

AnalisisDatos

September 27, 2024

```
[1]: # ESTADISTICA DESCRIPTIVA BASE DATOS OLIST ECOMMERCE
#### Analisis Datos base de datos Brazilian E-Commerce Public Dataset by Olist
#### Fuente: https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce
```

```
[2]: ## Instalacion librerias
#### pip install folium pandas psycopg2
#### pip install scikit-learn
#### pip install sqlalchemy
#### pip install pyclustertend pandas psycopg2 sqlalchemy
#### pip install matplotlib pandas psycopg2 sqlalchemy
#### pip install --upgrade psycopg2-binary SQLAlchemy
```

```
[3]: ## Importar Librerias
import numpy as np
import psycopg2
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import folium
from folium.plugins import HeatMap
from sklearn.preprocessing import MinMaxScaler
from sqlalchemy import create_engine
```

```
[4]: # Identificar tablas

#### Tabla 1 - olist_geolocation: Información de geolocalización.
#### Tabla 2 - olist_order_customers: Información de los clientes.
#### Tabla 3 - olist_orders: : Contiene la información de los pedidos.
#### Tabla 4 - olist_order_items: Detalles de los artículos de cada pedido.
#### Tabla 5 - olist_products: Información de los productos.
#### Tabla 6 - olist_sellers: Información sobre los vendedores.
#### Tabla 7 - olist_order_payments: Información de pagos.
#### Tabla 8 - olist_order_reviews: Reseñas de los pedidos.
#### Tabla 9 - product_category_name_translation: traducciones de los nombres
↳ de las categorías de productos desde portugués a ingles.
#### Tabla 10 - spatial_ref_sys: Extensión de PostgreSQL que permite manejar
↳ datos geográficos.
```

```
[5]: ## 1) Tabla olist_geolocation: Almacena las coordenadas tipo punto de cada
      ↪ código zip.
      ### Generaliza la capa para ubicar los estados a partir de centroide

      # Conectar a la base de datos PostgreSQL y ejecutar la consulta
      try:
          conn = psycopg2.connect(
              dbname="olist_ecommerce",
              user="postgres",
              password="postgres",
              host="localhost",
              port="5432",
              options="-c client_encoding=UTF8"
          )

          # Ejecutar la consulta SQL para calcular el centroide por geolocation_state
          query = """
          SELECT
              geolocation_state as estado,
              AVG(geolocation_lat) AS lat_centroide,
              AVG(geolocation_lng) AS lng_centroide
          FROM
              olist_geolocation
          GROUP BY
              geolocation_state
          ORDER BY
              geolocation_state;
          """

          df = pd.read_sql_query(query, conn)

          # Cerrar la conexión
          conn.close()

          # Crear un mapa centrado en Brasil
          m = folium.Map(location=[-14.2350, -51.9253], zoom_start=4)

          # Añadir los centroides al mapa
          for index, row in df.iterrows():
              folium.Marker(
                  location=[row['lat_centroide'], row['lng_centroide']],
                  popup=f"Estado: {row['estado']}"
              ).add_to(m)

          # Guardar el mapa en un archivo HTML
```

```

m.save('ubicacion_generalizada.html')
print("Mapa de centroides guardado como 'ubicacion_generalizada.html'")

except Exception as e:
    print(f"Error al conectar a la base de datos: {e}")

finally:
    if conn:
        conn.close()

```

C:\Users\jmelo\AppData\Local\Temp\ipykernel_71568\1339922584.py:31: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.

```
df = pd.read_sql_query(query, conn)
```

Mapa de centroides guardado como 'ubicacion_generalizada.html'

[6]: *## 2) Tabla olist_order_customers: Información de los clientes.*
Cuantos Clientes tiene cada estado

```

# Conectar a la base de datos PostgreSQL
try:
    conn = psycopg2.connect(
        dbname="olist_ecommerce",
        user="postgres",
        password="postgres",
        host="localhost",
        port="5432",
        options="-c client_encoding=UTF8"
    )

    # Crear un cursor para ejecutar una consulta
    cur = conn.cursor()

    # Ejecutar una consulta de prueba para verificar la conexión
    cur.execute("SELECT version();")

    # Obtener el resultado
    db_version = cur.fetchone()
    print(f"Conectado a: {db_version[0]}")

    # Consulta para contar el número de clientes por estado
    query = """
SELECT
    customer_state AS estado,

```

```

        COUNT(*) AS num_cliente
FROM
    olist_order_customers
GROUP BY
    customer_state
ORDER BY
    num_cliente DESC;
"""

# Ejecutar la consulta y leer los resultados en un DataFrame de Pandas
df = pd.read_sql(query, conn)

# Graficar los resultados
plt.figure(figsize=(10, 6))
plt.bar(df['estado'], df['num_cliente'], color='skyblue')
plt.xlabel('Estado')
plt.ylabel('Número de Clientes')
plt.title('Número de Clientes por Estado')
plt.xticks(rotation=45)
plt.show()

except Exception as e:
    print(f"Error al conectar a la base de datos: {e}")

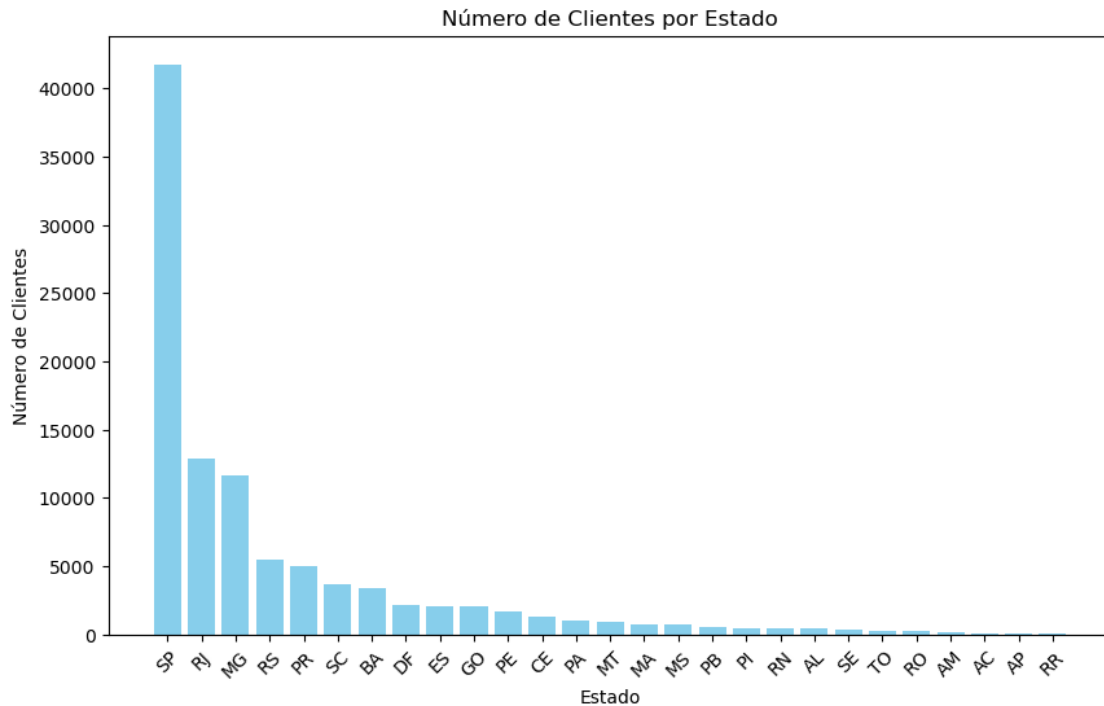
finally:
    # Cerrar la conexión si fue abierta
    if conn:
        cur.close()
        conn.close()
    print("Conexión cerrada.")

```

Conectado a: PostgreSQL 15.2, compiled by Visual C++ build 1914, 64-bit

C:\Users\jmelo\AppData\Local\Temp\ipykernel_71568\3638424074.py:41: UserWarning:
pandas only supports SQLAlchemy connectable (engine/connection) or database
string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested.
Please consider using SQLAlchemy.

```
df = pd.read_sql(query, conn)
```



Conexión cerrada.

```
[7]: ## Mapa de ubicacion de clientes

# Conectar a la base de datos PostgreSQL
try:
    conn = psycopg2.connect(
        dbname="olist_ecommerce",
        user="postgres",
        password="postgres",
        host="localhost",
        port="5432",
        options="-c client_encoding=UTF8"
    )

    # Crear un cursor para ejecutar una consulta
    cur = conn.cursor()

    # Ejecutar una consulta para obtener latitud, longitud y contar clientes_
    ↪por ciudad
    query = """
    SELECT
        customer_city,
        COUNT(*) AS num_cliente,
```

```

        AVG(ST_Y(location::geometry)) AS latitude,
        AVG(ST_X(location::geometry)) AS longitude
FROM
    olist_order_customers
WHERE
    location IS NOT NULL
GROUP BY
    customer_city
ORDER BY
    num_cliente DESC;
"""

# Ejecutar la consulta y leer los resultados en un DataFrame de Pandas
df = pd.read_sql(query, conn)

# Cerrar la conexión
cur.close()
conn.close()

# Crear el mapa Brasil
m = folium.Map(location=[-14.2350, -51.9253], zoom_start=4)

# Preparar los datos para el HeatMap (lista de [latitud, longitud, peso])
heat_data = [[row['latitude'], row['longitud'], row['num_cliente']] for
index, row in df.iterrows()]

# Añadir el HeatMap al mapa
HeatMap(heat_data).add_to(m)

# Guardar el mapa en un archivo HTML
m.save('clientes_estado_heatmap.html')
print("Mapa de calor guardado como 'clientes_estado_heatmap.html'")

except Exception as e:
    print(f"Error al conectar a la base de datos: {e}")

finally:
    # Asegurar que la conexión se cierre incluso si hay un error
    if conn:
        cur.close()
        conn.close()
    print("Conexión cerrada.")

```

C:\Users\jmelo\AppData\Local\Temp\ipykernel_71568\1303304301.py:35: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.

```
df = pd.read_sql(query, conn)
```

Mapa de calor guardado como 'clientes_estado_heatmap.html'
Conexión cerrada.

```
[10]: ### Ciudades con Más de 500 Pedidos en la Plataforma de E-commerce

# Crear la cadena de conexión con SQLAlchemy
engine = create_engine('postgresql+psycopg2://postgres:postgres@localhost:5432/
↳olist_ecommerce')

# Consulta SQL
query = """
SELECT
    customer_city,
    COUNT(customer_id) AS CantidadPedidos
FROM
    olist_order_customers
GROUP BY
    customer_city
HAVING
    COUNT(customer_id) > 500
ORDER BY
    CantidadPedidos DESC
LIMIT 10;
"""

# Ejecutar la consulta y cargar los datos en un DF
zona = pd.read_sql_query(query, engine)

# Mostrar los datos resultantes
print(zona)
```

	customer_city	cantidadpedidos
0	sao paulo	15540
1	rio de janeiro	6882
2	belo horizonte	2773
3	brasilia	2131
4	curitiba	1521
5	campinas	1444
6	porto alegre	1379
7	salvador	1245
8	guarulhos	1189
9	sao bernardo do campo	938

```
[11]: ### Cantidad de Pedidos por Estado

# Crear la cadena de conexión con SQLAlchemy
```

```

engine = create_engine('postgresql+psycopg2://postgres:postgres@localhost:5432/
↳olist_ecommerce')

try:
    # Consulta SQL
    query = """
    SELECT
        customer_city,
        customer_state,
        COUNT(customer_id) AS cantidadpedidos,
        AVG(ST_Y(location::geometry)) AS latitude,
        AVG(ST_X(location::geometry)) AS longitude
    FROM
        public.olist_order_customers
    WHERE
        location IS NOT NULL
    GROUP BY
        customer_city, customer_state
    HAVING
        COUNT(customer_id) > 1000
    ORDER BY
        cantidadpedidos DESC;
    """

    # Ejecutar la consulta y cargar los resultados en un DataFrame de Pandas
    df = pd.read_sql_query(query, engine)

    # Verificar el contenido del DataFrame
    print(df.head())

    # Crear un mapa centrado en Brasil
    m = folium.Map(location=[-14.2350, -51.9253], zoom_start=4)

    # Añadir los centroides al mapa
    for _, row in df.iterrows():
        folium.Marker(
            location=[row['latitude'], row['longitude']],
            popup=f"Ciudad: {row['customer_city']}, Pedidos:␣
↳{row['cantidadpedidos']}",
            icon=folium.Icon(color='green')
        ).add_to(m)

    # Guardar el mapa en un archivo HTML
    m.save('pedidos-estado.html')
    print("Mapa de centroides guardado como 'upedidos-estado.html'")

except Exception as e:

```



```
print(f"Error al conectar a la base de datos: {e}")
```

```
customer_city customer_state cantidadpedidos latitude longitude
0    sao paulo              SP          15538 -23.572600 -46.633861
1  rio de janeiro          RJ           6882 -22.923370 -43.328131
2  belo horizonte         MG           2773 -19.910936 -43.958786
3    brasilia             DF           1960 -15.814284 -47.960272
4    curitiba             PR           1521 -25.454796 -49.275508
Mapa de centroides guardado como 'upedidos-estado.html'
```

[12]: *#### visualizar las zonas geográficas donde se realizan más entregas de productos en una plataforma de e-commerce.*

[13]: *## 3) Tabla olist_orders: : Contiene la información de los pedidos.*

Relación entre el estado del pedido y el retraso en la entrega
orders = pd.read_sql_query("SELECT * FROM olist_orders;", con=engine)

```
orders_status = pd.read_sql_query("""
    SELECT
        order_status,
        COUNT(order_id) AS CantidadOrdenes
    FROM
        olist_orders
    GROUP BY
        order_status
    ORDER BY
        CantidadOrdenes DESC;
""", con=engine)

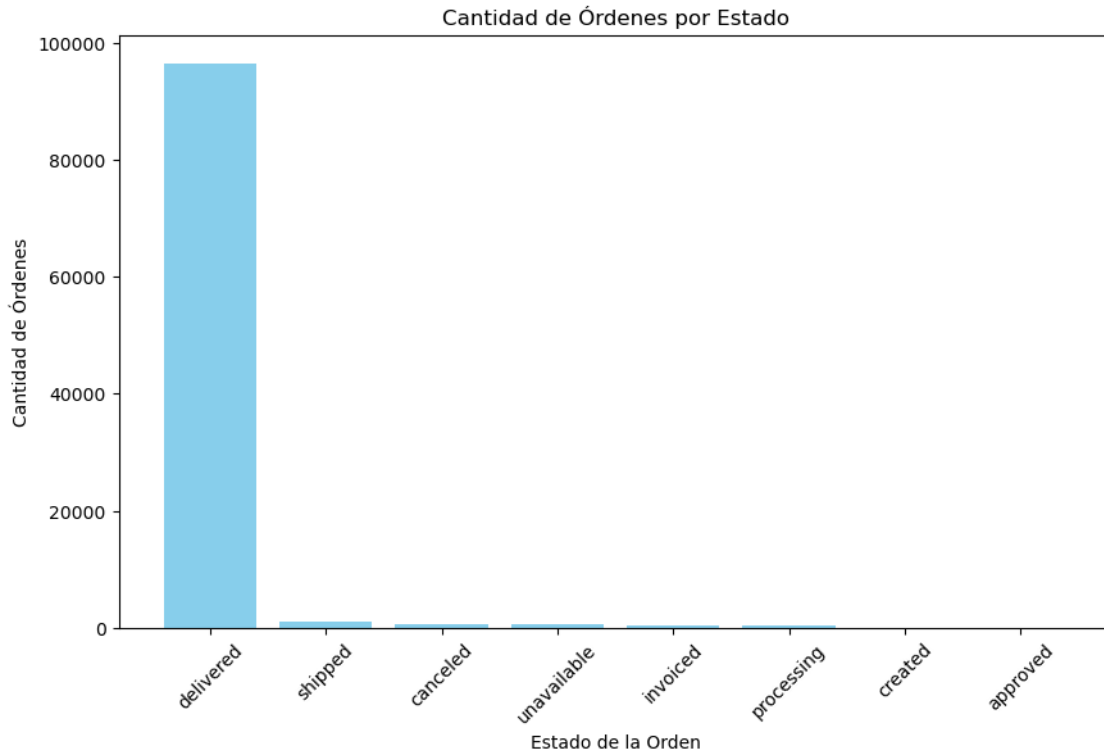
print(orders_status)
```

```
order_status  cantidadordenes
0    delivered          96478
1     shipped           1107
2    canceled            625
3 unavailable           609
4    invoiced            314
5  processing           301
6     created             5
7    approved             2
```

[14]: *### Gráfico Cantidad de Órdenes por Estado*

```
plt.figure(figsize=(10, 6))
plt.bar(orders_status['order_status'], orders_status['cantidadordenes'],
        color='skyblue')
plt.title('Cantidad de Órdenes por Estado')
```

```
plt.xlabel('Estado de la Orden')
plt.ylabel('Cantidad de Órdenes')
plt.xticks(rotation=45)
plt.show()
```



```
[15]: ### Estado del Pedido y Retraso en la Entrega

query_status_delay = """
SELECT
    order_id,
    order_status,
    order_delivered_customer_date,
    order_estimated_delivery_date
FROM olist_orders
WHERE order_delivered_customer_date IS NOT NULL;
"""

# Ejecutar la consulta SQL y cargar los resultados en un DataFrame
df_status = pd.read_sql_query(query_status_delay, con=engine)

# Calcular el retraso en la entrega (en días)
df_status['delivery_delay'] = (
```

```

    pd.to_datetime(df_status['order_delivered_customer_date']) - pd.
    ↪to_datetime(df_status['order_estimated_delivery_date'])
).dt.days

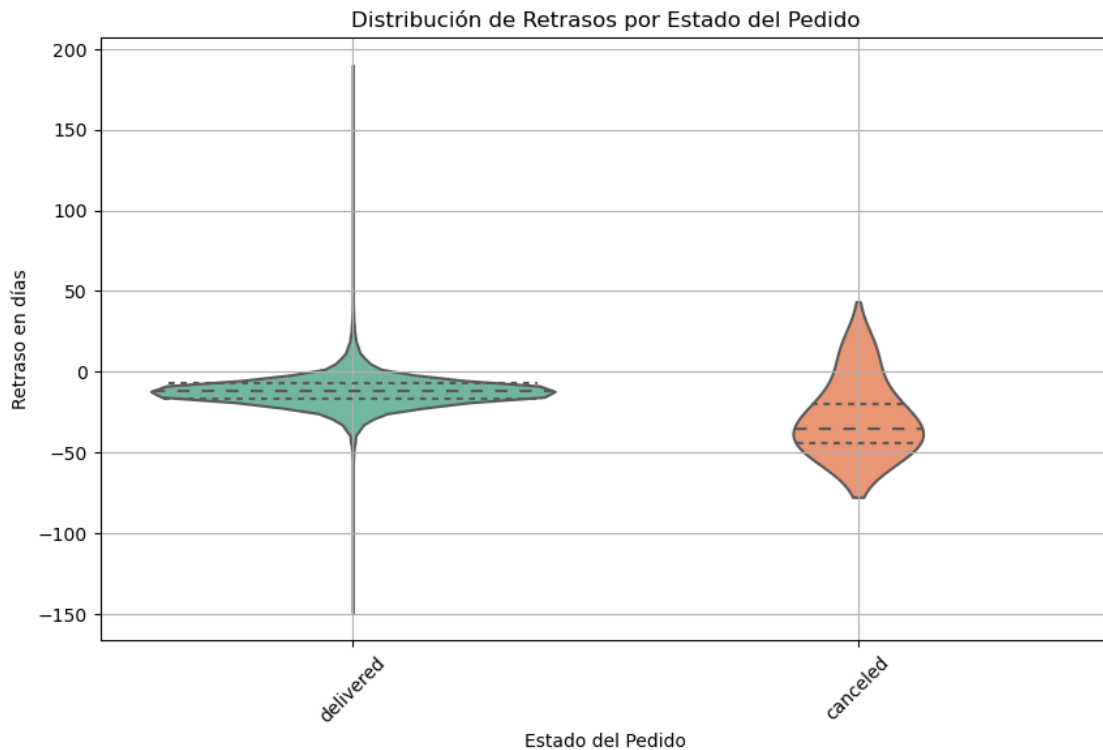
# Configuración del tamaño del gráfico
plt.figure(figsize=(10, 6))

# Gráfico de violín para mostrar la distribución del retraso según el estado
↪del pedido
sns.violinplot(x='order_status', y='delivery_delay', data=df_status,
↪inner="quartile", palette="Set2")

# Ajustes del gráfico
plt.title('Distribución de Retrasos por Estado del Pedido')
plt.xlabel('Estado del Pedido')
plt.ylabel('Retraso en días')
plt.xticks(rotation=45)
plt.grid(True)

# Mostrar el gráfico
plt.show()

```



```
[16]: '\nLa distribucion describe que entre los pedidos entregados se entregan_\n
      ↪anticipadamente (valores negativos) y otros con pequeños retrasos \n(valores_\n
      ↪positivos). En cuanto a pedidos cancelados se observan que fueron cancelados_\n
      ↪antes de la fecha de entrega estimada.\n'
```

```
[16]: '\nLa distribucion describe que entre los pedidos entregados se entregan
anticipadamente (valores negativos) y otros con pequeños retrasos \n(valores
positivos). En cuanto a pedidos cancelados se observan que fueron cancelados
antes de la fecha de entrega estimada.\n'
```

```
[17]: # Conectar a base de datos
try:
    conn = psycopg2.connect(
        dbname="olist_ecommerce",
        user="postgres",
        password="postgres",
        host="localhost",
        port="5432",
        options="-c client_encoding=UTF8"
    )

    # Consulta SQL fechas de entrega estimadas vs entregadas
    query = """
    SELECT
        order_id,
        order_estimated_delivery_date,
        order_delivered_customer_date
    FROM
        olist_orders
    WHERE
        order_delivered_customer_date IS NOT NULL
        AND order_estimated_delivery_date IS NOT NULL;
    """

    # Ejecutar la consulta y cargar los resultados en un DataFrame de Pandas
    df = pd.read_sql_query(query, conn)

    # Cerrar la conexión
    conn.close()

    # Calcular el retraso en días como la diferencia entre la fecha de entrega_\n
    ↪real y la estimada
    df['delivery_delay'] = (df['order_delivered_customer_date'] -_\n
    ↪df['order_estimated_delivery_date']).dt.days

    # Verificar si la columna 'delivery_delay' se ha calculado correctamente
    if 'delivery_delay' in df.columns:
```

```

    # Visualización gráfica: Histograma de retrasos en las entregas
    plt.figure(figsize=(8, 5))
    plt.hist(df['delivery_delay'], bins=20, color='skyblue',
    ↪edgecolor='black')
    plt.title('Distribución de Retrasos en la Entrega')
    plt.xlabel('Días de Retraso (positivos = entregas tardías, negativos = ↪
    ↪entregas anticipadas)')
    plt.ylabel('Cantidad de Pedidos')
    plt.grid(True)
    plt.show()
else:
    print("Error: La columna 'delivery_delay' no se calculó correctamente.")

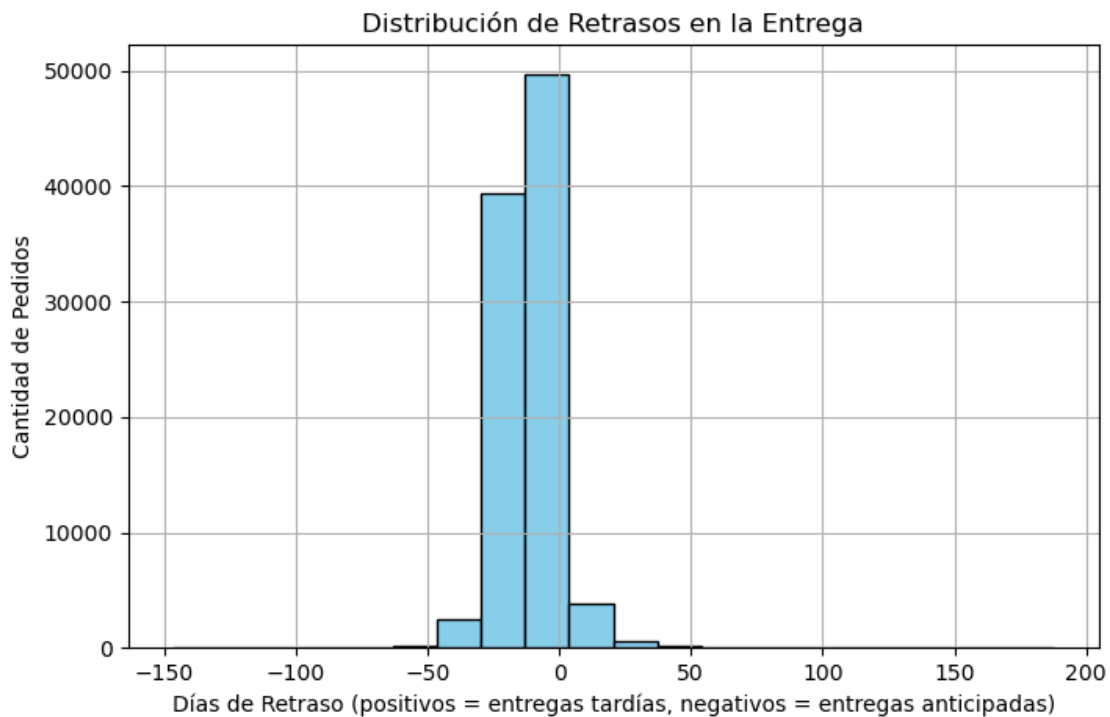
except Exception as e:
    print(f"Error al conectar a la base de datos: {e}")

finally:
    if conn:
        conn.close()

```

C:\Users\jmelo\AppData\Local\Temp\ipykernel_71568\1615506034.py:26: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.

```
df = pd.read_sql_query(query, conn)
```



```
[18]: ##### la mayoría de las entregas se realizan antes de la fecha estimada, con un  
      ↳ pico alrededor de los 10 a 30 días antes de la entrega esperada.
```

```
[19]: ### Detección de Atípicos en Retraso de Entrega

# Conectar a la base de datos PostgreSQL y ejecutar la consulta
try:
    conn = psycopg2.connect(
        dbname="olist_ecommerce",
        user="postgres",
        password="postgres",
        host="localhost",
        port="5432",
        options="-c client_encoding=UTF8"
    )

    # Consulta SQL para obtener las fechas de entrega estimadas y reales
    query = """
    SELECT
        order_id,
        order_estimated_delivery_date,
        order_delivered_customer_date
    FROM
        olist_orders
    WHERE
        order_delivered_customer_date IS NOT NULL
        AND order_estimated_delivery_date IS NOT NULL;
    """

    # Ejecutar la consulta y cargar los resultados en un DataFrame de Pandas
    df = pd.read_sql_query(query, conn)

    # Cerrar la conexión
    conn.close()

    # Calcular el retraso en días como la diferencia entre la fecha de entrega  
↳ real y la estimada
    df['delivery_delay'] = (df['order_delivered_customer_date'] -  
↳ df['order_estimated_delivery_date']).dt.days

    # Verificar si la columna 'delivery_delay' se ha calculado correctamente
    if 'delivery_delay' in df.columns:
        # Visualización gráfica: Boxplot para detectar outliers en los retrasos  
↳ de entrega
        plt.figure(figsize=(10, 6))
```

```

sns.boxplot(x=df['delivery_delay'], color='lightgreen')
plt.title('Detección de Atipicos en Retraso de Entrega')
plt.xlabel('Días de Retraso')
plt.grid(True)
plt.show()
else:
    print("Error: La columna 'delivery_delay' no se calculó correctamente.")

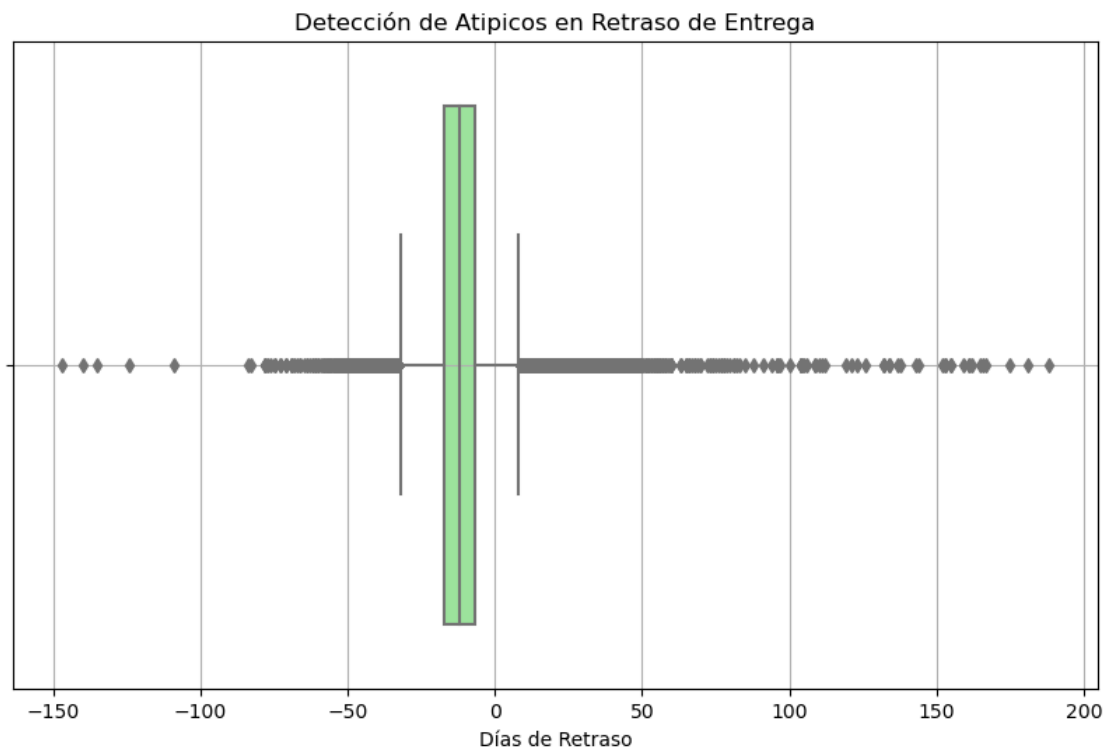
except Exception as e:
    print(f"Error al conectar a la base de datos: {e}")

finally:
    if conn:
        conn.close()

```

C:\Users\jmelo\AppData\Local\Temp\ipykernel_71568\4269501460.py:28: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.

```
df = pd.read_sql_query(query, conn)
```



[20]: *### Relación entre Precio y Retraso de la Entrega*

```

query_price_delay = """
SELECT  o.order_id,
        o.order_delivered_customer_date,
        o.order_estimated_delivery_date,
        oi.price
FROM olist_orders o

JOIN olist_order_items oi ON o.order_id = oi.order_id

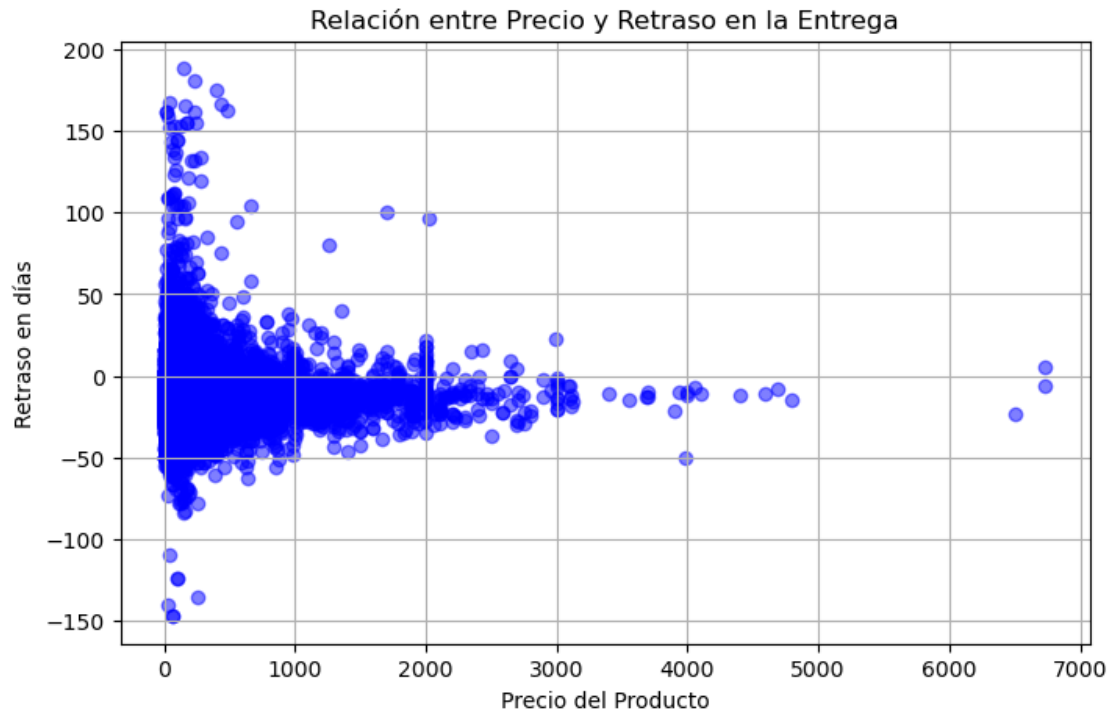
WHERE o.order_delivered_customer_date IS NOT NULL;
"""

# Ejecutar la consulta SQL y cargar los resultados en un DataFrame
df_price = pd.read_sql_query(query_price_delay, con=engine)

# Calcular el retraso en la entrega (en días)
df_price['delivery_delay'] = (
    pd.to_datetime(df_price['order_delivered_customer_date']) - pd.
    ↪to_datetime(df_price['order_estimated_delivery_date'])
).dt.days

# Visualización gráfica: Relación entre el precio y el retraso en la entrega
plt.figure(figsize=(8, 5))
plt.scatter(df_price['price'], df_price['delivery_delay'], color='blue', ↪
    ↪alpha=0.5)
plt.title('Relación entre Precio y Retraso en la Entrega')
plt.xlabel('Precio del Producto')
plt.ylabel('Retraso en días')
plt.grid(True)
plt.show()

```

```
[21]: """
La mayoría de los puntos se concentran en precios bajos, por debajo de 1000_
    ↪ unidades, para estos productos, los retrasos varían considerablemente,
    desde entregas anticipadas hasta entregas tardías. Entonces se puede indicar_
    ↪ que a medida que aumenta el precio del producto, la variabilidad
    en los retrasos disminuye.
"""
```

```
[21]: '\nLa mayoría de los puntos se concentran en precios bajos, por debajo de 1000
unidades, para estos productos, los retrasos varían considerablemente, \n desde
entregas anticipadas hasta entregas tardías. Entonces se puede indicar que a
medida que aumenta el precio del producto, la variabilidad \nen los retrasos
disminuye.\n'
```

```
[22]: ## 4) Tabla olist_order_items: Detalles de los artículos de cada pedido.
### Relación entre Año y Venta de Producto por Semestre.
```

```
# Conectar a la base de datos PostgreSQL
try:
    conn = psycopg2.connect(
        dbname="olist_ecommerce",
        user="postgres",
```

```

        password="postgres",
        host="localhost",
        port="5432",
        options="-c client_encoding=UTF8"
    )

    # Crear un cursor para ejecutar una consulta
    cur = conn.cursor()

    # Consulta SQL para obtener anio, semestre, venta_producto y costo_envio
    query = """
    SELECT
        EXTRACT(YEAR FROM shipping_limit_date) AS anio,
        CASE
            WHEN EXTRACT(MONTH FROM shipping_limit_date) <= 6 THEN '1 Semestre'
            ELSE '2 Semestre'
        END AS semestre,
        ROUND(SUM(price)) AS venta_producto,
        ROUND(SUM(freight_value)) AS costo_envio
    FROM
        olist_order_items
    WHERE
        EXTRACT(YEAR FROM shipping_limit_date) <> 2020
    GROUP BY
        anio, semestre
    ORDER BY
        anio, semestre;
    """

    # Ejecutar la consulta y leer los resultados en un DataFrame de Pandas
    df = pd.read_sql(query, conn)

    # Graficar la relación entre anio y venta_producto
    plt.figure(figsize=(10, 6))
    for semestre in df['semestre'].unique():
        df_semestre = df[df['semestre'] == semestre]
        plt.plot(df_semestre['anio'], df_semestre['venta_producto'],
            ↪marker='o', label=semestre)

    plt.title('Relación entre Año y Venta de Producto por Semestre')
    plt.xlabel('Año')
    plt.ylabel('Venta Producto (Sumatoria)')
    plt.xticks(df['anio'].unique())
    plt.grid(True)
    plt.legend(title='Semestre')
    plt.tight_layout()
    plt.show()

```

```

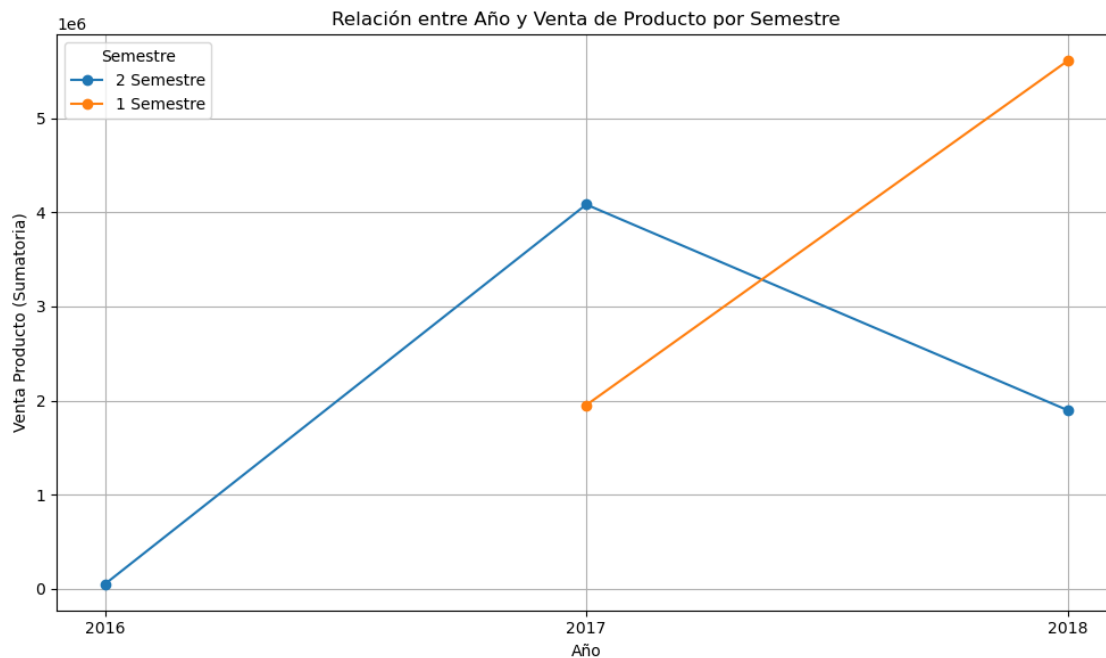
except Exception as e:
    print(f"Error al conectar a la base de datos: {e}")

finally:
    # Cerrar la conexión si fue abierta
    if conn:
        cur.close()
        conn.close()
        print("Conexión cerrada.")

```

C:\Users\jmelo\AppData\Local\Temp\ipykernel_71568\3427020997.py:40: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.

```
df = pd.read_sql(query, conn)
```



Conexión cerrada.

[23]: *### Tabla olist_orders items: Datos simplificados ventas por semestre*

```

# Conectar a la base de datos PostgreSQL y ejecutar la consulta
try:
    conn = psycopg2.connect(
        dbname="olist_ecommerce",

```

```

        user="postgres",
        password="postgres",
        host="localhost",
        port="5432",
        options="-c client_encoding=UTF8"
    )

    # Crear un cursor para ejecutar una consulta
    cur = conn.cursor()

    # Consulta SQL para obtener el año, semestre, venta_producto y costo_envio
    query = """
    SELECT
        EXTRACT(YEAR FROM shipping_limit_date) AS anio,
        CASE
            WHEN EXTRACT(MONTH FROM shipping_limit_date) <= 6 THEN '1 Semestre'
            ELSE '2 Semestre'
        END AS semestre,
        ROUND(SUM(price)) AS venta_producto,
        ROUND(SUM(freight_value)) AS costo_envio
    FROM
        olist_order_items
    WHERE
        EXTRACT(YEAR FROM shipping_limit_date) <> 2020
    GROUP BY
        anio, semestre
    ORDER BY
        anio, semestre;
    """

    # Ejecutar la consulta y leer los resultados en un DataFrame de Pandas
    df = pd.read_sql(query, conn)

    # Mostrar los primeros 5 registros del DataFrame para visualización
    print(df.head())

except Exception as e:
    print(f"Error al conectar a la base de datos: {e}")

finally:
    # Cerrar la conexión si fue abierta
    if conn:
        cur.close()
        conn.close()

```

	anio	semestre	venta_producto	costo_envio
0	2016.0	2 Semestre	49786.0	7397.0
1	2017.0	1 Semestre	1952154.0	302009.0

2	2017.0	2 Semestre	4082714.0	663731.0
3	2018.0	1 Semestre	5608861.0	936351.0
4	2018.0	2 Semestre	1897782.0	342269.0

C:\Users\jmelo\AppData\Local\Temp\ipykernel_71568\2716449766.py:39: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.

```
df = pd.read_sql(query, conn)
```

[24]: *## Tabla 5 - olist_products: Información de los productos.*

```
# Conectar a la base de datos PostgreSQL
engine = create_engine('postgresql+psycopg2://postgres:postgres@localhost:5432/
↳ olist_ecommerce')

# Consulta SQL
query = """
SELECT
    product_category_name AS categoria_producto,
    COUNT(product_id) AS cnt_productos,
    ROUND(AVG(product_weight_g)) AS media_peso
FROM
    public.olist_products
WHERE
    product_category_name IS NOT NULL
GROUP BY
    product_category_name
ORDER BY
    cnt_productos DESC;
"""

df = pd.read_sql_query(query, engine)

# Normalizar las características para ajustarlas a las visualizaciones
scaler = MinMaxScaler()
df_normalized = pd.DataFrame(scaler.fit_transform(df[['media_peso',
↳ 'cnt_productos']])),
                            columns=['media_peso', 'cnt_productos'])

# Añadir la columna de categoría de producto nuevamente al DataFrame normalizado
df_normalized['categoria_producto'] = df['categoria_producto']

# Función cara personalizada del producto
def draw_custom_face(ax, media_peso, cnt_productos, label):
    # Cara
    face = plt.Circle((0.5, 0.5), 0.4, color='orange', fill=True)
    ax.add_patch(face)
```

```

# Ojos conteo de productos
eye_y = 0.65
eye_x_dist = 0.15 + 0.1 * cnt_productos
eye_size = 0.05 + 0.05 * cnt_productos
left_eye = plt.Circle((0.5 - eye_x_dist, eye_y), eye_size, color='black')
right_eye = plt.Circle((0.5 + eye_x_dist, eye_y), eye_size, color='black')
ax.add_patch(left_eye)
ax.add_patch(right_eye)

# Boca
mouth_y = 0.35
mouth_width = 0.2 + 0.2 * cnt_productos
mouth_height = -0.1 * media_peso
ax.plot([0.5 - mouth_width / 2, 0.5 + mouth_width / 2],
        [mouth_y, mouth_y + mouth_height], color='red', linewidth=2)

# Etiqueta de la categoría
ax.text(0.5, -0.1, label, horizontalalignment='center', fontsize=12,
        fontweight='bold')

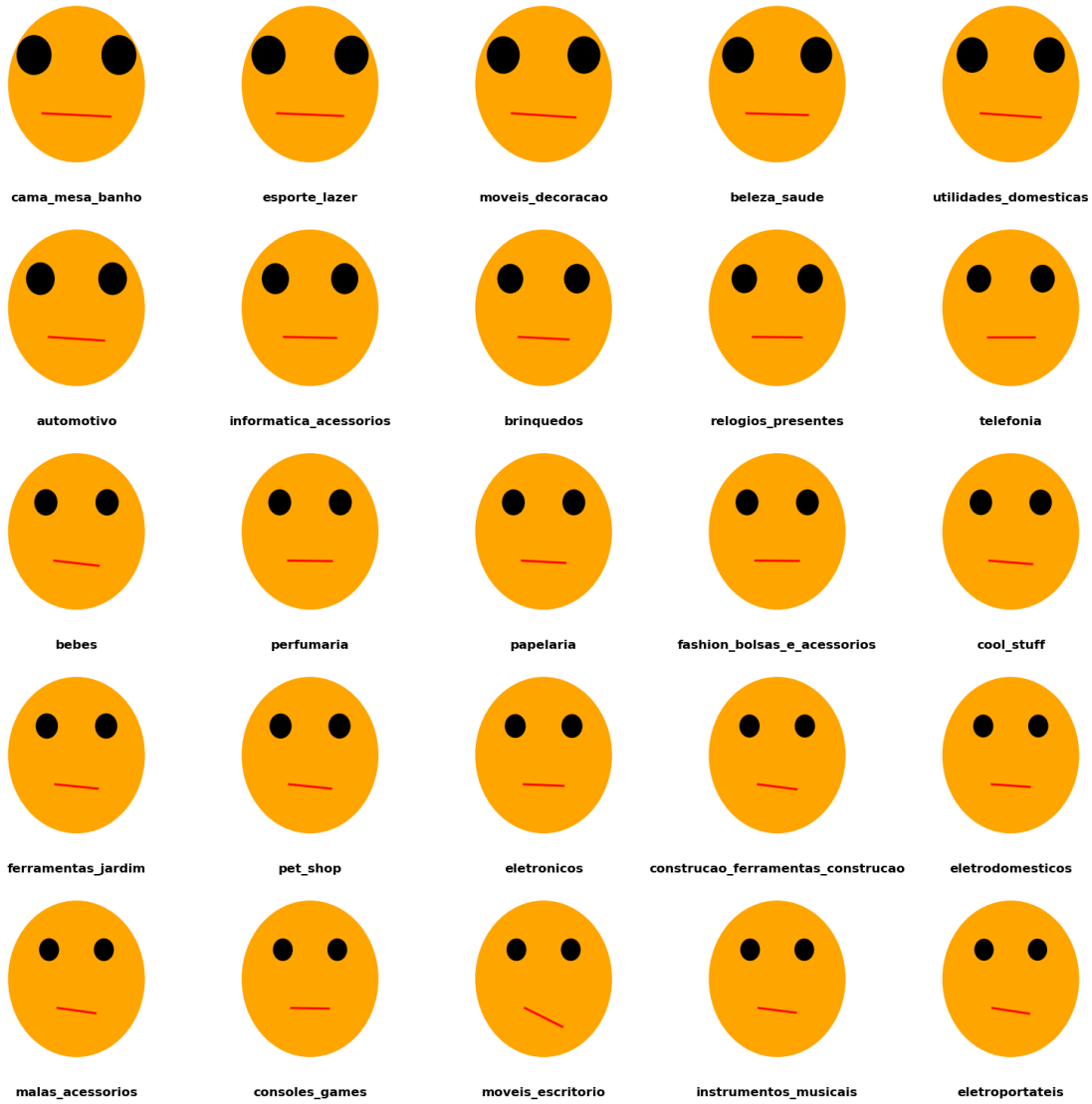
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.axis('off')

# Crear figuras para cada categoría de producto
fig, axs = plt.subplots(nrows=5, ncols=5, figsize=(15, 15))
axs = axs.flatten()

for i, (index, row) in enumerate(df_normalized.iterrows()):
    if i >= len(axs):
        break
    draw_custom_face(axs[i], row['media_peso'], row['cnt_productos'],
                    row['categoria_producto'])

plt.tight_layout()
plt.show()

```



```
[25]: """
La cara esta compuesta por:
Cara: Tipo de categorias.
Ojos: tamaño del numero de productos asociados a la categoria
Boca: Media del peso de los productos asociados a la categoria
"""
```

```
[25]: '\nLa cara esta compuesta por:\nCara: Tipo de categorias.\nOjos: tamaño del
numero de productos asociados a la categoria\nBoca: Media del peso de los
productos asociados a la categoria\n'
```

```
[27]: ## 6) Tabla olist_sellers: Información sobre los vendedores.

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sqlalchemy import create_engine

# Crear la cadena de conexión con SQLAlchemy
engine = create_engine('postgresql+psycopg2://postgres:postgres@localhost:5432/
↳olist_ecommerce')

# Ejecutar la consulta SQL y cargar los resultados en un DataFrame de Pandas
query = """
SELECT
    seller_city AS ciudad,
    COUNT(seller_id) AS cnt_vendedores
FROM
    public.olist_sellers
GROUP BY
    seller_city
ORDER BY
    cnt_vendedores DESC
LIMIT 5;
"""

df = pd.read_sql_query(query, engine)

# Verificar que las columnas se cargaron correctamente
print(df.columns)

# Crear un gráfico de estrellas para las cinco ciudades con más vendedores
def radar_chart(df):
    labels = df['ciudad']
    num_vars = len(labels)

    # Ángulos para cada eje en el gráfico de radar
    angles = np.linspace(0, 2 * np.pi, num_vars, endpoint=False).tolist()

    # La primera ciudad se repite para cerrar el gráfico de radar
    stats = df['cnt_vendedores'].tolist()
    stats += stats[:1]
    angles += angles[:1]

    # Inicializar el gráfico de radar
    fig, ax = plt.subplots(figsize=(6, 6), subplot_kw=dict(polar=True))

    ax.fill(angles, stats, color='skyblue', alpha=0.4)
```



```

ax.plot(angles, stats, color='blue', linewidth=2)

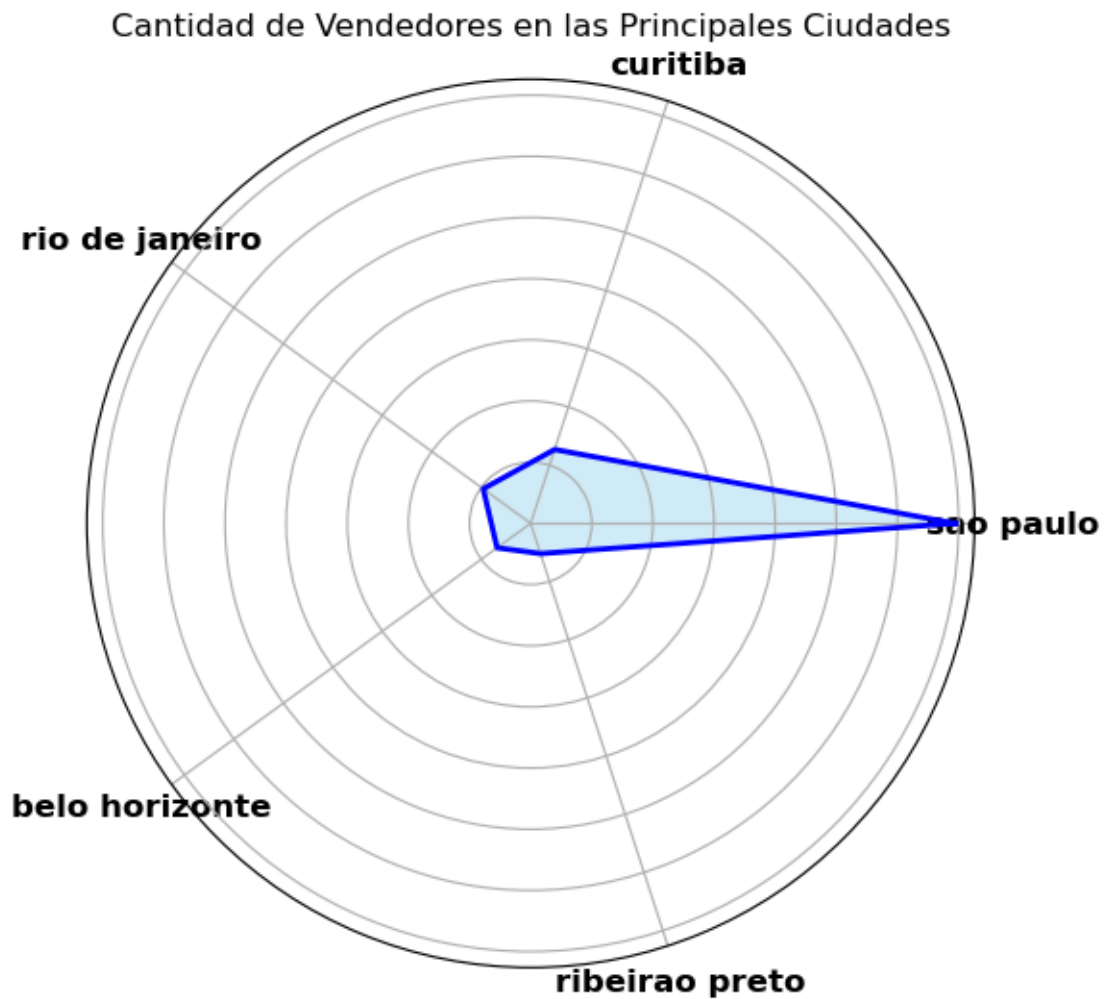
# Ajustar los atributos del gráfico
ax.set_yticklabels([])
ax.set_xticks(angles[:-1])
ax.set_xticklabels(labels, size=12, fontweight='bold')

plt.title('Cantidad de Vendedores en las Principales Ciudades')
plt.show()

# Crear el gráfico de radar
radar_chart(df)

```

```
Index(['ciudad', 'cnt_vendedores'], dtype='object')
```



[28]: *## 7) Tabla olist_order_payments: Información de pagos.*

```
# Conectar a la base de datos PostgreSQL y ejecutar la consulta
try:
    conn = psycopg2.connect(
        dbname="olist_ecommerce",
        user="postgres",
        password="postgres",
        host="localhost",
        port="5432",
        options="-c client_encoding=UTF8"
    )

    # Crear un cursor para ejecutar una consulta
    cur = conn.cursor()

    # Consulta SQL para obtener el método de pago y el valor de los pagos
    query = """
SELECT DISTINCT
    payment_type AS metodo_pago,
    ROUND(SUM(payment_value)) AS valor_pagos
FROM
    olist_order_payments
WHERE
    payment_type <> 'not_defined'
GROUP BY
    metodo_pago;
"""

    # Ejecutar la consulta y leer los resultados en un DataFrame de Pandas
    df = pd.read_sql_query(query, conn)

    # Cerrar la conexión
    cur.close()
    conn.close()

    # Crear el gráfico de tortas (pastel)
    plt.figure(figsize=(8, 8))
    plt.pie(df['valor_pagos'], labels=df['metodo_pago'], autopct='%1.1f%%',
    ↪startangle=140)
    plt.title('Métodos de Pago')
    plt.show()

except Exception as e:
    print(f"Error al conectar a la base de datos: {e}")
```

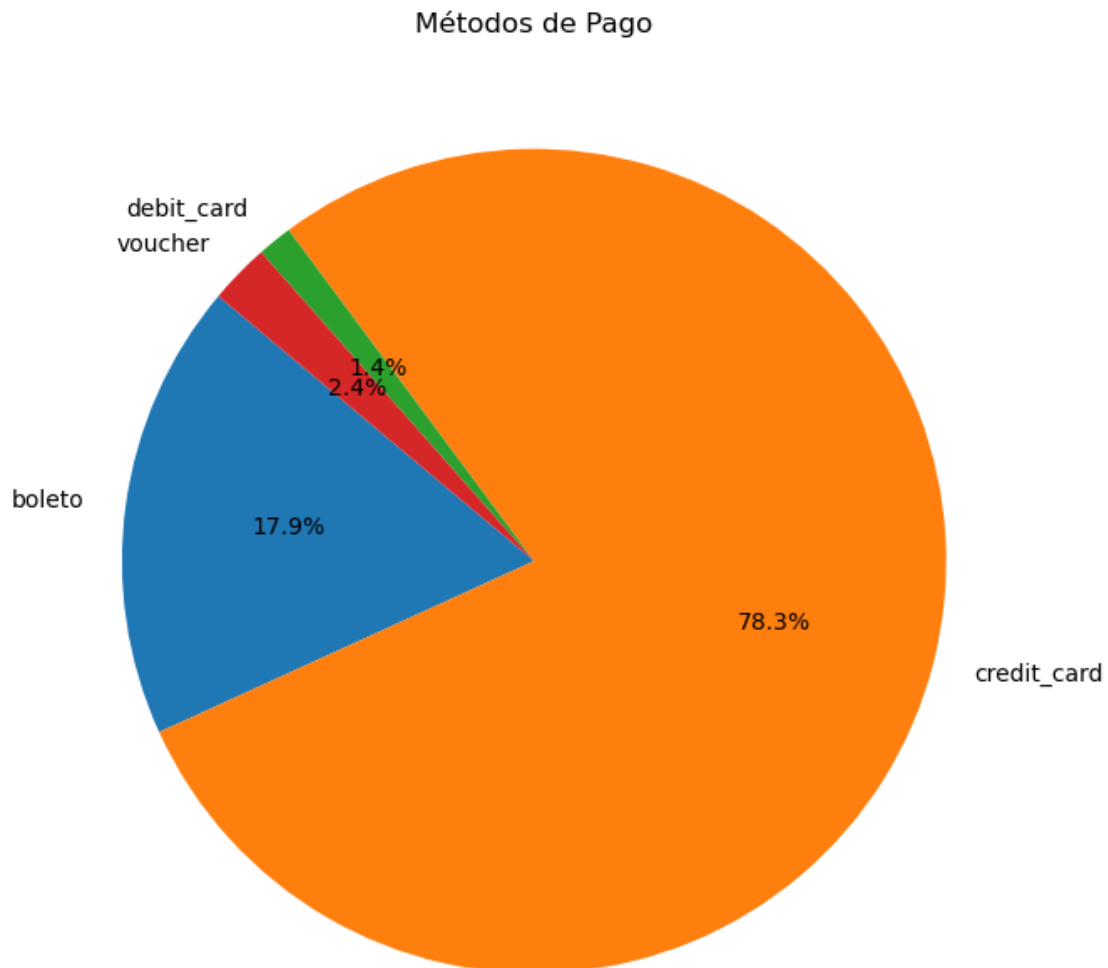
```

finally:
    # Asegurarse de que la conexión se cierra
    if conn:
        cur.close()
        conn.close()

```

C:\Users\jmelo\AppData\Local\Temp\ipykernel_71568\3609653470.py:32: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.

```
df = pd.read_sql_query(query, conn)
```



```

[29]: ### Gráfico de barras Pagos Tarjeta Credito vs Numero de cuotas

# Conectar a la base de datos PostgreSQL y ejecutar la consulta
try:
    conn = psycopg2.connect(
        dbname="olist_ecommerce",
        user="postgres",
        password="postgres",
        host="localhost",
        port="5432",
        options="-c client_encoding=UTF8"
    )

    # Crear un cursor para ejecutar una consulta
    cur = conn.cursor()

    # Consulta SQL para obtener el método de pago, número de cuotas, valor de
    ↪pagos y porcentaje
    query = """
    SELECT
        payment_type AS metodo_pago,
        payment_installments AS cuota,
        ROUND(SUM(payment_value)) AS valor_pagos,
        ROUND(CAST(SUM(payment_value) * 100.0 / SUM(SUM(payment_value)) OVER ()_
    ↪AS numeric), 2) AS porcentaje
    FROM
        olist_order_payments
    WHERE
        payment_type = 'credit_card'
    GROUP BY
        metodo_pago, cuota
    ORDER BY
        cuota ASC;
    """

    # Ejecutar la consulta y leer los resultados en un DataFrame de Pandas
    df = pd.read_sql_query(query, conn)

    # Agrupar cuotas entre 11 y 22
    df['cuota_grupo'] = df['cuota'].apply(lambda x: '11-22' if 11 <= x <= 22_
    ↪else str(x))

    # Agrupar por cuota_grupo y sumar valores de pagos
    df_grouped = df.groupby('cuota_grupo').agg({'valor_pagos': 'sum'}).
    ↪reset_index()

    # Ordenar por número de cuotas para visualización

```

```

df_grouped['cuota_grupo'] = pd.Categorical(df_grouped['cuota_grupo'],
categories=sorted(df_grouped['cuota_grupo'], key=lambda x: int(x.
split('-')[0]) if '-' in x else int(x)),
ordered=True)

# Crear el gráfico de barras
plt.figure(figsize=(10, 6))
plt.bar(df_grouped['cuota_grupo'], df_grouped['valor_pagos'],
color='skyblue')

# Añadir etiquetas y título
plt.xlabel('Número de Cuotas')
plt.ylabel('Valor Pagos (BRL)')
plt.title('Valor de Pagos por Número de Cuotas (con Tarjeta de Crédito)')
plt.grid(True)

# Mostrar el gráfico
plt.show()

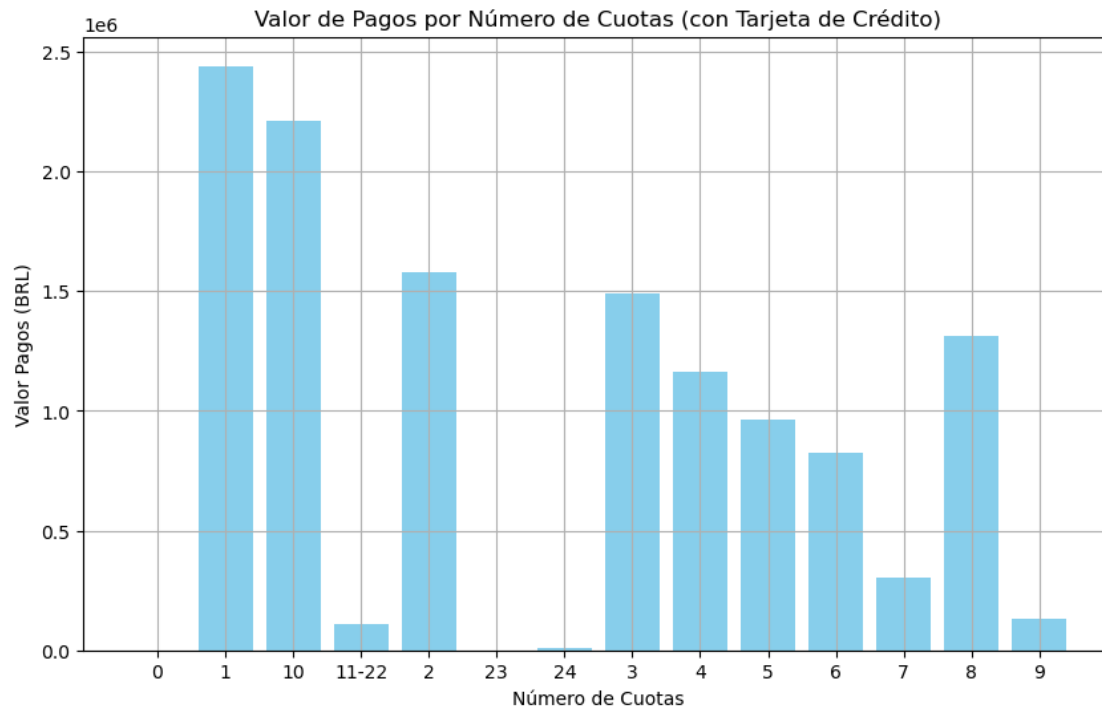
except Exception as e:
    print(f"Error al conectar a la base de datos: {e}")

finally:
    # Asegurarse de que la conexión se cierra
    if conn:
        cur.close()
        conn.close()

```

C:\Users\jmelo\AppData\Local\Temp\ipykernel_71568\3525691494.py:35: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.

```
df = pd.read_sql_query(query, conn)
```



[30]: *### Métodos de Pago y Retrasos de Entrega*

```
query_payments_delay = """
SELECT p.payment_type,
       o.order_delivered_customer_date, o.order_estimated_delivery_date
FROM olist_order_payments p
JOIN olist_orders o ON p.order_id = o.order_id
WHERE o.order_delivered_customer_date IS NOT NULL;
"""

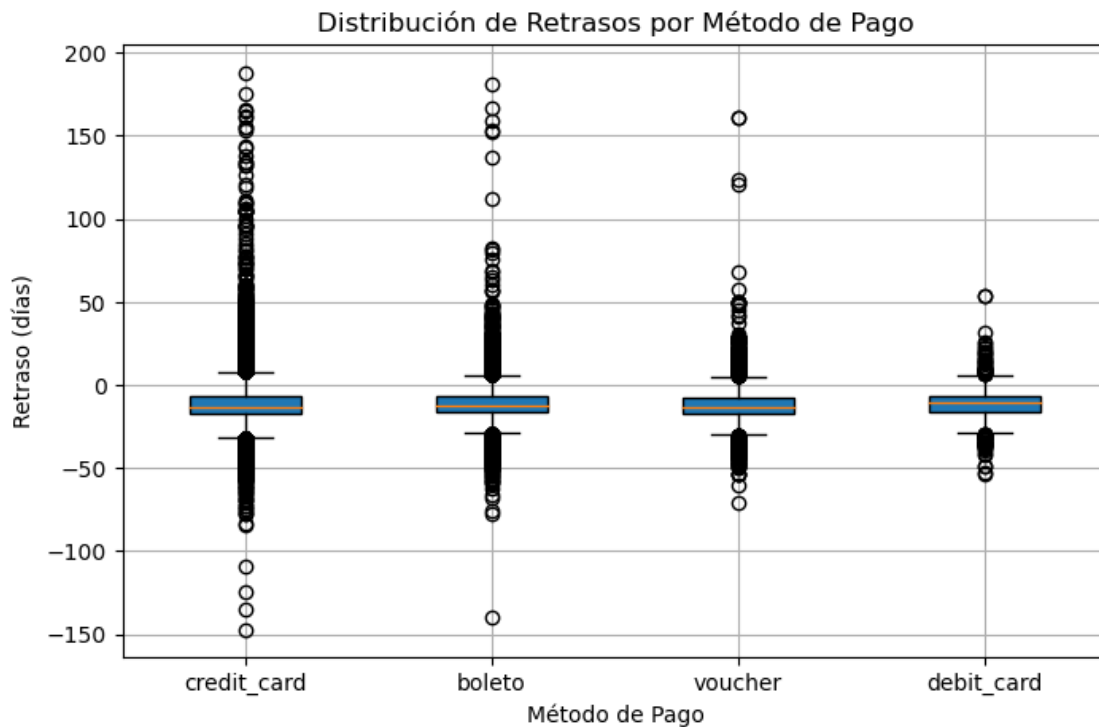
# Ejecutar la consulta SQL y cargar los resultados en un DataFrame
df_payments = pd.read_sql_query(query_payments_delay, con=engine)

# Calcular el retraso en la entrega (en días)
df_payments['delivery_delay'] = (
    pd.to_datetime(df_payments['order_delivered_customer_date']) - pd.
    to_datetime(df_payments['order_estimated_delivery_date'])
).dt.days

# Gráfico de caja y bigotes para mostrar la distribución de los retrasos por
# método de pago
plt.figure(figsize=(8, 5))
```

```
plt.boxplot([df_payments[df_payments['payment_type'] == payment][
    'delivery_delay'] for payment in df_payments['payment_type'].
    unique()],
            labels=df_payments['payment_type'].unique(), patch_artist=True)

plt.title('Distribución de Retrasos por Método de Pago')
plt.xlabel('Método de Pago')
plt.ylabel('Retraso (días)')
plt.grid(True)
plt.show()
```



[60]:

```
"""
Em el gráfico de caja y bigotes se observa la distribución de los retrasos en
    la entrega según el método de pago,
en promedio todos los métodos tienen entregas anticipadas, no se observan
    diferencias significativas entre
los métodos de pago. Existen valores atípicos notables, con algunas entregas
    extremadamente anticipadas o muy tardías,
lo que sugiere variabilidad en los tiempos de entrega independientemente del
    método de pago.
"""
```

[60]: '\nEm el gráfico de caja y bigotes se observa la distribución de los retrasos en la entrega según el método de pago, \nen promedio todos los métodos tienen entregas anticipadas, no se observan diferencias significativas entre \nlos métodos de pago. Existen valores atípicos notables, con algunas entregas extremadamente anticipadas o muy tardías, \nlo que sugiere variabilidad en los tiempos de entrega independientemente del método de pago.\n'

[31]: *## 8) Tabla olist_order_reviews: Reseñas de los pedidos.*

```
# Definir los detalles de la conexión
user = 'postgres'
password = 'postgres'
host = 'localhost'
port = '5432'
dbname = 'olist_ecommerce'

# Crear el engine usando SQLAlchemy
connection_string = f'postgresql://{user}:{password}@{host}:{port}/{dbname}'
engine = create_engine(connection_string)

# Definir y ejecutar la consulta SQL
query1 = """
SELECT
    review_score as Calificacion,
    COUNT(review_id) AS Cantidad,
    ROUND(COUNT(review_id) * 100.0 / (SELECT COUNT(*) FROM_
↵olist_order_reviews)) AS Porcentaje
FROM
    olist_order_reviews
GROUP BY
    review_score
ORDER BY
    review_score;
"""

# Ejecutar la consulta y cargar los datos en un DataFrame
opiniones_clientes = pd.read_sql_query(query1, engine)

# Mostrar los datos
print(opiniones_clientes)
```

	calificacion	cantidad	porcentaje
0	1	11282	11.0
1	2	3114	3.0
2	3	8097	8.0
3	4	19007	19.0
4	5	56910	58.0


```
[37]: """
El 58% de las calificaciones otorgadas por los clientes corresponde a la
    ↪puntuación más alta (5), lo que indica un alto nivel de satisfacción.
Un 11% de los clientes ha calificado con la puntuación más baja (0), lo que
    ↪podría reflejar un grupo insatisfecho o con problemas significativos en su
    ↪experiencia de compra.
En total, solo el 14% de las calificaciones se encuentran entre 1 y 2, lo que
    ↪muestra que los clientes tienden a polarizar sus opiniones, otorgando muy
    ↪pocas calificaciones bajas (entre 1 y 2).

"""
```

```
[37]: '\nEl 58% de las calificaciones otorgadas por los clientes corresponde a la
puntuación más alta (5), lo que indica un alto nivel de satisfacción.\nUn 11% de
los clientes ha calificado con la puntuación más baja (0), lo que podría
reflejar un grupo insatisfecho o con problemas significativos en su experiencia
de compra.\nEn total, solo el 14% de las calificaciones se encuentran entre 1 y
2, lo que muestra que los clientes tienden a polarizar sus opiniones, otorgando
muy pocas calificaciones bajas (entre 1 y 2).\n\n'
```

```
[32]: ### Pago Total
total_payment = pd.read_sql_query("""
    SELECT
        SUM(payment_value) AS PagoTotal
    FROM
        olist_order_payments;
""", con=engine)

print(total_payment)
```

```

    pagototal
0    1.600887e+07
```

```
[33]: # Union de Variables
```

```
[35]: ### Conexión BD
user = 'postgres'
password = 'postgres'
host = 'localhost'
port = '5432'
dbname = 'olist_ecommerce'

### Crear el engine usando SQLAlchemy
connection_string = f'postgresql://{user}:{password}@{host}:{port}/{dbname}'
engine = create_engine(connection_string)

### Consulta SQL
query1 = """
```

```

SELECT
    o.order_id,                                -- ID del pedido (olist_orders)
    o.order_delivered_customer_date,          -- Fecha de entrega al cliente
    ↪(olist_orders)
    o.order_estimated_delivery_date,          -- Fecha estimada de entrega
    ↪(olist_orders)
    c.customer_zip_code_prefix AS customer_zip, -- Código postal del cliente
    ↪(olist_order_customers)
    s.seller_zip_code_prefix AS seller_zip,    -- Código postal del vendedor
    ↪(olist_sellers)
    ST_Distance(c.location, s.location) AS distancia, -- Distancia entre el
    ↪cliente y el vendedor (calculado con PostGIS)
    r.review_score,                            -- Puntuación de la reseña del
    ↪pedido (olist_order_reviews)
    p.payment_type,                            -- Tipo de pago utilizado
    ↪(olist_order_payments)
    oi.price                                    -- Precio del producto
    ↪(olist_order_items)
FROM
    olist_orders o
JOIN
    olist_order_customers c ON o.customer_id = c.customer_id
JOIN
    olist_order_items oi ON o.order_id = oi.order_id
JOIN
    olist_sellers s ON oi.seller_id = s.seller_id
JOIN
    olist_order_reviews r ON o.order_id = r.order_id
JOIN
    olist_order_payments p ON o.order_id = p.order_id
WHERE
    o.order_delivered_customer_date IS NOT NULL
LIMIT 5;
"""

### Ejecutar la consulta y cargar los datos en un DataFrame
union_datos = pd.read_sql_query(query1, engine)

### Mostrar los datos
print(union_datos)

```

	order_id	order_delivered_customer_date	\
0	b81ef226f3fe1789b1e8b2acac839d17	2018-05-09 17:36:51	
1	a9810da82917af2d9aefd1278f1dcfa0	2018-06-29 20:32:09	
2	25e8ea4e93396b6fa0d3dd708e76c1bd	2017-12-18 17:24:41	
3	ba78997921bbcdc1373bb41e913ab953	2017-12-21 01:35:51	
4	ba78997921bbcdc1373bb41e913ab953	2017-12-21 01:35:51	

	order_estimated_delivery_date	customer_zip	seller_zip	distancia	\
0	2018-05-22	39801	13321	845683.597248	
1	2018-07-16	2422	4660	21319.322471	
2	2018-01-04	2652	9015	26790.100620	
3	2018-01-04	36060	13405	458723.781313	
4	2018-01-04	36060	13405	458723.781313	

	review_score	payment_type	price
0	1	credit_card	79.80
1	5	credit_card	17.00
2	5	credit_card	56.99
3	5	credit_card	89.90
4	5	credit_card	89.90

```
[38]: # Medidas Estadísticas
```

```
[39]: ## Resumen estadístico de las variables numéricas
print("Resumen Estadístico:")
print(df.describe())
```

Resumen Estadístico:

	cuota	valor_pagos	porcentaje
count	24.000000	2.400000e+01	24.000000
mean	11.708333	5.225868e+05	4.165417
std	7.369025	7.747085e+05	6.177154
min	0.000000	1.890000e+02	0.000000
25%	5.750000	2.171000e+03	0.017500
50%	11.500000	2.305300e+04	0.180000
75%	17.250000	1.011858e+06	8.065000
max	24.000000	2.440445e+06	19.460000

```
[40]: ## Análisis Estadístico de la Distribución de Calificaciones de los Clientes
## Importar NumPy
import numpy as np

## Extraer los valores de la columna 'cantidad'
review_counts = opiniones_clientes['cantidad'].values

## Calcular la media
mean = np.mean(review_counts)

## Calcular la mediana
median = np.median(review_counts)

## Calcular la desviación estándar
std_dev = np.std(review_counts)
```

```
## Mostrar los resultados
print(f"Media: {mean}")
print(f"Mediana: {median}")
print(f"Desviación Estándar: {std_dev}")
```

Media: 19682.0

Mediana: 11282.0

Desviación Estándar: 19316.575566077958

```
[41]: """
El análisis calcula tres métricas estadísticas de las calificaciones recibidas,
    ↪ en cada categoría (0 a 5) en la plataforma de e-commerce: entonces:
a) La Media o promedio de calificaciones es 19,682 indica que cada categoría,
    ↪ recibe una calificación.
b) La mediana es 11,282, lo que significa que la mitad de las categorías tienen,
    ↪ un número de calificaciones menor o igual, esto sugiere que,
    algunas calificaciones tienen una cantidad significativamente mayor de,
    ↪ opiniones, lo cual está respaldado por la concentración de altas,
    ↪ calificaciones
c) La desviación estándar es 19,316.58, lo que indica una gran variabilidad en,
    ↪ el número de calificaciones recibidas por cada categoría,
    esto significa que no todas las calificaciones tienen cantidades similares y,
    ↪ algunas categorías están por encima del promedio,
    mientras que otras tienen menos participación.
"""
```

```
[41]: '\nEl análisis calcula tres métricas estadísticas de las calificaciones
recibidas en cada categoría (0 a 5) en la plataforma de e-commerce:
entonces:\na) La Media o promedio de calificaciones es 19,682 indica que cada
categoría recibe una calificación.\nb) La mediana es 11,282, lo que significa
que la mitad de las categorías tienen un número de calificaciones menor o igual,
esto sugiere que,\nc) La desviación estándar es 19,316.58, lo que indica una gran
variabilidad en el número de calificaciones recibidas por cada categoría, \nesto
significa que no todas las calificaciones tienen cantidades similares y algunas
categorías están por encima del promedio, \nmientras que otras tienen menos
participación.\n'
```

```
[48]: import pandas as pd
import numpy as np
from sqlalchemy import create_engine

# Crear la cadena de conexión con SQLAlchemy
engine = create_engine('postgresql+psycopg2://postgres:postgres@localhost:5432/
    ↪olist_ecommerce')
```

```

# Cargar los datos de la tabla de órdenes, que debe contener las fechas de
↳ entrega
df = pd.read_sql_query('SELECT * FROM olist_orders', con=engine)

# Verificar las columnas presentes
print(df.columns)

# Cálculo del retraso en la entrega (en días)
if 'order_delivered_customer_date' in df.columns and
↳ 'order_estimated_delivery_date' in df.columns:
    df['delivery_delay'] = (
        pd.to_datetime(df['order_delivered_customer_date']) - pd.
↳ to_datetime(df['order_estimated_delivery_date'])
    ).dt.days

    # Medidas estadísticas descriptivas: Media y desviación estándar de los
↳ retrasos
    media_retraso = np.mean(df['delivery_delay'])
    desviacion_retraso = np.std(df['delivery_delay'])

    print(f"Media de retraso en días: {media_retraso}")
    print(f"Desviación estándar de retraso: {desviacion_retraso}")
else:
    print("Las columnas necesarias no están presentes en el DataFrame.")

```

```

Index(['order_id', 'customer_id', 'order_status', 'order_purchase_timestamp',
      'order_approved_at', 'order_delivered_carrier_date',
      'order_delivered_customer_date', 'order_estimated_delivery_date'],
      dtype='object')

```

Media de retraso en días: -11.876881296902857

Desviación estándar de retraso: 10.183801336071676

```

[ ]: """
Media de retraso en días -11.876881296902857: El valor negativo de la media
↳ sugiere que, en promedio, los pedidos fueron entregados 11.88 días antes
de la fecha estimada de entrega. Un valor negativo en el contexto del retraso
↳ indica que los pedidos fueron entregados más rápido de lo esperado.

Desviación estándar de retraso 10.183801336071676: La desviación estándar de
↳ aproximadamente 10.18 días indica que hay una variabilidad significativa
en los tiempos de entrega respecto a la fecha estimada, significa que, aunque
↳ en promedio los pedidos se entregaron antes, hay atípicos en los tiempos
de entrega.
"""

```

```

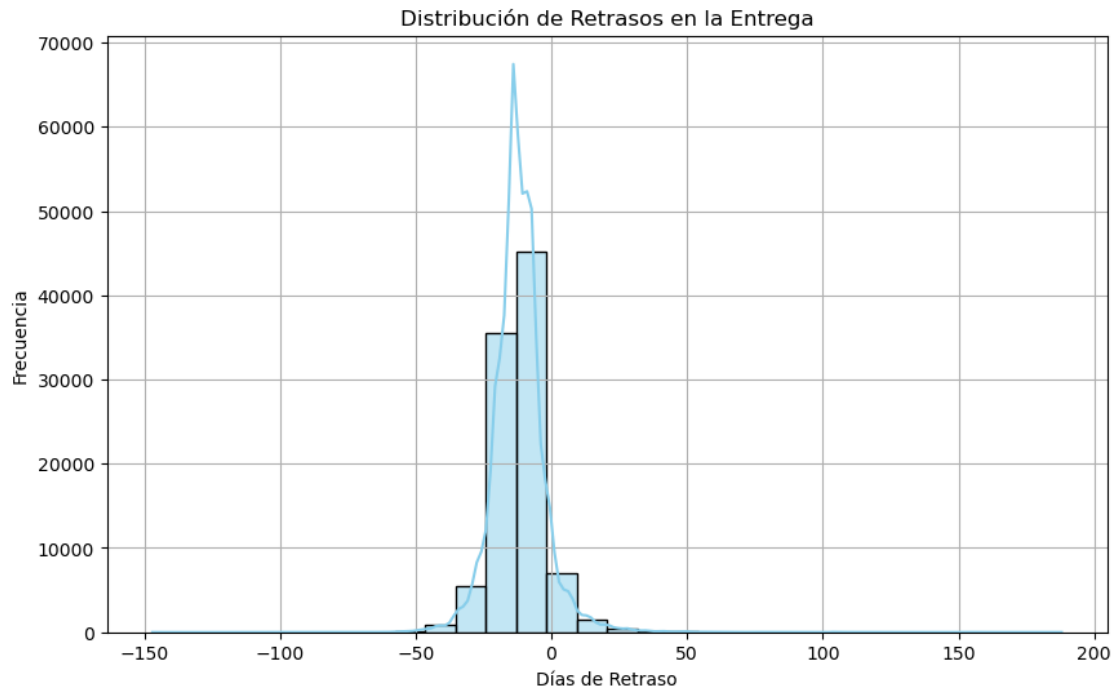
[50]: # Visualización de la Distribución de 'delivery_delay'
plt.figure(figsize=(10, 6))

```

```

sns.histplot(df['delivery_delay'], bins=30, kde=True, color='skyblue')
plt.title('Distribución de Retrasos en la Entrega')
plt.xlabel('Días de Retraso')
plt.ylabel('Frecuencia')
plt.grid(True)
plt.show()

```



```
[54]: # Regresion Lineal y Logistica
```

```
[ ]: ## Verifiquemos la correlacion
```

```

[71]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sqlalchemy import create_engine

# Conectar a la base de datos PostgreSQL
engine = create_engine('postgresql+psycopg2://postgres:postgres@localhost:5432/
↳olist_ecommerce')

# Ejecutar la consulta SQL y cargar los resultados en un DataFrame de Pandas
query = """
SELECT

```

```

        o.order_id,                -- ID del pedido (olist_orders)
        o.order_delivered_customer_date, -- Fecha de entrega al cliente_
    )
    (olist_orders)
        o.order_estimated_delivery_date, -- Fecha estimada de entrega_
    )
    (olist_orders)
        c.customer_zip_code_prefix AS customer_zip, -- Código postal del cliente_
    )
    (olist_order_customers)
        s.seller_zip_code_prefix AS seller_zip, -- Código postal del vendedor_
    )
    (olist_sellers)
        ST_Distance(c.location, s.location) AS distancia, -- Distancia entre el_
    )
    cliente y el vendedor (calculado con PostGIS)
        r.review_score,                -- Puntuación de la reseña del_
    )
    pedido (olist_order_reviews)
        p.payment_type,                -- Tipo de pago utilizado_
    )
    (olist_order_payments)
        oi.price                        -- Precio del producto_
    )
    (olist_order_items)
FROM
    olist_orders o
JOIN
    olist_order_customers c ON o.customer_id = c.customer_id
JOIN
    olist_order_items oi ON o.order_id = oi.order_id
JOIN
    olist_sellers s ON oi.seller_id = s.seller_id
JOIN
    olist_order_reviews r ON o.order_id = r.order_id
JOIN
    olist_order_payments p ON o.order_id = p.order_id
WHERE
    o.order_delivered_customer_date IS NOT NULL
LIMIT 5;
"""

# Cargar los datos en un DataFrame
df = pd.read_sql_query(query, engine)

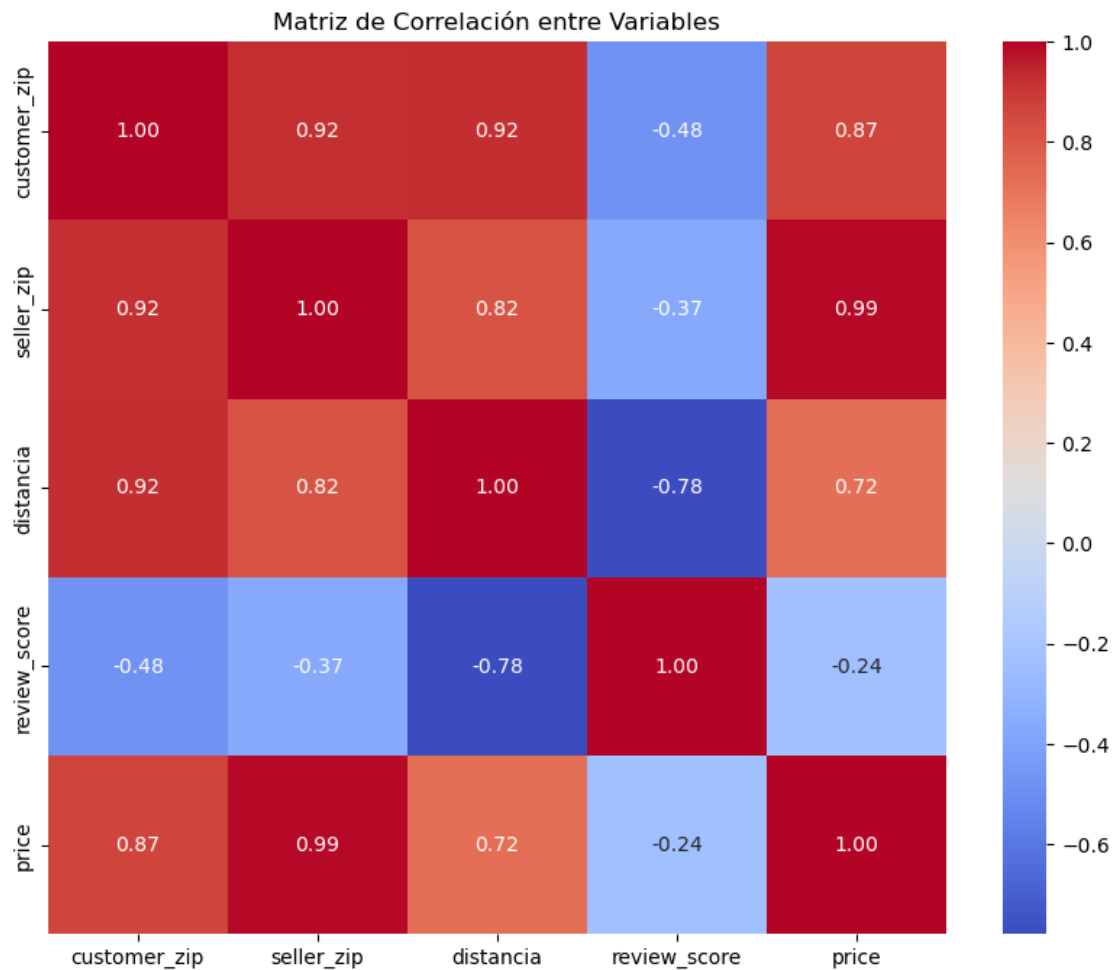
# Filtrar solo las columnas numéricas para la matriz de correlación
df_numeric = df.select_dtypes(include=[np.number])

# Manejo de valores NaN (eliminación en este caso)
df_numeric = df_numeric.dropna()

# Calcular la matriz de correlación
corr_matrix = df_numeric.corr()

```

```
# Visualizar la matriz de correlación
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Matriz de Correlación entre Variables')
plt.show()
```



[58]: """

- 1) Hay una alta correlación positiva entre el código postal del cliente y el del vendedor. por ello se identifica que los clientes y vendedores tienden a estar geográficamente cercanos
- 2) Hay una correlación negativa significativa entre la distancia y la puntuación de la reseña. Los clientes que están más lejos del vendedor tienden a dar peores puntuaciones, posiblemente debido a tiempos de entrega más largos o mayores costos de envío.

price vs seller_zip (0.99):

*3) El precio tiene una correlación casi perfecta con el código postal del
→vendedor, ciertos vendedores en áreas específicas tienden a vender productos
a precios similares por ello se indica que los precios están asociados a
→ciertas regiones.
""*

[58]: '\n1) Hay una alta correlación positiva entre el código postal del cliente y el
del vendedor. por ello se identifica que los clientes y vendedores \ntienden
a estar geográficamente cercanos\n\n2) Hay una correlación negativa
significativa entre la distancia y la puntuación de la reseña. Los clientes que
están más lejos del vendedor \ntienden a dar peores puntuaciones, posiblemente
debido a tiempos de entrega más largos o mayores costos de envío.\nprice vs
seller_zip (0.99):\n\n3) El precio tiene una correlación casi perfecta con el
código postal del vendedor, ciertos vendedores en áreas específicas tienden a
vender productos \na precios similares por ello se indica que los precios están
asociados a ciertas regiones.\n'