de dos niveles

```
# 🔹 3. Crear y entrenar el Árbol de Decisión
model = DecisionTreeClassifier(criterion=”entropy”, max_depth=2,
 random_state=42)
model.fit(X_train, y_train)
```

Como cuarto paso, realizamos las predicciones con los datos de validación:

```
# 🔹 4. Hacer predicciones
y_pred = model.predict(X_test)
```

Como quinto paso, evaluamos el modelo, mostramos la matriz de confusión e imprimimos el reporte

```
# 🔷 5. Evaluar el modelo, mostrar matriz de confusión e imprimir reporte
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

print("\nMatriz de Confusión:")
print(confusion_matrix(y_test, y_pred))

print("\nReporte de Clasificación:")
print(classification_report(y_test, y_pred))
```

Finalmente, visualizamos el árbol de decisión generado

```
# 🔷 6. Visualizar el árbol de decisión
plt.figure(figsize=(12, 8))
plot_tree(model, feature_names=X.columns, class_names = ['Cliente Activo', 'Cliente Retirado'], filled=True)
plt.show()
```

```
Accuracy: 0.81

Matriz de Confusión:
[[1352  247]
 [ 137  265]]

Reporte de Clasificación:
              precision    recall  f1-score   support

           0       0.91      0.85      0.88      1599
           1       0.52      0.66      0.58       402

    accuracy                           0.81      2001
   macro avg       0.71      0.75      0.73      2001
weighted avg       0.83      0.81      0.82      2001
```

Figura 32. Matriz de confusión del Modelo 1: dos niveles de profundidad.

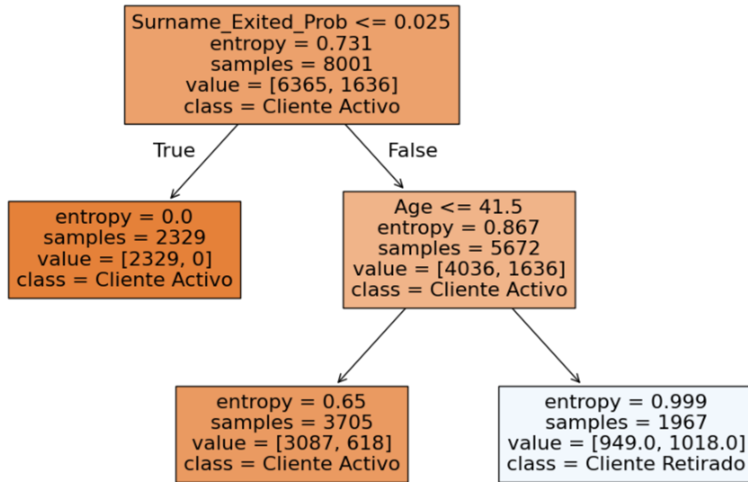

Figura 33. Árbol de decisión del Modelo 1: dos niveles de profundidad.

El árbol de decisión comienza analizando la variable Surname_Exited_Prob, que representa la probabilidad de abandono asociada al apellido del cliente.

- **Primera pregunta del árbol:**

  Si el valor de Surname_Exited_Prob **es menor o igual a 0.025**, el modelo concluye directamente que el cliente es un **Cliente Activo**. Esto significa que, históricamente, los clientes con esta característica casi nunca abandonan el banco.

- **Si** Surname_Exited_Prob **es mayor a 0.025**, el modelo realiza una segunda pregunta utilizando la variable Age:

  - Si la **edad del cliente es menor o igual a 41.5 años**, el modelo clasifica nuevamente al cliente como **Cliente Activo**.

  - Si la **edad es mayor a 41.5 años**, el modelo clasifica al cliente como **Cliente Retirado**.

Con estos dos niveles de profundidad, obtuvimos un *accuracy* del 81%. Vemos en la matriz de confusión que, de los 402 casos de retiro, el modelo identifica correctamente 265. Aún tenemos un margen de mejora importante, por lo que aumentaremos el nivel de profundidad.

```
Accuracy: 0.87

Matriz de Confusión:
[[1504   95]
 [ 156  246]]

Reporte de Clasificación:
              precision    recall  f1-score   support

           0       0.91      0.94      0.92      1599
           1       0.72      0.61      0.66       402

    accuracy                           0.87      2001
   macro avg       0.81      0.78      0.79      2001
weighted avg       0.87      0.87      0.87      2001
```

Figura 34. Matriz de confusión del Modelo 2: cuatro niveles de profundidad.

Figura 35. Árbol de decisión del Modelo 2: cuatro niveles de profundidad.

Con cuatro niveles de profundidad, pasamos de un

acc = 0.81 a un acc = 0.87. No obstante, de los casos de retiro, ahora identifica correctamente sólo 246 casos vs los 265 del modelo inicial.

Entonces, ¿cómo sabemos cuál modelo seleccionar? Esto depende de cuál sea la prioridad del banco.

- Si el objetivo es identificar a los clientes que están próximos a retirarse para intentar convencerlos de no hacerlo, entonces la métrica más importante a comparar es el Recall de la clase 1 (clientes retirados). En el primer modelo se obtuvo un *Recall* de 0.66, mientras que en el segundo modelo se obtuvo un valor de 0.61. Si yo fuese el gerente del banco, me quedaría con el **Modelo 1**, de acuerdo con este criterio, aun cuando esto implique contactar también a algunos clientes que en realidad sí desean continuar.

- Pero si, la prioridad del banco no es contactar a todos los posibles clientes que podrían retirarse, sino evitar al máximo contactar a clientes que en realidad no se van a ir, ya que cada contacto implica un costo económico (llamadas, correos, personal, campañas, etc.), entonces la mejor opción es **Modelo 2**, dado que ahora debemos enfocarnos en *Precision* de la clase 1, que en el Modelo 1 fue de 0.52, mientras que la del Modelo 2 fue de 0.72.