First Steps for Lipstick Data

Jazelle Saligumba

Planning and Exploration

My first step was to gather a dataset of all lipsticks on Sephora with variables like price, brand name, and number of reviews. This was already tricky as Sephora doesn't have a public API anymore and isn't friendly to common parsers like Selenium or BeautifulSoup; their bot-detection is extremely strict, and a day's attempt to break past it proved unsuccessful, which is also ethically questionable. Past projects that scraped Sephora are dated (they used the official API when it existed or bypassed detection more easily). To stay within Sephora's site constraints, I did not and could not use automated scraping and had to figure out a manual way to collect the data.

Methods

Google Chrome's "Inspect Element" exposes what the page loads, including relevant text and the HTML. But Sephora is a dynamic site, unlike a static one, it doesn't render everything at once. You have to scroll to load more products, so a simple "Save Page" or copy-pasted HTML won't capture all items. Regularly copying the HTML of www.sephora.com/shop/lips-makeup still misses a lot of crucial information.

To capture everything, I used the Network tab in DevTools and recorded while I manually scrolled and I could grab the same JSON the page fetches as I browse. The full Lips category is huge (500+ products), so I narrowed to **lipsticks** only (144 products) both to focus the analysis and keep the dataset manageable and clean.

The Capture Process

I opened https://www.sephora.com/shop/lipstick, went to **Network**, checked **Preserve log**, hit **Refresh**, then scrolled slowly and clicked "Load more" until I reached the end. Since I'm looking at longer-term patterns and not day-to-day changes, a single careful capture is enough for this stage.

From the 1200+ recorded requests, I pulled the relevant ones. The data ended up split across **one HTML file** (with inline product objects) and **two JSON files**. With 144 products, the site effectively paged into three chunks (max 60 per page). The first batch landed in the HTML; the second and third were JSON.

I saved those responses as `page1.html`, `page2.json`, and `page3.json`, then parsed them into a single CSV with help from ChatGPT. Below are the core code chunks I used.

```
import json, re
from pathlib import Path
from urllib.parse import urljoin


HTML_PATH = Path("page1.html")
JSON_FILES = [Path("page2.json"), Path("page3.json")]
OUT_CSV = Path("sephora_products.csv")


BASE = "https://www.sephora.com"

def to_float(x):
    if x is None: return None
    s = str(x).strip().replace(",", "")
    if s.startswith("$"): s = s[1:]
    try: return float(s)
    except ValueError: return None

def to_int(x):
    if x is None: return None
    try: return int(str(x).replace(",", "").strip())
    except ValueError: return None

def first(*vals):
    for v in vals:
        if v not in (None, "", [], {}):
            return v
    return None

def norm_url(u):
    if not u: return None
    return urljoin(BASE, u)
```

First, I extract product-like JSON objects embedded directly in the HTML (using a small

brace-matching parser):

```
def extract_inline_products_from_html(html_text: str):
    """
    Finds JSON objects that look like products embedded in HTML/JS.
    We scan for '{"brandName"' starts and then brace-match to the closing '}'.
    """
    objs = []
    needle = '{"brandName"'
    i = 0
    n = len(html_text)
    while True:
        start = html_text.find(needle, i)
        if start == -1:
            break
        # back up one char if there's a leading '{' before the needle
        # (our needle already starts with '{', so 'start' is at the '{')
        brace_count = 0
        j = start
        in_str = False
        esc = False
        # brace-match until we close the top-level object
        while j < n:
            ch = html_text[j]
            if in_str:
                if esc:
                    esc = False
                elif ch == '\\':
                    esc = True
                elif ch == '"':
                    in_str = False
            else:
                if ch == '"':
                    in_str = True
                elif ch == '{':
                    brace_count += 1
                elif ch == '}':
                    brace_count -= 1
                    if brace_count == 0:
                        # end of object
                        block = html_text[start:j+1]
                        # try to load JSON
                        try:
```

```
                            obj = json.loads(block)
                            objs.append(obj)
                        except Exception:
                            pass
                        i = j + 1
                        break
            j += 1
        else:
            # ran out of text
            break
    return objs
```

Next, I walk the nested JSON structures.

```
# ---------- B) Walk any JSON structure to yield product-like dicts ----------
def walk_products(node):
    """
    Yield dicts that look like product/sku records anywhere in JSON.
    """
    if isinstance(node, dict):
        if any(k in node for k in ("productId","displayName","productName","currentSku","bra
            yield node
        for v in node.values():
            yield from walk_products(v)
    elif isinstance(node, list):
        for v in node:
            yield from walk_products(v)
```

Then, due to their slightly different formats, the chunk below normalizes the data for consistency.

```
# ---------- C) Normalize to tidy rows ----------
def normalize(prod: dict):
    # IDs
    product_id = first(prod.get("productId"), prod.get("id"))
    current_sku = prod.get("currentSku") or {}
    sku_id = first(prod.get("skuId"),
                   (prod.get("sku") or {}).get("skuId"),
                   current_sku.get("skuId"))
    # Names
    product_name = first(prod.get("displayName"), prod.get("productName"), prod.get("name"))
```

```
    # Brand
    brand = prod.get("brand")
    brand_name = first(prod.get("brandName"),
                       (brand or {}).get("brandName"),
                       (brand or {}).get("name"))
    # Prices
    list_price = to_float(first(prod.get("listPrice"),
                                current_sku.get("listPrice"),
                                (prod.get("price") or {}).get("listPrice"),
                                (prod.get("price") or {}).get("formattedPrice")))
    sale_price = to_float(first(prod.get("salePrice"),
                                current_sku.get("salePrice"),
                                (prod.get("price") or {}).get("salePrice")))
    # Rating / reviews
    rating = to_float(first(prod.get("rating"),
                            prod.get("starRatings"),
                            prod.get("reviewRating")))
    reviews = to_int(first(prod.get("reviews"),
                           prod.get("reviewsCount"),
                           prod.get("reviewCount")))
    # URLs
    url = first(prod.get("targetUrl"), prod.get("productUrl"),
                prod.get("canonicalUrl"), prod.get("targetURL"))
    if url and isinstance(url, str) and url.startswith("/"):
        url = norm_url(url)
    if not url and isinstance(product_id, str) and product_id.startswith("P"):
        url = f"{BASE}/product/{product_id}"
    # Images
    image_url = first(
        prod.get("heroImage"),
        prod.get("altImage"),
        (prod.get("skuImages") or {}).get("imageUrl") if isinstance(prod.get("skuImages"), di
        (current_sku.get("skuImage") or {}).get("imageUrl"),
        (prod.get("image") or {}).get("src"),
        (prod.get("image") or {}).get("url"),
        (prod.get("images")[0].get("src") if isinstance(prod.get("images"), list) and prod["
    )
    return {
        "skuId": sku_id,
        "productId": product_id,
        "brandName": brand_name,
        "productName": product_name,
```

```
        "listPrice": list_price,
        "salePrice": sale_price,
        "rating": rating,
        "reviews": reviews,
        "imageUrl": image_url,
        "url": url,
        "isNew": bool(current_sku.get("isNew", False)),
        "isLimitedEdition": bool(current_sku.get("isLimitedEdition", False)),
        "isSephoraExclusive": bool(current_sku.get("isSephoraExclusive", False)),
    }
```

The rest of the code calls in the defined functions, loads the data, and creates the CSV file.

```
# ---------- D) Load everything, dedupe, and export ----------
all_nodes = []

# 1) HTML inline products
if HTML_PATH.exists():
    html_text = HTML_PATH.read_text(encoding="utf-8", errors="ignore")
    inline = extract_inline_products_from_html(html_text)
    all_nodes.extend(inline)
else:
    print(f"Warning: HTML not found: {HTML_PATH}")

# 2) JSON files (page2.json, page3.json, etc.)
for jf in JSON_FILES:
    if jf.exists():
        try:
            data = json.loads(jf.read_text(encoding="utf-8"))
            all_nodes.extend(walk_products(data))
        except Exception as e:
            print(f"Skipping {jf.name} (JSON load error): {e}")
    else:
        print(f"Warning: JSON not found: {jf}")

# 3) Normalize + dedupe by skuId (fallback to productId)
rows, seen = [], set()
for node in all_nodes:
    if not isinstance(node, dict):
        continue
    row = normalize(node)
    key = row["skuId"] or (f"PID:{row['productId']}" if row["productId"] else None)
```

```
    # require at least name or brand to avoid noise
    if not key or not (row["productName"] or row["brandName"]):
        continue
    if key in seen:
        continue
    seen.add(key)
    rows.append(row)

print(f"Parsed {len(rows)} unique products.")
for r in rows[:10]:
    print("-", r["brandName"], "|", r["productName"], "|", r["listPrice"], "|", r["url"])

# 4) Save CSV
if rows:
    import csv
    OUT_CSV.parent.mkdir(parents=True, exist_ok=True)
    with OUT_CSV.open("w", newline="", encoding="utf-8") as f:
        w = csv.DictWriter(f, fieldnames=rows[0].keys())
        w.writeheader()
        w.writerows(rows)
    print("Wrote:", OUT_CSV)
```

```
Parsed 144 unique products.
- SEPHORA COLLECTION | Cream Lip Stain 10HR Liquid Lipstick | 16.0 | https://www.sephora.com/
lip-stain-liquid-lipstick-P281411?skuId=2760981
- Anastasia Beverly Hills | Full-Pigment Matte & Satin Velvet Lipstick | 26.0 | https://www.s
beverly-hills-matte-satin-velvet-lipstick-P480576?skuId=2882280
- SEPHORA COLLECTION | Satin Hydrating Lipstick | 16.0 | https://www.sephora.com/product/sati
hydrating-lipstick-P501496?skuId=2564474
- Fenty Beauty by Rihanna | Gloss Bomb Stix High-Shine Gloss Stick | 26.0 | https://www.sepho
beauty-rihanna-gloss-bomb-stix-high-shine-gloss-stick-P511572?skuId=2787497
- SEPHORA COLLECTION | Matte Velvet Lipstick | None | https://www.sephora.com/product/matte-
velvet-lipstick-P506548?skuId=2666840
- MERIT | Signature Lip Lightweight Lipstick | 26.0 | https://www.sephora.com/product/merit-
signature-lip-lightweight-lipstick-P481403?skuId=2792802
- DIOR | Dior Addict Shine Lipstick | None | https://www.sephora.com/product/dior-
dior-addict-refillable-shine-lipstick-P481969?skuId=2767093
- MAKEUP BY MARIO | SuperSatin® Lipstick | 28.0 | https://www.sephora.com/product/makeup-
by-mario-supersatin-lipstick-P509305?skuId=2731636
- DIOR | Rouge Dior Refillable Lipstick | 50.0 | https://www.sephora.com/product/dior-
rouge-dior-lipstick-P467760?skuId=2750966
```

```
- Charlotte Tilbury | K.I.S.S.I.N.G Satin Shine Lipstick | None | https://www.sephora.com/pr
i-s-s-i-n-g-lipstick-P433531?skuId=2736056
Wrote: sephora_products.csv
```

Results

This created a sephora_products.csv file with 144 rows with each one to represent a lipstick and variables like brandName, productName, reviews, and price. The .csv is within the folder.

Next Steps

This is just one of the datasets I will use. Given the manual capturing, this approach is extremely unfeasible for my next dataset which is recording individual lipstick reviews (to use the posted date as a proxy for sales), where many products have over a thousand reviews. I found a recent project that successfully scraped Sephora Malaysia. Unlike the U.S. site, Sephora Malaysia uses a different structure, and the project's BeautifulSoup pipeline works cleanly there. While there are many differences such as fewer overlapping products, a site geared more toward an Asian audience, and prices listed in RMB, it's still useful because many Malaysian product pages carry over reviews from Sephora U.S., explicitly marked "Reviewed in United States" and "Originally posted on sephora.com." Reviews can also be filtered by location.

My plan is to start with that pipeline to prototype review parsing and then use the **United States** reviews for my next dataset, paired with the union of lipstick SKUs present in both countries. Given this direction, Professor Cordova suggested a comparative exploration of the U.S. economic cycle versus Asia's (the Malaysia site also skews toward Asia and Oceania in its reviews), which is an interesting but ambitious addition.

Reflection

Even though I discovered both the roadblock on the U.S. Sephora site early and Sephora Malaysia as the easier workaround, I wanted to challenge myself to think outside the box and see what I could do without leaning on familiar packages. I was also curious about applying this approach to other sites, especially as many companies have deprecated their public APIs in recent years. The manual capturing allowed me to understand the contents and structure of a webpage more, and to be much less of a black box that the 'Inspect Element' once was for me.