FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2

[7506-9558] Organización/Ciencia de Datos
Cátedra Martinelli
Segundo cuatrimestre de 2025

Alumno	Padrón	Email
Giannantonio, Maurizio	108119	mgiannantonio@fi.uba.ar

Índice

1. Introducción

En el presente trabajo se realiza el análisis y resolución de consultas sobre un conjunto de datos compuesto por los siguientes *datasets*: `orders`, `order_items`, `categories`, `customers`, `reviews`, `products` e `inventory_logs`. Los archivos, provistos por la cátedra en formato `.csv`, fueron cargados en Apache Spark mediante una SparkSession y procesados principalmente a través de la API de RDDs, complementando cuando corresponde con la API de DataFrames para comparativas de desempeño.

En una primera etapa se efectuó la limpieza y normalización de datos: estandarización de cadenas y manejo de valores inválidos, derivación de atributos (por ejemplo, extracción de estado y código postal desde direcciones) y verificación de claves para los cruces entre tablas. Los distintos datasets se integran mediante identificadores (order_id, product_id, customer_id, etc.), lo cual permite realizar uniones y agregaciones distribuidas de manera eficiente.

El objetivo principal es formular y resolver seis consultas específicas exclusivamente con Spark, priorizando la implementación con RDDs para afianzar el modelo de programación distribuida.
El código completo utilizado para cargar datos, limpiar, transformar y ejecutar las consultas se encuentra disponible en el repositorio correspondiente. En consecuencia, este informe se centra en el razonamiento seguido, las decisiones de modelado/limpieza, y los hallazgos derivados de la ejecución en Spark.Algunas de las funciones utilizadas para la limpieza de datos estan definidas en el código.

2. Consultas de la cátedra

2.1. Cuál es el estado que más descuentos tiene en total? y en promedio?

Objetivo. Determinar, a partir de las órdenes (RDD `rddOrders`), qué estado acumula:(i) el mayor descuento total (`discount_amount` sumado) Y (ii) el mayor descuento promedio por orden.

Supuestos.

- Se extrae de `shipping_address` el código de dos letras inmediatamente antes del código postal (p. ej., "..., `MA 26926`").
- Se excluyen registros con `shipping_address` nulo y/o `discount_amount` nulo o no positivo.

Metodología (RDD).

1. *Limpieza y mapeo:* Filtramos órdenes válidas y mapeamos a pares (`state`, (`discount`, 1)).
2. *Agregación por clave:* `reduceByKey` acumula (`suma_descuento`, `conteo_ordenes`) por estado. El resultado se cachea para reutilizarlo.
3. *Máximo por total:* Seleccionamos el estado con mayor `suma_descuento`.
4. *Máximo por promedio:* Convertimos a promedio `suma/n` y seleccionamos el máximo.

Complejidad y performance. El pipeline realiza una sola barajada principal (`reduceByKey`). El `cache()` sobre el RDD agregado evita recomputar cuando se calculan ambos máximos (total y promedio).Estas cifras dependen del dataset y de los filtros mencionados.

Código.

```

rddEstadosConDescuentos = rddOrders.filter(lambda r: r.shipping_address and r.discount_amount is not None).map(lambda r: (extract_state(r.shipping_address), to_float(r.discount_amount))).filter(lambda kv: kv[0] is not None and kv[1] is not None and kv[1] > 0).mapValues(lambda v: (v, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1])).cache()

# Estado con MÁS DESCUENTO TOTAL ($)

rddEstadosConDescuentos.mapValues(lambda sc: sc[0]).reduce(lambda a,b: a if a[1] > b[1] else b)

# Estado con MÁS DESCUENTO TOTAL ($)
```

```
estado_max_total = rddEstadosConDescuentos.mapValues(lambda sc: sc[0]).reduce(lambda a,b: a if a[1] > b[1] else b)

# Estado con MAYOR PROMEDIO de descuento ($ por registro con descuento)
estado_max_prom = rddEstadosConDescuentos.mapValues(lambda sc: sc[0] / sc[1]).reduce(lambda a,b: a if a[1] > b[1] else b)

print(f"estadoConMasDescuentosTotales: {estado_max_total[0]}, ${estado_max_total[1]:,.2f}")
print(f"estadoConMasDescuentosEnPromedio: {estado_max_prom[0]}, ${estado_max_prom[1]:,.2f}")

```

2.2. ¿Cuáles son los 5 códigos postales más comunes para las órdenes con estado 'Refunded'? ¿Y cuál es el nombre más frecuente entre los clientes de esas direcciones?

Objetivo. (i) Encontrar los 5 códigos postales más comunes entre las órdenes con estado `Refunded`. (ii) Para **cada** uno de esos ZIPs, determinar el **nombre de pila más frecuente** entre los clientes que tienen órdenes `Refunded` a esas direcciones.

Supuestos.

- El estado de la orden se toma desde `orders.status` y se compara de manera *case-insensitive* con `Refunded`.
- El código postal (ZIP) se extrae de `shipping_address`.
- Los nombres de clientes provienen de `customers.first_name` y se normalizan con capitalización tipo `Title Case`.

Metodología (RDD).

1. *Filtrado y extracción de ZIP:* de `rddOrders` nos quedamos con las órdenes `Refunded` y mapeamos a $(\texttt{zip}, \texttt{customer\_id})$.
2. *Frecuencias de ZIPs:* `map` + `reduceByKey` para contar ocurrencias; luego `takeOrdered(5)` para el Top-5.
3. *Broadcast del Top-5:* difundimos el conjunto de ZIPs top para filtrar rápido las órdenes relevantes.
4. *Join con clientes:* $(\texttt{customer\_id}, \texttt{zip})$ `join` $(\texttt{customer\_id}, \texttt{first\_name}) \rightarrow (\texttt{zip}, \texttt{first\_name})$.
5. *Top nombre por ZIP:* contamos $(\texttt{zip}, \texttt{first\_name})$; resolvemos empates por orden alfabético del nombre.

Complejidad y performance. Una barajada principal en el `reduceByKey` de ZIPs y otra en el `join` con clientes. El broadcast del Top-5 evita mover datos innecesarios en esa etapa.

Resultados (dataset de cátedra).

- **Top-5 ZIPs (Refunded):** `70696 (6)`, `47612 (5)`, `11954 (5)`, `83755 (5)`, `59883 (5)`.
- **Nombre más frecuente por ZIP:** `11954` → `Anthony (1)`, `47612` → `Debbie (1)`, `59883` → `Carolyn (1)`, `70696` → `Amber (1)`, `83755` → `Alexis (1)`.

Estas cifras dependen del dataset y de los criterios de limpieza.

Código.

```
rddOrdersRefundedZipCus = rddOrders.filter(lambda x: normalizar_string(x.status) == 'Refunded' and x.shipping_address is not None).map(lambda x: (extraer_codigo_postal(x.shipping_address), x.customer_id)).filter(lambda kv: kv[0] is not None)

rddTop5CodigoPostales = rddOrdersRefundedZipCus.map(lambda x: (x[0],1)).reduceByKey(lambda a,b: a+b).cache()

# top 5 codigos postales
top5 = rddTop5CodigoPostales.takeOrdered(5, key = lambda x: -x[1])

top5Zips = [z for z, _ in top5]
btop5 = sc.broadcast(set(top5Zips))
```

```
10
11 ordersRefundedTop5Bycust = rddOrdersRefundedZipCus.filter(lambda zc: zc[0] in btop5.value).map(lambda
   ↪  zc: (zc[1], zc[0]))
12
13
14 rddbyCus = rddcustomers.map(lambda clientes:
   ↪  (clientes.customer_id,normalizar_string(clientes.first_name))).filter(lambda x: x[1]is not None)
15
16 joined  = ordersRefundedTop5Bycust.join(rddbyCus)
17
18 nameCountByZip = joined.map(lambda x: ((x[1][0], x[1][1]), 1)).reduceByKey(lambda a,b: a[1]+b[1])
19
20 # Top nombre por ZIP (con desempate alfabético)
21 topNamePerZip = nameCountByZip.map(lambda znc: (znc[0][0], (znc[0][1], znc[1]))).reduceByKey(lambda a,
   ↪  b: a if (a[1] > b[1]) or (a[1] == b[1] and a[0] <= b[0]) else b)
22
```

2.3. Para cada tipo de pago y segmento de cliente, devolver la suma y el promedio expresado como porcentaje, de clientes activos y de consentimiento de marketing.

Objetivo. Para cada combinación de **método de pago** y **segmento de cliente**, calcular: (i) la **suma** (cantidad total de clientes únicos) y (ii) el **promedio expresado como porcentaje** de clientes activos y con consentimiento de marketing.

Supuestos.

- Consideramos clientes únicos por método de pago: si un cliente aparece varias veces con el mismo método, cuenta una sola vez.
- Se excluyen clientes sin `registration_date`, `customer_segment`, `is_active` o `marketing_consent`.
- Normalizamos `payment_method` y `customer_segment` a *Title Case* y comparamos ignorando mayúsculas/minúsculas.

Metodología (RDD).

1. *Método por cliente (único):* de `rddOrders` mapeamos (`customer_id`, `payment_method`), normalizamos y deduplicamos por (`customer`, `método`).
2. *Atributos del cliente:* de `rddcustomers` tomamos (`segmento`, `is_active`, `marketing_consent`) filtrando nulos.
3. *Join por `customer_id`:* obtenemos (`método`, `segmento`, `activo`, `marketing`).
4. *Agregación:* por clave (`método`, `segmento`) sumamos activos, marketing y total).
5. *Porcentajes:* calculamos $\frac{100 \cdot \text{activos}}{\text{total}}$ y $\frac{100 \cdot \text{marketing}}{\text{total}}$.

Complejidad y performance. Los filtros y mapeos son baratos (recorren los datos una vez). Lo más costoso es el `join` entre órdenes y clientes; para alivianarlo repartimos ambos RDDs con `partitionBy(200)` y así el `join` trabaja más "local". La agregación final por (método,segmento) y el `sortBy` operan sobre pocos grupos, por eso su impacto es bajo. Usamos `cache` solo si vamos a reutilizar resultados; si no, puede ocupar memoria de más.

Código.

```
1 customersMethod = rddOrders.filter(lambda x: x.customer_id is not None and x.payment_method is not
  ↪  None).map(lambda x: ((x.customer_id, normalizar_string(x.payment_method)),1)).filter(lambda x
  ↪  :x[0][1] is not None).reduceByKey(lambda a,b: 1).map(lambda kv: (kv[0][0], kv[0][1])).cache()
2
3 CustomersPerId = rddcustomers.filter(lambda x:  x.customer_segment is not None
4 and x.registration_date is not None and x.is_active is not None and x.marketing_consent is not
  ↪  None).map(lambda cliente: (cliente.customer_id, (normalizar_string(cliente.customer_segment)
  ↪  ,cliente.is_active, cliente.marketing_consent))).filter(lambda x :x[1][0] is not None
  ↪  ).reduceByKey(lambda a,b: b)
5
6 NP = 200
7 customersMethod = customersMethod.partitionBy(NP).cache()
```

```
CustomersSegmentMarketing = CustomersPerId.partitionBy(NP).cache()

joined = customersMethod.join(CustomersSegmentMarketing)

MethodAndSegment = joined.map(lambda kv: ((kv[1][0], kv[1][1][0]),# clave = (metodo, segmento)
(1 if kv[1][1][1] else 0,                    # suma activos
1 if kv[1][1][2] else 0,                     #suma marketing
1))).reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1], a[2]+b[2]))

Porcents = MethodAndSegment.mapValues(lambda t: (
    t[2],                          # total_clientes
    100.0 * t[0] / t[2],           # % clientes_activos
    100.0 * t[1] / t[2]            # % marketing
  )
)
tablaFinal = Porcents.map(lambda x: (
      x[0][1],  # segmento_cliente
      x[0][0],  # metodo_pago
      x[1][0],  # total_clientes
      x[1][1],  # % clientes_activos
      x[1][2]   # % marketing
    )).sortBy(lambda x: (x[0], x[1])).cache()  # ordena por segmento, luego método de pago


```

2.4. Peso total de inventario por marca para productos con 'stuff'

Objetivo. Entre los productos cuya `description` contiene la palabra *"stuff"* (ignorando mayúsculas/minúsculas), calcular el **peso total de inventario** por **marca** y mostrar las **5 marcas** con mayor peso total.

Supuestos.

- El peso total de inventario por producto se calcula como `weight_kg` × `stock_quantity`.
- Se excluyen registros con `brand`, `description`, `weight_kg` o `stock_quantity` nulos/no válidos, y valores ≤ 0.
- Las marcas se normalizan (*Title Case*) para unificar variantes.

Metodología (RDD).

1. *Filtrado por palabra clave y datos válidos:* de `rddProducts` nos quedamos con productos cuyo `description` contiene "stuff" y con peso/stock/marca válidos.
2. *Mapeo a pares:* transformamos a (`brand`, `peso_total`), donde `peso_total` = `weight_kg`×`stock_quantity`.
3. *Agregación por marca:* usamos `reduceByKey` para sumar `peso_total` por `brand`.
4. *Top-5:* aplicamos `takeOrdered(5)` con clave negativa para obtener las 5 marcas más pesadas.

Complejidad y performance. Los filtros y mapeos recorren los productos una vez. La parte más costosa es la `reduceByKey` (una barajada) que agrupa por marca; como el número de marcas es mucho menor que el de productos, el *Top-5* final es liviano.

Código.

```
brandStuff = rddProducts.filter(lambda product: (product.description is not None and product.weight_kg
↪  is not None and product.stock_quantity is not None)  and ('stuff' in product.description.lower())
↪  and (product.brand is not None)).map(lambda product: (normalizar_string(product.brand),
↪  (product.weight_kg * product.stock_quantity))).filter(lambda kv: kv[0] is not None)

brandStuffByWeight = brandStuff.reduceByKey(lambda a,b: a+b)

top5Weight = brandStuffByWeight.takeOrdered(5, key = lambda x: -x[1])

# ----------------- Resultados -----------------
for i, (brand, weight) in enumerate(top5Weight, start=1):
    print(f"{i}. {brand}: {weight:,.2f} kg.")
```

1. 3M: 1,851,279.46 kg.
2. Adidas: 1,578,522.37 kg.
3. Hasbro: 1,506,279.13 kg.
4. Wayfair: 1,448,357.55 kg.
5. Nike: 1,385,209.17 kg.

2.5. porcentaje de productos cuyo stock es al menos 20 porciento más alto que el stock promedio de su marca.

Objetivo. Calcular el **porcentaje global** de productos cuyo `stock_quantity` es $\geq 120\,\%$ del **stock promedio de su marca**. Ej.: si Adidas promedia 100, cuentan los productos Adidas con stock > 120.

Supuestos.

- Se consideran solo productos con `brand` no nula y `stock_quantity` > 0.
- La marca se normaliza para unificar variantes (espacios/casos).
- El promedio se calcula por marca y luego se evalúa la condición producto a producto.

Metodología (RDD).

1. *Filtrado y mapeo:* (`brand`, `stock`) para productos válidos.
2. *Promedio por marca:* `mapValues` a (*suma*, *conteo*) + `reduceByKey`; luego `suma`/`conteo`.
3. *Join producto-promedio:* (`brand`, (`stock`, `avg_brand`)).
4. *Marcado y porcentaje:* marcar 1 si `stock` $> 1{,}2 \cdot$ `avg_brand`, sumar y dividir por el total.

Complejidad y performance. Los filtros y mapeos recorren una vez los productos. La parte más costosa es la agregación por marca (`reduceByKey`) y el `join` contra los promedios; como el número de marcas es mucho menor que el de productos, el cálculo del porcentaje final es liviano.

Código.

```
productos = rddProducts.filter(lambda product: (product.stock_quantity is not None and
↪  product.stock_quantity > 0) and product.brand is not None).map(lambda product:
↪  (normalizar_string(product.brand), product.stock_quantity)).filter(lambda kv: kv[0] is not
↪  None).map(lambda x: (x[0],x[1])).cache()

sumatoria = productos.mapValues(lambda x: (x, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1]))
promedios = sumatoria.mapValues(lambda s : s[0]/s[1]) # (marca, avg_stock)

# Uno cada producto con el promedio de su marca
# productos: (marca, stock)  promedios: (marca, avg)
joined = productos.join(promedios)

marcados = joined.map(lambda x: 1 if x[1][0] > 1.2 * x[1][1] else 0)

cumplen = marcados.sum()
total   = productos.count()          # total de productos válidos
porcentaje_global = 100.0 * cumplen / total if total else 0.0
# ----------------- Resultados -----------------
print(f"Porcentaje global de productos con stock > 120% del promedio de su marca:
↪  {porcentaje_global:.4f}%")
#Porcentaje global de productos con stock > 120% del promedio de su marca: 39.9650%

```

2.6. cantidad de órdenes que no hayan comprado ninguno de los 10 productos más vendidos.

Objetivo. Obtener la **cantidad de órdenes** cuyos items no incluyen **ninguno** de los **10 productos más vendidos**.

Supuestos.

- Los más vendidos se definen por **cantidad de items** en `order_items` (no por facturación).

- Se usan `product_id` y `order_id` no nulos.
- La lista Top-10 se difunde con `broadcast` para filtrar rápido.

Metodología (RDD).

1. *Top-10 productos:* contar (`product_id` → `cantidad`) con `reduceByKey` y tomar el Top-10.
2. *Marcar órdenes:* para cada *item* mapear (`order_id` → 1) si su `product_id` está en el Top-10, si no (`order_id` → 0), y sumar por orden.
3. *Contar:* filtrar las órdenes con suma = 0 (no compraron Top-10) y contar.

Complejidad y performance. Hay dos pasos costosos: el conteo por `product_id` y la suma por `order_id` (ambos usan `reduceByKey`). El `broadcast` del Top-10 es pequeño y barato. El `takeOrdered(10)` es liviano porque opera sobre el resumen de productos.

Código.

```
1
2  ProductsSellers = rddOrderItems.filter(lambda x: x.product_id is not None and x.order_id is not
   ↪  None).map(lambda x: (x.product_id,1)).reduceByKey(lambda a,b : a + b)
3
4  top10ProductsBestSellers = ProductsSellers.takeOrdered(10, key = lambda x: -x[1])
5
6  top10_ids = set(pid for (pid, _) in top10ProductsBestSellers)
7  btop10 = sc.broadcast(top10_ids)
8
9  # Para cada ítem de orden, marco pertenece al top10
10 # (order_id -> 1 si contiene un producto del top10, 0 si no)
11 orderHasTop10 = rddOrderItems.filter(lambda x: x.product_id is not None and x.order_id is not
   ↪  None).map(lambda x: (x.order_id, 1 if x.product_id in btop10.value else 0)).reduceByKey(lambda a,
   ↪  b: a + b)  # suma de flags por orden
12
13 #Órdenes que NO compraron ninguno del top10: suma == 0
14 ordersNotInTop10 = orderHasTop10.filter(lambda kv: kv[1] == 0).keys()
15
16 # Cantidad pedida
17 count_orders_not_in_top10 = ordersNotInTop10.count()
18 # ----------------- Resultados -----------------
19 print("Cantidad de órdenes que no compraron ninguno de los 10 productos más vendidos:",
   ↪  count_orders_not_in_top10)
20
21 #Cantidad de órdenes que no compraron ninguno de los 10 productos más vendidos: 99490
```

3. Consultas exploratorias complementarias

3.1. Top-3 métodos de pago por cantidad de órdenes completadas en 2024.

Objetivo. Obtener el **Top-3** de **métodos de pago** con más **órdenes completadas** durante **2024**.

Supuestos.

- El estado de una orden completada es `Completed` (comparación *case-insensitive*).
- El año se extrae de un campo de fecha ISO (p. ej., `order_date` con formato `YYYY-MM-DD...`).
- `payment_method` se normaliza (*Title Case*) para unificar variantes.

Metodología (RDD).

1. *Filtrado y normalización:* conservar órdenes con `payment_method`, `order_date` y `status` válidos; normalizar método y estado.
2. *Selección de 2024 completadas:* quedarnos con (`método`, 1) cuando `status` = `Completed` y año = 2024.
3. *Conteo y Top-3:* `reduceByKey` para contar por método y `takeOrdered(3)` en orden descendente.

Complejidad y performance. El costo principal es el conteo por método con `reduceByKey` (una barajada). Los filtros y mapeos son lineales; el *Top-3* es liviano.

Código.

```

def extraer_anio_iso(s):
    if not s: return None
    s = str(s).strip()
    return int(s[:4]) if len(s) >= 4 and s[:4].isdigit() else None

metodosDePagoCompletadas2024 = rddOrders.filter(lambda order: order.payment_method is not None and order.created_at is not None and order.status is not None).map(lambda x:(normalizar_string(x.payment_method), (normalizar_string(x.status), extraer_anio_iso(x.order_date)))).filter(lambda t: t[0] is not None and t[1][0] == 'Completed' and t[1][1] == 2024).map(lambda x: (x[0],1)).reduceByKey(lambda a,b: a+b)
metodosDePagoCompletadas2024.takeOrdered(3,lambda x: -x[1])
```

3.2. Precio promedio por categoría padre

Objetivo. Calcular el **precio promedio** de los productos por **categoría padre** y mostrar las **5 categorías padre** con mayor promedio.

Supuestos.

- Se consideran productos con `category_id` y `price` válidos, y `price` > 0.
- Se consideran categorías con `category_id`, `parent_category` y `created_at` no nulos.
- `parent_category` se normaliza (Title Case) para unificar variantes.

Metodología (RDD).

1. *Productos:* mapear a (`category_id`, `price`).
2. *Categorías:* mapear a (`category_id`, `parent_category`) normalizada.
3. *Join por `category_id`:* obtener (`parent_category`, `price`) por producto.
4. *Promedio por padre:* agregar (`suma`, `conteo`) y luego dividir; tomar **Top-5** descendente.

Complejidad y performance. El paso dominante es el `join` por `category_id`. La agregación por `parent_category` (`reduceByKey`) introduce una barajada adicional. Los filtros y mapeos son lineales; el *Top-5* es liviano.

Código.

```
catPrice = rddProducts.filter(lambda prod: prod.price and prod.category_id is not None and prod.price > 0).map(lambda x: (x.category_id, x.price))

categories = rddCategories.filter(lambda cat:cat.category_id and cat.parent_category and cat.created_at is not None).map(lambda cat: (cat.category_id,
normalizar_string(cat.parent_category))).filter(lambda x:x[1] is not None)

joinCategoriesPrice = categories.join(catPrice)

sumas_conteos = joinCategoriesPrice.map(lambda p: (p[1][0],(p[1][1],1))).reduceByKey(lambda a,b: (a[0]+b[0], a[1] +b[1]))

promedio_por_parent = sumas_conteos.mapValues(lambda sc: sc[0] / sc[1])

top5Parent = promedio_por_parent.takeOrdered(5, key=lambda kv: -kv[1])

for cat, avg in top5Parent:
    print(f"{cat}\t{avg:.2f}")

#Electronics        1504.47
#Automotive         1013.22
#Handmade           506.24
#Grocery & Gourmet Food         506.12
#Industrial & Scientific         505.82

```

Enlace al repositorio: https://github.com/MaurizioG28/TP_CIENCIA_DATOS.git