Optimizing Online Sports Retail Revenue

1. Counting missing values

Sports clothing and athleisure attire is a huge industry, worth approximately $193 billion in 2021 with a strong growth forecast over the next decade!

In this notebook, we play the role of a product analyst for an online sports clothing company. The company is specifically interested in how it can improve revenue. We will dive into product data such as pricing, reviews, descriptions, and ratings, as well as revenue and website traffic, to produce recommendations for its marketing and sales teams.

The database provided to us, `sports`, contains five tables, with `product_id` being the primary key for all of them:

`info`

column	data type	description
`product_name`	`varchar`	Name of the product
`product_id`	`varchar`	Unique ID for product
`description`	`varchar`	Description of the product

`finance`

column	data type	description
`product_id`	`varchar`	Unique ID for product
`listing_price`	`float`	Listing price for product
`sale_price`	`float`	Price of the product when on sale
`discount`	`float`	Discount, as a decimal, applied to the sale price
`revenue`	`float`	Amount of revenue generated by each product, in US dollars

`reviews`

column	data type	description
`product_name`	`varchar`	Name of the product
`product_id`	`varchar`	Unique ID for product
`rating`	`float`	Product rating, scored from `1.0` to `5.0`
`reviews`	`float`	Number of reviews for the product

`traffic`

column	data type	description
`product_id`	`varchar`	Unique ID for product
`last_visited`	`timestamp`	Date and time the product was last viewed on the website

`brands`

column	data type	description
`product_id`	`varchar`	Unique ID for product
`brand`	`varchar`	Brand of the product

We will be dealing with missing data as well as numeric, string, and timestamp data types to draw insights about the products in the online store. Let's start by finding out how complete the data is.

In [1]:
```
%%sql
postgresql:///sports

SELECT COUNT(*) AS total_rows,
COUNT(info.description) as count_description,
COUNT(finance.listing_price) as count_listing_price,
COUNT(traffic.last_visited) as count_last_visited
FROM info
INNER JOIN finance
ON finance.product_id = info.product_id
INNER JOIN traffic
ON traffic.product_id = finance.product_id
;
```

1 rows affected.

Out[1]:

total_rows	count_description	count_listing_price	count_last_visited
3179	3117	3120	2928

In [81]:
```
%%nose

last_output = _
last_output_df = last_output.DataFrame()

def test_columns():
    assert "total_rows" in set(last_output_df.columns), \
    """Did you alias the count of all products as "total_rows"?"""
    assert set(last_output_df.columns) == set(['total_rows', 'count_description', 'count_listing_price',
           'count_last_visited']), \
    """Did you select four columns and use the aliases in the instructions?"""

def test_shape():
    assert last_output_df.shape[0] == 1, \
    """Did you return a single row containing the count of values for each column?"""
    assert last_output_df.shape[1] == 4, \
    """Did you select four columns?"""

def test_values():
    assert last_output_df.values.tolist() == [[3179, 3117, 3120, 2928]], \
    """Did you correctly calculate the values for each column? Expected different results."""
```

Out[81]: 3/3 tests passed

2. Nike vs Adidas pricing

We can see the database contains 3,179 products in total. Of the columns we previewed, only one — `last_visited` — is missing more than five percent of its values. Now let's turn our attention to pricing.

How do the price points of Nike and Adidas products differ? Answering this question can help us build a picture of the company's stock range and customer market. We will run a query to produce a distribution of the `listing_price` and the count for each price, grouped by `brand`.

In [2]:

```
%%sql

SELECT brands.brand, CAST(finance.listing_price AS INTEGER), COUNT(finance.*)
FROM finance
INNER JOIN brands
ON brands.product_id = finance.product_id
WHERE finance.listing_price > 0
GROUP BY brands.brand, finance.listing_price
ORDER BY finance.listing_price DESC;
```

```
 * postgresql:///sports
77 rows affected.
```

Out[2]:

brand	listing_price	count
Adidas	300	2
Adidas	280	4
Adidas	240	5
Adidas	230	8
Adidas	220	11
Adidas	200	8
Nike	200	1
Adidas	190	7
Nike	190	2
Adidas	180	34
Nike	180	4
Adidas	170	27
Nike	170	14
Adidas	160	28
Nike	160	31
Adidas	150	41
Nike	150	6
Adidas	140	36
Nike	140	12
Adidas	130	96
Nike	130	12
Adidas	120	115
Nike	120	16
Adidas	110	91
Nike	110	17
Adidas	100	72
Nike	100	14
Adidas	96	2
Nike	95	1
Adidas	90	89
Nike	90	13
Adidas	86	7
Adidas	85	1
Nike	85	5
Adidas	80	322
Nike	80	16
Nike	79	1
Adidas	76	149
Adidas	75	1
Nike	75	7
Adidas	70	87
Nike	70	4
Adidas	66	102
Nike	65	1
Adidas	63	1
Adidas	60	211
Nike	60	2
Adidas	56	174
Adidas	55	2
Adidas	53	43
Adidas	50	183
Nike	50	5
Adidas	48	42
Nike	48	1
Adidas	46	163
Adidas	45	1
Nike	45	3
Adidas	43	51
Adidas	40	81
Nike	40	1
Adidas	38	24
Adidas	36	25
Adidas	33	24

Adidas	30	37
Nike	30	2
Adidas	28	38
Adidas	27	18
Adidas	25	28
Adidas	23	1
Adidas	20	8
Adidas	18	4
Adidas	16	4
Adidas	15	27
Adidas	13	27
Adidas	12	1
Adidas	10	11
Adidas	9	1

In [83]:

```
%%nose

last_output = _
last_output_df = last_output.DataFrame()

def test_columns():
    assert set(last_output_df.columns) == set(['brand', 'count', 'listing_price']), \
    """Did you select the correct columns and alias the first as "total_rows"?"""

def test_shape():
    assert last_output_df.shape[0] == 77, \
    """Did you correctly aggregate by brand? Expected the output to contain 77 products."""
    assert last_output_df.shape[1] == 3, \
    """The output should contain three columns: "brand", "listing_price", and "count"?"""

def test_values():
    assert last_output_df.iloc[0].values.tolist() == ['Adidas', 300, 2], \
    """Did you sort the results by "listing_price" in descending order?"""
```

Out[83]: 3/3 tests passed

3. Labeling price ranges

It turns out there are 77 unique prices for the products in our database, which makes the output of our last query quite difficult to analyze.

Let's build on our previous query by assigning labels to different price ranges, grouping by `brand` and `label`. We will also include the total `revenue` for each price range and `brand`.

In [3]:

```
%%sql

SELECT b.brand, count(f.*), sum(f.revenue) as total_revenue,
CASE WHEN f.listing_price < 42 THEN 'Budget'
WHEN f.listing_price >= 42 AND f.listing_price < 74 THEN 'Average'
WHEN f.listing_price >= 74 AND f.listing_price < 129 THEN 'Expensive'
ELSE 'Elite'
END AS price_category
FROM finance as f
INNER JOIN brands as b
ON f.product_id = b.product_id
WHERE b.brand IS NOT NULL
GROUP BY b.brand, price_category
ORDER BY total_revenue DESC;
```

```
 * postgresql:///sports
8 rows affected.
```

Out[3]:

brand	count	total_revenue	price_category
Adidas	849	4626980.069999999	Expensive
Adidas	1060	3233661.060000001	Average
Adidas	307	3014316.8299999987	Elite
Adidas	359	651661.1200000002	Budget
Nike	357	595341.0199999992	Budget
Nike	82	128475.59000000003	Elite
Nike	90	71843.15000000004	Expensive
Nike	16	6623.5	Average

In [85]:

```
%%nose

last_output = _
last_output_df = last_output.DataFrame()

def test_columns():
    assert set(last_output_df.columns) == set(['brand', 'price_category', 'count', 'total_revenue']), \
    """Did you select the correct columns? Expected "brand", "price_category", "count", and "total_revenue"."""

def test_shape():
    assert last_output_df.shape[0] == 8, \
    "Did you group by brand and labels? Expected there to be eight rows."
    assert last_output_df.shape[1] == 4, \
    "Did you select four columns?"

def test_values():
    assert last_output_df[:4].values.tolist() == [['Adidas', 849, 4626980.069999999, 'Expensive'],
     ['Adidas', 1060, 3233661.060000001, 'Average'],
     ['Adidas', 307, 3014316.8299999987, 'Elite'],
     ['Adidas', 359, 651661.1200000002, 'Budget']], \
    "Did you correctly calculate values for Adidas products? Expected something different."
    assert last_output_df[4:].values.tolist() == [['Nike', 357, 595341.0199999992, 'Budget'],
     ['Nike', 82, 128475.59000000003, 'Elite'],
     ['Nike', 90, 71843.15000000004, 'Expensive'],
     ['Nike', 16, 6623.5, 'Average']], \
    "Did you correctly calculate values for Nike products? Expected something different."
```

Out[85]: 3/3 tests passed

4. Average discount by brand

Interestingly, grouping products by brand and price range allows us to see that Adidas items generate more total revenue regardless of price category! Specifically, `"Elite"` Adidas products priced $129 or more typically generate the highest revenue, so the company can potentially increase revenue by shifting their stock to have a larger proportion of these products!

Note we have been looking at `listing_price` so far. The `listing_price` may not be the price that the product is ultimately sold for. To understand `revenue` better, let's take a look at the `discount`, which is the percent reduction in the `listing_price` when the product is actually sold. We would like to know whether there is a difference in the amount of `discount` offered between brands, as this could be influencing `revenue`.

In [4]:
```
%%sql

SELECT b.brand, AVG(f.discount) * 100 AS average_discount
FROM brands AS b
INNER JOIN finance AS f
    ON b.product_id = f.product_id
GROUP BY b.brand
HAVING b.brand IS NOT NULL
ORDER BY average_discount;
```

```
 * postgresql:///sports
2 rows affected.
```

Out[4]:

brand	average_discount
Nike	0.0
Adidas	33.452427184465606

In [87]:
```
%%nose

last_output = _
last_output_df = last_output.DataFrame()

def test_columns():
    assert set(last_output_df.columns) == set(['brand', 'average_discount']), \
    """Did you select the correct columns? Expected "brand" and "average_discount"."""

def test_shape():
    assert last_output_df.shape[0] == 2, \
    "Did you group by brand? Expected two rows, one per brand."
    assert last_output_df.shape[1] == 2, \
    "Did you select two columns?"

def test_values():
    assert last_output_df.iloc[:, 1].values.tolist() == [0.0, 33.452427184465606], \
    "Did you correctly calculate the average discount for the two brands?"
```

Out[87]:
```
3/3 tests passed
```

5. Correlation between revenue and reviews

Strangely, no `discount` is offered on Nike products! In comparison, not only do Adidas products generate the most revenue, but these products are also heavily discounted!

To improve revenue further, the company could try to reduce the amount of discount offered on Adidas products, and monitor sales volume to see if it remains stable. Alternatively, it could try offering a small discount on Nike products. This would reduce average revenue for these products, but may increase revenue overall if there is an increase in the volume of Nike products sold.

Now explore whether relationships exist between the columns in our database. We will check the strength and direction of a correlation between `revenue` and `reviews`.

In [5]:
```
%%sql

SELECT corr(r.reviews, f.revenue) as review_revenue_corr
FROM reviews as r
INNER JOIN finance as f
ON r.product_id = f.product_id
```

```
 * postgresql:///sports
1 rows affected.
```

Out[5]:

review_revenue_corr
0.6518512283481301

In [89]:
```
%%nose

last_output = _
last_output_df = last_output.DataFrame()

def test_columns():
    assert set(last_output_df.columns) == set(['review_revenue_corr']), \
    """Did you calculate the correlation between reviews and revenue, aliasing as "review_revenue_corr"?"""

def test_shape():
    assert last_output_df.shape == (1, 1), \
    "Did you calculate the correlation between reviews and revenue?"

def test_values():
    assert last_output_df.values.tolist() == [[0.6518512283481301]], \
    "Did you correctly calculate how reviews correlates with revenue?"
```

Out[89]:
```
3/3 tests passed
```

6. Ratings and reviews by product description length

Interestingly, there is a strong positive correlation between `revenue` and `reviews`. This means, potentially, if we can get more reviews on the company's website, it may increase sales of those items with a larger number of reviews.

Perhaps the length of a product's `description` might influence a product's `rating` and `reviews` — if so, the company can produce content guidelines for listing products on their website and test if this influences `revenue`. Let's check this out!

In [6]:
```
%%sql

SELECT TRUNC(length(i.description), -2) as description_length, ROUND(AVG(r.rating::numeric), 2) as average_rating
FROM reviews as r
INNER JOIN info as i
ON r.product_id = i.product_id
WHERE i.description IS NOT NULL
GROUP BY description_length
ORDER BY description_length;
```

```
 * postgresql:///sports
7 rows affected.
```

Out[6]:

description_length	average_rating
0	1.87
100	3.21
200	3.27
300	3.29

400	3.32
500	3.12
600	3.65

In [91]:

```
%%nose

last_output = _
last_output_df = last_output.DataFrame()

def test_columns():
    assert set(last_output_df.columns) == set(['description_length', 'average_rating']), \
    """Did you select the correct columns use the aliases "description_length" and "average_rating"?"""

def test_shape():
    assert last_output_df.shape[0] == 7, \
    """Did you create bins of 100 characters for "description_length"? Expected the output to contain seven rows."""
    assert last_output_df.shape[1] == 2, \
    "Expected the output to contain two columns."

def test_values():
    last_output_df = last_output.DataFrame().values.astype("float")
    assert last_output_df[0].tolist() == [0.0, 1.87], \
    """Did you sort the results by "description_length" in ascending order?"""
    assert last_output_df[-1].tolist() == [600.0, 3.65], \
    "Did you correctly calculate the results? Expected a different average rating for the largest description length bin."
```

Out[91]:

```
3/3 tests passed
```

7. Reviews by month and brand

Unfortunately, there doesn't appear to be a clear pattern between the length of a product's `description` and its `rating`.

As we know a correlation exists between `reviews` and `revenue`, one approach the company could take is to run experiments with different sales processes encouraging more reviews from customers about their purchases, such as by offering a small discount on future purchases.

Let's take a look at the volume of `reviews` by month to see if there are any trends or gaps we can look to exploit.

In [7]:

```
%%sql

SELECT b.brand, EXTRACT(MONTH from t.last_visited) as month, count(r.product_id) as num_reviews
FROM traffic as t
INNER JOIN reviews as r
ON t.product_id = r.product_id
INNER JOIN brands as b
ON r.product_id = b.product_id
GROUP BY brand, month
HAVING b.brand IS NOT NULL and EXTRACT(MONTH from t.last_visited) IS NOT NULL
ORDER BY brand, month;
```

```
 * postgresql:///sports
24 rows affected.
```

Out[7]:

brand	month	num_reviews
Adidas	1	253
Adidas	2	272
Adidas	3	269
Adidas	4	180
Adidas	5	172
Adidas	6	159
Adidas	7	170
Adidas	8	189
Adidas	9	181
Adidas	10	192
Adidas	11	150
Adidas	12	190
Nike	1	52
Nike	2	52
Nike	3	55
Nike	4	42
Nike	5	41
Nike	6	43
Nike	7	37
Nike	8	29
Nike	9	28
Nike	10	47
Nike	11	38
Nike	12	35

In [93]:

```
%%nose

last_output = _
last_output_df = last_output.DataFrame()

def test_columns():
    assert set(last_output_df.columns) == set(['brand', 'month', 'num_reviews']), \
    """Did you select the correct columns? Expected "brand", "month", and "num_reviews"."""

def test_shape():
    assert last_output_df.shape[0] == 24, \
    "Did you group by brand and month?"
    assert last_output_df.shape[1] == 3, \
    "Did you select three columns?"

def test_values():
    assert last_output_df.iloc[0].values.tolist() == ['Adidas', 1.0, 253], \
    "Expected the first row to contain the number of reviews for Adidas products in January."
    assert last_output_df.iloc[-1].values.tolist() == ['Nike', 12.0, 35.0], \
    "Expected the last row to contain the number of reviews for Nike products in December."
    assert max(last_output_df["num_reviews"]) == 272, \
    "Did you correctly calculate the number of reviews? Expected the largest number of reviews to be 272 for Adidas products in February."
```

Out[93]: 3/3 tests passed

8. Footwear product performance

Looks like product reviews are highest in the first quarter of the calendar year, so there is scope to run experiments aiming to increase the volume of reviews in the other nine months!

So far, we have been primarily analyzing Adidas vs Nike products. Now, let's switch our attention to the type of products being sold. As there are no labels for product type, we will create a Common Table Expression (CTE) that filters `description` for keywords, then use the results to find out how much of the company's stock consists of footwear products and the median `revenue` generated by these items.

In [8]:
```
%%sql

WITH footwear AS
(SELECT i.description, f.revenue
 FROM info as i
 INNER JOIN finance as f
 ON i.product_id = f.product_id
 WHERE description ILIKE '%shoe%' OR description ILIKE '%trainer%' OR description ILIKE '%foot%' AND description IS NOT NULL
)

SELECT COUNT(*) AS num_footwear_products,
    percentile_disc(0.5) WITHIN GROUP (ORDER BY revenue) AS median_footwear_revenue
FROM footwear;
```

```
 * postgresql:///sports
1 rows affected.
```

Out[8]:

num_footwear_products	median_footwear_revenue
2700	3118.36

In [95]:
```
%%nose

last_output = _
last_output_df = last_output.DataFrame()

def test_columns():
    assert set(last_output_df.columns) == set(['num_footwear_products', 'median_footwear_revenue']), \
    "Did you select the correct columns and use the aliases specified in the instructions?"

def test_shape():
    assert last_output_df.shape[0] == 1, \
    "Expected the output to contain one row."
    assert last_output_df.shape[1] == 2, \
    "Expected the output to contain two columns."

def test_values():
    assert last_output_df.iloc[0,0] == 2700, \
    "Did you count the number of footwear products?"
    assert last_output_df.iloc[0,1] == 3118.36, \
    "Did you calculate the median revenue for footwear products?"
```

Out[95]: 3/3 tests passed

9. Clothing product performance

Recall from the first task that we found there are 3,117 products without missing values for `description`. Of those, 2,700 are footwear products, which accounts for around 85% of the company's stock. They also generate a median revenue of over $3000 dollars!

This is interesting, but we have no point of reference for whether footwear's `median_revenue` is good or bad compared to other products. So, for our final task, let's examine how this differs to clothing products. We will re-use `footwear`, adding a filter afterward to count the number of products and `median_revenue` of products that are not in `footwear`.

In [9]:
```
%%sql

WITH footwear AS
(SELECT i.description, f.revenue
 FROM info as i
 INNER JOIN finance as f
 ON i.product_id = f.product_id
 WHERE description ILIKE '%shoe%' OR description ILIKE '%trainer%' OR description ILIKE '%foot%' AND description IS NOT NULL
)

SELECT count(i.product_id) as num_clothing_products, PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY f.revenue) as median_clothing_revenue
FROM info as i
INNER JOIN finance as f
ON i.product_id = f.product_id
WHERE i.description NOT IN (SELECT description FROM footwear);
```

```
 * postgresql:///sports
1 rows affected.
```

Out[9]:

num_clothing_products	median_clothing_revenue
417	503.82

In [97]:
```
%%nose

last_output = _
last_output_df = last_output.DataFrame()

def test_columns():
    assert set(last_output_df.columns) == set(['num_clothing_products', 'median_clothing_revenue']), \
    "Did you select the correct columns and use the aliases specified in the instructions?"

def test_shape():
    assert last_output_df.shape[0] == 1, \
    "Expected the output to contain one row."
    assert last_output_df.shape[1] == 2, \
    "Expected the output to contain two columns."

def test_values():
    assert last_output_df.iloc[0,0] == 417, \
    "Did you count the number of clothing products? Expected there to be 417 items."
    assert last_output_df.iloc[0,1] == 503.82, \
    "Did you calculate the median revenue for clothing products? Expected it to be $503.82."
```

Out[97]: 3/3 tests passed